Novel Techniques for Hardware/Software Partitioning and Emulation

by
Iakovos Mavroidis

A dissertation submitted in partial fulfillment of the
requirements of the degree of
Doctor of Philosophy

in the

Department of Electronic and Computer Engineering
of the
Technical University of Crete at Greece

Committee in charge:

Professor Ioannis Papaefstathiou
Professor Apostolos Dollas
Professor Dionisios Pnevmatikatos

April 2011

Abstract

Novel Techniques for Hardware/Software Partitioning and Emulation

Over the last several years, uniprocessor systems, in an effort to overcome the limits of deeper pipelining, instruction-level parallelism and power dissipation, evolved from one processing core to tens or hundreds of cores. At the same time, multi-chip systems and Systems on Board (SoB), have started giving their place to Systems on Chip (SoC) that exploit the latest nanometer technologies. This has also caused a tremendous shift in the system development process towards embedded systems, hardware/software co-design, SoC designs, multi-core designs, and hardware accelerators. Nowadays, one of the key issues for continued performance scaling is the development of advanced CAD tools that can efficiently support the design and verification of these new platforms and the requirements of today's complex applications.

This thesis focuses on three important aspects of the system development process: hardware/software partitioning, simulation and verification. Since the time consumed in those tasks is usually a large percentage of the overall development time, speeding them up can significantly reduce the ever important time to market.

Hardware emulation on FPGAs has been widely used as a significantly faster and more accurate approach for the verification of complex designs than software simulation. In this approach, Hardware Simulation Accelerator and Emulator co-processor units are used to offload calculation-intensive tasks from software simulators. One of the biggest problems however is that the communication overhead between the software simulator, where the behavioral testbench usually runs, and the hardware emulator where the Design Under Test (DUT) is emulated, is becoming a new critical bottleneck. Another problem is that in a hardware emulation environment it is impossible to bring outside of the chip a large number of internal signals for verification purposes. Therefore, on-chip observability has become a significant issue. Finally, one more

crucial issue is the decision that has to be made on how to partition the system components into two distinct sets: those that will be implemented in hardware and those that will run in software. In this thesis we analyze all the aforementioned problems and propose novel techniques that can be used to attack them.

First, we introduce a novel emulation framework that automatically transforms certain HDL parts of the testbench into synthesizable code in order to offload them from the software simulator and, more importantly, minimize the aforementioned communication overhead. In particular, we partition the testbench running on the software simulator into two sections: the testbench HDL code that communicates directly with the DUT and the rest, C-like, testbench code. The former section is transformed into synthesizable code while the latter runs in a general purpose CPU. Next, we extend this architecture by adding multiple fast scan-chain paths in the design in order to provide full circuit observability and controllability on the fly. Finally, we develop a fully automated hardware/software partitioning tool that incorporates a novel flow with new cost metrics and functions to provide fast and efficient solutions. The tool employs two separate partitioning algorithms; Simulated Annealing (SA) and a novel greedy algorithm, the Grouping Mapping Partitioning (GMP).

Our experiments demonstrate that our methodologies provide cost-effective solutions for the hardware/software partitioning and emulation of large and complex systems.

Abbreviations

API	*Application Programming Interface*
ASIC	*Application Specific Integrated Circuit*
CAD	*Computer-Aided Design*
COTS	*Commercial-Off-The-Shelf*
CPU	*Central Processing Unit*
CPS	*Cycles Per Second*
DCM	*Digital Clock Management*
DSP	*Digital Signal Processing*
DUT	*Design Under Test*
EDA	*Electronic Design Automation*
ELA	*Embedded Logic Analyzer*
ESL	*Electronic System Level*
FF	*Flip-Flop*
FIFO	*First-In-First-Out*
FPGA	*Field Programmable Gate Array*
GA	*Genetic Algorithm*
GMP	*Grouping Mapping Partitioning*
GPGPU	*General Purpose Graphics Processor Unit*
GUI	*Graphical User Interface*
HDL	*Hardware Description Language*
HPC	*High Performance Computing*
HW	*HardWare*
I/F	*Interface*
ISS	*Instruction Set Simulator*
MIPS	*Million Instructions Per Second*
MTPS	*Million Transactions Per Second*
OS	*Operating System*
OSCI	*Open SystemC Initiative*
PnR	*Place and Route*
PLI	*Programming Language Interface*
PLL	*Phase Lock Loop*
RAM	*Random Access Memory*
RC	*Reconfigurable Computing*
RISC	*Reduced Instruction Set Computer*
RTL	*Register Transfer Level*
SA	*Simulated Annealing*
SCE-MI	*Standard Co-Emulation API - Modeling Interface*
SDK	*Software Development Kit*
SIMD	*Single Instruction Multiple Data*
SoC	*System on Chip*
SW	*SoftWare*
TB	*TestBench*
TLM	*Transaction Level Modeling*
VHDL	*VHSIC Hardware Description Language*
VLSI	*Very Large Scale of Integration*

Contents

List of Figures

Acknowledgements

During the years of my studies, I had the opportunity to work with many wonderful people. This work would not be possible without their generous support.

First, I am very grateful to my advisor, Ioannis Papaefstathiou, for his overall guidance, support and friendship. He helped me develop a taste for research and encouraged me to all my efforts. His technical expertise, dedication, immense patience, and availability to students made him a great advisor. This thesis would not have been possible without his help.

I would like to thank Dionisios Pnevmatikatos and Apostolos Dollas who provided significant insight comments on my research. Our discussions have influenced my thoughts and research a lot.

I am especially grateful to Manolis Katevenis, my professor at University of Crete, for initiating me into computer architecture. I am also indebted to Dave Patterson, my MSc degree advisor at University of California at Berkeley, for his enthusiastic teaching and valuable advices.

I would like to thank the Greek Secretariat for Research and Technology (GSRT) and the European Union for their financial support.

I am also very thankful to all my friends for their moral support.

I am indebted to my brother Ioannis for countless hours of discussion, proofreading of draft papers, and feedback of any kind. Finally, I want to thank my parents, Evangelia and Manolis, and my brother Dimitrios, for their love, support and encouragement.

Chapter 1. Introduction

"No architecture is so haughty as that which is simple."

John Ruskin

Development and testing of large complex systems is a time consuming process that continuously evolves over time following the market requirements of every era. Nowadays, the rising design complexity combined with the reduced time-to-market window has revolutionized the embedded system design process. The traditional design techniques (i.e. independent hardware and software design) are now being challenged when heterogeneous models and applications are integrated in complex systems on chip. In hardware/software co-design, designers analyze the trade-offs in the way the hardware and the software components of a system work together so as to exhibit a specified behavior, given a set of performance goals and technology.

One very promising approach in developing such an embedded system is to use a hardware/software co-design platform in order to (a) partition the design into hardware and software components, (b) design all the partitioned components, (c) co-simulate the entire system, and (d) finally test and verify the system, as shown in Figure 1.1.

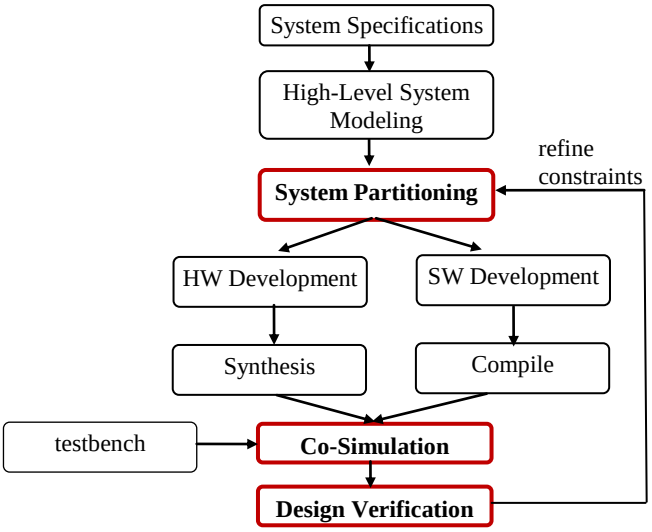

Figure 1.1. Hardware/Software Co-design Flow

This thesis focuses on three important aspects of the system development process: hardware/software partitioning, simulation and verification. The time spent in those tasks is usually such a significant portion of the overall development time, so that by reducing it we significantly reduce the (nowadays so important) time to market. In this thesis, first we introduce a novel hardware emulation framework that performs fast system simulations on an FPGA-based platform. Next, we extend this architecture by adding multiple fast scan-chain paths in the design in order to provide full circuit observability and controllability during the simulation. Finally, we propose a hardware/software partitioning methodology that provides cost-efficient solutions in a much faster way than traditional partitioning algorithms.

1.1 Background

Computer-aided design (CAD) tools have severally increased user productivity in recent decades. CAD tools help designers produce higher quality designs in less time, while powerful analysis and simulation tools help engineers develop better products in less time. As CAD tools have become more powerful, they have also become more complex and specialized. Numerous techniques have been employed in order to perform faster, more efficient and with less human involvement the required steps in the process of designing and verifying a new system.

Such widely used techniques pertained to the simulation and verification of complex designs as well as to the hardware/software partitioning of embedded systems are the following:

- Hardware simulation accelerators and emulators that can imitate the behavior of one or more pieces of hardware (typically a system under design) by executing simulations on another piece of hardware, typically a special purpose emulation system, which is significantly faster than if the simulation is executed on a general purpose CPU.
- Embedded logic analyzers that can be inserted inside an FPGA design in order to sample signals for analysis and verification.
- Partitioning frameworks and heuristics in order to decide which components of the system should be realized on hardware and which ones in software.

Next, we describe these techniques and their limitations.

1.1.1 Hardware Simulation Accelerator and Emulator

Simulation and verification of large complex systems is a time consuming process that requires significant man-power. The largest fraction of silicon integrated circuit respins are due to functional errors. Thus, comprehensive functional verification is a key factor in reducing development costs and delivering a product in time.

Functional verification of a design is most often performed using logic simulation and/or prototyping. There are advantages and disadvantages for each of those approaches and thus often both are used. Logic simulation is easy, accurate, flexible, and low cost. However, simulation is often not fast enough for large designs and almost always too slow to run the complete application software on top of the hardware design. Software simulation of embedded designs that need emulation of I/O interfaces or full emulation of embedded CPUs tend to be extremely slow.

In contrast to software simulations, FPGA-based prototypes are fast. It has been a common practice for hardware engineers to perform design validation on the FPGA hardware itself since back-end verification in hardware provides a faster, more accurate and closer-to-reality model than software simulations. Direct hardware execution is thousands of times faster than software

simulation, and the reconfigurability of FPGAs allows any design modifications to be recompiled and reloaded directly onto the FPGA. But the time required to implement a large design into several FPGAs can be very long and is error-prone. The changes needed in order to fix design flaws also take a long time to implement and may require board wiring changes. Since FPGA prototypes have limited debugging capabilities, probing signals inside the FPGAs in real time is very difficult, if not impossible, and recompiling the FPGA designs in order to change the probes takes too long. Moreover, such approach may delay significantly the development of an embedded system since the software can be tested only after the hardware implementation is complete. This is shown in Figure 1.2 where a typical schedule of an embedded system design is depicted. A flexible hardware/software co-simulation platform that can simulate and test the software and the hardware sections of the design in parallel is essential.

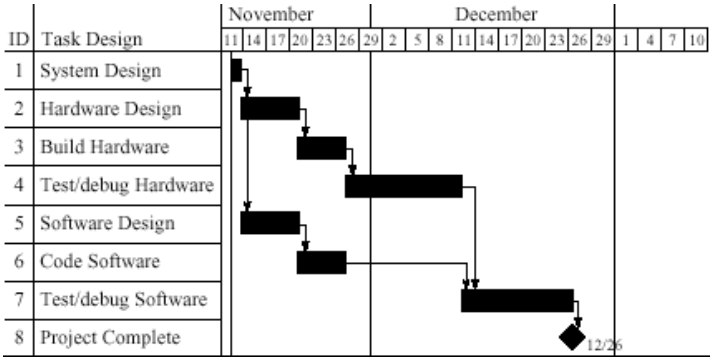

Figure 1.2. Typical embedded system project schedule

In the last decade, hardware simulation accelerators and hardware emulators have come into picture in order to tackle the aforementioned limitations of software simulations and FPGA-based prototypes. Hardware simulation accelerators**,** designed primarily to speed-up front-end simulation, have been available to large design centers with large budgets and extensive design tool support. The hardware accelerator schemes are based on using circuit boards populated with multiple special-purpose ASICs, each of which contains a number of specialized processors and lots of local memory (typically 80% to 90% of these devices are memory). In those systems, the High Description Language (HDL) representation of the design is compiled into machine code, which is subsequently distributed amongst the various processors.

The alternative to such systems, hardware emulators (or in-circuit emulators), have also been proposed as a moderate-cost solution, mainly satisfying the needs of back-end verification. The hardware emulator schemes are based on using circuit boards populated with FPGAs, in

which case the HDL design is typically synthesized into a gate-level equivalent, which is partitioned across, and loaded into, the various FPGAs. In this case a processor executes the non-synthesizable code such as the testbench.

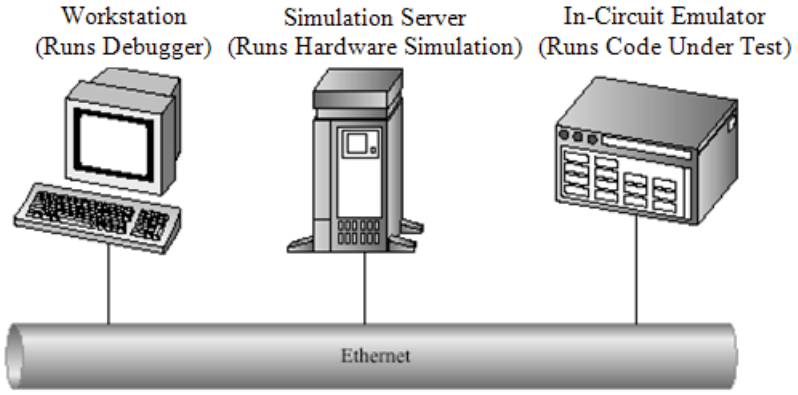

Figure 1.3. Hardware emulation

These approaches lighten the burden of hardware design verification by using custom hardware to aid the verification process. Hardware accelerator and emulator co-processor units are used to offload calculation-intensive tasks from software simulators. However, both techniques can address the performance shortcomings of software simulation only to a certain extent; even though the design is mapped into a hardware accelerator and thus it is executed much faster, the testbench (and any behavioral design code) continues to run on a CPU-based platform such as a workstation. A high-bandwidth, low latency channel connects the software simulation (workstation) and the hardware accelerator/emulator supporting the signal data exchange between the testbench and the design as shown in Figure 1.3. By Amdahl's law, the slowest device in the chain will determine the achievable speed and normally, this is either the testbench executed on the software simulator or the communication channel connecting the software simulator and the hardware emulator. With a very efficient testbench (written in C or transaction-based), the channel may become the bottleneck. So the communication overhead between the software simulator and the hardware emulator is becoming a new critical bottleneck.

Overall performance is typically limited by the communications channel between the emulator and the workstation and by the testbench execution time of the components running on the workstation. Therefore, we introduce a novel way to tackle this problem in Chapter 4 by partitioning the code running on the software simulator into two parts.

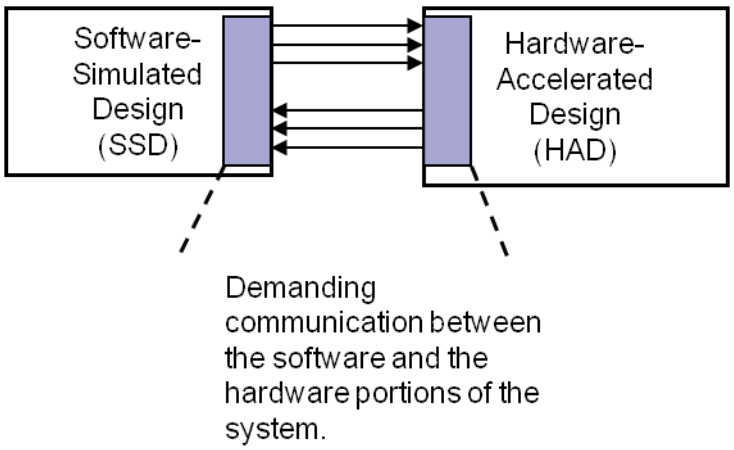

Figure 1.4. Software - Hardware communication overhead

1.1.2 Embedded Logic Analyzer

Even if the hardware-software communication overhead is heavily reduced, the existing hardware emulation schemes still face important limitations; in order to be very effective in the verification process, the hardware emulation framework should provide the same level of testability as a software HDL simulator does. With the term testability we imply observability (i.e. the ability to view or probe the output of a gate) and controllability (i.e. the ability to manipulate the inputs of a gate or the state of a flip-flop). Towards this end, FPGA vendors have provided integrated solutions, such as Embedded Logic Analyzers (ELAs), which show the transient behavior of the design. Such tools allow the designer to easily probe the internal signals of the design inside an FPGA, much as he/she would do with an external logic analyzer device.

Figure 1.5 shows an example ELA, where the Design Under Test (DUT) is running on the FPGA and a trigger event (for example an internal signal of the DUT reaching a specific value) determines when certain internal signals should be captured and stored in the internal memory of the FPGA. In this way, the ELA, that has internal access to the test buses, the clocks and certain test events, can be used to debug the actual chip. Additionally, the configuration of the ELA and the observations of the acquired results in the shared memory can be accessed through normal control interfaces of the chip and do not require special test cards.

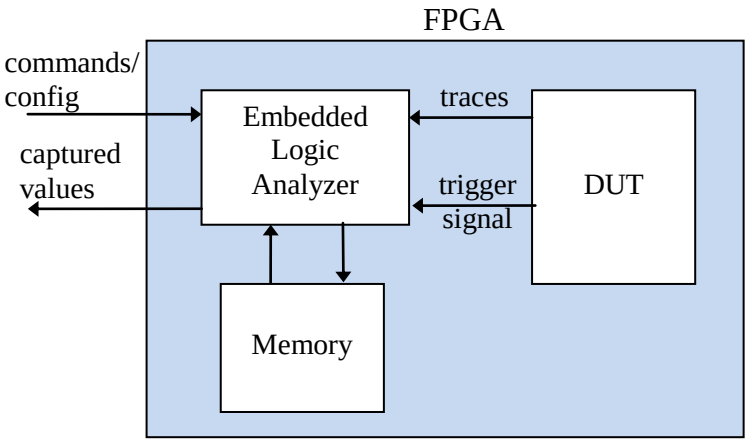

Figure 1.5. Architecture of Embedded Logic Analyzer

While ELAs lighten the burden of hardware design verification by adding on-FPGA circuitry in order to observe the design on-the-fly, the observability and controllability they provide are still limited. Compared to software HDL simulators, the existing ELAs have some important limitations:

- Changing specific parameters, such as the signal probes or the depth of the sample buffer, in most cases, requires a time-consuming full recompilation of the user design.
- ELAs utilize the limited FPGA memory in order to store their traces. As a result, their sample memory, which determines the maximum trace period, is limited by the memory resources of the FPGA. In a design that uses much of the FPGA's memory, there may not be enough memory left over for the ELA.
- Basic debug operations such as breakpoints, and step by step execution are not supported.
- There is no controllability of the design; the user cannot force an internal signal to a specific value.

We tackle all the aforementioned problems in Chapter 5 by extending the hardware emulator environment introduced in Chapter 4 with multiple fast scan chains added in the user’s design and organized in an innovative manner.

1.1.3 Hardware/Software Partitioning

While fast and efficient hardware emulators are very significant in the system development, CAD tools should also provide system level guidelines and techniques. Defining the system architecture in a software/hardware co-design environment is not a trivial process and any wrong decisions at that point may result in significant delays in the development time and/or in inefficient designs that cannot meet the system constraints. When considering the design costs/time, software implementation is almost always more cost efficient than hardware implementation. This is mainly because hardware development requires more effort and money than software development. As a result hardware/software co-design has emerged as a key first step in the design of complex embedded systems. The ever increasing design complexity and the ability of current FPGAs to utilize large numbers of embedded CPUs, as well as special purpose hardware modules, makes hardware-software co-design even more important.

Probably the most important aspect of co-design is the actual hardware/software partitioning. Hardware engineers usually partition their system into hardware and software entities at an early design stage. Even though the internal implementation and characteristics of a design usually are not well specified at the initial design phases, hardware-software partitioning is decided a priori and is adhered to as much as possible, because even small refinements in the partitioning may trigger extensive redesign. In general the most common design practice is to initially try to map everything in software, and then gradually off-load only the most time-critical parts of the design to hardware in order to meet the timing constraints.

Today's designs consist of several, usually hundreds of, *design components* which operate and communicate with each other in parallel. The design components can range from small modules such as FIFOs, arithmetic units, etc., to larger ones such as compression or encryption engines, Signal Processing Units, etc. The level of granularity at which partitioning is performed, which specifies the size of the design components, is determined at the beginning of the partitioning process. Instruction-level granularity, block-level granularity as well as function-level granularity have all been employed in the past (see Section 3.5). These design components will finally need to be mapped to and implemented by the available *system entities*, which can range from general-purpose CPUs, to FPGA slices or to custom-built ASIC gates. In general,

depending on the entity the implementation of the component will either be in software or in hardware.

The *Partitioning Problem* is, at its simplest form, a "best fit" problem that tries to identify the allocation of the design components into two distinct sets, those that will be executed in embedded CPUs and those that will be implemented in specific hardware modules; the overall aim is to minimize or maximize a given metric or a number of metrics.

Since executing a design component in a soft or hard-core CPU is always easier and more flexible than implementing the same functionality in hardware, a common practice is to try to map as many components as possible into software, making sure at the same time that all the performance requirements are met. As a result, most partitioning algorithms try to find the partitioning that minimizes the hardware footprint of the design.

Of course, since the partitioning algorithm is run a priori, in order to decide which components should be implemented in software and which in hardware, the algorithm usually has to deal with design components that are described in a more abstract behavioral model. For this reason, at an initial stage, most algorithms evaluate each design component according to several *cost metrics* such as its performance, power, and size if implemented in hardware. Similarly, *capacity metrics* are associated with each system entity, such as the maximum processing power or bandwidth the entity can provide.

In Reconfigurable Computing environments (see definition of RC in Section 2.1.2) the partitioning algorithm can be applied several times so as to create many different designs that can be altered at run time. Therefore, performing fast and efficient hardware/software partitioning is especially important in RC. Despite the extensive research on partitioning algorithms no commercial tool can yet provide a complete, fast and efficient solution. As our results demonstrate in Chapter 6 our tool provides clear, fast and effective solutions to all the partitioning problems that arise during the partitioning process.

1.2 Contributions

The main contributions of this thesis are threefold: a) we introduce an emulation platform that overcomes the communication problem of existing hardware emulators, b) we provide full

circuit observability and controllability of the emulated design and c) we propose a fast and efficient approach for hardware/software partitioning.

In Chapter 4 we propose a new approach that outperforms, in a number of real-world cases, all the current hardware emulator systems by a factor of more than 15. Moreover, our method is transparent to the designer (unlike some emulators that require re-writing of the testbench) and can also be used on top of existing emulation platforms triggering an even more significant acceleration.

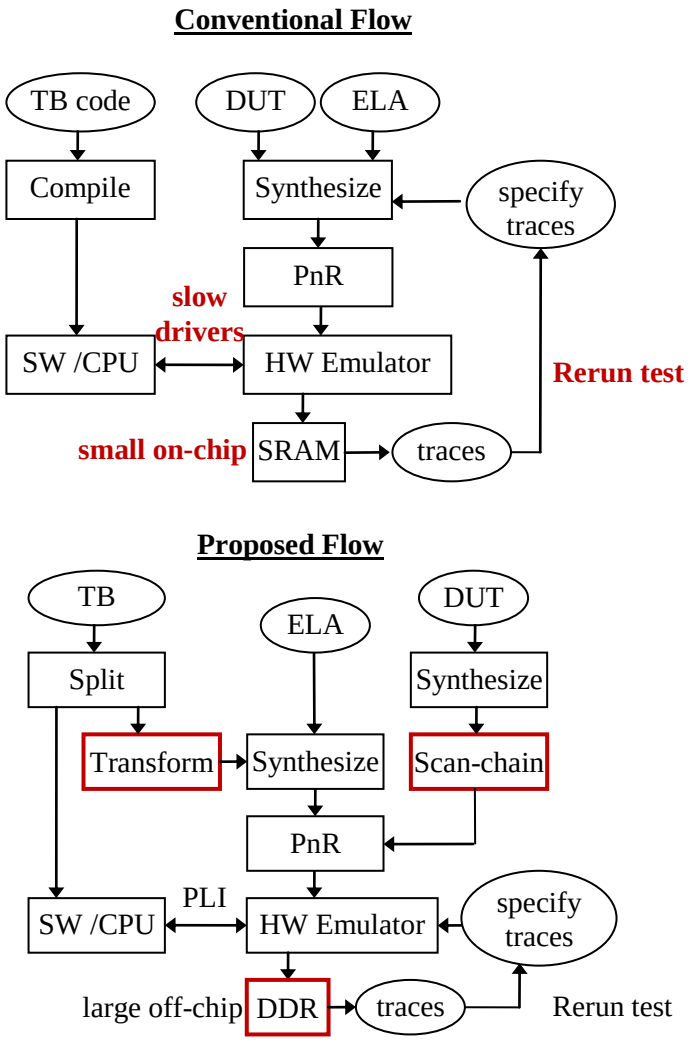

Figure 1.6. New Emulation Flow transforms portion of TB code into HW and adds scan-chains in DUT.

In Chapter 5, we tackle the problems of existing ELAs, described in Section 1.1.2, by extending the hardware emulator environment introduced in Chapter 4 with multiple fast scan chains added in the user's design and organized in an innovative manner. A scan circuitry links all the storage elements of a design, or part of them, and eventually creates a large shift register, the so-called scan chain. The major advantage of this approach lies in the fact that, in the scan mode, we have full observabiliy and controllability of the memory elements included in the scan chains.

Figure 1.6 shows a conventional emulation flow and our modified flow. The conventional flow executes the testbench code on a CPU since it is a non-synthesizable code. Instead of that, we split the testbench code and synthesize a certain portion of it. Moreover, we add scan chains in the synthesized DUT. The main advantages of these innovative techniques, when compared with the existing schemes are:

- The accesses on the Programming Language Interface (PLI), shown in Figure 1.6, are much more infrequent than the accesses at the DUT-testbench boundaries. By ***implementing a portion of the testbench on HW*** we managed to reduce significantly the communication overhead between SW and HW, and the testbench execution time outperforming conventional hardware emulation systems by a factor of more than 15.
- By ***adding scan-chains in the DUT*** the ELA can easily trace any signal in the DUT. In this way, in order to rerun a test with different set of traces or set a new trigger condition the user simply has to modify the configuration memory of the ELA, instead of performing a very time consuming design re-synthesis, re-placement and re-routing. This is feasible because the set of traces and the trigger condition are specified in the configuration memory of the ELA (see Section 5.4) instead of being hardwired in the ELA.
- Our scheme ***holds the traces of the emulation in an external large memory*** instead of the limited on-chip FPGA memory utilized by the existing schemes. This is feasible due to the breakpoint operation our methodology provides (see Section 4.8).
- Run-time modifications of the values of any of the internal signals of the DUT during execution can be easily performed through the scan chains.

These advantages are more significant in complex designs or when time to market is a critical factor. Simulators simply are too slow to fully verify complex embedded designs at gate level and provide insufficient feedback to the developer. We propose a platform for efficient evaluation of complex designs where any modification in the implementation has to be verified through numerous regression tests.

Regarding the hardware/software partitioning problem, we propose in Chapter 6 an innovative and fast approach so as to provide cost-efficient systems in a timely manner. Our novel algorithm produces very similar results (the difference is less than 3%) to those triggered by the most widely used algorithms (such as simulated annealing) while it is more than 2500 times faster. Performing fast hardware/software partitioning is especially important in Reconfigurable Computing (RC) since in RC environments the partitioning algorithm can be applied several times so as to create many different designs that can be altered at run time [KK04].

1.3 Outline

The outline of the rest of this thesis is as follows.

Chapter 2 provides the background and demonstrates the trends of existing technologies. It discusses the characteristics and limitations of the FPGAs and the multiprocessor systems. It provides significant knowledge and guidelines for proposing and developing cost-efficient algorithms and methodologies in the rest of the thesis.

Chapter 3 describes the related work. It provides information about the existing hardware simulation accelerators, hardware emulators and ELAs. In parallel, it describes existing approaches employed in order to tackle the various limitations of current simulation systems, described in Section 1.1. Finally, it describes the existing algorithms and research on hardware/software partitioning and shows how this work has influenced our decisions.

Chapter 4 describes the proposed hardware emulator platform which solves the communication bottleneck between the testbench and the DUT. We first introduced this platform in [MP07, MP08].

Chapter 5 proposes a scan chain methodology in order to provide full chip observability and controllability at run time. Connecting the registers of the design in multiple scan chains we achieved full observabiliy and controllability of the memory elements included in the scan chains. We first described our scan chain methodology in [MP09].

Chapter 6 presents our hardware/software partitioning tool and methodology. It explains all the steps of the partitioning process such as the graph representation of the design and the cost metrics derived through simulations. The tool supports a novel greedy algorithm for partitioning the design which produces results very close to those of the most successful partitioning approaches, such as simulated annealing, while it is several times faster. We introduced this partitioning tool in [MP10].

Chapter 7 provides a detailed evaluation of the proposed methodologies and platforms. Our approaches are compared against existing methodologies and a performance analysis based on system simulations of real world and random test scenarios is provided. First, we analyze the hardware emulator and the scan chain methodology and finally we evaluate the hardware/software partitioning algorithm.

Finally, Chapter 8 provides summary, conclusion remarks and future directions.

Chapter 2. Technology Trends

"Man is still the most extraordinary computer of all."

John F. Kennedy

Currently, the most important problem in designing complex devices is that traditional approaches in system development as well as the associated tools do not scale well. The designers encounter significant difficulties in delivering competitive products to the market, since traditional design methods cannot satisfy, at the same time, issues such as short time-to-market, low cost, and complex, reliable, high performance and low power designs. This Chapter focuses on the state-of-the-art technologies used for developing competitive systems in order to meet the high market expectations. This study will provide us significant knowledge and guidelines for proposing an efficient Hardware Emulator platform (Chapter 4 and Chapter 5) as well as to understand the requirements of today's embedded applications in order to provide an efficient hardware/software partitioning methodology (Chapter 6).

Several different platforms ranging from multi-core CPUs (such as Cell processor), to high-density FPGAs and ASICs come to meet the requirements of today's market. However, each platform has different advantages and disadvantages. While programming a multi-core CPU is not so trivial, the effort required for implementing an application in an FPGA or ASIC is far more significant because the description has to be taken down way beyond the assembly coding

level, all the way to the micro electronics gate logic level. On the other hand, a hardware implementation (FPGA/ASIC) provides faster solutions than a software implementation (CPU). Moreover, FPGA-based platforms offer higher performance and better energy efficiency than CPU-based platforms for numerous applications.

A designer can trade off between ease of implementation, system performance and low power dissipation by using the aforementioned available platforms or a combination of them. The manpower, regarding the development and testing of a system, and the performance and power dissipation of the system are highly coupled with the selected technologies.

2.1 Field Programmable Gate Arrays (FPGAs)

A Field Programmable Gate Array (FPGA) is a semiconductor device containing programmable logic components and programmable interconnects. The programmable logic components can be programmed to duplicate the functionality of basic logic gates such as AND, OR, XOR, NOT or more complex combinational functions such as decoders or math functions. In most FPGAs, these programmable logic components also include memory elements, which may be simple flip-flops or complete blocks of highly-dense memories. A hierarchy of programmable interconnects allows the logic blocks of an FPGA to be interconnected as needed by the system designer. These logic blocks and interconnects can be programmed after the manufacturing process by the customer/designer (hence the term "field programmable", i.e. programmable in the field) so that the FPGA can perform whatever logical function is needed. State of the art FPGAs also provide embedded processors, transceivers and floating point units interconnected in a modular way.

FPGAs have a faster growth of transistor density even than that of general processors, as it can be seen in Figure 2.1. The largest FPGAs now in the market, part of the Xilinx Virtex6 and Virtex7 family devices, provide more than ten million "equivalent gates" (the relative density of logic). These advanced devices also offer features such as built-in hardwired processors (such as PowerPC and ARM), substantial amounts of memory in the range of MBytes, clock management systems, and support for many of the latest, very fast device-to-device signaling technologies such as gigabit transceivers.

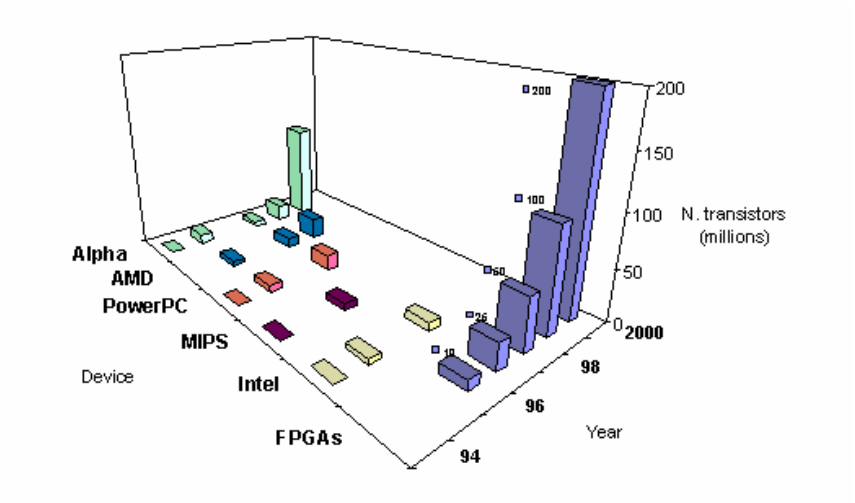

Figure 2.1. The growing of transistor density in processors and FPGAs devices.

2.1.1 System on Chip and HW/SW Co-design

System on Chip (SoC) is an idea of integrating all components of a computer or other electronic system into a single integrated circuit (chip). SoC designers often have to speed up critical portions of their design by implementing them in hardware, because general-purpose processor cores cannot meet the required performance goals.

Most SoCs are developed from pre-qualified hardware blocks (called Intellectual Property (IP) cores) together with the software drivers that control their operation. By simply interconnecting pre-qualified IP cores the SoC development can be accelerated significantly. The hardware blocks are interconnected together using CAD tools and the software modules are integrated using a software development environment. Certain tools, such as Xilinx EDK, support SoC design flows on FPGAs.

A typical application of SoC is in the area of embedded systems. An embedded system is a combination of computer software and hardware, either fixed in capability or programmable, that is specifically designed for a particular function. Industrial machines, automobiles, medical equipment, cameras, household appliances, airplanes, vending machines and toys, as well as the widely used cellular phone and PDA are among the myriad possible hosts of an embedded system. Since the embedded system is dedicated to specific tasks, design engineers can optimize it in terms of size and cost as well as in terms of reliability and performance; some embedded systems are mass-produced, benefiting also from economies of scale.

Current practice, in the vast majority of the cases, in embedded design makes use of a sequential scheme: the designer decides the platform (FPGA fabric, multi-core CPU, DSP, etc.) for implementing each design component (arithmetic units, compression, encryption engines, signal processing units, etc.), then the hardware platform design is completed, then an operating system and/or middleware is chosen and tested on the hardware prototype platform, and finally the embedded software is ported on the operating system and/or the middleware. In this complicated practice**,** the designer faces the problem of a very long design cycle. A unified, holistic process to embedded system design will enable the design of such a system to be performed at a high level of abstraction, where every component of the system has a specified behavior and several implementation paths, in software or in hardware. Hardware/software codesign is a unified process, which enables system design to take place free of any implementation-specific details, by means of defining applications using an abstract system model, and then by providing a concurrent mapping path into a mix of hardware and software. The design flow for a SoC aims to seamlessly develop the hardware and software components of the system in parallel. Essentially an ideal codesign process should decouple the design of the system from the implementation details of the underneath hardware or software platform providing a unified model for hardware and software development.

A key step in the SoC design flow is emulation: the hardware is mapped onto an emulation platform based on a field programmable gate array (FPGA) that mimics the behavior of the SoC, and the software modules are loaded into the memory of the emulation platform. Once programmed, the emulation platform enables the hardware and software of the SoC to be tested and debugged at a speed close to its full operational speed.

2.1.2 Reconfigurable Computing

Reconfigurable Computing (RC) is “computer processing with reconfigurable computing devices” (such as FPGAs). The principal difference when compared to using ordinary microprocessors is the ability to make substantial changes to the hardware itself (i.e. the FPGA fabric) on the fly. Configuration of these reconfigurable systems can happen at deployment time, between execution phases, or during execution (in this latter case it is called run time

reconfiguration). Implementing the critical operations of a process on hardware instead of using a general purpose microprocessor can result in significant performance speedup.

The main characteristic of RC is the presence of programmable hardware that can be reconfigured to implement a specific functionality, which is more suitable for specially tailored hardware than for a processor. RC systems potentially combine microprocessors and programmable hardware in order to take advantage of the combined strengths of hardware and software and have been used in applications ranging from embedded systems to high performance computing. In reconfigurable computing certain critical parts are implemented in hardware and therefore heavily accelerated; those parts would have been executed much slower in general-purpose processor cores.

Hardware, like software, can be designed modularly, by creating subcomponents and then higher-level components to instantiate them. In many cases it is useful to be able to swap out one or several of these subcomponents while the FPGA is still operating. Partial Reconfiguration allows reconfiguration of selected areas of an FPGA anytime after its initial configuration. You can do this while the design is operational and the device is active (known as active partial reconfiguration) or when the device is inactive in shutdown mode (known as static partial reconfiguration). One of the largest FPGA manufacturers, Xilinx, has supported partial reconfiguration in many generations of its devices. Partial reconfiguration can be used to save space for big designs by swapping in and out different portions of the design on the same FPGA area. With the introduction of FPGAs with faster reconfiguration times and partial reconfiguration support, it is possible to use FPGAs in a dynamically reconfigurable environment. This technology makes possible the concept of unlimited hardware or "virtual hardware".

2.2 High Performance Computing

The term High Performance Computing (HPC) refers to the use of parallel supercomputers and computer clusters, that is, computing systems comprised of multiple (usually mass-produced) processors linked together in a single system with commercially available interconnects. The TOP500 project (*http://www.top500.org/*) was started in 1993 so as to provide a reliable basis for tracking and detecting trends in HPC. Twice a year, a list of the sites operating the 500 most powerful computer systems is assembled and released. "HPC in Europe

Taskforce" plans are aimed at creating a sustainable supercomputer infrastructure in Europe to support science that also includes a world class supercomputer system as the top of the pyramid.

The computer industry has switched to delivering multiple core processors rather than increasing clock speed. Recent trends in HPC systems have shown that future increases in performance will only be achieved through increases in system scale, i.e., using a larger number of components and not by improvements in single-processor performance. The fact that future single CPU-chips need higher Gigahertz rates, resulting in higher energy consumption, developing more heat and bringing the chips to their physical limits was the real stimulus for the multi-core processor technology. In the last couple of years we have witnessed multi-core systems, within large clusters, and parallel computing becoming a must[1]. The chart in Figure 2.2 shows the increase in the number of processors employed in a system during the last decades.

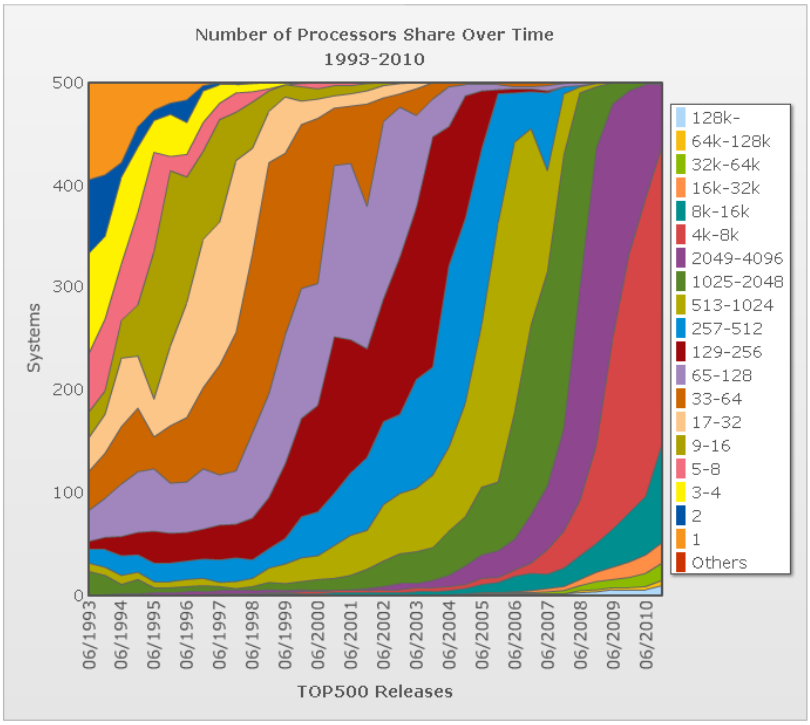

Figure 2.2. Number of processors over time.

According to the TOP500 list of the world's most powerful supercomputers, seven systems achieved performance at or above 1 petaflop/s. The most powerful system is the Chinese Tianhe-1A system at the National Supercomputer Center in Tianjin, achieving a performance level of 2.57 petaflop/s (quadrillions of calculations per second). The Cray XT5 "Jaguar" system at the

[1] http://www.top500.org/blog/2009/05/20/top_trends_high_performance_computing

U.S. Department of Energy's (DOE) Oak Ridge Leadership Computing Facility in Tennessee is ranked in the second place achieving 1.75 petaflop/s when Linpack (the TOP500 benchmark application) runs.

Five of the systems in the Top 10 were built in 2010. Of the Top 10, five are in the United States and the others are in China, Japan, France, and Germany. The most powerful system in Europe is a Bull system at the French CEA (Commissariat à l'Énergie Atomique), ranked at number six.

Several HPC systems have been used in simulation environments. Multiprocessor systems can provide cost-efficient solutions for simulating parallel applications. For example, hardware simulator accelerators, described in 1.1.1, use multiple CPUs in order to simulate the behavior of a design. Figure 2.3 shows the most important areas of applications where HPC systems have been employed.

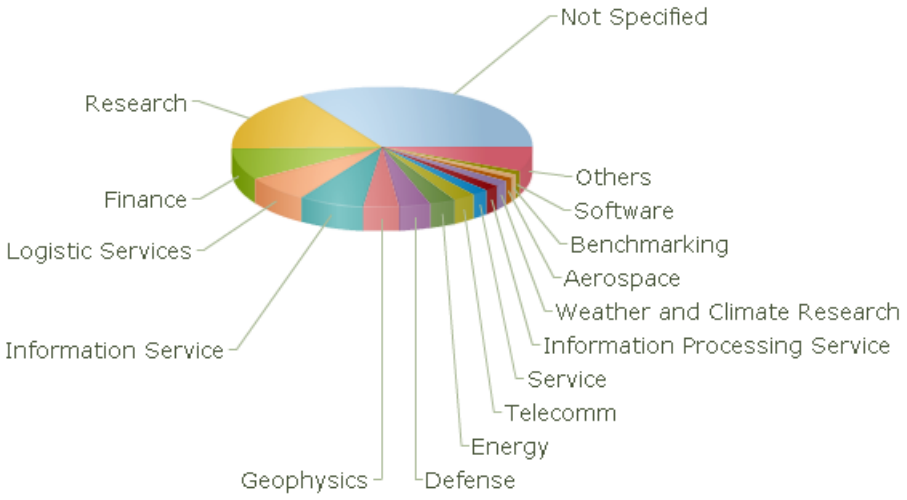

Figure 2.3. Application areas over number of systems

2.3 FPGAs vs Microprocessors

Programmable Logic Devices offer a cost effective alternative to custom microprocessors due to their generic nature with the added benefits of short time-to-market, no NRE costs, off-the shelf availability, relatively low power dissipation, and high performance. An FPGA device can be reprogrammed to do any logic task that can be fitted into its gates. The logic gates can be

rewired and configured to any possible task while microprocessors already have their own circuitry and instruction set that the programmer must follow. On the other hand a CPU can be programmed in a much faster and simpler way than an FPGA.

State of the art supercomputers, such as the Chinese Tianhe-1A system at the National Supercomputer Center in Tianjin, can achieve PetaFlop processing power. In a scalable multi-processor infrastructure the computation power is proportional to the number of the processors.

On the other hand, special-purpose platforms can be very effective, both performance-wise and cost-wise, compared to general-purpose multi-processor platforms. The performance of a serial algorithm ported to an FPGA is usually in the order of 50 to 100 times faster, than the same algorithm running on a state-of-the-art general-purpose processor; several studies[2] demonstrate a 90x speedup in parallel applications when employing FPGA-based systems.

Similar conclusions can be derived from hardware simulator accelerator and emulator systems. The hardware simulator accelerator is based on using circuit boards populated with multiple processors and lots of local memory. In this case, the HDL representation of the design is compiled into machine code, which is subsequently distributed amongst the various processors. The alternative, the hardware emulator, is to use circuit boards populated with FPGAs, in which case the HDL design is typically synthesized down into a gate-level equivalent, which is partitioned across, and loaded into, the various FPGAs. A hardware emulator (FPGA-based simulator) is about 100 times faster than a hardware accelerator (CPU-based simulator) of similar cost.

In summary, the pros and cons of the FPGAs and the microprocessors are the following:

Advantages of CPUs over FPGAs

- Ease of implementation of a CPU-based design, since the user can develop and test the system in a purely software environment. FPGAs are quite cumbersome to program. It seems to be more suited to electronic engineers (who are generally the ones who work on FPGAs) than software developers.
- Fast turn-around time (time between major modifications of the model) of a CPU-based platform compared to the time required to generate a new bitstream for FPGA.

[2] *http://www.soccentral.com/results.asp?CatID=488&EntryID=13654, http://www.drugdiscoverynews.com/index.php?newsarticle=371*

- FPGA machines are rarely large enough to encode entire interesting programs all at once. Smaller configurations handling different pieces of a program must be swapped-in over time. However, configuration time is too expensive for any configuration to be used only briefly and discarded. In real programs, much code is not repeated often enough to be worth loading into an FPGA.
- No circuit constructed with an FPGA can be as efficient as the same circuit in dedicated hardware. Standard functions like multiplications and double precision floating-point operations are big and slow in an FPGA when compared to their counterparts in ordinary processors.
- Problems that are worth solving with FPGAs usually involve more data than can be kept in the FPGAs themselves. No standard model exists for attaching external memory to FPGAs. FPGA-based machines typically include ad hoc memory systems, designed specifically for the first application envisaged for the machine/board.

Advantages of FPGAs over CPUs

- FPGAs are great for real time systems, where even 1ms of delay might be too long. FPGAs can be significantly faster for certain applications, (for example for well-defined digital signal processing usages (e.g. radar data)), than even the best CPUs available.
- An FPGA-based design consumes less power than a CPU-based design.

An interesting combination of FPGAs and multiprocessor systems is the Multiprocessor System-on-Chip (MPSoC) which is a SoC that uses multiple processors usually targeted for embedded applications. It is used by platforms that contain multiple, usually heterogeneous, processing elements. All these components are linked to each other by an on-chip interconnect. These architectures usually meet the performance needs of multimedia applications, telecommunication architectures, network security and other application domains while limiting the power consumption through the use of specialized processing elements and architecture.

2.4 Summary

Some scientific and technical applications are very demanding in terms of computational intensity, size of data sets and number of I/O channels. These applications usually perform High-

Performance Computing-type computations under real-time constraints. As processing capabilities increase and parallel programming barriers decrease, we expect this trend to continue and most likely accelerate in terms of Tera and Giga-Floating Point Operations Per Second (TFLOPS and GFLOPS) and Giga Multiply-ACcumulate operations per Second (GMACS).

In order to respond to these and other related challenges, new technologies for CAD tools are under development. Such technologies are the following:

- FPGAs that can implement high-performance low power designs using programmable logic blocks in a modular way.
- Multi-core CPUs that can be easily programmed to execute parallel programs.

By combining different technologies and hardware components available as Commercial-Off-The-Shelf (COTS) technologies, a hybrid, heterogeneous architecture can be easily configured which can combine the advantages from all those technologies.

Chapter 3. Related Work

"A generation which ignores history has no past and no future"

Robert Heinlein

With the advent of multi-processor embedded systems (see Chapter 2), system development has changed from a simple design and verification process to a multi-step process where advanced CAD tools play a significant role.

Taking a step back in time, 1981 marks the beginning of EDA as an industry. For many years, some of the largest electronic companies, such as Hewlett Packard, Tektronix, and Intel, had pursued EDA internally. Within a few years there were many companies specializing in EDA, each with a slightly different emphasis. The first trade show for EDA was held at the Design Automation Conference in 1984. In 1986, Verilog, a nowadays popular HDL (Hardware Description Language), was first introduced by Gateway Design Automation. In 1987, the U.S. Department of Defence funded the creation of VHDL as a specification language. Simulators quickly followed these introductions, permitting direct simulation of chip designs, the so called "executable specifications". In a few more years, back-ends were developed in order to perform logic synthesis.

By late 1980s, an embedded system was the norm rather than the exception for almost all electronics devices. The specific constraints that must be satisfied by embedded systems, such as

timeliness, energy efficiency of battery-operated devices, dependable operation in safety-relevant scenarios, short time-to-market and low cost, particularly in consumer products, coupled with the never-ending pressure to increase the functionality, lead to an enormous growth in the complexity of the design at the system level. System complexity challenges imply super-exponentially increasing complexity in the design process. Hardware Emulators, Embedded Logic Analyzers and hardware/software partitioning tools have become common parts of the design process and essential tools to realize a cost-effective design.

3.1 Hardware Simulation Accelerators and Emulators

Nowadays, with the rising design complexity, there is an increased interest in hardware simulation accelerators and emulators (see Section 1.1.1). Speeding up simulation and verification of complex embedded systems can save design teams a lot of money and effort. Therefore, more and more companies build systems for hardware emulation.

Among the available simulation acceleration and emulator systems, we note the following:

- The *Palladium* system from Cadence [CAP], which provides hardware acceleration and in-circuit emulation, can speed up verification 100 to 1M times when compared with software-based RTL simulation. Palladium is described as an array of “massively parallel Boolean compute engines”, and supports the latest industry standards, such as OSCI SystemC TLM 1.0/2.0 (transaction-level modeling), IEEE 1850 PSL (assertion-based verification), and SCE-MI 2.0 (transaction-based acceleration). The designer can apply the Palladium’s capabilities to mixed SystemC-HDL designs across both hardware-based and software-based simulation environments. The tool supports advanced SystemC simulation features such as save/restore, transaction-level recording, and multi-language hierarchical visualization and analysis.
- The *Veloce SoC* verification platform from Mentor Graphics [MGV] delivers high performance simulation acceleration and SoC in-circuit emulation to speed-up the verification of complex designs from 8 to 512 million gates. The product family platform and accompanying software allow designers to create reconfigurable hardware representations of a SoC design, thus enabling pre-silicon testing and debug at hardware speeds with real-world

data. Veloce employs the SCE-MI 2.0 communication protocol to provide a simulation-like debug environment that allows full signal visibility into the design.

- The *Zebu* system emulator from EVE [EVZ], which can handle 200 million gates, and up to five of them can be combined to handle up to 1-billion gates. It is aimed primarily at large-scale chip and system emulation applications, and it is offered in a modular, 19-inch rack-mountable configuration, which accepts up to 64 Xilinx Virtex-II XC2V8000 FPGAs. EVE has also acquired Tharas Systems, the company behind the *Hammer* accelerator system which contains up to 128 specialized processors connected through a proprietary backplane.
- The *Riviera-IPT* system from Aldec [ALR], an FPGA-based PCI board that is tightly coupled to Aldec's own software simulator. A single board has a simulation capacity of up to 12M gates. Multiple boards can be connected on the same PCI bus for larger capacity.

The following table summarizes the features of the above systems.

Product	Accel.	Emul.	Capacity	Speedup	Emulation Speed
Cadence (Palladium)	√	√	256M	1M	2MHz
Mentor (Veloce)	√	√	512M	400	1.5MHz
EVE (Zebu)		√	200M	?	30MHz
ALDEC (Riviera)		√	12M	?	1MHz

Speedup column shows the emulation speedup compared to the RTL simulation

Figure 3.1. Hardware Accelerator/Emulator Systems

3.2 Hardware/Software Communication Bottleneck

A major challenge that all these hardware-software co-simulation systems must solve is how to optimize the communication at the hardware-software interface. Even though this issue has largely been identified as of critical importance by all these systems, their individual solutions have not been made publicly available. Most of these systems are commercial systems and therefore no details have been disclosed regarding their hardware/software communication schemes, apart from the use of the Standard Co-Emulation Modeling Interface (SCE-MI), an interface introduced by Accelera [AC07].

SCE-MI attacks the hardware/software communication bottleneck between the DUT and the testbench, by defining both a methodology as well as a protocol. According to SCE-MI the part of the testbench closer to the DUT is split into two parts; one that is comprised of "Transactors" and is written in synthesizable HDL and emulated on-chip together with the DUT, and one that is comprised of "Message Port Proxies" and is written in software and simulated on a host workstation.

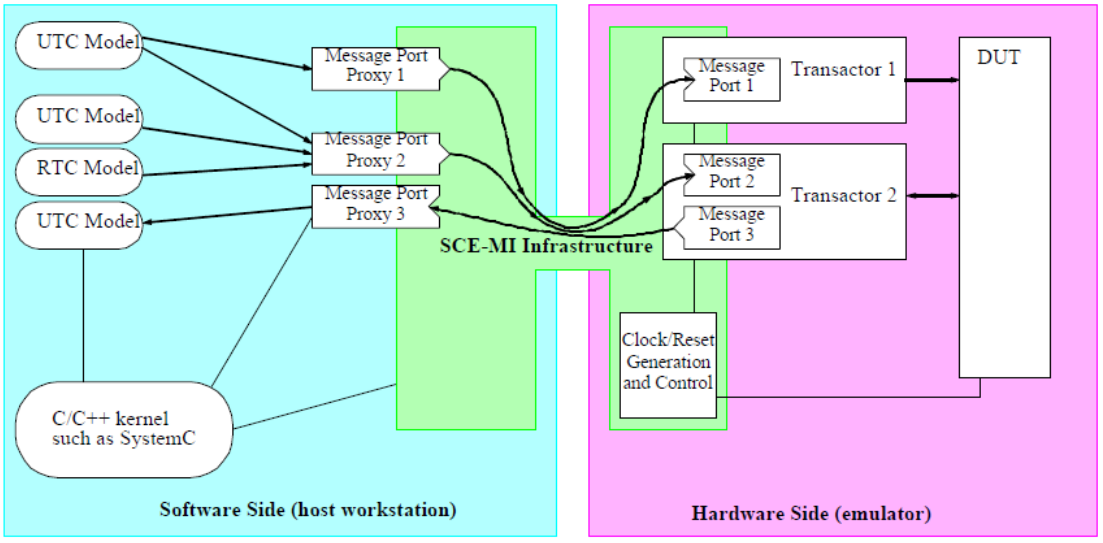

Figure 3.2. High-level view of SCE-MI's run-time components

Figure 3.2, derived from [AC07], shows the main components of the SCE-MI standard. By splitting the testbench into two parts, the hardware/software communication has now been moved to the proxies-transactors interface. The idea is to make the communication through this interface as coarse-grained and consequently as low-traffic as possible, thereby eliminating the hardware/software communication bottelneck.

In order to do this, the two parts communicate using high-level messages (“transactions”), and it is up to the transactors to decompose each incoming message into a series of cycle-accurate clocked events for the DUT, and, vice-versa, compose a series of clocked events from the DUT into a single outgoing message. On the software side, the message port proxies translate untimed messages to cycle-accurate messages, and vice versa.

SCE-MI also standardizes the proxies-transactors interface, by defining a cycle-accurate protocol for it, thus increasing the interoperability of transactors between testbenches. The protocol supports a superset of SystemVerilog’s Direct Programming Interface (DPI – an interface intended to allow the efficient connection of an HDL model with a C model).

The main disadvantages of SCE-MI’s approach compared to ours are the following:

- Using SCE-MI’s methodology, the designer has to manually implement a part of the testbench (the “transactors”) in synthesizable code. In contrast, our methodology will automatically transform HDL testbench into synthesizable code that can run on our emulation environment (Section 4.4).
- SCE-MI performs more transactions between the hardware emulator and the host processor than our approach, because we implement a larger portion of the testbench in hardware. By performing fewer transactions, the host processor is not involved so often in the emulation and therefore the simulation is executed faster in our approach (Section 7.1.2). Additionally, we provide a memory controller and floating point unit in order to offload the host processor (Section 4.3).

[YM07] presents a new scheme that reduces the modeling efforts for a transactor while retaining the performance of transaction-based verification for hardware/software co-emulation systems. While a conventional transaction-based verification requires the designer to develop a synthesizable transactor block which interfaces with the DUT and its unfamiliar system dependent protocols, the proposed method locates the transactor in the software side instead of in the hardware emulator; this allows the designer to develop the transactor in a high-level language. Moreover, to reduce the communication time between testbench and DUT, the authors make the signal flow uni-directional.

In [MB99] the authors distinguish the behavioral functionality, which further remains on the simulator, with those parts which can be hardware accelerated. The acceleratable parts have to be

detected in the testbench description and must be remodeled in a Register-Transfer Level (RTL) description. A transaction-level interface is used to reduce the amount of communication data. However, no automatic way is provided for finding these hardware accelerated parts of the testbench and remodeling them in a RTL description.

GateRocket [GRR] provides a platform which allows the designer to place synthesizable portions of the DUT into the *RocketDrive* and emulate them on the disk-drive sized verification platform. RocketDrive plugs into a standard disk drive slot of a workstation and it is available in several configurations, each containing a different FPGA device from Xilinx or Altera. This platform can accelerate the simulation by up to an order of magnitude or more depending on the design and the testbench. However, the software driver can easily become the performance bottleneck.

In [RH03] the authors introduce a new technology that accelerates functional system verification. The authors propose a seamless flow from a behavioral testbench to a re-use-oriented synthesizable testbench fully compatible with the original testbench. In this way, the authors combine the flexibility of a behavioral testbench and the high performance of a synthesizable testbench, while greatly reducing the modeling overhead. The approach itself is hardware independent. The proposed platform was applied on a hard disc controller achieving a speed-up factor of 5000 versus software simulation. No automatic method is provided for applying this approach to any generic testbench.

In [YW04, YC04] the authors propose a methodology to reduce the communication overhead by exploiting burst data transfer and parallelism, which is obtained by splitting the testbench and moving a part of it into a hardware accelerator. The authors try to identify a part of the testbench which is involved in generating the next input stimulus using only output results from the DUT; this part is then moved into hardware and merged with the emulated DUT. Their experiments demonstrated that the proposed method reduces the communication overhead by a factor of about 40 compared to conventional hardware accelerated simulation while maintaining the cycle accuracy and compatibility with the original testbench. The authors also propose a hybrid dynamic simulation scheme, called TPartitioning, which implements a part of the simulator in software running on a processor and maps the rest onto a programmable hardware accelerator. The proposed algorithm for hardware synthesis of simple behavioral testbenches enables better partitions, thus resulting in lower communication costs between the two

components. However, the efficiency of their algorithm depends on the behavior of the testbench since the authors assumed that they could find “autonomous” testbench parts where the generating stimulus depends only on the DUT outputs.

Axis has developed *SEmulation* [AXS] which compiles a RTL design into "computing elements", essentially coprocessors, which are then mapped into FPGAs without going through logic synthesis. Axis claims to be able to compile 500,000 to 1 million gates in an hour on eight distributed workstations. Their approach is still slower than a conventional emulator since everything runs essentially on software, but considerably faster than an accelerator.

Verisity has developed eCelerator [VEE] which focuses on reducing the communication overhead by using innovative synthesis technology to transform the most frequently executed sections of e-testbenches in hardware. By shifting the computationally most expensive parts onto hardware, the tool achieves significant performance gains in the verification process, ranging from 10x to 50x speedup. With eCelerator, the designer can still create e-testbenches to generate tests for the design, perform complex data and protocol checking and collect functional coverage. Verisity has worked with its acceleration/emulation vendor partners to create a new buffered transaction-based scheme. The ability to buffer many transactions in the design allows for much higher communication bandwidth and removes the need for Specman Elite (Verisity’s functional verification tool) to communicate on a cycle-by-cycle basis with the hardware. In addition, the interface provides visibility to the synthesized testbench and DUT, enabling full visibility to the entire testbench. However, eCelerator is focused only on e-testbenches.

Finally, [HS06] presents a synthesizable testbench architecture addressing the same problem, which is based on a defined instruction set for standalone mode verification. A set of instructions describes the transitions of a signal. The instructions are loaded on the emulator’s memory. The proposed approach allows for fast emulation and increases flexibility and reusability by using a specific instruction set. However, in this case the original testbench has to be rewritten following their defined instruction set.

3.3 Embedded Logic Analyzers

The major FPGA vendors have recognized the value of the ELAs, and have released proprietary packages that work with their platforms; the most efficient such systems are the following:

- ChipScope [XCS] by Xilinx provides an embedded, software based logic analyzer that can monitor the signals of the design. ChipScope tool inserts logic analyzer, system analyzer, and virtual I/O low-profile software cores directly into the design, allowing the designer to view any internal signal or node, including embedded hard or soft processors. The signals are captured in the system at the operational speed and brought out through the programming interface, freeing up FPGA pins for the design itself.
- SignalTap [AST] by Altera enables efficient design verification by allowing the designer to quickly route internal signals to I/O pins without affecting the design. The designer can define custom trigger-condition logic in order to investigate possible problems. All captured signal data are stored in the device memory for further analysis. The SignalTap embedded logic analyzer supports up to 1K channels and a sample depth of up to 128K bits. The architecture of this ELA is shown in Figure 3.3.
- ClearBlue [DCB] by DAFCA provides an advanced verification platform. Pre-silicon, the ClearBlue Instrumentation Studio software delivers a user-directed environment for insertion of the Reconfigurable Debug Instruments (ReDU) into the SoC design. Post-silicon, the ClearBlue Debugger offers a wide spectrum of configurable, at execution speed, analysis capabilities, including signal trace, on-chip logic analyzers, event-based and assertion-based debug, and performance monitoring, that all feed directly into standard graphical debugging software tools.
- Configurable Logic Analyzer Module (CLAM) by First Silicon Solution (FS2) provides logic analyzer capabilities for Actel’s Flash-based FPGAs. It provides an intuitive and easy way to view internal signals and debug the logic design. The system features FS2 On-Chip Instrumentation (OCI) in the form of Configurable Logic Analyzer Module (CLAM) logic. It can trace and trigger up to 32 channels, selectable in groups of 32, from an available 128 predefined signals in the FPGA fabric.

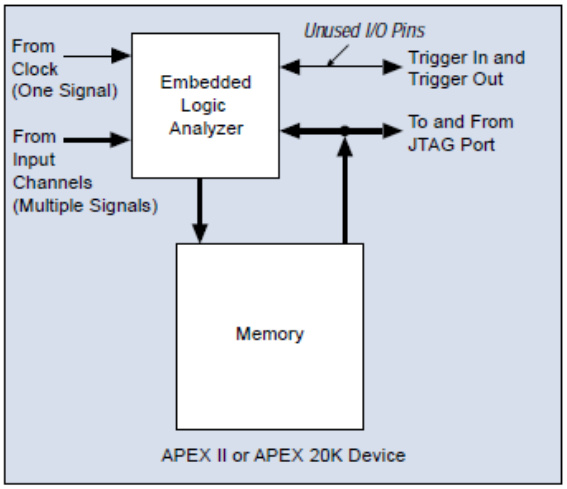

Figure 3.3. Architecture of Altera's Embedded Logic Analyzer

Moreover, PALMiCE FPGA [HGP] by HiTech Global is an external analyzer used for debugging designs on Xilinx FPGAs. PALMiCE connects to the FPGA either with a dedicated 38-pin "MICTOR" connector or "clip-on" connectors. The important feature of this logic analyzer is that it allows routing of the internal nodes to the pins of the FPGA to be added or changed without re-synthesizing the circuit. However, the place-and-route step, which is usually a long process, has to be repeated.

Embedded Logic Analyzers are also developed for application-specific integrated chips (ASICs). Cisco's embedded logic analyzer module (ELAM) is a debugging device used for many of Cisco's ASICs. The ELAM is used to capture data and store it for analysis purposes. The user enters a trigger expression containing data fields of interest in the form of a logical equation. The data fields associated with the trigger expression are stored in a set of Match and Mask (MM) registers. Incoming data packets are matched against these registers, and if the user-specified data pattern is detected, the ELAM starts capturing data until the end of a predetermined period. The ELAM is tailored to the requirements of the Cisco's packet based architecture.

3.4 Circuit Observability and Controllability

All the aforementioned products lack circuit controllability (with the exception of ClearBlue) while they provide limited circuit observability; in order to change the watched signals they require design recompilation while the size of the data capture buffer is limited by the internal memory of the FPGA.

Scan-chain is not a new technique. Design-level scan is a structured technique proposed within the Design For Testability (DFT) process. For example, [WP82] takes advantages of traditional DFT methods like scan chain, as well as certain DFT methods which control clocks for testability, and proposes a hybrid DFT method that reduces the hardware overhead and the test generation time. Moreover, the IEEE 1500 standard [IS05] defines a mechanism for the testing of the IP-cores within a SoC by employing scan-chains.

The concept of using multiple scan chains has also been utilized in DFT architectures in order to reduce the test application time as described below.

In [HP99] a new testability technique is introduced, called Parallel Serial Full Scan (PSFS), for reducing the test application time for full-scan embedded cores. Test application time reduction is achieved by dividing the scan chain into multiple partitions and shifting in the same vector to each scan chain through a single scan input.

In [JC05] the authors propose an algorithm, based on a framework of reconfigurable multiple scan-chains for a System on Chip, to minimize test application time. The test application time is minimized by using a balancing method to assign registers into multiple scan-chains. The experimental results show that this technique significantly reduces the test application time.

In [LH04] the authors present an efficient multiple scan chain architecture for reducing power dissipation and test time. This paper shows a DFT technique employing clustering of the unspecified bits in the response test cubes so as to reduce power consumption and test time. The unspecified bits in the response test cubes are clustered by reordering scan latches; the multiple scan chain architecture is modified by inserting multiplexers (MUXes) in each scan chain in order to implement this reordering.

Finally, the multiple scan chain approach is also used in [TA06] where the testing methodology of the UltraSPARC T1 microprocessor is presented.

The FreedomChip by Lattice Semiconductor [LS07] is the first FPGA-based design methodology to employ scan-chain structures in the fabric. The user’s design is implemented in low-cost, custom-tested silicon through automatic insertion of scan logic and dedicated silicon test features. This eliminates the difficult and error prone back-end design conversion associated

with traditional structured ASICs. Fault coverage of over 99% typically is achieved using these test techniques. However, the scan chains are used only for DFT and not for circuit observability.

Wheeler et. al. in [WG01] demonstrate how "design-level scan" can provide an efficient approach for the monitoring and control of the status of an FPGA. This paper uses a scan chain methodology for providing full circuit observability and controllability for functionally debugging FPGA designs. The proposed design-level scan technique includes all FPGA flip-flops and RAMs in a serial scan chain using the FPGA logic rather than custom-made transistor logic. This paper describes the general procedure for modifying designs with design-level scan chains. The authors measured that scan chains result in an average FPGA resource overhead of 84%. However, the authors demonstrate their technique without providing a way to apply it at run time in order to investigate the transient behavior of a design.

Finally, Tiwari et. al. in [TT04] propose a framework to define trigger conditions in an ELA utilizing a scan chain methodology. This paper describes a watch-point implementation utilizing scan chains which is applied to the hardware design running on the FPGA in order to help in debugging and verification. The hardware debugging procedure proposed, which uses the look-up table shift registers (srluts) of the FPGA, does not require any recompilation of the design in order to change the watch-point conditions and thus is very fast. In this paper, the area overhead resulting from adding this scan-chain based watch-point logic is discussed and it is compared with other proposed debugging techniques. The observed average area overhead was 46% for the ITC benchmark circuits with varying widths of watch-point signals. This work is orthogonal to our approach; their scheme can be used on top of our framework so as to define the trigger conditions in an optimal way.

The methodology we propose is based on the hardware emulator environment introduced in Chapter 4. In Chapter 5 we extend the aforementioned basic ideas from [JC05, HP99, WG01, TT04] by combining a multiple scan chain methodology (Section 5.3), a novel ELA (Section 5.4) and the synthesizable testbench methodology (Chapter 4), so as to build an integrated tool that supports fast emulations and efficient circuit testability at run time. We show that our methodology is, to the best of our knowledge, the first that combines the flexibility of a software simulator with the high speed of a hardware emulator. We also evaluate our approach in terms of area, cost and speed, and show that the best tradeoffs are achieved for a certain range of scan chains independent of the DUT size (Section 7.2).

3.5 Hardware/Software Partitioning

The first papers on hardware/software partitioning were presented in early nineties (for example [EH93]) and then for a few years this was considered a very active research topic in the CAD community. In the early 2000 the topic was considered “uninterested” and that was valid until recently. However, nowadays, with the rising design complexity, there is again an increased interest in hardware/software co-design. Even though, this is an active topic more or less for about 20 years several issues remain still open due to the unavoidable complexity of the actual partitioning problem.

The partitioning problem is NP-complete since it requires the exploration of a design space whose size grows exponentially with the number of design components. Several researchers have proposed partitioning heuristics that an automated tool can follow. They differ in the initial specification, the level of granularity at which partitioning is performed, the degree of automation of the partitioning process, the cost metrics, the cost function, as well as the actual partitioning algorithm. The effectiveness of a partitioning tool as well as the time required to partition a system depend on the aforementioned characteristics of the partitioning process.

Common description languages that have been used for the initial specifications of an unpartitioned system are C [EH93], HardwareC [GM93], VHDL [EP97], and object oriented languages. In our work we have selected SystemC which is, nowadays, a popular high-level language for describing a system.

The *granularity* of a partitioning approach determines the size of each design component that will be considered to be implemented either in hardware or in software. Instruction-level granularity [AS93], block-level granularity [HE01] as well as function-level granularity [SN04] have all been employed in the past. We follow a thread-based approach (as described in Section 6.4.2) considering that each SystemC thread is a design component implementing a single function and therefore it should not be split into more than one system entities.

Most partitioning approaches model and analyze a system using annotated process graphs. Common cost metrics that determine the annotated values on the graphs involve the I/O delay and the rate and execution time of each node. LOTOS [CA96] and QUEST [SR98] can estimate

the delay associated with a combinational circuit from its high-level description. In [EP97] the authors describe the *Computational load* of a block and the *Communication intensity* on a channel; those metrics are similar to the MIPS (Million Instructions Per Second) and MTPS (Million Transactions Per Second) metrics that we will introduce in Chapter 6. We strongly believe that these metrics provide a better insight of the system requirements and of the allocation of the system resources.

A common approach to estimate the cost metrics is through profiling [FZ05, HE98] and simulations. In our approach we employ high-level system simulation which seems to be the most accurate methodology in order to extract realistic design characteristics. The increased time for calculating the cost metrics based on high-level simulations is compensated by the improved quality and accuracy of the results.

The effectiveness of a partitioning process is also determined by the actual partitioning algorithm which partitions the systems into hardware and software entities based on the cost metrics of the system blocks. Related research on the most effective partitioning algorithms is described below.

Integer Programming

The translation of the HW/SW partitioning problem into a set of integer programming (IP) constraints is described in [NM96]. The advantage of using IP is that optimal results are calculated respective to the chosen objective function. This partitioning approach works in a fully automatic way and it supports multi-processor systems, interfacing and hardware sharing. In contrast to other approaches where special estimators are used, the authors used compilation and synthesis tools for cost estimation. The increased time for calculating the cost metrics is compensated by an quality of the estimations compared to the results of estimators.

Greedy Algorithms

Kalavade and Lee introduced the Global Criticality/Local Phase (GCLP) algorithm to solve the two-way partitioning problem [KL94] for tasks of moderate to large granularity. The authors note that two possible objective functions could be used in order to decide whether a task should be mapped into hardware or software: minimization of the execution time of that node, and minimization of the solution size (hardware or software area) of the node's implementation. To this end, the authors devise a global criticality measurement, which is re-evaluated at each step

of the algorithm to determine whether time or area is more critical in the design. As the list of functional tasks is traversed, the global criticality measurement is checked so as to determine the current design requirement. If time is critical, the mapping minimizes the finish time of the task; otherwise the resource consumption of the task is minimized. In addition to the global system requirements, local optimality is sought by classifying each task as either an extremity (meaning it consumes an extreme amount of resources), a repeller (meaning the task is intrinsically preferred to have either a software or hardware implementation), or a normal task. This classification of each task, and its weighty consideration in the choice of hardware or software mapping, represents the local phase of a given task. The running time of the GCLP algorithm is extremely efficient ($O|N|^2$), and the partitions it determines are no more than 30% larger than the optimal solution.

Dynamic Programming

Just as Kalavade and Lee incorporated a dynamic performance metric into their partitioning decision, Henkel and Ernst incorporated "dynamic functional granularity" [EH93]. The authors' partitioning method allowed the dynamic clustering of fine-grain tasks (at the basic block or instruction level) into larger units of operation (as large as a procedure/subroutine). The rationalization for having a flexible functional granularity are that large partitioning objects should contain complete control constructs (in the form of loop bodies or procedures), and that only a few moves should be necessary (between hardware and software) in order to determine a good partition. The innovation comes from the hierarchical search of the design space and the fast retrieval of a good solution.

Bhasyam et al [KB03] propose a dynamic programming framework for hardware/software partitioning which incorporates the cost of communication delays between components of two different partitions. Their work attempts to find a minimum latency solution within finite resource constraints. A pruning technique is introduced in order to reduce the runtime of the worst-case scenario for partitioning directed acyclic graphs (DAGs). The algorithm has a polynomial run time complexity.

Kernighan-Lin/Fiduccia-Matheyes (KLFM)

Vahid and Le extended the Kernighan-Lin (KL) circuit partitioning heuristic to explore the design space of Hardware/Software functional partitioning [VL97]. The chief advantage of the

KL heuristic is its ability to overcome local minima without making excessive numbers of moves. The basic strategy of KL is to make the least costly swap of two nodes in different partitions, and then to lock those nodes. This continues until all nodes are locked. The best partition *bestp* is selected from this set. All nodes are subsequently unlocked, and the previous *bestp* becomes the starting point for the next set of node swaps. This swapping, locking, selection of *bestp*, and subsequent unlocking and looping continues until no subsequent improvement over the former *bestp* exists. Vahid and Le extend the KL heuristic by replacing its cost function with a combined execution-time/area/communication metric, by redefining a move as a movement of a functional node across partitions (rather than a swap of nodes), and by reducing the running time of the "next move selection" procedure . Via these means, the authors are able to achieve nearly equal-quality partitions to simulated annealing in an order of magnitude less time. The running time of the algorithm is accelerated by considering task nodes at a subroutine-level granularity.

Ramani and Markov adapted the Fiduccia Mattheyses (FM) hypergraph partitioning heuristic to Boolean Satisfiability (SAT), and the WalkSAT SAT solver to hypergraph partitioning [RM03]. They developed a SAT solver based on the FM algorithm, and a hypergraph partitioner, WalkPart, based on the WalkSAT algorithm (i.e. a stochastic local search heuristic for Boolean Satisfiability).

Hill Climbing and Simulated Annealing

Ernst and Henkel developed a hill-climbing partitioning heuristic that sought to minimize the amount of hardware used, while meeting a set of performance constraints [HE98]. Their work operated on the basic block level of functional granularity. They started with an initial partitioning that was improved on subsequent iterations. However, to escape convergence to a local minimum, they utilized simulated annealing to explore design cost. Unlike greedy heuristics, simulated annealing often accepts changes which decrease the quality of a design, in hopes of achieving a more optimal final design. Ernst and Henkel began the process with an all-software partition, seeking to minimize hardware costs by starting with less hardware. In order to prevent annealing before a performance-satisfying partition has been reached, they used a heavily weighted cost function that provided high penalties for violating runtime constraints. This choice proved effective in minimizing hardware costs. To provide performance

enhancement estimates for hardware implementation, Ernst and Henkel utilized simulation and profiling information to determine the most frequently executed and computationally intensive regions of functionality.

In [SN04], Banerjee and Dutt represent applications as procedural call-graphs and they prove that during partitioning, the execution time metric for moving a vertex needs to be updated only for the immediate neighbours of the vertex, rather than for all ancestors along all the paths to the root vertex. Consequently, move-based partitioning algorithms such as Simulated Annealing (SA), can process call graphs with thousands of vertices in less time. The authors also introduce a new cost function for SA that allows frequent discovery of better partitioning solutions by searching spaces overlooked by traditional SA cost functions. By optimizing the SA, several thousand configurations are partitioned in minutes as compared to several hours or days using traditional SA. Furthermore, this approach can derive better design points in most cases with over 10% improvement in application execution time compared to the solutions derived from a Kernighan-Lin partitioning algorithm starting with an all-SW partitioning.

Genetic Algorithms

[CA02] presents a Genetic Algorithm (GA) based approach for Hardware/Software partitioning targeting an architecture composed of a processor and a dynamically reconfigurable datapath (FPGA). From an acyclic task graph and a set of Area-Time implementation trade off points for each task, the GA performs HW/SW partitioning and scheduling such that the global application execution time is minimized.

In [MZ06] the authors propose an enhanced genetic algorithm which selects the most "interesting" code-parts of the program to be implemented in hardware using a dynamically-weighted fitness function. The novelty of their approach resides in the reduction of the search space obtained by specific optimizations passes that are conducted on each generation. Moreover, by considering different granularities during the evolution process, very fast and effective convergence (in the order of a few seconds) can be attained.

[LL09] presents an immune algorithm based on the Pareto concept of multi-objective optimization problems. The immune algorithm has many merits, such as high searching efficiency, avoiding immature convergence, colony optimization, keeping individual varieties

and so on. Experimental results show that the algorithm can achieve the global optimal solution of the HW/SW partitioning problem based on certain system constraints.

Finally, in [XW09] the Non-dominated Sorting Genetic Algorithm (NSGA-II) is applied to HW/SW partitioning. Each run of the algorithm can produce many Pareto-optional solutions. This method can provide an effective tool for measuring the performance of different objective functions.

Tabu Search

In [EP97] two heuristics for automatic hardware/software partitioning of system level specifications are presented and compared. Partitioning is performed at the granularity of blocks, loops, subprograms, and processes with the objective of performance optimization with a limited hardware and software cost. The goal of the partitioning process is to minimize the communication cost and improve the overall parallelism. One heuristic is based on simulated annealing and the other on tabu search. Results of extensive experiments, including real-life examples, show the clear superiority of the tabu search based algorithm.

Similarly, [WC02] compares three heuristic search algorithms: genetic algorithm (GA), simulated annealing (SA) and tabu search (TS) and shows that TS is superior to SA and GA in terms of both search time and quality of solutions. In addition, the authors have implemented an intensification strategy in TS called *penalty reward*, which can further improve the quality of results.

Ant Colony Optimization

A recent approach for reconfigurable system partitioning is based on the Ant System (AS) algorithm, a heuristic optimization method inspired by the behaviors of ants. In this algorithm, a collection of agents cooperate together to search for a good partitioning solution.

[WL08] presents a collaborative partition approach of coarse-grained reconfigurable system design using ant colony optimization. The authors create a distributed collaborative design environment for system decision engineers, software designers, hardware designers and algorithm developers. The method utilizes the advantages of ant colony optimization in searching for global optimal solutions in order to provide a framework for multi-field experts to work collaboratively.

In Chapter 6 we propose a novel two-stage greedy partitioning algorithm (as analytically described in Section 6.2) that can partition an embedded design in a cost-efficient manner. The second stage of the algorithm creates sub-graphs similar to the algorithm described in [PA04]. The proposed algorithm provides accurate results and can process large graphs with hundreds of nodes in less than a second and thus it is much faster than traditional algorithms such as SA and KLFM. This is especially important, for example, to FPGA designers who may need differently partitioned systems, all executed on the same platform, so as to efficiently utilize the run-time reconfiguration characteristics of today's FPGAs.

Finally, there are limited research frameworks and tools that have been used in order to develop and test hardware/software partitioning algorithms, such as the following:

- LOTOS [CA96] design flow that performs the partitioning on a Process Communication Graph (PCG). Each node in a PCG is first decomposed into a Control and Data Flow Graph (CDFG) whose nodes represent basic data-dependent operations. The CDFG represents a flow made up of *control* nodes, which are synthesizable components from a design library. Each control node may further be associated with a Data Flow Graph (DFG). The DFG consists of data flowing to and from *operator* nodes. The response time for the operator nodes are fixed and are available from an operator library.

- QUEST [SR98] which is an estimation tool that finds reasonably accurate area and delay values from high-level designs. The input, in this case, is an RTL description of the system. The basic idea is to implement a small subset of the design in the target technology and extract prediction parameters.

- MUSIC [JE99] which is a high-level synthesis tool that can be used to convert a high level description of a system into a VHDL RTL specification for hardware implementation.

- POLIS [CH94] that represents the system design as a network of Codesign Finite State Machines (CFSM). The next level of abstraction for software is a set of *s-graphs* that are derived from the CFSMs. The *s-graph* is then converted to "C" code using a straightforward translation. Delay can be estimated either at the CFSM level or from the *s-graphs*.

Chapter 4. Testbench Code Synthesis

"What you get free costs too much."

Jean Anouilh

The rising complexity of modern embedded systems is causing a significant increase in the verification effort required by hardware designers and software developers, leading to the "design verification crisis", as it is known among engineers. Today's verification challenges require powerful testbenches and high-performance simulation solutions such as Hardware Simulation Accelerators and Hardware Emulators that have been in use in hardware and electronic system design centers for approximately the last decade. In particular, in order to accelerate functional simulation, hardware emulation is used so as to offload calculation-intensive tasks from the software simulator. However, the communication overhead between the software simulator and hardware emulator is becoming a new critical bottleneck as described in 1.1.1. In this Chapter, we introduce a novel emulation framework that automatically transforms into synthesizable code certain HDL parts of the testbench, in order to offload them from the software simulator and, more importantly, minimize the aforementioned communication overhead. Our experiments (see Section 7.1), using real-world designs, demonstrate that (i) our approach is at least 1000 times faster than conventional software simulation, and (ii) the

proposed method reduces significantly the communication overhead and outperforms the conventional hardware emulation systems by a factor of more than 15.

4.1 Background and Motivation

In order to prototype a system, several steps are performed that usually involve a lot of effort and associated risks. The approach taken by employing an initial verification of the design on FPGA devices is a simple cost-effective one. However, while FPGA vendors provide several evaluation boards supporting many different external interfaces (Ethernet, PCI, Serial, etc.) connected to a central FPGA or an array of FPGAs, these commercial evaluation boards cannot still cover all possible requirements of any design, and therefore several times the designer has to build a custom evaluation board according to the exact requirements of the design. Moreover, any non-trivial software development and verification can only start once the hardware design is tested.

Hardware Simulation Accelerators and Hardware Emulators aim to simplify the design process by performing the simulations faster and more accurately in hardware, and to provide an application development platform earlier in the design process. These verification platforms can simulate any testbench, including the behavior of any system interface which is usually part of the testbench, and therefore they can be used for the verification of any possible design. Using a Hardware Emulator the designer can simulate the design with silicon-level accuracy and identify problems that typically go undetected until system-level debugging in the lab. In this way, the designer can spot problems early in the design cycle when they are much easier to find and fix. Since performing back-end verification in hardware provides an accurate model, we can even ignore the FPGA prototyping and testing steps, as long as we have a reliable testbench that exercises the system under all possible scenarios. Figure 4.1 shows the conventional and modified design processes.

While hardware emulation is used to offload calculation-intensive tasks from the software simulator, the communication overhead between the software simulator and the hardware emulator is becoming a new critical bottleneck as described in Section 1.1.1. To facilitate the communication path between the hardware and the software sections of an emulator, most commercial platforms use a certain transaction-level interface so as to reduce the amount of

communication data (most frequently the SCE- MI 2.0 communication protocol introduced by Accelera (see Section 3.2)). However this method requires designers to rewrite the testbench in a synthesizable fashion while the transaction-level interface can easily become the new emulation bottleneck.

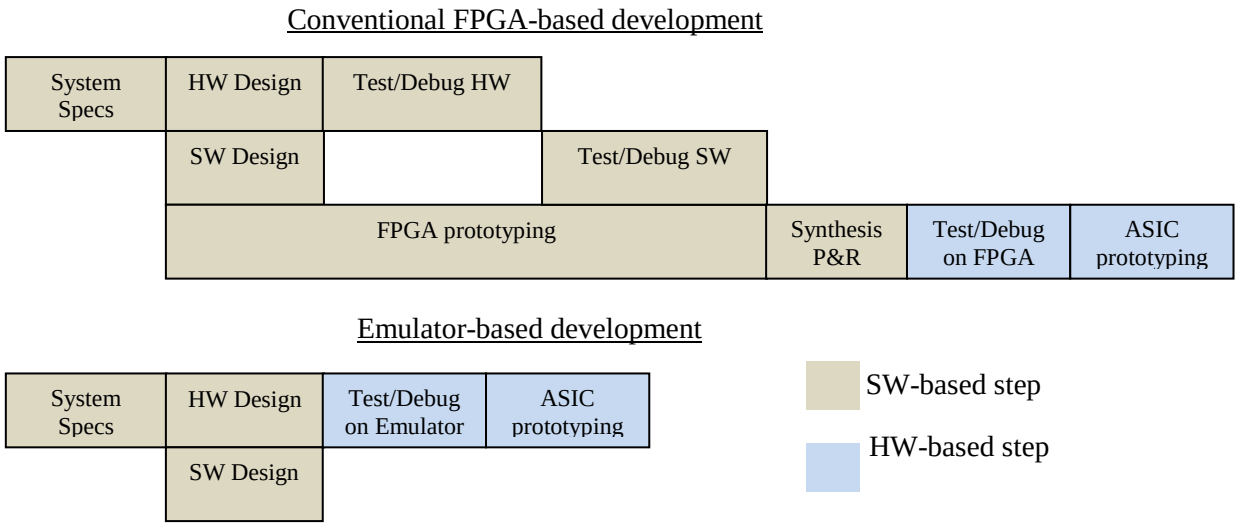

Figure 4.1. New design process requires less man-power (less number of steps) and less total time.

In this Chapter we provide a solution which is transparent to the designer. Our methodology is able to move this emulation bottleneck away from the interface at the DUT boundaries, and thus reduce it significantly. This is done in an automated way, largely without any intervention from the designer, by using a code transformation tool and certain HDL Block libraries that we have developed. The main idea of our proposed methodology is to split the testbench into two sections and transform the portion of the testbench that communicates very frequently with the DUT to synthesizable code.

4.2 Communication Bottleneck

Hardware emulators allow designers to implement a circuit on an FPGA, thereby running simulations of the circuit at a much higher clock frequency than a software simulator can provide. When hardware emulators first became available, the complete circuit had to reside in one or more FPGAs, but today's emulators can communicate with a software simulator and allow designers to use all the models that the software simulator supports.

Although ISS models, TLMs, and pure C or C++ models all provide system designers with the means to create powerful testbenches in order to verify and evaluate their designs, it is extremely difficult, if at all possible, to synthesize these models and implement them on an FPGA. In practice, such testbench code runs in a software simulation environment usually on a general purpose CPU, and uses custom communication protocols to communicate with the synthesizable DUT (Design-Under-Test) that runs in hardware on an FPGA. This leads to a communication overhead between the testbench and the synthesizable DUT. A software testbench in a hardware-assisted environment is likely to create a major communication bottleneck.

The proposed solution reduces the communication overhead by synthesizing the portion of the testbench code that directly communicates with the DUT. In particular, the process involves the following steps:

1. Partition the testbench code into two parts: the testbench HDL code that directly interfaces to the DUT, and the testbench C-like code that interfaces to the testbench HDL code.
2. Transform the testbench HDL code part into synthesizable code.
3. Put everything on the same FPGA with any supporting modules as needed:
   - The C-like testbench runs on one (or more) of the FPGA's embedded processors.
   - The transformed HDL testbench is synthesized together with the DUT and a library of blocks that we have created so as to provide the environment for transparent communication with the DUT and the C-like testbench.

Usually the portion of the testbench that communicates directly to the DUT is written in an HDL, such as Verilog or VHDL, while high-level operations and behavioral models are written in a C-like language. Figure 4.2 shows how the testbench is split into these two parts. The communication path between the testbench and the DUT has now been synthesized into hardware and therefore the transactions are performed in a much faster way.

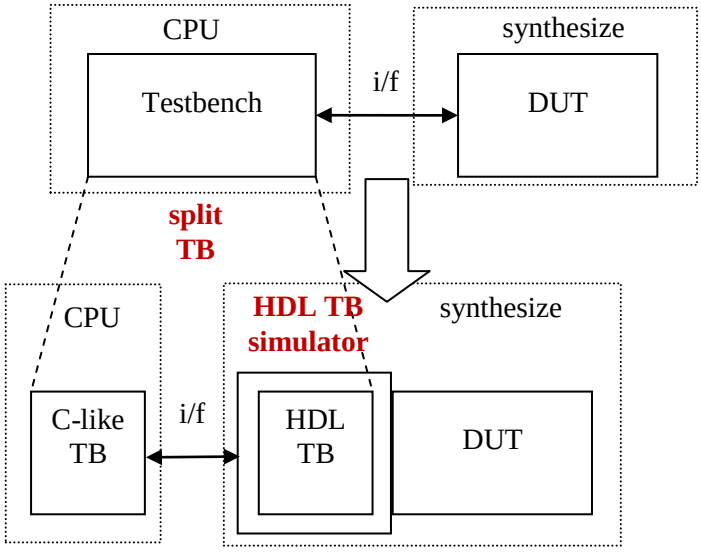

Figure 4.2. Splitting of the testbench.

4.3 System Architecture

A high-performance verification system should incorporate both processors and FPGAs. A processor-only or FPGA-only solution is limited in terms of performance or flexibility in simulating various types of models. First, in terms of the performance achieved, the maximum clock frequency of FPGAs lags behind that of processors implemented in contemporary ASIC. Therefore, processors with higher clock frequency execute behavioral models faster than FPGAs. On the other hand, FPGAs are more appropriate for executing simultaneous events and computation-intensive processes in parallel. Moreover, testbenches are commonly created using HDL such as Verilog or VHDL, sometimes including C-like programming language linked to the HDL simulator through e.g. the Programming Language Interface (PLI). This technique is used when the testbench needs to simulate more complex and more abstract functions. FPGAs are not capable of simulating models created in C-like languages and/or behavioral HDL that is not synthesizable. Therefore, processors and FPGAs have mutually complementary natures for high-performance verification systems. Modern large FPGAs provide on-chip general purpose CPUs and configurable CPU bus architecture facilitating the communication between the FPGA fabric and the CPUs.

In the proposed architecture, shown in Figure 4.3, the embedded CPU(s) located on the FPGA run the C-like behavioral part of the testbench, execute testbench floating point expressions, and access large arrays and external files as determined by the testbench code.

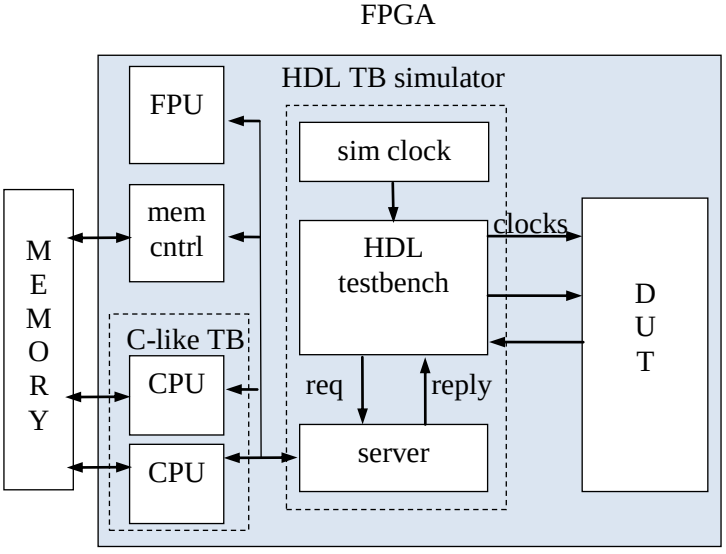

Figure 4.3. Proposed Architecture

The *Server Block* is responsible for serving the requests from the HDL testbench block. This block communicates with the embedded CPU(s) in order to execute PLI-like requests generated from the testbench code. Moreover it can execute other requests such as memory references, file accesses or floating point instructions. The large arrays of the transformed testbench code are stored in the external memory.

The *HDL Testbench Simulator* generates a simulation clock that coordinates the flow of the simulation. The clock speed of the simulator is defined by that of the testbench. The transformed HDL testbench block can pause the whole simulation environment in order to send requests such as PLI calls, memory references, file accesses or floating point instructions to the server block. In parallel, the HDL testbench block provides all the input signals including the clock signals to the DUT. This is more elaborated in Section 4.4.

A pipelined DDR memory controller and a single-precision FPU are used in order to offload the CPU. In this way, the Server Block can send memory requests directly to the external memory and perform floating point operations without the intervention of the slow CPU. This

saves many cycles on external requests leading to better performance results as shown in Section 7.1.

Two soft-core CPUs serve the PLI requests in parallel. If the requests sent to each CPU are independent (they access different memory areas for example) the CPUs can work independently. The parallelism exposed in almost any testbench code favors the existence of more than one embedded CPUs. State-of-the-art FPGAs can support multiple embedded high performance CPUs; for example Xilinx's Virtex-5-FXT FPGAs include two hardcore PowerPC processors and they can embed several MicroBlaze soft processors.

In this architecture the communication bottleneck between the software and the hardware sections of the simulation is pushed into the server-CPU(s), server-memory controller and server-FPU interfaces. However, the accesses on these interfaces are much more infrequent than the accesses at the DUT boundaries. Moreover, an additional and very important advantage is that these are fixed interfaces, independent of the emulated DUT.

In the next Sections, we describe the testbench code transformation, giving also emphasis on the pause/resume mechanism that the architecture provides.

4.4 Testbench Transformation

The original HDL behavioral testbench is transformed into synthesizable code that can run in the environment provided by the HDL Testbench Simulator of Figure 4.3. The tool we developed transforms a testbench written in VHDL language; same concepts can be applied to a Verilog testbench. The process body of a VHDL testbench includes various code sections that are not synthesizable. Such sections are mainly timing statements such as the VHDL *wait* statement, large arrays that are impractical or even impossible to be mapped onto FPGA embedded memories, floating point calculation and file handling.

A VHDL process of the transformed VHDL testbench running in the HDL Testbench Simulator can access the CPUs, the external memory and the FPU by sending requests to the Server Block. We have enhanced the functionality of the VHDL processes in the transformed testbench in such a way that they can pause the simulation time of the HDL Testbench Simulator in order to transfer requests to the Server Block (Section 4.7). In every simulation clock cycle the

HDL Testbench Simulator serves all the pending requests before advancing the simulation time counter.

We use the tree structure of the VHDL code to transfer the requests from the body of a process to the Server Block; a code segment that receives the requests from a process body and forwards them to a scheduler block is attached to each process. In every VHDL module a scheduler block is responsible to gather the requests from all the processes in the module and advance them a layer higher in the VHDL hierarchy. The scheduler block can serve the requests in any order since the simulation is paused when any request is pending. This process is illustrated in Figure 4.4.

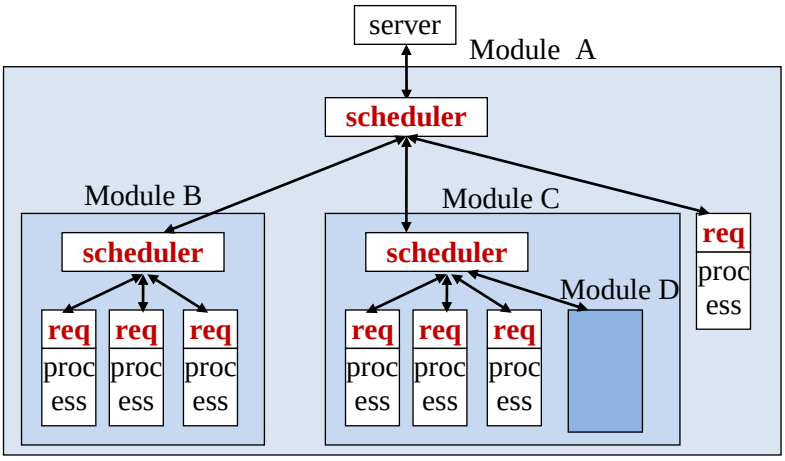

Figure 4.4. Tree-like Scheduling of Requests

4.5 Simulation Clock and Clock Management

The HDL Testbench Simulator provides the simulation clock that coordinates the functions of the simulation by translating all timing references in the original testbench code into simulation clock cycles. Every simulation clock cycle is divided into four simulation ticks, where the tick period is equal to the clock period of the synthesized testbench. The four tick time interval is essential for the operations performed by the transformed, by our toolset, VHDL process during a simulation cycle. This is because any transformed process requires four ticks at most in order to perform all its operations involved in a simulation cycle unless a request is generated, as the next section clearly demonstrates. Upon a request from a process the simulation time stalls until the request has been served and the four tick time interval starts over.

A common practice in several designs is to feed the external clocks to on-chip clock phase-locked loops (PLLs) and/or delay-locked loops (DLLs) in order to generate stable frequencies, recover a signal from a noisy communication channel, generate a phase shifted clock, or distribute clock timing pulses in the design. Such an example digital circuit used in Xilinx FPGAs is the Digital Clock Management (DCM) which supports clock delay locked loops, digital frequency synthesizers, digital phase shifters, and digital spread spectrums. Figure 4.5 shows the inputs and outputs of the Xilinx DCM. DCM includes a clock delay locked loop used to minimize the clock skew in the Xilinx devices. It synchronizes the clock signal at the feedback clock input (CLKFB) to the clock signal at the input clock (CLKIN). The locked output (LOCKED) is high when the two signals are in phase. The signals are considered to be in phase when their rising edges are within a specified time (ps) of each other.

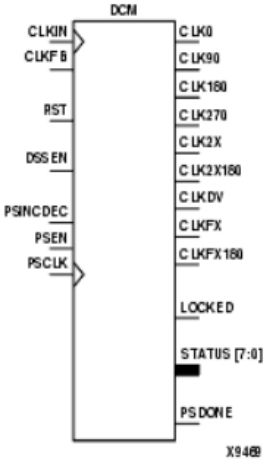

Figure 4.5. Xilinx' Digital Clock Management

On-chip clock management circuits cannot operate properly in the simulation environment of our proposed platform since the DUT clocks provided by the HDL TB Simulator block can be paused at any time. Since our emulator provides functional verification of the design simulating the system several times slower than the final implementation, clock skew or clock instability is not an issue during the emulation. Therefore, there is no need for internal PLLs/DLLs and such circuits have to be replaced with their corresponding simpler behavioral models. For example, an internal PLL that is used in order to provide a stable clock to the design can be removed and replaced by a simple wire that connects the external input clock directly to the output generated clock.

4.6 Testbench Simulation Flow

The transformation of the testbench code is process-based. During a simulation tick a VHDL process either (a) executes a code segment or (b) waits for the transition of a signal or (c) waits for some time interval or (d) sends a request to the Server Block. In order to achieve the aforementioned functionality every VHDL process is transformed according to the FSM shown in Figure 4.6.

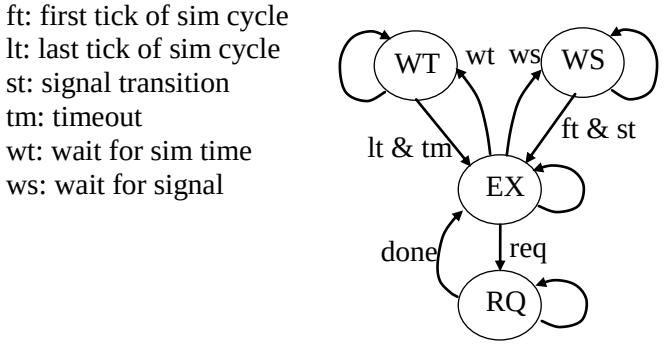

Figure 4.6. Process State Transition Diagram

In the *EX* state the process executes the synthesizable code of the testbench. A process stays in the *EX* state for 1 or 2 cycles depending on the original code. On a timing statement, such as the VHDL *wait* instruction, the process jumps in the *WT* or *WS* state. Finally, the process enters the *RQ* state in order to send a request to the server block. An example timing diagram that shows three processes and their state transitions is shown in Figure 4.7.

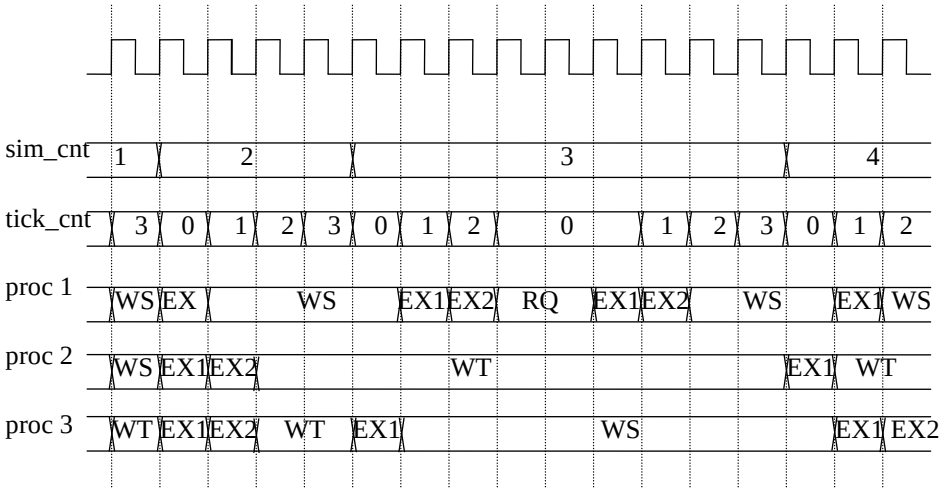

Figure 4.7. Process Timing Diagram.

The clock signals generated from the transformed testbench are fed to the clock buffers of the FPGA that drive in their turn the clock trees of the DUT. The transition from the *WT* state to the *EX* state can happen in the last tick of a simulation cycle while the transition from the *WS* state to the *EX* state can happen in the first tick of a simulation cycle, as shown in Figure 4.8. In this way, the synchronous signals of the transformed testbench change their values one simulation tick after the clock signals change their values and thus we prevent setup and hold time violations of the signals sent from the testbench to the DUT. This is depicted in Figure 4.8. Assuming that the *clk* and *val* signals are sent to the DUT, the *val* signal will arrive one tick after the *clk* signal which is certainly the correct behavior for functional verification.

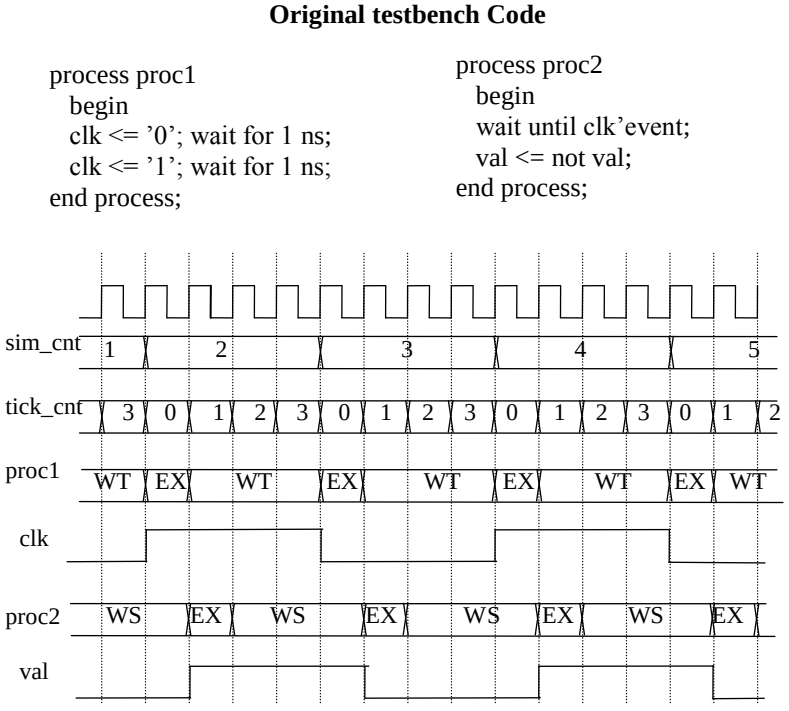

Figure 4.8. Setup and Hold Time Violations Prevention.

4.7 Pause and Resume Process State

Whenever a VHDL process executes a wait instruction or sends a request to the server block the process must stall, pause its state and resume at some time later. In order to add this functionality to a process all the statements in its body are transformed to conditional statements.

Any point in the process body can become an exit point by setting an exit condition at that point. Similarly the last exit point can become an entry point using conditional instructions.

Take for instance the code segment below and its transformation. The wait instruction becomes the exit point when it is first executed and the entry point after 10 simulation cycles, assuming that the simulation cycle is 1 ns.

Original code:

```
If clk = '1' then
    val <= '1';
    wait 10 ns;
    val <= '0';
end if;
```

Transformed code:

```
If reset = '1' then
    exit_point := 0;
else
    If (exit_point = 0 or exit_point = 1) and clk = '1' then
        if exit_point = 0 then
            val <= '1';
        end if;
        if exit_point = 0 then
            proc_state <= WT; -- enter WT state
            wait_time <= 10;    -- stay in WT for 10 sim cycles
            exit_point := 1;       -- exit point
        elsif exit_point = '1' then
                exit_point := 0; -- entry point
        end if;
    if exit_point = 0 then
            val <= '0';
        end if;
    end if;
end if;
```

If a process can pause its state at any instruction and resume it at some time later then non-blocking assignments may erroneously become blocking assignments. In order to avoid this erroneous behavior we transform all the non-blocking assignments of the original code into blocking assignments by using extra variables. Every extra variable corresponds to a variable used in a non-blocking assignment. The extra variable holds the value of its corresponding variable in the last simulation cycle. A VHDL process in the transformed code assigns the values of all the extra variables in the last tick of every simulation cycle.

Consider the code segment below and its transformation. In the transformed code, the last non-blocking assignment of the original code uses the last value of a, *a_last*, in order to avoid an erroneous assignment once the memory request has been served by the server block.

Original code:

```
process
    a <= '1';
    c[100] <= '1';
    b <= a;
end process;
```

Transformed code:

```
process  begin -- fix non-blocking assignments
      if last_tick = '1' then -- last tick of sim cycle
         a_last <= a;   -- all variables used in
         …              -- non-blocking assignments
      end if;
   end process;

   process begin
   If exit_point = 0 then
      a <= '1';
   end if;
   if exit_point = 0 then
      proc_state  <= RQ; -- memory reference
      req_type <= REQ_MWRITE;
      req_addr <= 100; req_val <= 1;
      exit_point := 1;
   elsif exit_point = 1 then
      exit_point := 0;
   end if;
   if exit_point = 0 then
       b <= a_last; -- use the old value of a
   end if;
       end process;
```

4.8 Simulation Breakpoint

Even though FPGA-based emulators have become an essential tool for many companies and research labs, it is surprising that no such system provides the equivalent of a software breakpoint: a mechanism to pause the emulation, observe its current state, and resume it.

In our proposed architecture this ability is readily available since we can do this simply by pausing or resuming the global simulation time counter of the clock generator in the HDL Testbench Simulator (Figure 4.3). This is the mechanism that the Server Block uses in order to pause the execution when serving a request.

Figure 4.9 shows the Flow Controller block which uses the same mechanism to temporarily pause the simulation upon the reception of a trigger event (such as a signal in the DUT receiving a specific value), providing in this way the equivalent of software breakpoints.

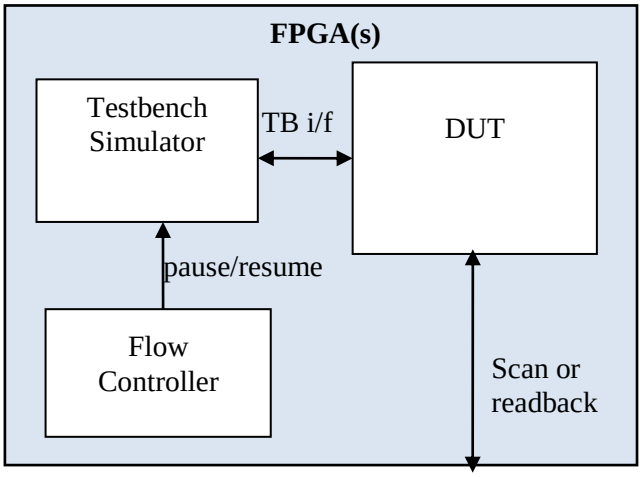

Figure 4.9. Flow Controller Block can pause/resume simulation

The mechanism to pause and resume the hardware emulation can provide us comparable benefits to a software simulator. Wheeler et. al. in [WG01] demonstrate how design-level scan can provide an efficient solution in order to monitor and control the status of the FPGA. However, this technique can work at the end of a hardware execution (not on the fly) while the authors do not show how this can work in real-time-systems. Combining this scan chain methodology and our proposed architecture we can have a hardware simulation platform that can pause and modify the state of the design on the fly. In particular, we can read, save and modify the state of the memory elements in the DUT while the emulation is paused. FPGA vendors also provide other techniques to read the state of an FPGA, such as the built-in Xilinx readback.

It is important to notice that we can read and modify any signal of the DUT without performing any time-consuming full recompilation of the DUT that most embedded logic analyzers (such as Xilinx chipscope) would require. This is more elaborated in Chapter 5.

Moreover, the ability to insert breakpoints in hardware emulation can be used to accelerate the execution of multiple simulations with common long startup parts. The designer only needs to run the common startup part once, save the state of the DUT block and restore it for each subsequent simulation using different testbench configurations. Figure 4.10 depicts this process.

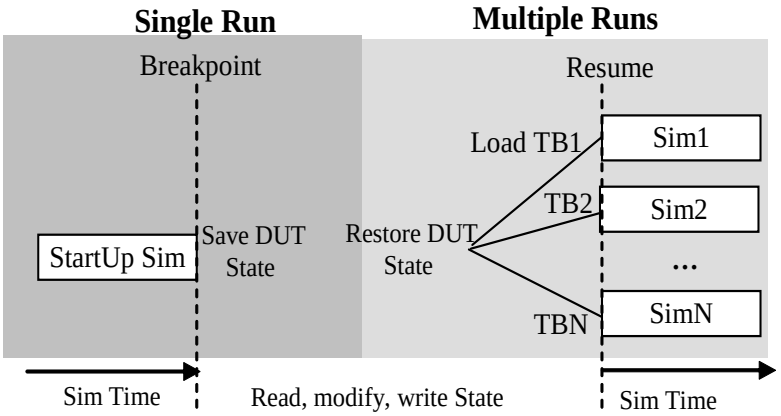

Figure 4.10. Using a “breakpoint” to speed-up the execution of multiple simulations with common startups

4.9 Transformations overview

Several other functions are performed by the tool we developed, so as to be able to reduce the communication overhead in a hardware emulator environment. Briefly, we mention the following code transformations:

- Timing references are transformed to simulation cycles.
- Large multi-dimensional arrays and their references are transformed into one-dimensional arrays in order to simplify their mapping to the external memory.
- *VHDL assertion* statements are sent to the external CPU.
- VHDL a*fter* statements are transformed into VHDL processes that are triggered when the after statements are executed.
- VHDL *select* statements are transformed into *if/else* statements.
- Processes that describe combinational logic which sends requests to the server block are transformed into sequential logic that is clocked with the simulation clock.

4.10 Summary

Hardware emulators and FPGA prototypes have long provided the highest performance when compared with all the verification approaches in the industry, but they have also suffered from a number of severe drawbacks. One of the most important problems is that complex emulator systems demand high communication throughput between the testbench and the synthesizable DUT which can eventually limit the performance of the simulation. To address the above shortcoming, we proposed to split the testbench into two sections and transform the portion of the testbench that communicates very frequently with the DUT to synthesizable code. Therefore, we built a tool that transforms a behavioral VHDL code to synthesizable code that can be implemented in our hardware simulation environment. In this way, we claim that we can overcome the testbench-DUT communication bottleneck and therefore increase the capabilities of today's hardware emulators.

Chapter 5. Circuit Observability and Controllability

"I have always wished for a computer that would be as easy to use as my telephone. My wish came true. I no longer know how to use my telephone."

Bjarne Stronstrup

Performing hardware emulation on FPGAs provides a faster, more accurate and closer-to-reality model than software simulations. However, it is impossible to bring all the necessary communication signals outside of the chip; on-chip visibility has become a significant issue. Recognizing this problem, FPGA vendors have provided tools to help designers understand what happens internally. However, circuit observability is still limited. Modification to the signals being captured or to the size of the data capture buffer often requires a time-consuming full recompilation of the user design. In addition, these tools provide no controllability of the Design Under Test (DUT). In this Chapter, we tackle these problems by adding multiple fast scan-chain paths in the design in order to provide full circuit visibility and controllability in a hardware emulator environment. The scan chain technique proposed provides an easy way for observing and/or modifying the state of hardware emulation on the fly. Our experiments (see Section 7.2) demonstrate that using around 25 scan chains is the optimum solution in terms of speed and area.

5.1 Background and Motivation

Apart from a fast emulation environment, engineers urgently need more efficient techniques, than the ones provided by existing systems, for debugging their complex IC designs. The existing hardware emulation schemes still face important limitations; in order to be very effective in the verification process, the hardware emulation framework should provide the same level of observability and controllability as a software HDL simulator does.

Towards this end, FPGA vendors have provided integrated solutions, such as Embedded Logic Analyzers (ELAs), which show the transient behavior of the design. Such tools allow the designer to easily probe the internal signals of the design inside an FPGA, much as he/she would do with a logic analyzer. For example, while the design is running on the FPGA, a trigger event determines when specific internal signals should be captured. In Section 3.3 we presented several ELAs such as the ChipScope by Xilinx, the SignalTap by Altera, the ClearBlue by DAFCA, the Configurable Logic Analyzer Module by FS2, and the embedded logic analyzer module by Cisco.

All the aforementioned products lack circuit controllability (with the exception of ClearBlue) while they provide limited circuit observability. Compared to software HDL simulators, the existing ELAs have some important limitations:

- Changing specific parameters, such as the signal probes or the depth of the sample buffer, in most cases, requires a time-consuming full recompilation of the user design.
- The sample memory of the analyzer, which determines the maximum trace period, is limited by the memory resources of the FPGA. In a design that uses much of the FPGA's memory, there may not be much memory left over for the ELA.
- Basic debug operations such as breakpoints, and step by step execution are not supported.
- There is no controllability of the design; the user can not set the value of an internal signal.

We tackle these problems in Sections 5.2 and 5.3 by adding multiple fast scan-chain paths in the design in order to provide full circuit visibility and controllability in a hardware emulator environment. The scan chain technique proposed provides an easy way for observing and/or modifying the state of hardware emulation on the fly. Moreover, based on the scan-chain

methodology, we propose, in Section 5.4, the architecture of a novel Embedded Logic Analyzer along with a software toolset supporting full circuit observability and controllability.

5.1.1 Scan-Chain Methodology

Scan-chain path insertion includes wiring up the memory elements, such as flip-flops (FFs), in such a way so as to have the state bits contained in these elements exit the circuit serially through a *ScanOut* signal whenever the *ScanEnable* control signal is asserted. New state data concurrently enters the circuit serially on the *ScanIn* pin. When *ScanEnable* is deasserted, the circuit returns to normal operation. While scan-chain paths are usually employed to find defects in the silicon, we use this technique in order to provide full circuit observability and controllability.

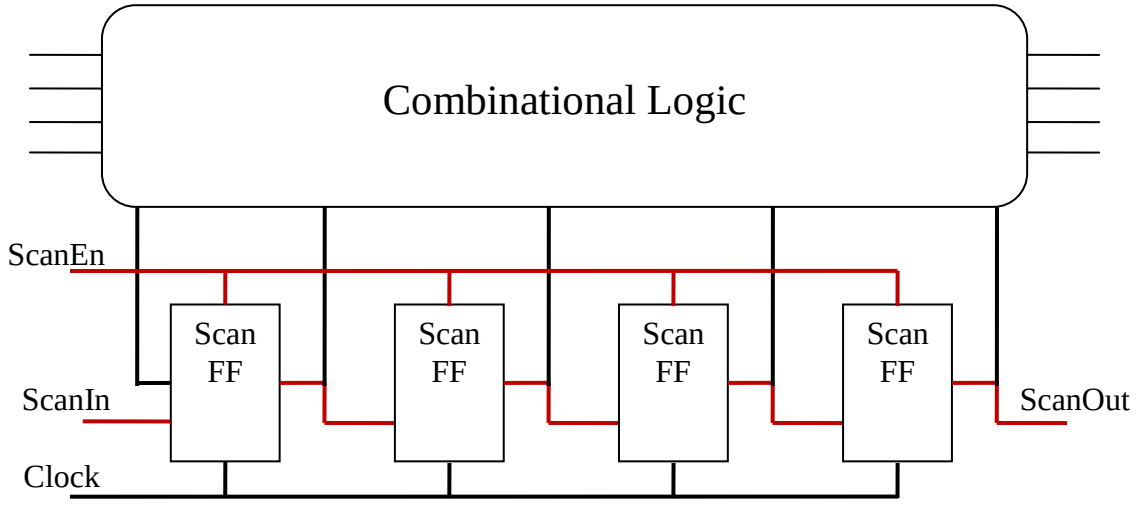

Figure 5.1. Scan Chain Architecture

Since FPGAs do not support scan FFs (except for the FreedomChip [LS07]) we developed a tool (described in Section 5.3) that automatically adds the scan circuitry to the synthesized DUT. Figure 5.2 shows how a FF of a design can be inserted into a scan chain by attaching a multiplexor (mux) and logic gates at the inputs of the FF.

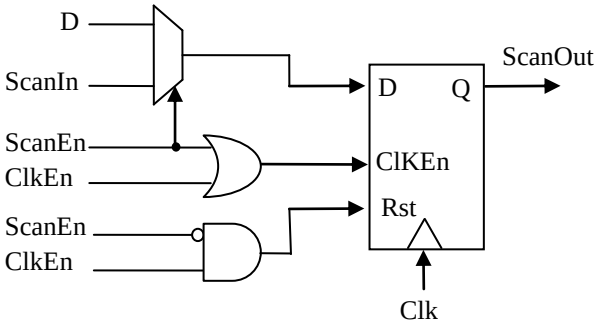

Figure 5.2. Scan Flip-Flop

5.2 System Architecture and Methodology

The system presented here consists of the FPGA debug infrastructure and the supporting toolset needed so as to provide full chip observability and controllability. The tool provides an environment to the designer, similar to that of a software HDL simulator, where he can execute the standard hardware emulations on the FPGA, trace internal signals, modify signal values and perform step-by-step execution.

The common process is to define a trigger condition and a trace period where the tool captures the DUT traces. During this period the **system repeats automatically three steps: pause execution, scan of DUT signals, resume execution.** The proposed architecture is depicted in Figure 5.3. The ELA block reads/modifies the status of the DUT by employing an extension of the scan chain methodology (see Section 5.3) proposed in [WG01]. The ELA block takes several clock cycles to read/modify the state of the DUT through the scan chains every time the execution is paused and enters the scan mode. In contrast to conventional ELAs that read the state of the DUT while the emulation is running, our ELA reads the state of the DUT when the FPGA emulation is paused.

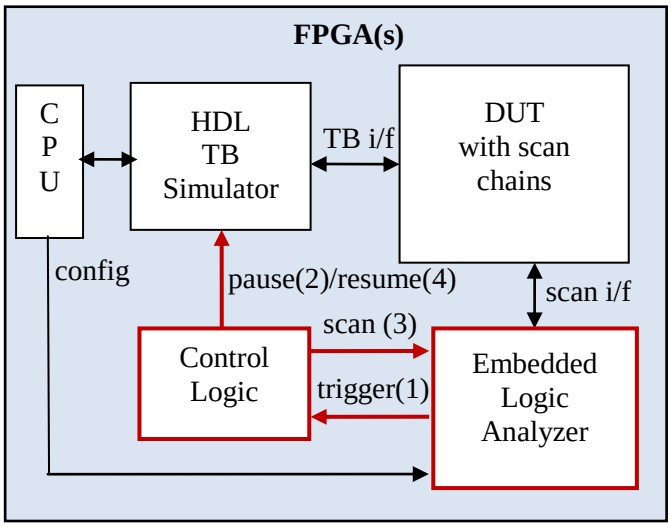

Figure 5.3. System Architecture

Thus, providing a mechanism for pausing temporarily the hardware emulation gives the opportunity to the ELA to observe and/or modify the status of the design using the scan-chains. The key idea for supporting real-time circuit testability is the use of the HDL Testbench Simulator. The HDL Testbench Simulator block uses a clock generator to drive all the clock signals of the DUT. Chapter 4 describes in detail how to control (pause and resume) the emulation execution by pausing or advancing the internal simulation time counter of the clock generator.

The embedded CPU executes the C-like testbench code and configures the ELA at the beginning of an emulation.

The Control Logic is a relatively simple block that sends requests to the HDL Testbench Simulator and the ELA blocks when a trigger event is satisfied in order to capture the trace data. Once the ELA catches a trigger event (i.e. the trigger condition becomes true) it notifies the Control Logic. The Control Logic then reads the simulation time from the HDL Testbench Simulator in order to send pause and resume requests periodically. Upon a pause request, it also sends a scan request to the Logic Analyzer. Figure 5.4 shows an example timing diagram of the aforementioned functionality, where sim_clk is the clock signal used by the HDL Testbench Simulator block; this signal is used to coordinate the emulation and generate all the DUT clock signals. In this example the Control Logic is configured to send a pause and a scan request once

every DUT clock cycle (DUTclk) where the period of the DUT clock is assumed to be equal to two simulation cycles (sim_clk).

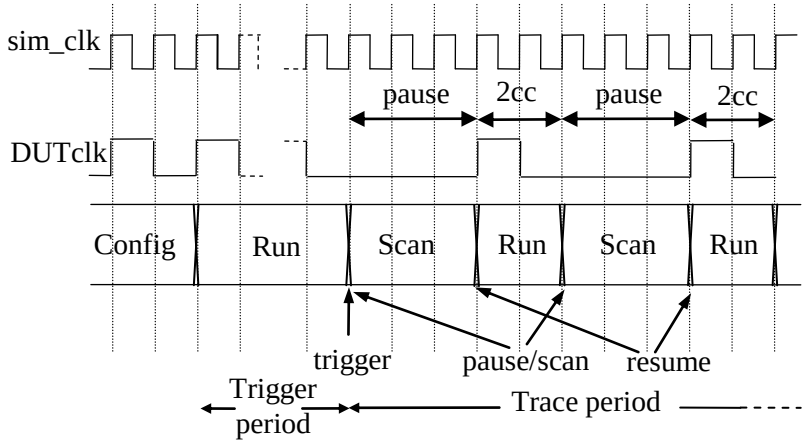

Figure 5.4. Timing Diagram

During the Scan Period of Figure 5.4 the Logic Analyzer reads all the signals of all the scan chains in order to capture the ones that the designer has specified. As we demonstrate in Section 7.2, the length of the scan period depends on the design size and the number of scan chains.

5.3 Multiple Scan-Chain Paths

Since the time to process a scan chain is proportional to its length, we reduce this processing time by employing multiple scan chains that are accessed in parallel. The same technique is used in DFT [JC05, HP99].

The tool, developed in Perl, processes the synthesized DUT in order to add multiple scan chains. The tool tries to produce the optimal routing of the scan chains in order to minimize the area occupancy and the access time by placing the FFs in the scan chains according to their topology in the design; FFs logically close in the design are also close in the scan chain path. Moreover, the tool tries to balance the lengths of the constructed scan chains. The number of scan chains generated by the tool is parameterized. This number affects the area of the ELA as well as the time to process the scan chains. This is further discussed and evaluated in Section 7.2. The steps performed by the tool are the following:

1. Reading of the FPGA vendor libraries in order to determine the inputs and the outputs of all the primitive elements.
2. Reading of the design in order to find all the instances of the primitive elements as well as their input and output signals.
3. Processing of the design instances in order to partition the FFs into separate clock domains. Two FFs belong to the same clock domain if they use the same clock and there is a combinational or sequential path that connects them.
4. Processing of the clock domains in order to generate the FF connectivity graphs. A FF is connected to another FF in the graph of a clock domain only if there is a combinational logic path that connects them. Each clock domain has a separate connectivity graph.
5. Generation of long scan chains. For each connectivity graph a single scan chain is formed where the FFs are placed according to their topology in the connectivity graph; i.e. FFs close in the graph will remain close in the scan chain. Moreover, FFs belonging to the same bus are placed together in the scan chain.
6. Partitioning of the long scan chains into multiple shorter scan chains. The number of the short scan chains is defined by the user. The target is to minimize the length (number of FFs) of the longest scan chain, which is succeeded by trying to equalize the sizes of all the scan chains.

5.4 Embedded Logic Analyzer

Embedding a Logic Analyzer in a programmable logic device allows signals to be captured after a trigger condition is true usually for a small period. An ELA captures and stores logic signals and provides them to a graphical user interface (waveform viewer). The proposed ELA has no limitation regarding the size of the data capture buffer and the number of signals. Moreover, the user can modify the signal values or run a simulation in step-by-step mode providing full chip controllability and observability.

5.4.1 Functionality of the Logic Analyzer

The ELA starts in the Trigger period (see Figure 5.4) by continuously checking for the trigger condition. There are two ways to monitor the trigger signals in order to evaluate the trigger condition:

- The common way is to route the trigger signals from the DUT to the ELA. In this case, the user has to reroute the trigger signals and recompile the design whenever the trigger signals change.
- Alternatively, the ELA can read all the scan chains in every cycle of the hardware emulation, even before the trigger condition becomes true, in order to capture and test the trigger signals periodically. **In this case there is no need for a recompilation of the design when the trigger signals change. However, the ELA delays the entire emulation significantly and therefore its performance is critical.** As we show in Section 7.2.3 this is not a problem for the proposed ELA which satisfies the speed needs of today's DUTs and can therefore adopt this innovative technique. One more advantage following this approach is that if the ELA can trace any signal in the DUT during the Trigger period, the user can easily program the ELA (through the embedded CPU) to support multiple trigger conditions that are altered on the fly depending on the trigger results.

Once the trigger condition becomes true the system enters the Trace period. In typical hardware emulations the Trace period is usually short, when compared to the entire execution time, and therefore it is not considered critical.

The performance of the ELA determines the Scan period. The ELA enters a Scan period when it receives a scan request by the Control Logic. During this period the ELA accesses all the scan chains in parallel and captures/modifies the values of the data according to its configuration. The length of the longest scan chain as well as the amount of the captured data determine the duration of each scan period as will be discussed in Section 7.2.

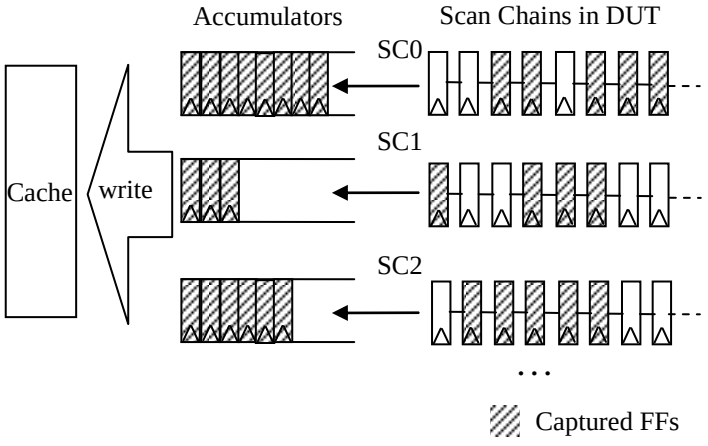

Figure 5.5. Captured FFs by the Logic Analyzer.

During a scan period, the scan chains in the DUT are accessed by the ELA synchronously (i.e. at exactly the same frequency, in lock-step with one another). In each access cycle the FFs at the head of the scan chains are accessed. A 32-bit register per scan chain accumulates the values of the FFs that we want to capture as shown in Figure 5.5. When an accumulator becomes full it is written to a fast embedded cache that holds the captured FFs' values. Since the FFs that we want to capture have well defined positions in the constructed scan chains, and the scan chains are accessed and written to the accumulator registers with a well defined and predetermined timing, we can calculate the position in the embedded cache where each captured signal will end-up to. These calculations are automatically done for each captured signal by a post-processing tool we have developed. This tool reads all captured values, and converts them to a format suitable for display in a waveform viewer.

Since the embedded cache is not big enough to hold all the captured data, a DRAM memory controller periodically transfers the cache contents to an external DRAM. To prevent the embedded cache from overflowing, in case the ELA writes trace data faster than the DRAM controller can read them, a flow control signal can temporarily stall the ELA. **In this way, the maximum amount of data that we can capture is determined by the size of the external DRAM memory, instead of the size of the limited on-FPGA memory (which is the case for most existing systems)**.

5.4.2 Configuration of the Logic Analyzer

The ELA can capture and/or modify the values of the DUT FFs. First, the designer specifies the trigger condition and the trace data, as well as the length of the Trace period. A software tool that is aware of the positions of the FFs in the scan chains configures the ELA accordingly.

In each cycle a word with the values of the FFs at the head of the scan chains is formed. A 16-bit embedded pointer memory, shown in Figure 5.6, holds the positions of the words with the traces (i.e. words including at least one bit that should be captured). Each 16-bit entry of this memory specifies a number of continuous words. This is done by specifying the position of the first word and the number of the following words. In this way, an entry can be used to specify a group of adjacent words.

A 32-bit mask in a mask memory (Figure 5.6) is associated with each word pointer and its group of pointed words. The mask specifies the positions of the captured FFs in this group of words. If some words from different groups overlap then their masks are ORed together. The entries of the configuration memory are placed in order, based on the positions of the pointed words. In every scan period, the ELA processes the configuration memory sequentially while it reads the scan chains.

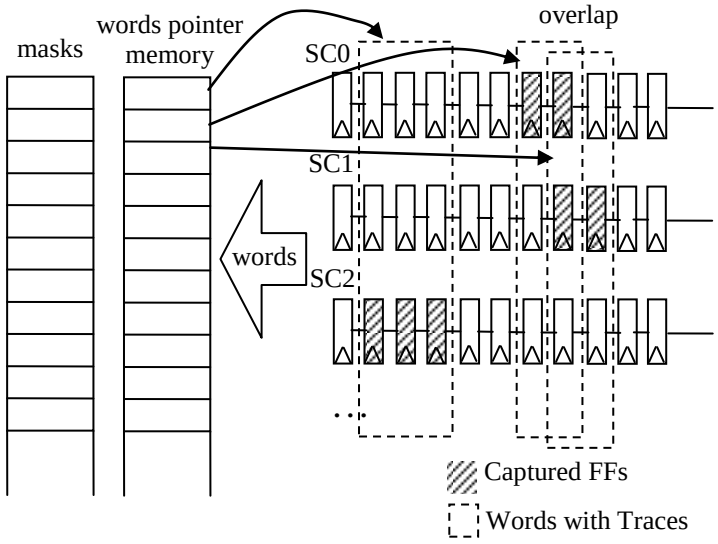

Figure 5.6. Configuration Memory of the Logic Analyzer.

The ELA can also modify the value of a DUT signal during the Scan period. In this case the configuration memory is not used. The ELA supports three configuration registers (not shown in Figure 5.6) for this operation; the WordUpdPos and WordUpdMask registers that specify the position of a word in the scan chains and its FFs that will be modified respectively, and the WordUpdVal register that holds the new value. For every modification of a word the user has to update these registers and send a scan/modify request.

5.4.3 Architecture of the Logic Analyzer

A simplified block diagram of the ELA is shown in Figure 5.7. The ELA can capture and/or modify the values of the DUT FFs. In order to do so, the designer first specifies the trigger condition and the trace data, as well as the length of the trace period. Our software tool, that has constructed the scan chains in the DUT, and therefore it is aware of the positions of the FFs in

the scan chains, configures the ELA accordingly. In particular, the tool generates the configuration code running on the embedded CPU which sets the Configuration Memory of the ELA at the beginning of the emulation.

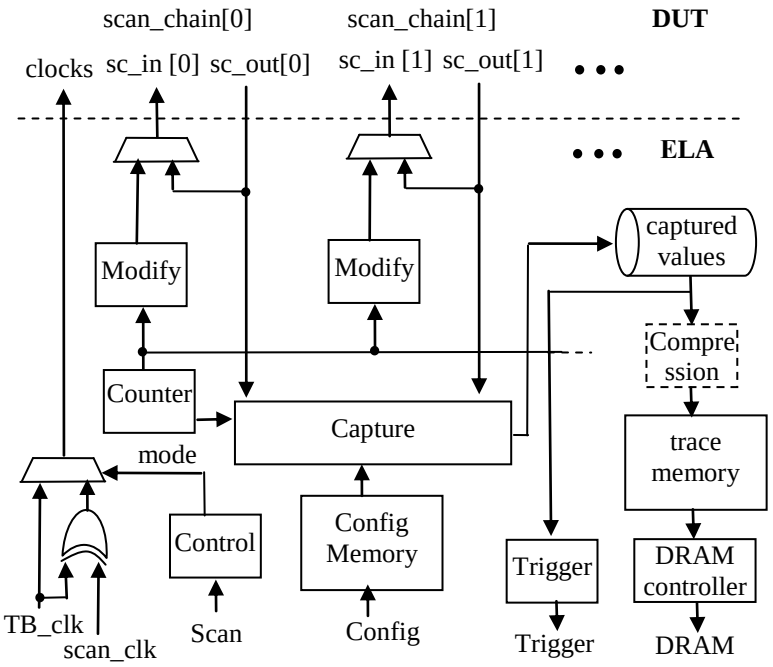

Figure 5.7. Logic Analyzer Architecture

The Counter is reset when a new scan request is received and is incremented whenever a new value is read out of the scan chains. Each Modify block is responsible for modifying the value of a signal in the corresponding scan chain according to the ELA configuration registers (see Section 5.4.2) and the value of the Counter. The Capture block compares the value of the Counter with the information in the Configuration Memory in order to capture the traces using the accumulators of Figure 5.5 where the requested traces are collected. The traces are stored temporarily in the trace memory before being forwarded to the external DRAM, as explained in Section 5.4.1. A compression unit can optionally be used to reduce the amount of data stored in the trace memory.

Figure 5.7 also shows how the DUT clock signals are set during a scan period. In normal mode these signals are generated by the HDL TB Simulator block (Figure 4.3) and provided to the DUT as described in Section 4.6. In scan mode the clocks are connected to the scan_clk input which is also used by the ELA. However, if the value of a clock signal, generated by the HDL TB Simulator, is 1 when the ELA enters the scan period, the state of the DUT registers using this

clock signal will not be restored correctly at the end of the scan period; using an XOR gate (as in Figure 5.7) solves this problem by inverting scan_clk.

The architecture of the ELA, although relatively simple, is very effective as Section 7.2 clearly demonstrates. Moreover, by using the scan command and the trigger block appropriately, our system is, to the best of our knowledge, the only one that provides: (a) step by step execution, (b) setting of break points and (c) setting of specific input test vectors at run-time.

5.5 Testing and functional verification

In order to verify the proposed methodology we built an environment for circuit emulations where we applied the proposed framework. In particular, we used the XUP Virtex-II Pro Development System from Xilinx which supports a Virtex-2P-30K FPGA which is a widely used state-of-the-art FPGA and described in more detail in Section 7.1.1.

We have tested the functionality of our platform with several DUTs and testbenches. One of the real world scenarios we emulated is a TDM card that was developed on a Xilinx FPGA in order to connect a network with hundreds of clients. The next Section describes this test case.

5.5.1 Test Case

The test case we employed in order to verify the proposed methodology is the FPGA design of a line card. The FPGA is responsible for the communication between Plain Old Telephone Service (POTS) interfaces and a back plane. In particular, it forwards the voice data from the POTS to the back plane and vice versa. Two separate designs have been emulated which correspond to the two supported modes of the line card: the Time-Division Multiplexing (TDM) mode and the Fast Ethernet mode. In the TDM mode the data is sent to the back plane through two TDM channels, while in the Fast Ethernet mode the data is sent to the back plane through two Fast Ethernet interfaces.

Figure 5.8 depicts the line card and a rough block diagram of the design when it operates in TDM mode. Eighteen Si3241 Quad Codec chips are connected to the three TDM ports of the FPGA. The voice data arrives from the codec chips to the Client TDM block interleaved. The Inter2Ser block receives the interleaved voice data from Client TDM block, serializes them and sends them to the activated Backplane TDM block.

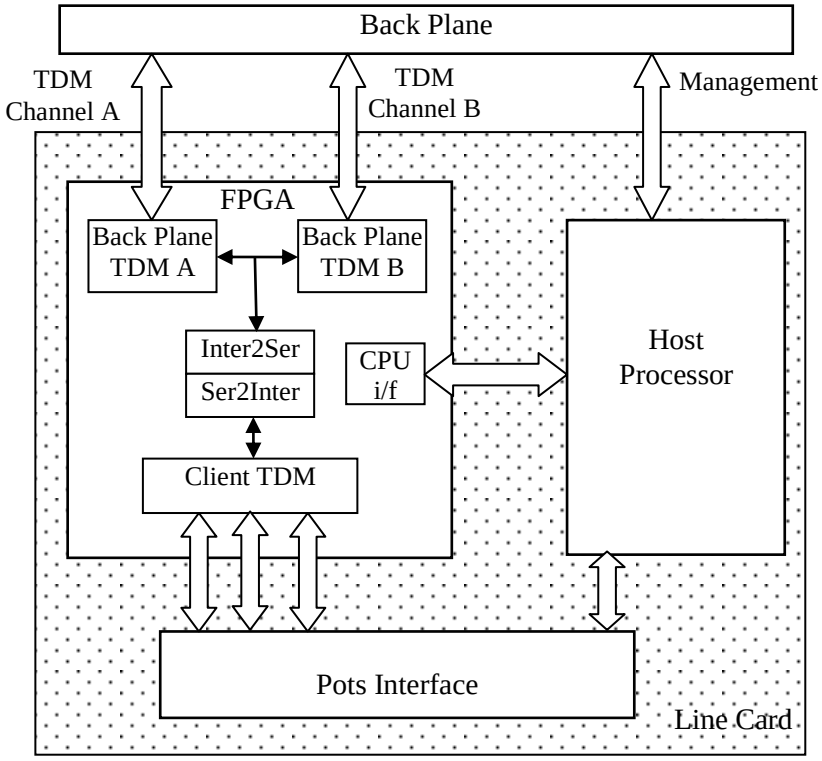

Figure 5.8. Line Card in TDM mode

In the opposite direction the voice data from the activated Backplane TDM block are transmitted to the Ser2Inter block. The Ser2Inter block is responsible for changing the order of the voice data bytes and sends them interleaved to the Client TDM block.

The CPU block is responsible for the signaling interface to the host processor. It also holds the configuration registers of the FPGA that are accessed from the host processor through the parallel interface.

The architecture of the Line Card in Fast Ethernet mode is similar to the one in the TDM mode. Two blocks, the MAC TX and the MAC RX blocks, have taken the place of the Backplane TDM blocks. The MAC RX block receives Ethernet frames and extracts the voice data for the codec chips while the MAC TX block generates Ethernet frames with voice data from the codec chips.

5.5.2 FPGA Clocks

In order to verify the correct functionality of the emulated system we need to take into account the clock domains and make sure that the clock in each domain functions properly. This

test case provides several clock domains allowing us to test the functionality of the hardware emulator and the ELA thoroughly.

Figure 5.9 depicts the clock domains of the FPGA when the Line Card operates in TDM mode. The three clock domains are colored. The interface to the POTS and the CPU serial interface of the CPU use a 2.048MHz clock. The backplane TDM interface and the inter2ser and ser2inter blocks use a 16.384MHz clock received from the backplane. Finally, the CPU parallel interface use a 48MHz clock received from the external CPU.

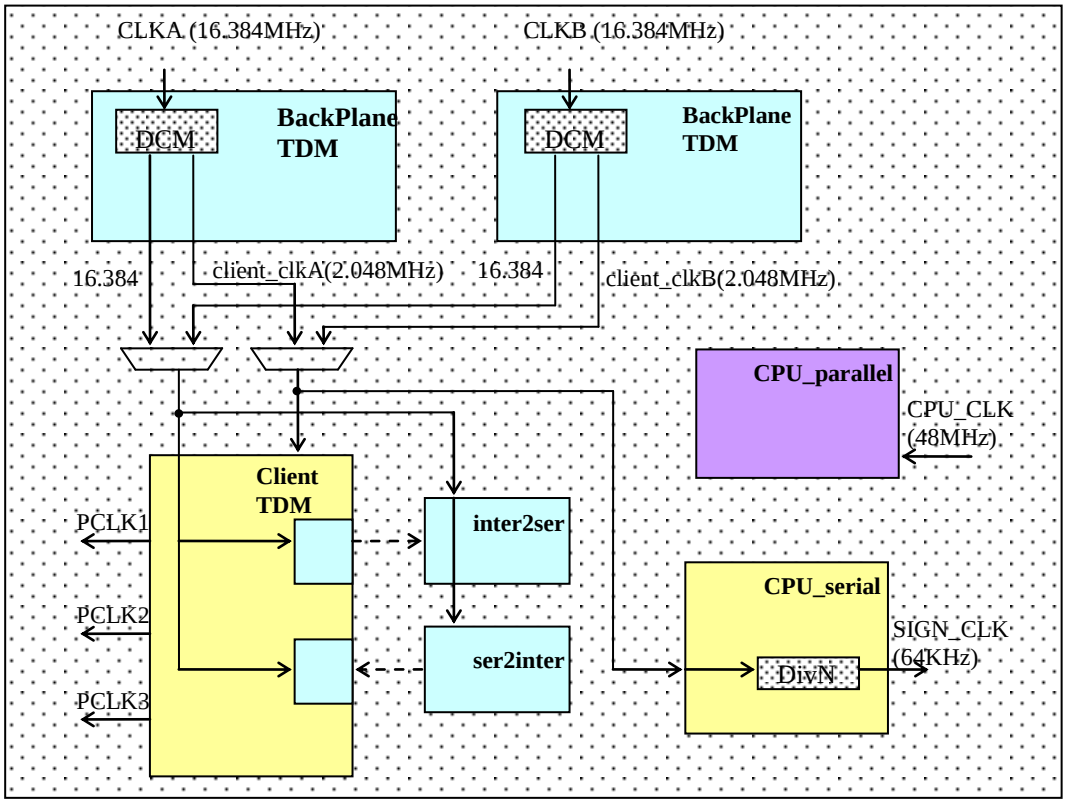

Figure 5.9. Clock domains in TDM mode

When the Line Card operates in Fast Ethernet mode the design employs four clock domains: physical transmit at 25MHz, physical receive at 25MHz, client TDM at 2.048MHz and the CPU parallel at 48MHz.

5.5.3 Testing Environment

In order to verify the functionality of the proposed hardware emulator and ELA we connected the XUP evaluation board with a workstation through a serial interface. In this way, we could read the captured signals, as shown in Figure 5.10. The on-chip PowePC microprocessor communicates with a uart block that supports an RS-232 link in order to send the values of the captured signals to a Hyper Terminal running on the workstation. The Hyper Terminal stores the data in a text file which is parsed by the *dump_signals.pl* script in order to create the waveform file. Next, a waveform viewer, such as the one used by modelsim, depicts graphically the waveforms.

The verification process is the following: First, the PowerPC sets the configuration memory and registers of the ELA such as the number of scan chains in the DUT, the size of each scan chain, the positions of the signals in the scan chains that should be captured, the trigger condition, and the length of the trace period. Next, the ELA controller starts the emulation. When the trigger condition is true, the ELA starts capturing the signals of the DUT. The values of the captured signals are stored in the on-chip trace memory. In parallel, the PowerPC reads the values from the trace memory and transfers them to the workstation through the serial port. If the ELA fills up the trace memory faster than the read frequency of the PowerPC, the ELA as well as the whole emulation is paused temporarily in order to prevent any memory overflow, as described in Section 5.4.1. The ELA stores data in the trace memory for the Trace period configured by the PowerPC.

In order to rerun a test and capture different traces or set a new trigger condition the user simply has to modify the configuration memory of the ELA (i.e. the PowerPC has to execute different configuration code), instead of performing a very time consuming design re-synthesis, re-placement and re-routing. Therefore, by simply modifying the code running on the PowerPC we could:

- Capture different signals in the DUT.
- Change the trigger condition.
- Change the length of the Trace period.

Essentially, there is no constraint on the number of the captured signals and the length of the Trace period. The trace memory never overflows since the emulation is paused in order to allow the PowerPC to empty it. The final destination of the captured values is the traces.txt file and not the trace memory which is a small buffer. Therefore, the external disk space of the workstation determines the buffer size used for capturing the traces which is orders of magnitude larger than any on-chip SRAM memory used by existing ELAs.

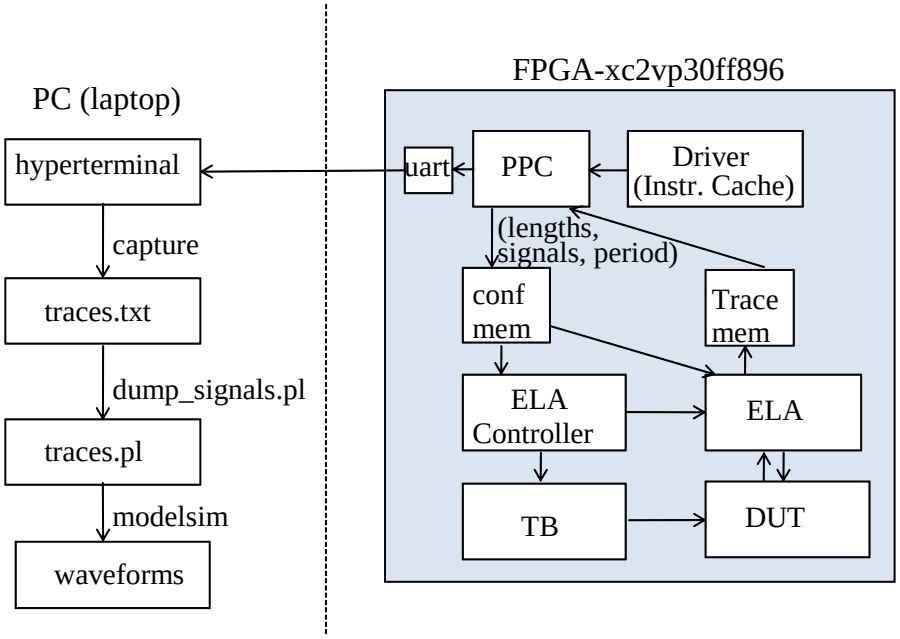

Figure 5.10. Testing Environment for the ELA

The design described in Section 5.5.1 has been extensively used in order to verify the functionality of the proposed platform. In particular, we tested the emulator by running several hardware emulations with different input voice data and comparing the captured traces with the corresponding traces from software simulations.

5.6 Summary

FPGA emulation has long provided the highest performance when compared with all the verification approaches in the industry, but it has also suffered from a number of severe drawbacks. One of the most important restrictions is the limited circuit observability and controllability provided to the designer. While Embedded Logic Analyzers provide circuit observability there are still significant limitations, such as a) the small number of the probed

signals and the short probing period due to the small internal memory of the FPGAs and b) the time-consuming process of changing the trigger condition or the probed signals due to the recompilation of the design. To address the above shortcoming, we proposed a novel methodology based on multiple scan chains that can access quickly any register in the DUT. Moreover, we proposed the architecture of a novel Embedded Logic Analyzer along with a software toolset that compose an emulation environment supporting full circuit observability and controllability on the fly.

Chapter 6. Hardware/Software Partitioning

"Computers WORK, people THINK."

IBM Corporation Old Adage

One of the most crucial tasks in today's complex embedded systems is to split them into their design components and allocate these design components to the available hardware and software system entities in a cost-effective manner.

This Chapter introduces a fully automated partitioning tool incorporating a novel flow with new cost metrics and functions. The tool employs two separate partitioning algorithms; Simulated Annealing (SA) and a novel greedy algorithm, the Grouping Mapping Partitioning (GMP). By selecting the partitioning algorithm, the designer can trade off between partitioning time and effectiveness. The innovative GMP algorithm operates in two stages: the first stage groups the design components according to how closely they interact, and in the second stage the grouped components are mapped into the system entities. System-level simulations provide accurate estimations in order to guide the tool to the most effective partitioning. The tool also interacts with the end user; a feature that is crucial in complex designs. Our experiments (see Section 7.3) demonstrate that the tool provides cost-efficient solutions in complex and large designs and derives close to optimum results. In particular our pioneering GMP algorithm produces results very close to those of SA while it is more than 2500 faster.

6.1 Background and Motivation

Partitioning is a fundamental CAD optimization problem. It is used at almost every level of abstraction during the synthesis of a digital system. The partitioning problem attempts to take a set of connected modules and group them to satisfy a set of constraints and optimize a set of design variables. During physical synthesis, partitioning is used during the floorplanning and placement tasks. In this case, the modules are gates that are connected by nets and they are partitioned in such a way that highly connected gates are in the same partition. As we move to higher levels of abstraction, the modules become larger; from standard cells to macro cells (logic level) to blocks (architecture level). Partitioning at higher levels of abstraction will impact the system performance in a more drastic way since the interconnect delay at the higher levels of abstraction is more pronounced. Even though the internal implementation and characteristics of a design usually are not well specified at the initial design phases, hardware-software partitioning is decided a priori and is adhered to as much as possible, because even small refinements in the partitioning may trigger extensive redesign. A good system partitioning is essential for the overall quality of the circuit.

The partitioning task is very significant for current FPGA designers since the vast majority of today's systems implemented in state-of-the-art FPGAs consist of a number of (mainly) soft-core CPUs as well as dedicated hardware modules. For architectures consisting of a processor and one or more fully dynamic run-time reconfigurable (RTR) devices, the nature of the partitioning problem changes, as a spatial as well as temporal partitioning must be performed [KK04]. In Reconfigurable Computing (RC) environments the partitioning algorithm can be applied several times so as to create many different designs that can be altered at run time. Therefore, performing fast hardware/software partitioning is especially important in RC. Greedy partitioning algorithms, such as the GMP algorithm described in this Chapter, can provide faster solutions than other partitioning algorithms.

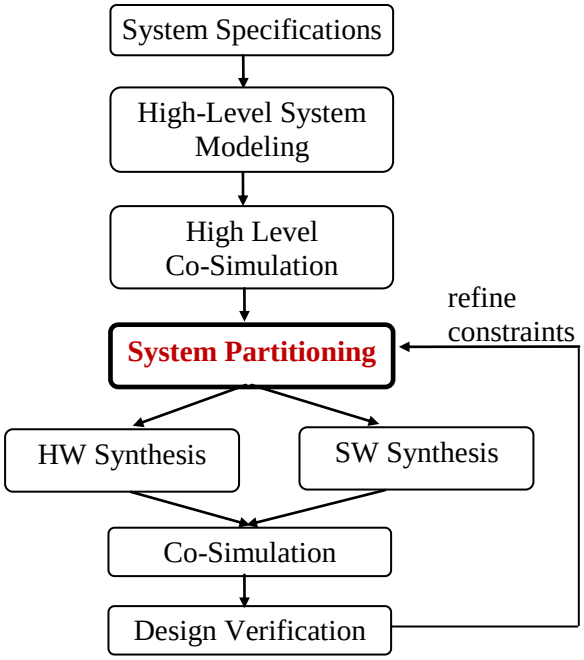

Figure 6.1. Hardware/Software Co-design Flow

It has been a common practice for designers to strive to make everything fit in software, and off-load only the most time critical parts of the design to hardware so as to meet timing constraints. Based on this approach, as Figure 6.1 shows, the designer first models the whole system in a System Level Modeling Language, such as SystemC. Then, the system level hardware/software co-simulation is a way to verify the functionality of the system and to give the designer a better and deeper understanding of the various characteristics of the design, such as attainable throughput and latency. An automated tool can then collect the information from the co-simulation phase and give designers certain feedback regarding their design choices such as the hardware/software partitioning, the CPU selection and the interconnection schemes of the system entities. Moreover, the designer needs a tool that can provide an integrated environment allowing her/him to quickly evaluate any such selections through various feedback mechanisms based on either formal verification or system co-simulation.

The partitioning algorithm is composed of two major tasks: *the creation of a system representation* which is usually an annotated graph, and the *partitioning of the system representation*. The partitioning problem is NP-complete as it requires the exploration of a design space whose size grows exponentially with the number of design components. Several investigations propose partitioning heuristics that an automated tool can follow. The most

effective ones include dynamic programming, greedy algorithms, hill climbing, simulated annealing, genetic algorithms, tabu search, integer programming, and ant colony optimization as described in Section 3.5.

Today's CAD tools although they provide efficient system level simulation and hardware/software co-simulation schemes, they still fail to address the system-level partitioning problem effectively. Unfortunately, many design platforms cannot provide trustworthy partitioning solutions yet and therefore they leave the actual hardware/software partitioning choice to the system designer, or allow the designer to interactively explore the design space of partitioning options.

In this Chapter we propose a partitioning tool that implements an innovative and fast approach so as to provide cost-efficient systems in a timely manner. We introduce the GMP partitioning algorithm that can produce very similar results (the difference is less than 3%) to those triggered by the most widely used algorithm (such as SA) while it is several times faster as shown in Section 7.3. Moreover, our tool supports also a standard SA algorithm implementation, if even higher quality partitioning results are required. In this way, by selecting the partitioning algorithm, the designer can trade off between partitioning time and effectiveness.

6.2 Partitioning Process

Before trying to solve the generic hardware/software intractable partitioning problem for systems with multiple diverse hardware and software entities, we focus on the simpler, while very important, problem of splitting the system into the parts that will be implemented in specifically designed hardware modules and the functions that will be executed on the embedded CPUs. **The proposed methodology tries to fit the maximum possible portion of the system into software as long as the capacity of the available software entities is not exceeded.**

In order to perform the actual partitioning the following distinct steps are executed:

- **System Description.** The system description usually determines the granularity of the design components. Our scheme gets as input a description of all the design components in a system level description language (such as SystemC), while the interfaces of the components follow the *Transaction Level Modeling* (such as SystemC TLM). The

functionality of each design component can be described in any of several abstraction levels, ranging from a detailed HDL-like description, to a very abstract behavioral one.

- **Cost Metrics Measurement.** The Size, MIPS and MTPS metrics of the components are measured. The original code is transformed and simulated in order to trace the transactions on the components' interconnections and the executed CPU instructions.
- **Transaction Graph (TG) Creation.** A graph of the description is constructed, with nodes representing the design components, and arcs representing their communication channels. The cost metrics are annotated on the graph.
- **Software Entities Specifications.** The MIPS and the external bus MTPS values of the software entities are specified by the designer.
- **Grouping Algorithm.** The graph is processed so as to produce a second graph in a more compact and manageable form. Each node in the new compact graph represents a group of highly-related design components.
- **Mapping Algorithm.** The graph is gradually splitted into two distinct parts: one part contains the nodes with the components that should be executed in software and the other contains the components that should be implemented in hardware.

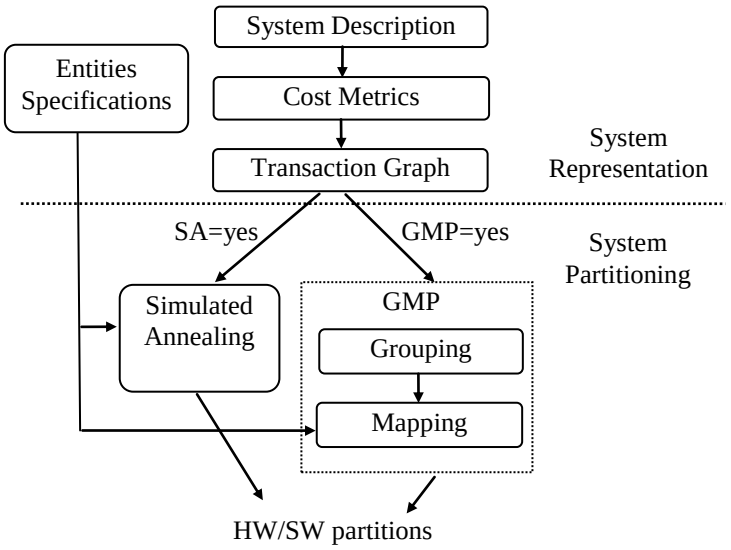

Figure 6.2. Steps of Partitioning Tool

The system partitioning algorithm, shown below the dashed line in Figure 6.2, is performed iteratively for each software entity until no more design components can be assigned to the software entities. In the following sub-section each of the steps is further described.

6.3 System Representation

The first main partitioning task is the representation of the system in a form that can accurately describe both the behavior of the design components as well as the available system resources.

6.3.1 System Description

The proposed tool takes as input a design described at TLM; such a description provides the transactions at the design components boundaries while it may conceal the internal implementation details. The designer should also provide the testbench code to the tool along with corner-case scenarios where the inputs, as well as the internal signals of the design, are switched at the maximum possible rates.

Moreover, the frequency of each clock in the design is specified in order to derive the transaction rates of the signals between the design components. If the rate of a system input is not fixed (for example it depends on the performance of the system), the whole partitioning process can be executed several times using different rates and clock frequencies. For example, this is a common case when the system reads data from a memory and the read rate may depend on the data processing rate.

6.3.2 Cost Metrics Measurements

In our tool we define three new cost metrics: (a) the *MTPS* (Million Transactions Per Second) of a design component's interconnection bus, (b) the *MIPS* (Million Instructions Per Second) and (c) the *Size* of a design component.

In order to understand the MTPS cost metric, let's consider an example with two design components, *A* and *B*, that are interconnected trough bus *X* as shown in Figure 6.3. If, say, component *A* is to be implemented in software and component *B* is to be implemented in hardware, their communication through bus *X* should be carried over the bus that interconnects the corresponding software (i.e. CPUs) and hardware system entities, say bus *Z*. This bus *Z*, which may implement a standard protocol such as AMBA in the case of ARM CPUs, will need to perform the transactions that are carried across the simulated bus *X*. The MTPS value

measures the transaction throughput of the bus that interconnects the software and hardware entities (bus *Z* in this example) that is needed in order to sustain the throughput of data exchange on the simulated bus that interconnects the actual design components (bus *X*). The MTPS value will of course depend on the transaction rate of the simulated bus *X* and the ratio of the widths of the two busses. For example, if 100 128-bit transactions per second are performed on bus *X* and we have a 64-bit bus *Z*, the MTPS value of the interconnection between *A* and *B* will be 200.

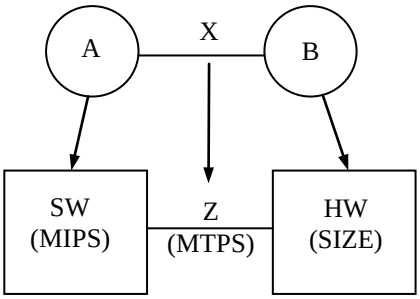

Figure 6.3. MIPS and MTPS metrics of component A

Continuing with the above example, the software system entity (i.e. CPU) that will implement design component *A* will need to run at a certain minimum speed in order to sustain the MTPS value for the interconnection busses of component *A*. The MIPS value of a design component defines the performance requirements for a software system entity, which simulates this component, in order to sustain the total MTPS value across all its interconnection busses. This value depends on the number of assembly instructions that should be executed if this component is mapped into software. The MIPS value, to the best of our knowledge, has not previously been used as a metric in hardware/software partitioning, whereas as the efficiency of our algorithm demonstrates it is a value that can precisely characterize the corresponding entities.

The clock frequency of a system entity or of a design component affects linearly its MIPS and MTPS values. Both MIPS and MTPS metrics incorporate the clock frequency which makes them ideal performance metrics for systems with multiple clock domains.

Our system first transforms the original code of the design so as to trace the transactions at the components' boundaries and the number of executed instructions. A Hardware Description Language (HDL), such as VHDL, Verilog and SystemC, usually employs a tree-like structure which we also utilize in order to create separate *flat names* for each design component. In Figure

6.4 each cycle represents an HDL basic block (module, thread, process, etc.) that instantiates other HDL blocks. The transformed code creates flat names for all block instantiations; those names are used in order to distinguish the operations of the design components in the trace file.

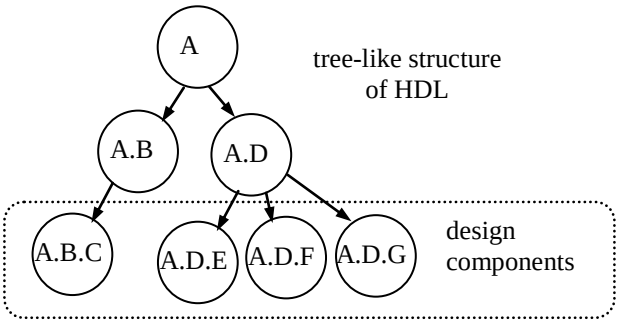

Figure 6.4. Flat names of design components

The partitioning tool performs simulations in order to generate a detailed trace file with all the bus transactions, the executed instructions and the clock transactions. Using the trace file the tool can automatically calculate the MTPS values of the interconnections and the MIPS values of the components. In order to take into account performance bursts (i.e. short periods that many instructions are executed), the MTPS and MIPS calculations are performed in successive short time periods and the maximum values are selected.

The Size of a design component is a metric demonstrating its silicon area if implemented in hardware. The area of a component can be estimated from the synthesis of its functions. A synthesis tool, such as [AGC] for SystemC code, can provide a rough estimate of the area of each component. An open-source SystemC synthesis tool is also developed in [OS10] that we can use for the area estimation of the components. Alternatively, the tool can measure the number of assembly instructions of the compiled description of the component in order to create a rough estimation of its size.

6.3.3 Transaction Graph Creation

In this step, the design is flattened and an annotated *Transaction Graph* (TG) is constructed. In this graph each node initially includes a single design component while the arcs correspond to the communication paths between the design components of the nodes. The cost metrics are annotated on the graph, as the example in Figure 6.5 shows.

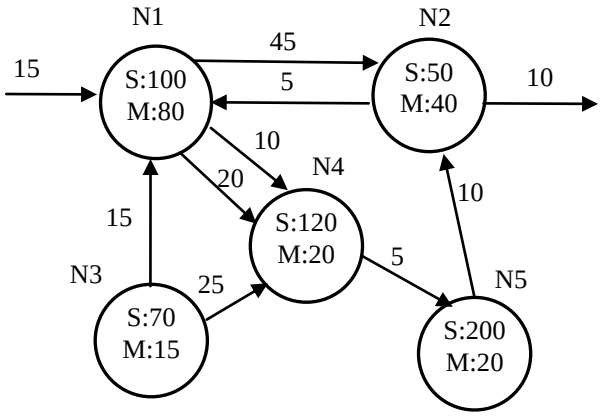

Figure 6.5. Example Transaction Graph where nodes are annotated with Size(S) and MIPS(M) metrics and arcs are annotated with MTPS metric.

In order to merge the nodes of a TG, the Sizes and the MIPS values of the initial nodes are added and the MTPS values of the interconnections are recalculated.

6.3.4 Software Entities Specifications

Regarding the software entities' specifications, the MTPS supported by the external bus of each software entity is specified by the designer. The protocol overhead can substantially reduce the utilization and therefore the maximum MTPS of a bus, so this should also be taken into account when the designer specifies the MTPS.

Moreover, the designer specifies the MIPS of each software entity as well as the Operating System overhead (performance percentage) triggered in order to serve multiple design components in parallel.

6.4 System Partitioning

The tool employs two separate partitioning algorithms, which are described in more detail in this section. The GMP algorithm is an innovative algorithm invented by us, whereas the tool implements also the SA algorithm which, based on certain studies such as [SN04], produce very good partitioning results.

6.4.1 GMP Algorithm

The proposed partitioning is performed in two stages: *Grouping* and *Mapping*. While the Grouping stage is not mandatory, it facilitates the Mapping algorithm leading to more effective results as shown in Section 7.3.

Grouping Algorithm

While there can be many potential partitioning solutions for a given design, we argue that there are groups of *closely related* design components, in almost every design, that will end up together in any cost-efficient partitioning solution. The Grouping stage aims to group together such design components early in the partitioning process using as a guideline the load of their intercommunication channels. Figure 6.6 shows the distinct steps of the Grouping algorithm. The steps in dash boxes are optional.

The algorithm consists of a coarse-grain grouping followed by a finer-grain grouping; in the latter phase the design components of each node can be regrouped.

In step 1 the designer can manually merge together nodes. These nodes usually implement a common function of the system and should be implemented on the same system entity. The designer can also specify that a group of nodes will be implemented in a specific-purpose hardware module or executed on a CPU; the specified nodes are merged together and remain intact throughout the whole partitioning process.

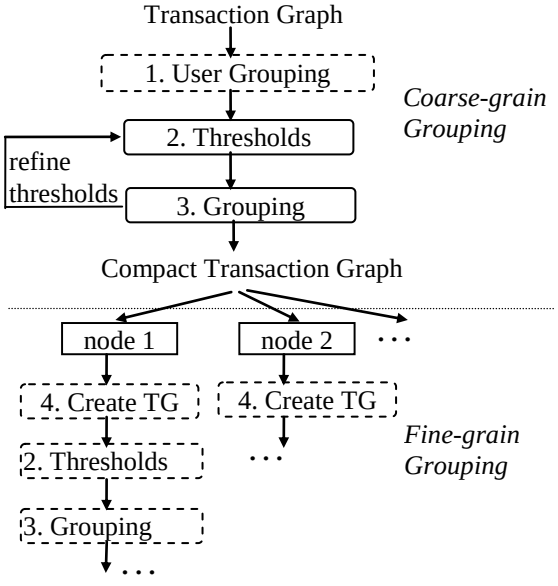

Figure 6.6. Steps in Grouping

In step 2 the designer defines the high *MTPS threshold* and the low and high *group-size thresholds*. These thresholds determine the level of *aggressiveness* (i.e. the grouping tendency) of the Grouping algorithm; the lower the MTPS threshold or the higher the low group-size threshold, the more nodes that will be grouped together. The high group-size threshold is used in order to limit the Sizes of the nodes in the resulting compact TG. Typical threshold values are shown in Section 7.3.

In step 3, a greedy algorithm has been implemented in order to group the nodes of the TG according to the aforementioned thresholds. This algorithm is described in the following pseudo-code:

a) Find unmarked node A with the minimum Size (all the nodes are initially unmarked and they are marked in step d). If all nodes are marked exit.

b) Find the adjacent nodes (nodes directly connected to node A) that their Sizes, when added to the Size of node A, do not exceed the high group size threshold. If no such adjacent node exists jump to d.

c) Select node B from the adjacent nodes with the maximum total MTPS value on the arcs connecting it with node A. If this MTPS value is greater than the MTPS threshold or the size of node A is less than the low group-size threshold merge nodes A and B and jump back to a. Otherwise continue with d.

d) Mark node A. Jump back to a.

In the TG created by this algorithm, for any two nodes that have total Size less than the high group-size threshold, the total MTPS value on the communication channels connecting them will certainly be less than the MTPS threshold. If the designer is not satisfied with the grouping results, she/he can refine the thresholds and rerun the Grouping algorithm.

In step 4 a node can be disassembled and a new TG can be formed from its design components. This TG is a sub-graph of the initial TG including only the design components of the node. Steps 4, 2 and 3 can be iteratively executed for each node of the TG. In this way, the designer can create a new sub-TG from the design components included in a single node and rerun the whole Grouping algorithm using different thresholds.

Mapping Algorithm

The Mapping algorithm maps the nodes of the TG derived from the Grouping step into the available entities of the system. The algorithm can map more than one node into the same entity

essentially merging the nodes together. **The primary goal of the Mapping algorithm is to fit as much logic as possible into the software entities (i.e. maximize the total SIZE of the software components).**

A greedy algorithm is employed which derives accurate results in a much faster way than existing algorithms. The partitioning methodology proposed here begins with the assumption that the entire system is implemented in custom hardware. The algorithm then visits the various nodes in the TG, finding those that can be executed in the specified CPUs without breaking any timing constraints.

In order to decide whether a node can be implemented on a software entity we need to compare its MIPS value and the total MTPS value of its arcs against the available MIPS and MTPS values of the specified CPU taking also into account the OS overhead. In general, our mapping algorithm creates sub-graphs and assigns them to the software entities. Using the cost function described below it selects a node and subsequently picks adjacent nodes. The specific steps of the algorithm are the following:

a) *Find a large node A in the TG with small MIPS and MTPS values that can be executed on the software entity.* Nodes that require high throughput and high processing power are obviously more suited for hardware implementation. On the other hand, certain nodes that are large and have low throughput and process requirements can better be executed on a CPU. For example, node 5 in Figure 6.5 is better suited for software implementation than node 1. Considering this goal, the node with the minimum value of cost function F_{SW} is selected as the best candidate for software implementation, where the cost function F_{SW} is defined as following:

$$F_{SW} = \max\left(\frac{MIPS_{node}}{MIPS_{avail}}, \frac{MTPS_{diff}}{MTPS_{avail}}\right) / Size_{node}$$

The $MIPS_{avail}$ and $MTPS_{avail}$ values are the remaining available MIPS and MTPS values of the software entity. The $MTPS_{diff}$ value is the actual numerical difference, in the remaining available MTPS value of the entity, if the node is selected to be implemented in software. The $MIPS_{node}$ and $Size_{node}$ are the MIPS and Size values of the node. This formula roughly shows the percentage of the available resources that a node will allocate if executed on this software entinty over its Size. This formula

characterizes accurately any system ranging from systems with high throughput requirements to systems with high process requirements since it dynamically employs the available throughput percentage or the available process percentage.

b) Calculate the new $MIPS_{avail}$ and $MTPS_{avail}$ values of the software entity:

$$MIPS_{avail(new)} = MIPS_{avail(old)} + MIPS_{node}$$
$$MTPS_{avail(new)} = MTPS_{avail(old)} + MTPS_{diff}$$

c) Use the same cost function F_{SW} in order to select adjacent nodes and create the maximum sub-graph starting from node A by merging nodes together as long as the sub-graph can be implemented on the specified software entity.

d) Map the sub-graph into the software entity. Jump back to a.

6.4.2 Simulated Annealing

Simulated annealing is a programming method that attempts to simulate the physical process of annealing. Annealing is where a material (as steel or glass) is heated and then cooled usually for softening and making the material less brittle. By analogy with this physical process, each step of the SA algorithm replaces the current solution by a random "nearby" solution, chosen with a probability that depends both on the difference between the corresponding function values and on a global parameter *T* (called the *temperature*), that is gradually decreased during the process. The dependency is such that the current solution changes almost randomly when *T* is large, but increasingly "downhill" (i.e. better solutions) as *T* goes to zero. The allowance for "uphill" moves (i.e. worst solutions) saves the method from becoming stuck at local optima, which are the bane of greedier methods. For any given finite problem, the probability that the simulated annealing algorithm terminates with the global optimal solution approaches 1 as the annealing schedule is extended. This theoretical result, however, is not particularly helpful, since the time required to ensure a significant probability of success will usually exceed the time required for a complete search of the solution space.

The most important factor in SA algorithm, as described in [WC02], is the employed cost function and whether this function allows the algorithm to *hill-climb* over suboptimal solutions.

The SA approach creates valid neighborhood solutions by moving a single node from one partition to the other. The algorithm is described in the pseudo code below:

Construct initial partitioning Pnow with all nodes in HW
Initialize Temperature T=TI
while stopping criterion not met {
  for i=1 to TL {
    Generate a random neighboring solution Pneigh
    Compute cost func. change ΔC=C(Pnow) – C(Pneigh)
    if ΔC >= 0 then Pnow = Pneigh
    else {
      Generate q=random(0,1)
      if $q < e^{-\Delta C/T}$ *then Pnow = Pneigh*
    }
  *Set new temperature T= a * T*
}
return solution Pnow
(where TI = 400, TL = 100, α = 0.98)

The parameters *TI* (initial temperature), *TL* (temperature length), *a* (cooling ratio) and *stopping criterion* specify the cooling schedule. The stopping criterion becomes true if no new solution has been accepted for three consecutive temperatures. The formula of the cost function we used, so as to take advantage of all our metrics, is the following:

$$C = -0.8 * \frac{Size_{SW}}{Size_{total}} - 0.1 * \frac{MTPS_{avail}}{MTPS_{SW}} - 0.1 * \frac{MIPS_{avail}}{MIPS_{SW}}$$

The $Size_{SW}$ is the total Size of the nodes that are selected to be implemented on the software entity and the $Size_{total}$ is the total Size of all the nodes.

6.5 Partitioning Tool Implementation

The input, to our tool, is an embedded system described in SystemC. A *SystemC thread*, similar to a Verilog module or a VHDL process, is the building block of any SystemC design and describes the functionality of a design component. Therefore, the developed tool assumes that each SystemC thread is a separate design component.

The tool is available at [OP10]. A snapshot of its GUI is shown in Figure 6.7, where the design is depicted as an annotated graph as well as a tree-like structure.

In order to verify the functionality of the developed platform we applied it on the SystemC descriptions of a digital filter and two small RISC CPU-based designs. The original codes of the digital filter and one of the RISC CPUs were derived from the SystemC examples publicly available at *www.systemc.org*. The other RISC CPU (named Mephisto CPU) is a Digital Signal Processing module that has been designed by the French research organization CEA (Commissariat à l'Énergie Atomique). These designs consist of 6, 12 and 18 threads respectively. The digital filter and the CPU from CEA are described in the following Sections.

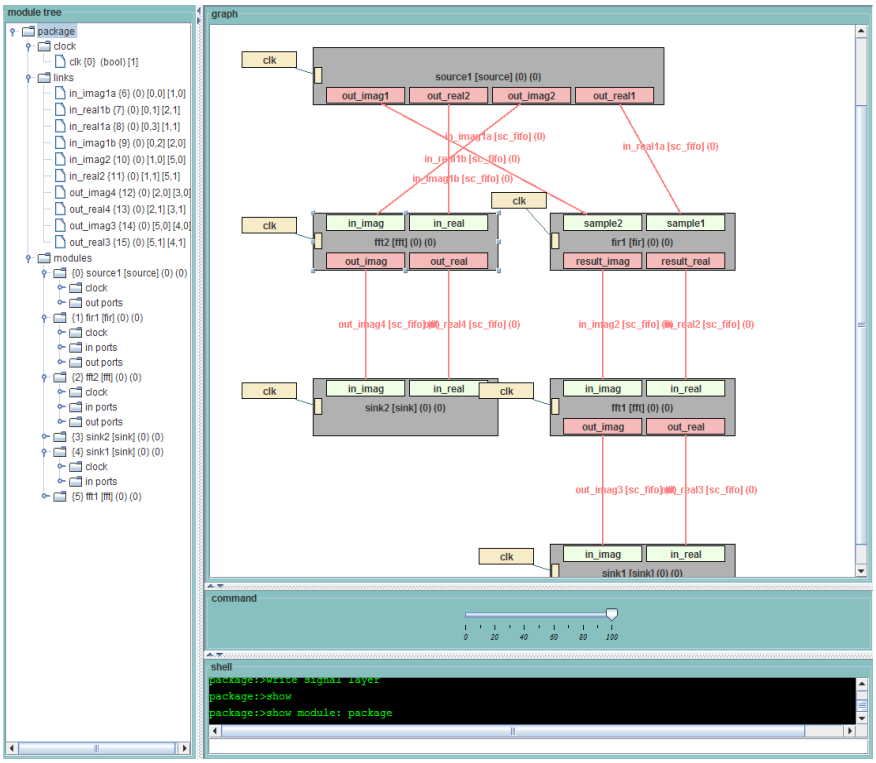

Figure 6.7. Graphical User Interface of Partitioning Tool

6.6 First Test Case

The example digital filter consists of six design components: one SOURCE block, two FFT blocks, one FIR block and two SINK blocks, as shown in Figure 6.8.

The tool first parses the design and creates a graph, where the nodes represent the design components and the arcs represent the communication paths between the components. The user can select to execute any of the following commands, as shown in Figure 6.9:

- *Flat:* A flat graph of the design is constructed with nodes representing the design components and edges representing the interconnections between the components.
- *Annotate*: The design is simulated in order to generate a graph annotated with the cost metrics of the design components.
- *Merge*: The user can select nodes and manually merge them together.
- *Grouping*: The Grouping algorithm runs in order to group closely related nodes.
- *Mapping*: The Mapping algorithm is executed in order to allocate the nodes into the available hardware and software entities.

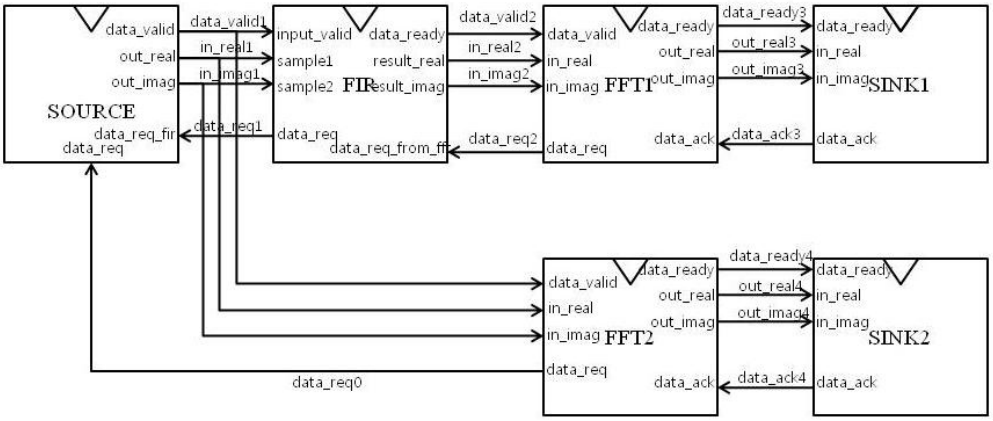

Figure 6.8. Block Diagram of digital filter

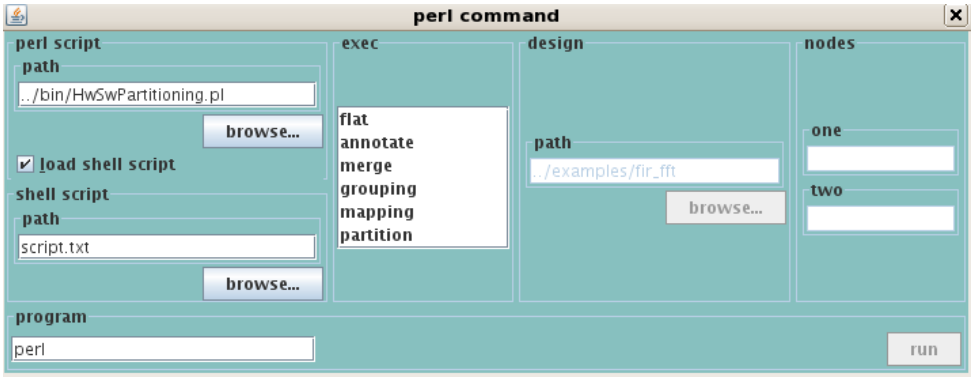

Figure 6.9. User commands supported by the tool

In Figure 6.10, the designer has manually merged SOURCE1 node with FFT2 node (green nodes) and FIR1 node with FFT1 node (teal nodes).

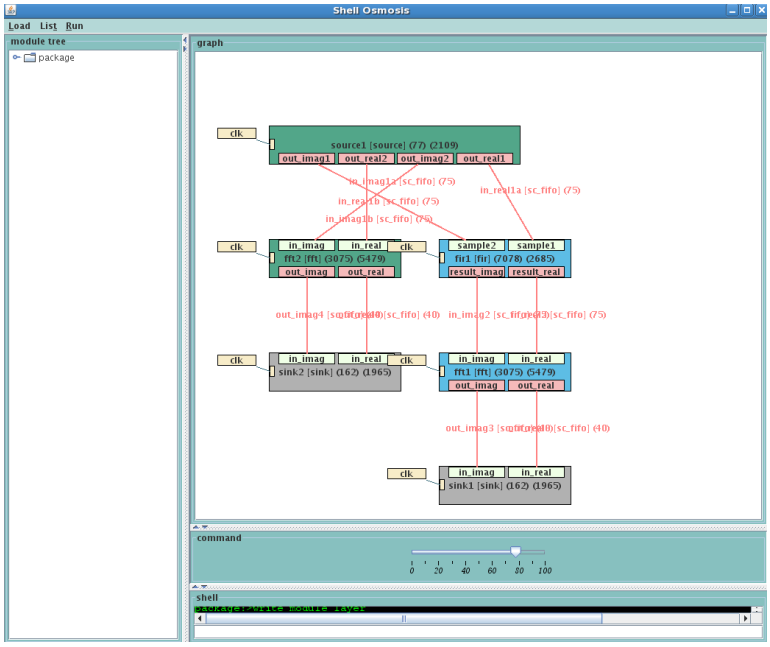

Figure 6.10. Manual merging of design components

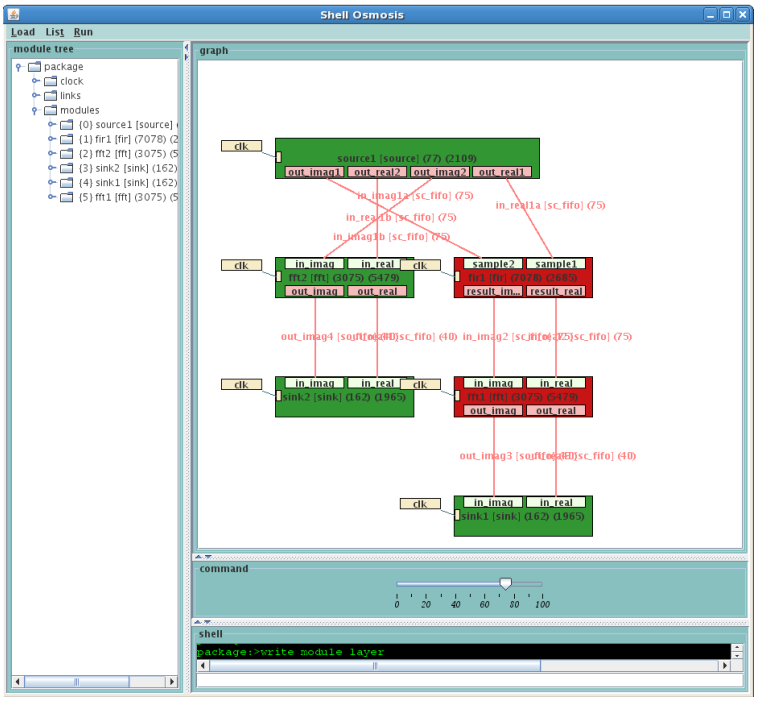

Figure 6.11. Partitioned design where green nodes are assigned to the SW entity and red nodes to the HW entity

Figure 6.11 depicts an example partitioning of the digital filter. The green nodes are assigned to the software entity while the red nodes are assigned to the hardware entity.

6.7 Second Test Case

In order to further verify the functionality of the HW/SW partitioning tool we applied it on the SystemC description of Mephisto CPU, which has been developed by CEA. The CPU was used for 3GPP telecommunication. The design consists of 18 components. The block diagram of the Mephisto's design is shown in Figure 6.12 and the hierarchy of the design components is illustrated in Figure 6.13.

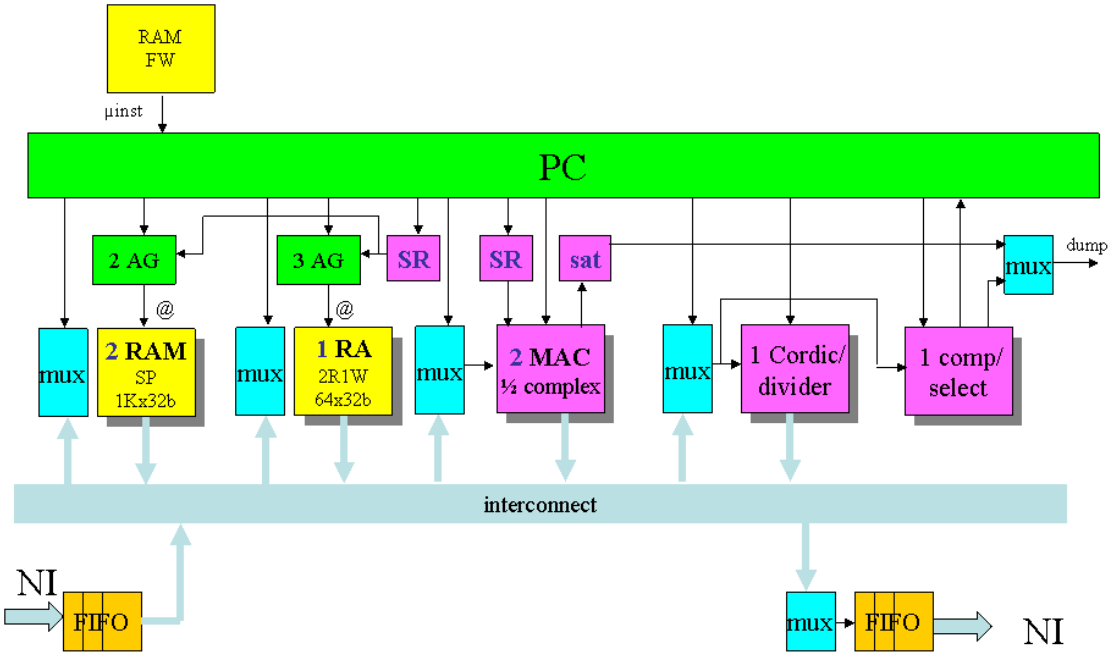

Figure 6.12. Block Diagram of Mephisto

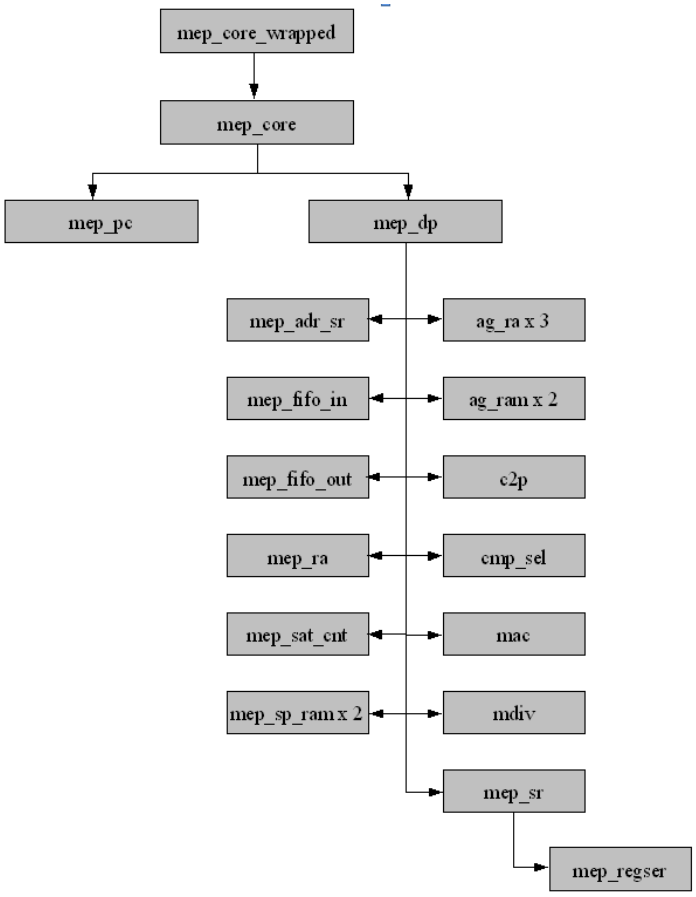

Figure 6.13. Design Components of Mephisto

In order to parse and partition this real word design the partitioning tool had to support more complicated SystemC declarations. First, all the SystemC files of the design are parsed in order to derive the design components and their interconnections. In this step, the tool finds all SystemC modules, instantiations and inter-block signals. A flat graph of the design is constructed, with nodes representing the design components, and edges representing their communication channels. A snapshot of the GUI is given in Figure 6.14, where the design is depicted as a flat graph as well as a tree-like structure.

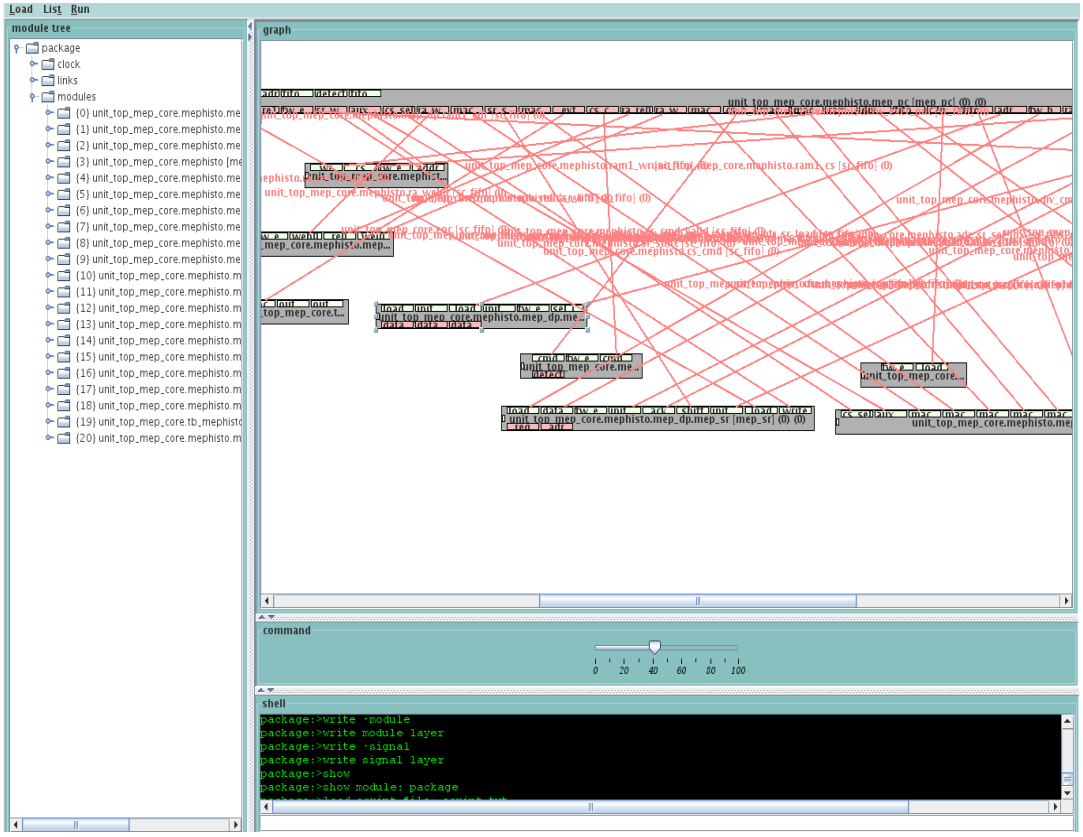

Figure 6.14. Flat graph of Mephisto depicted in the GUI of the Partitioning Tool

The tool transforms the original code of the design so as to trace the transactions at the components' boundaries and the number of executed instructions. System simulations are performed in order to generate a detailed trace file with all the bus transactions, the executed instructions and the clock transactions. Using the trace file the tool can automatically calculate the MTPS values of the interconnections and the MIPS values of the components. In order to take into account performance bursts (i.e. short periods that many instructions are executed), the MTPS and MIPS calculations are performed in successive short time periods and the maximum values are selected.

The table below shows the measured MIPS and Size values of the Mephisto's design components that were derived from the trace file.

Component	Total MIPS	Size (instruction count)
mep_sat_cnt	300	77594
mep_regser	1	78562
ag_ra0	198	86042
mep_sp_ram0	97	79365
mep_sp_ram1	97	79365
mep_fifo_out	500	90213
mep_ra	310	86228
cmp_sel	138	91560
mep_fifo_in	526	86073
ag_ra1	197	86042
ag_ra2	210	86042
mep_adr_sr	1	89882
c2p	141	81873
Mdiv	1	90099
mep_sr	300	92812
ag_ram0	1	89019
ag_ram1	1	89019
mep_dp	548	185156

The cost metrics are annotated on the graph as shows in Figure 6.15. The MIPS and total MTPS values of the design component are annotated on the nodes while the MTPS values of the communication paths are annotated on the edges.

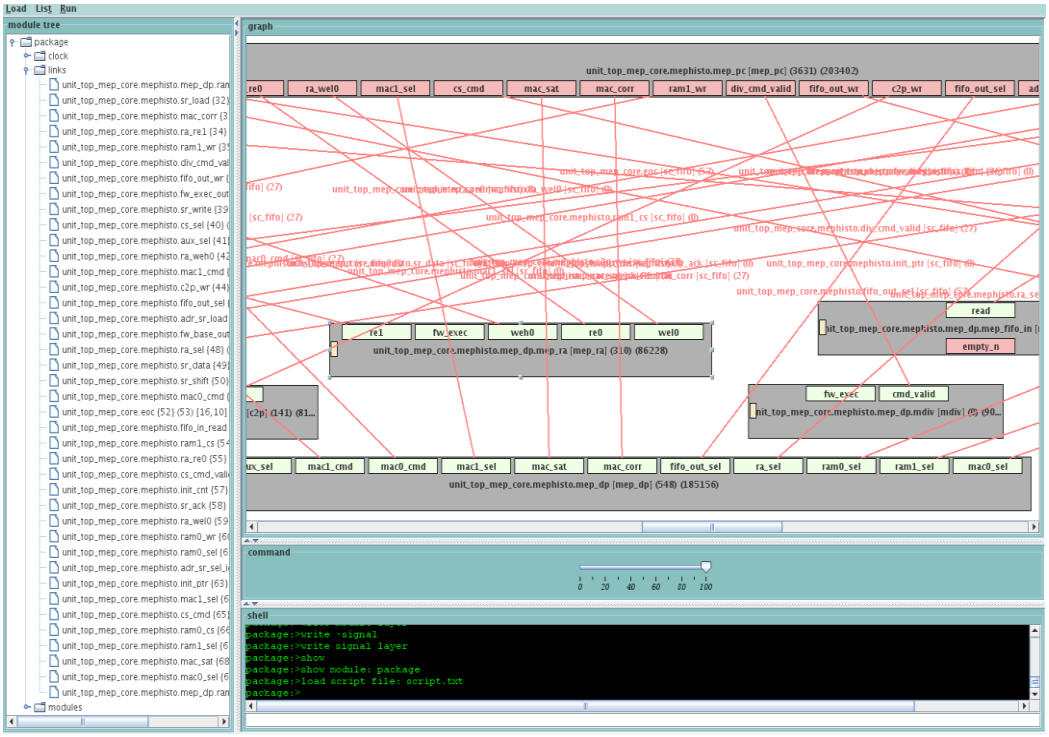

Figure 6.15. Annotated graph of Mephisto

The partitioning of the design is based on the GMP (Grouping Mapping Partitioning) we have developed and consists of two steps: The Grouping algorithm and the Mapping algorithm. The Grouping algorithm is performed where the graph is processed so as to produce a graph in a more compact and manageable form. Each node in the new compact graph represents a group of highly-related design components. The threshold values of the Grouping algorithm are set in the configuration file of the tool. The user defines the high *MTPS threshold* and the low and high *group-size thresholds*. These thresholds determine the level of *aggressiveness* (i.e. the grouping tendency) of the Grouping algorithm as described in Section 6.4.1.

Figure 6.17 shows the results after the Grouping algorithm. As shown nodes *mep_pc* and *mep_dp* have been merged together because there are many communication paths between them.

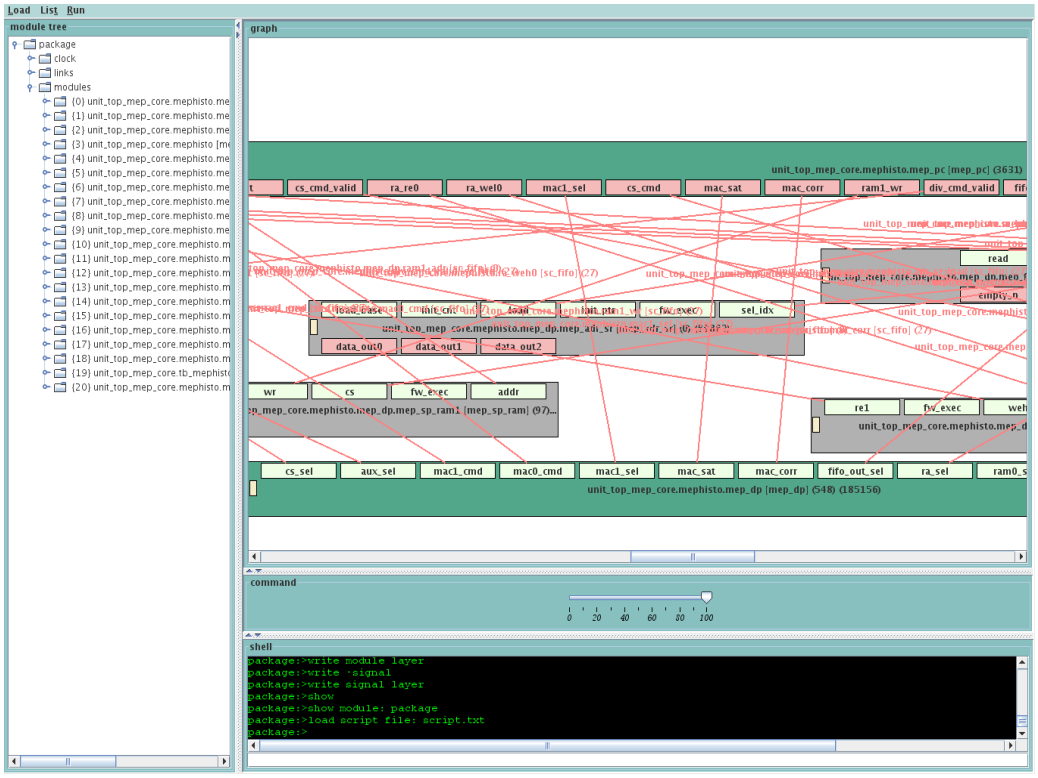

Figure 6.16. Colored nodes have been merged together after the Grouping algorithm is applied

Next the Mapping algorithm is applied where the graph is gradually splitted into two distinct parts: one part contains the nodes with the components that should be executed in software and the other contains the components that should be implemented in hardware. In the example of

Figure 6.17 we used a software entity that supports 1000 MIPS and 400 MTPS on its external bus.

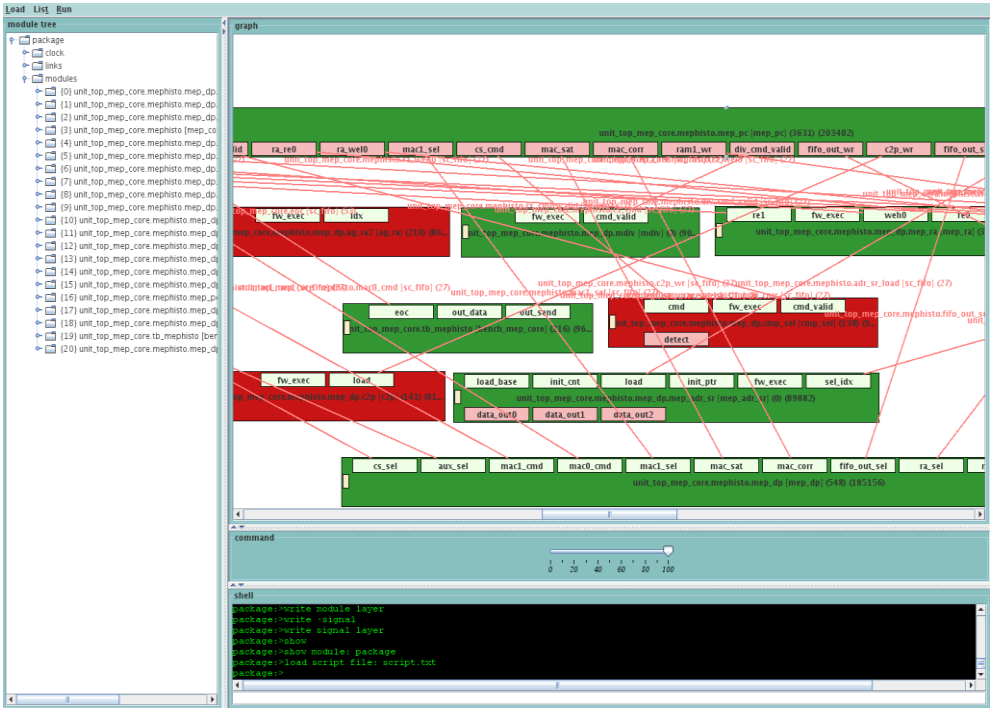

Figure 6.17. HW/SW partitioning of Mephisto where red nodes should be implemented in HW and green nodes should be executed in SW.

In order to verify the functionality of our partitioning tool, the tool has been extensively used in partitioning the digital filter and the Memphisto CPU using different software entities and partitioning parameters. Tweaking the thresholds of the Grouping algorithm we could derive the optimum partitioning results. These designs are small in order to extensively evaluate the tool and therefore a very large set of possible graphs were used in Section 7.3.

6.8 Summary

Although Hardware/Software partitioning has been extensively studied in the last twenty years, there are still many important open issues. At the same time, the partitioning task is very important for current SoC designers since today virtually all such systems consist of a number of CPUs as well as dedicated hardware modules. Moreover, performing fast and efficient hardware/software partitioning is especially important in RC where the partitioning algorithm

can be applied several times at run time. In order to address some of the open issues we first tried to provide a better understanding of the partitioning problem and then we presented an open-source tool that provides a complete and efficient solution.

The developed tool utilizes new cost metrics and supports two separate partitioning algorithms. The tool first constructs a unified performance graph of the embedded system where the sizes and instructions per second of the nodes, as well as the transactions of the interconnections between the nodes, are annotated. Next, our innovative GMP algorithm processes the graph, groups highly-related nodes together and indicates which should be implemented in hardware and which should be executed on the embedded CPUs. The main advantage of our GMP-based approach is that it provides cost-efficient solutions in a much faster way than traditional partitioning algorithms/tools. Alternatively, our tool employs also an implementation of the SA algorithm which is considered to derive very efficient partitioning results.

We strongly believe that since this is the first known such open-source framework it can provide a concrete base of a family of more advanced partitioning design tools that employ various cost and capacity metrics and accurate cost functions.

Chapter 7. Performance Analysis and Evaluation

"Man is a slow, sloppy and brilliant thinker; the machine is fast, accurate and stupid."

William M. Kelly

This Chapter provides a detailed evaluation of the proposed methodologies and platforms. We compare our approaches against existing methodologies and provide a performance analysis based on real world test scenarios. First, we analyze the hardware emulator (Section 7.1) and the scan chain methodology (Section 7.2) and finally we evaluate the hardware/software partitioning algorithm (Section 7.3).

7.1 Hardware Emulator

In order to evaluate the tool and quantify the proposed methodology we created a typical hardware emulator system in which we applied the proposed framework.

7.1.1 Evaluation board

We used the Xilinx University Program (XUP) Virtex-II Pro Development System from Xilinx which supports a Virtex-2P-30K FPGA, a widely used FPGA. The Xilinx Virtex-II Pro family of devices incorporates small yet powerful PowerPC 405 processor hard cores and supports Microblaze processor soft cores. Xilinx uses CoreConnect as the bus infrastructure for

all of their embedded processor designs; the CoreConnect is a microprocessor bus-architecture from IBM for SoC designs and it is used extensively in their PowerPC-based designs.

Advanced features of the Virtex-II Pro FPGAs include powerful system connectivity solutions, digitally controlled impedance (DCI) technology, comprehensive clocking solutions, high-speed Active Interconnect routing architecture, and bitstream encryption. Figure 7.1 shows the Xilinx evaluation board which we used for our Hardware Emulator platform.

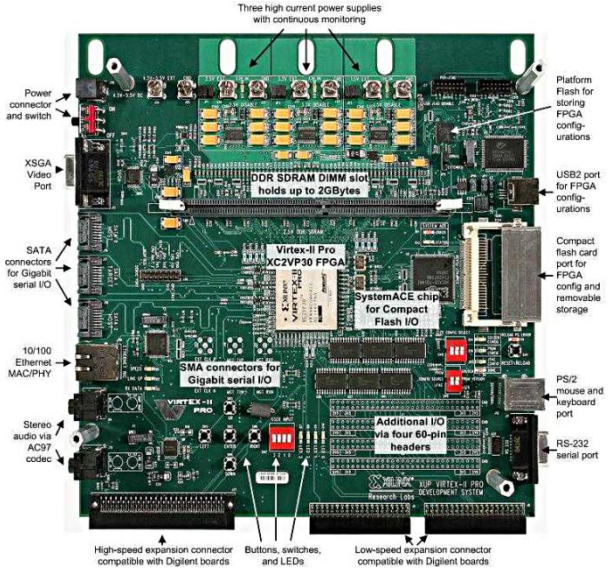

Figure 7.1. XUP Virtex-II Pro Development board

The XUP Virtex-II Pro Development System provides an advanced hardware platform that consists of a high performance Virtex-II Pro Platform FPGA surrounded by a comprehensive collection of peripheral components that can be used to create a complex system and to demonstrate the capability of the Virtex-II Pro Platform FPGA. Figure 7.2 shows a block diagram of the XUP Virtex-II Pro Development System.

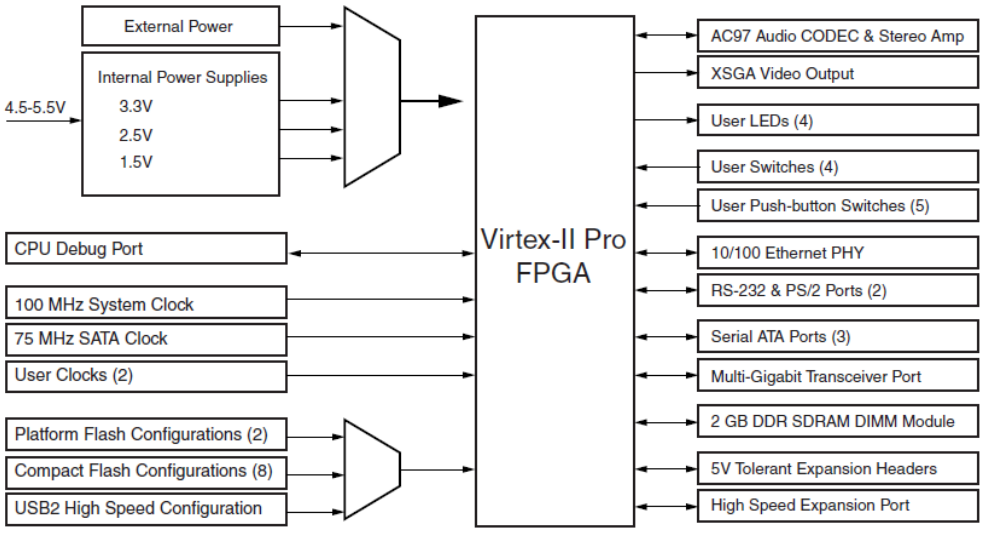

Figure 7.2. XUP block diagram

We have tested our proposed methodology against several real-world DUTs and testbenches, and always derived significant speed-ups compared to conventional emulation systems and came to the same final conclusions. The speed-up depends directly on the number of DUT-testbench signals. In Section 7.1.2 we compare the proposed emulator with a conventional emulation scheme, and in Section 7.1.3 we analyze a test scenario that we have used.

7.1.2 Performance Evaluation

In this Section, "frequency" of an event is defined as the number of times this event occurs over the total simulated clock cycles.

In a transaction-based conventional emulator, such as the ones described in Section 3.1 that follow the SCE-MI standard, the FPGA that emulates the DUT and the transactors, communicates with the external CPU that runs the proxies and software portion of the testbench, via a fast off-chip bus. The average time to simulate a clock cycle is given by the following formula:

$$(1)\ SimulationCycle_{time} = EmulatorCycle_{time} + TRANSACTION_{frequency} \times (BUS_{time} + CPU_{time})$$

Where:

- "$EmulatorCycle_{time}$" is the time required by the FPGA to emulate one clock cycle of the DUT.

- "$TRANSACTION_{frequency}$" denotes the frequency of the transactions.
- "BUS_{time}" denotes the average bus time to execute the SCE-MI protocol and send the transaction and its data from the FPGA to the CPU and vice-versa.
- "CPU_{time}" denotes the average CPU time consumed in the host workstation to process a transaction.

In our case, the average time to simulate a clock cycle is given by the following formula:

$$(2)\quad SimulationCycle_{time}' = EmulatorCycle_{time}' + PLI_{frequency} \times (BUS_{time}' + CPU_{time}') + MEM_{frequency} \times MEM_{time} + FP_{frequency} \times FP_{time}$$

Where:

- "$EmulatorCycle_{time}'$" is the time required by the FPGA to emulate one clock cycle of the DUT.
- "$PLI_{frequency}$", "$MEM_{frequency}$" and "$FP_{frequency}$" denote the frequencies of the PLI calls, the memory references and the floating point operations, respectively.
- "BUS_{time}'" denotes the average bus time to execute the bus protocol (e.g. IBM's CoreConnect) and send the PLI call or results from the synthesized testbench to the embedded CPU and vice-versa.
- "CPU_{time}'" denotes the average time consumed by the embedded CPU for processing a PLI call.
- "MEM_{time}" denotes the access time of the external memory.
- "FP_{time}" denotes the execution time of a floating point operation by the embedded FPU.

Let us now compare these two formulas. One of the biggest advantages of our approach is that it will usually result in a much higher portion of the testbench than just the transactors, being converted into synthesizable HDL code and emulated on the FPGA together with the DUT. The result is that the emulation will need to halt for PLI calls or other external operations much less frequently than a conventional emulator will need to halt for a transaction. Thus, compared to a conventional emulator:

$$PLI_{frequency} + MEM_{frequency} + FP_{frequency} \leq TRANSACTIONS_{frequency}$$

Moreover, memory and floating-point operations are executed much faster in our approach since we do not need any software intervention for them, in contrast to a conventional emulator that performs these operations in software via corresponding off-chip transactions. Thus:

$MEM_{time} < BUS_{time} + Average\ CPU_{time}$

$FP_{time} < BUS_{time} + Average\ CPU_{time}$

Additionally, our approach utilizes on-chip high-bandwidth low-latency buses (running e.g. CoreConnect), in contrast to conventional emulators that usually utilize off-chip buses (running SCE-MI). Thus:

$BUS_{time}' < BUS_{time}$

Putting it all together, our approach turns out to be considerably faster than a conventional emulator.

7.1.3 Test Case

The DUT of our reference test scenario, for which we present detailed results in this Section, is a hardware module containing two memory controllers that make periodically pseudo-random accesses to a parameterized number of SRAM and DRAM chips. The testbench includes the memory models for these chips.

The memory models that we used are those of a 32-bit ZBT SRAM and a 16-bit SDRAM DDR, acquired from Micron Technology, Inc. [MTI]. Thus, even though it is a small-scale test case, it includes real-world, widely-used code, that has been developed by engineers in a large semiconductor company. We performed several tests by varying the number of memory chips that are instantiated.

The tool that we have developed was able to successfully transform the original testbench, including the memory models from Micron Technology, into synthesizable HDL code. The large memory arrays of the models are stored in the DDR memory of the XUP board that is accessed by the embedded PowerPC processor and the DDR memory controller. The whole system, including the transformed testbench and the DUT, was synthesized and simulated using the ISE 8.1 EDA software from Xilinx.

Performance Measurements

The FPGA which contains the DUT and the synthesized testbench runs at 125 MHz, with the critical path in the SDRAM model, as expected. An SRAM memory transaction, which involves a single request between the HDL testbench block and the server (Section 4.3), averaged 30 ticks in a large number of test runs. A SDRAM burst transaction, which involves 4 requests for a burst size equal to 4, took 140 ticks on average. The DUT performs one access to every SRAM chip and one to every SDRAM chip in parallel every 200 simulation cycles on average. Taking all this into account we can measure the simulated Cycles Per Second (CPS). The results are depicted in Figure 7.3.

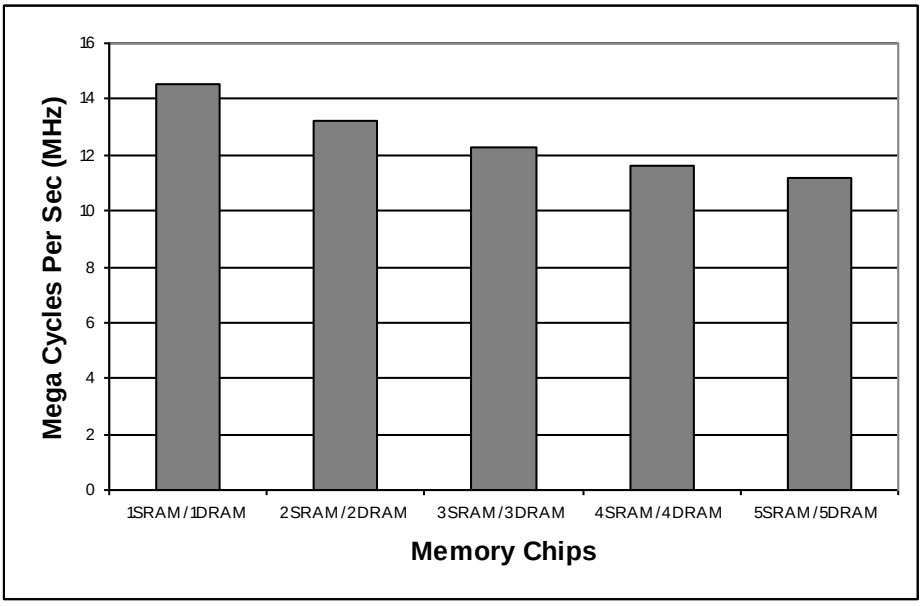

Figure 7.3. Simulation Speed.

In our tests we varied the number of SRAM and SDRAM chips in order to measure how this affects the simulation speed. As we can see in the Figure, the simulation speed remains largely unaffected by the number of memory chips. Increasing the number of memory chips will certainly result in a higher number of signals between the DUT and the testbench (90 signals per SRAM, and 48 signals per SDRAM). Since the testbench is synthesized and emulated on the same FPGA together with the DUT, the increased number of signals does not affect the FPGA clock cycle $\text{EmulatorCycle}_{\text{time}}'$, but the number of memory chips affects linearly the frequency of the memory accesses $\text{MEM}_{\text{frequency}}$. However, the server is able to access the external memory

where the long arrays of the memory models reside without any software intervention ($PLI_{frequency} = 0$) and consequently the emulation speed is slightly affected.

On the other hand, a conventional emulator, using transactors and message port proxies, needs to go off-chip several times for every memory access. The memory accesses need to be transferred from the FPGA emulator to the CPU simulating the software part of the testbench, executed in software that accesses an external memory, and have their results sent back via the SCE-MI bus. The resulting communication and execution overhead when compared to our approach can be as high as proportional to the number of memory chips.

As Figure 7.3 shows, our approach reaches a simulation speed of over 10M CPS for 5 SRAM and 5 SDRAM chips. In comparison, Palladium and VStation Pro report emulation speeds of 600K CPS, and Kim and Kyung [YC04] measure the speed of a conventional emulator and their improved emulation system between 38K CPS and 701K CPS which is about 15 times slower than our approach.

In order to further quantify our approach, we compared the simulation frequency of a conventional system against that achieved by our proposed framework. Following the transaction-based model, we assumed that the designer provides the transactors that implement the functionality of the SRAM and SDRAM models. Figure 7.4 shows the speed-up gained by our approach over a conventional emulator running our test case. We have measured that the conventional emulator requires an average of 250 cycles at 125 MHz per transaction (BUS_{time} + Average CPU_{time}).

These results are in favour of the conventional emulator since several delays incorporated in the communication, such as the latency of the off-chip communication link, that cannot be accurately measured or estimated, were assumed to be zero.

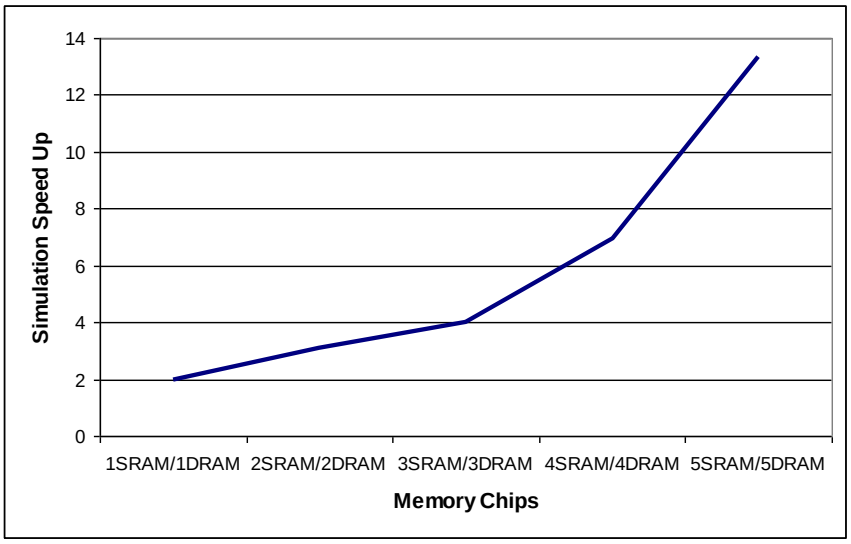

Figure 7.4. Comparison of the proposed architecture.

Finally, we wanted to compare our proposed architecture against a widely-used software simulator such as *modelsim* or *vcs*. To this end, we used Intel's VTune [INV] to analyze a number of simulations running on *modelsim*. The main drawback of a software simulator is that the simulation speed is directly affected by the size of the DUT, in contrast to a hardware emulator where the DUT is synthesized and the speed is not as severely affected by its size. VTune shows millions of instructions being executed even for small simulations, while OS device drivers consume a significant percentage of the simulation time. Our approach operates at 1000 to 10000 times faster than *modelsim* depending on the actual complexity of the DUT**.**

7.2 Embedded Logic Analyzer

In order to evaluate and quantify the proposed ELA (see Section 5.4) we used the emulation environment described in Section 7.1 where we applied the scan chain methodology. The hardware emulator uses the XUP Virtex-II Pro Development System from Xilinx, described in detail in Section 7.1.1. This evaluation environment supports an embedded PowerPC which we used in order to configure and monitor the status of the ELA.

In order to measure the performance and evaluate the scan chain technique and the ELA, we applied our methodology on the code of the TDM line card described in Section 5.5.1, which connects a backbone network with hundreds of clients. The DUT includes 786 FFs partitioned into 5 long scan chains by the scan chain tool (Section 5.3). The final conclusions from the

measurements are independent of the specific DUT we used. The system which includes the testbench block, the DUT and the ELA was synthesized using the Xilinx ISE 7.1 tool.

7.2.1 DUT Scan Circuitry Evaluation

The area overhead of the scan circuitry is an important issue. [SM97] claims that a D Flip Flop instrumented for scan is only 10% larger than the original one and adding scan logic in VLSI requires a 5% to 30% area overhead. This, unfortunately, is not the case in FPGAs (see also [WG01]), where instrumenting a FF for scan effectively doubles its size since the FF and the scan mux have the same size (each block covers half of the Logic Element (LE)). The size may even triple or quadruple by using additional LEs for the clock enable and set/reset scan logic. In our evaluation environment the DUT with the multiple scan chains occupies around 90% more area than the original design as our measurements for different DUTs demonstrated. If the future FPGAs internally support scan structures, such as FreedomChip, this area overhead will obviously be drastically reduced.

Moreover, in [WG01] the authors claim that adding scan logic, on average, reduces the speed of the circuit by 20%. The measured speed reduction in our experiments was slightly lower but in general this is not a significant issue for a hardware emulator; such systems perform functional verification by emulating the circuits of the user design, orders of magnitude slower than reality (i.e. the final real hardware implementation).

7.2.2 Logic Analyzer Evaluation

The scan period is important when the ELA is activated during the complete hardware run, even before the trigger condition is true, in order to capture and test the trigger signals. The length of the longest scan chain in the DUT affects directly the scan period of the ELA. By increasing the number of scan chains generated in the DUT, the length of the longest scan chain diminishes. This is depicted in Figure 7.5 for our example design.

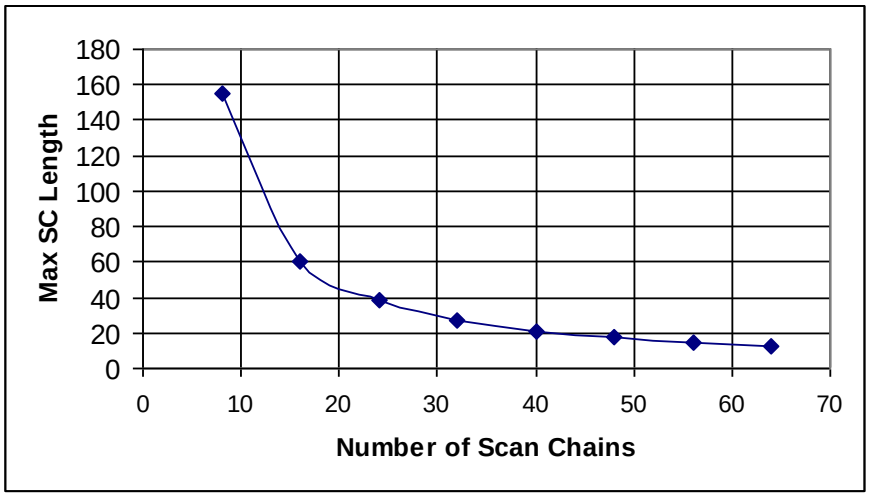

Figure 7.5. Length of longest scan chain

Figure 7.5 shows that the length of the longest scan chain decreases slightly when the number of scan chains is greater than 25. This is obvious since the equation that gives the length of the longest scan chain when the scan chains are balanced (see Section 5.3) is:

$$(1)\ Scan\ Chain\ Length_{max} = \lceil Number\ of\ Registers_{Total} / Number\ of\ Scan\ Chains \rceil$$

The number of the scan chains supported by the ELA is parameterized in the VHDL code. This number affects the size of the ELA linearly, as shown in Figure 7.6. Therefore, it is important to keep it as small as possible. We used the Xilinx ISE synthesis tool in order to estimate the size of the ELA. As shown in Figure 5.7 the logic blocks as well as the length of the internal signals in the ELA are directly affected by the number of the scan chains. Using the Xilinx Virtex-2P-30K FPGA, an ELA that supports 32 scan chains occupies 13% of the FPGA area, while an ELA that supports 64 scan chains occupies 20% of the FPGA area.

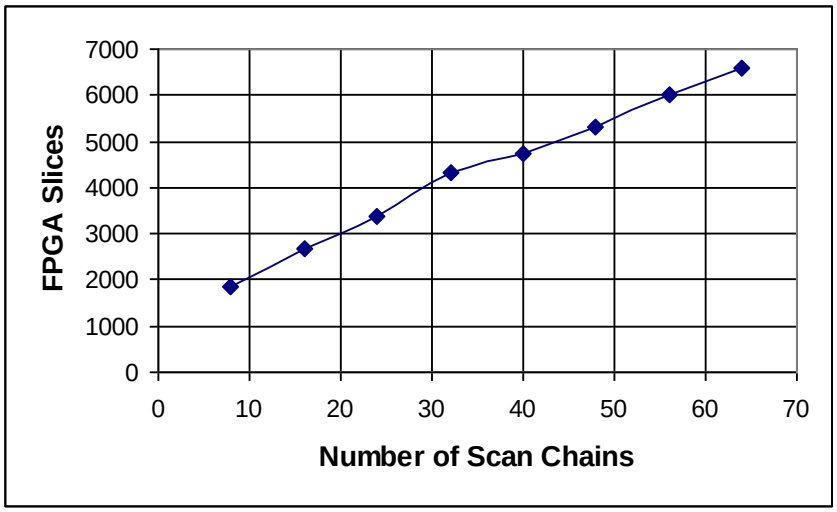

Figure 7.6. Area of ELA

Figure 7.7 shows the clock frequency as a function of the number of the scan chains supported by the ELA. These numbers were derived from the Xilinx ISE synthesis tool. As the number of scan chains increases the area of the ELA increases (Figure 7.6); this means that the internal logic as well as the length of the internal wires increases also. This results in a small clock frequency drop as shown in the figure below.

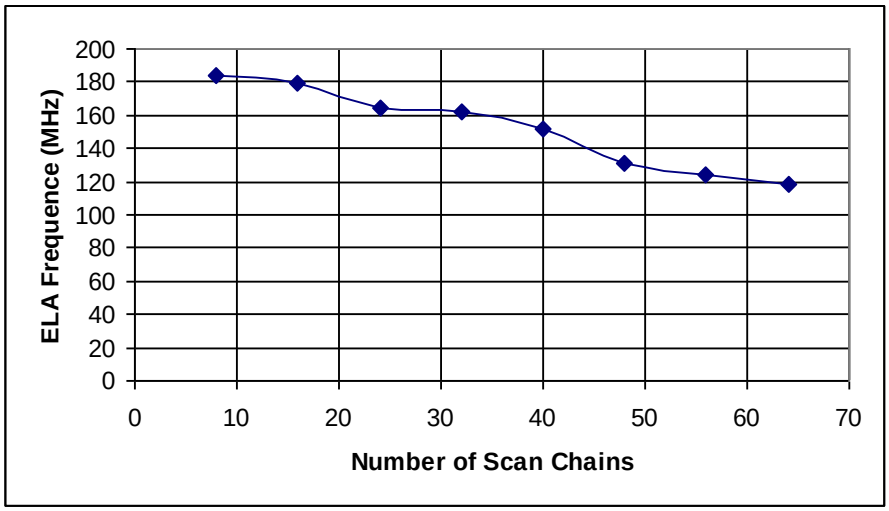

Figure 7.7. Frequency of ELA

The length of the scan period is an important measurement that characterizes the performance of the proposed methodology. This number depends on the length of the longest scan chain in the DUT (or the number of the scan chains), the number of the signals to be traced,

and the speed of the ELA. Figure 7.8 shows the scan period as a function of the number of the scan chains in the evaluation environment, when we trace the values of 15 signals/busses.

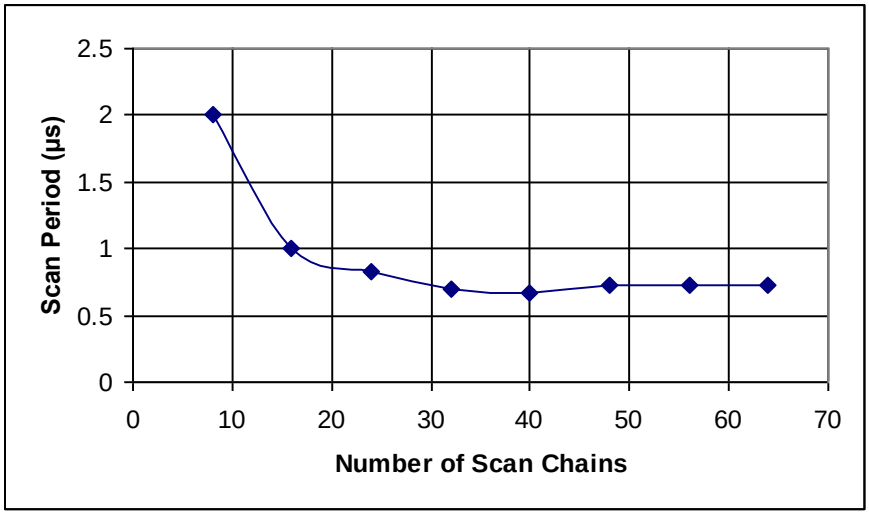

Figure 7.8. Scan Period

Figure 7.8 shows that the scan period is almost fixed when the number of scan chains is greater than 25. The reason is that the length of the longest scan chain slightly decreases (Figure 7.5) while in parallel the speed of the ELA drops (Figure 7.7) and the clock cycle of the ELA increases. The length of the scan period is roughly proportional to the length of the longest scan chain and to the clock cycle of the ELA (there are also some overhead cycles due to the captured signals and the size of the captured buffer) as described in the following equation:

$$(2)\ Scan\ Period_{length} \approx Scan\ Chain\ Length_{max} \times ELA_{clock\ cycle}$$

The aforementioned plots demonstrate that around 25 scan chains is the optimum solution, while implementing more scan chains results in increasing the area of the ELA without succeeding smaller scan period. This conclusion is independent of the DUT we used since the DUT size (Number of $Registers_{Total}$) affects linearly the measurements of Figure 7.5 and Figure 7.8 as derived from equations (1) and (2).

7.2.3 Evaluation of the Trigger Condition

The ELA must be activated before the trigger condition is true in order to avoid recompiling the design when the trigger signals change (see Section 5.4.1). In our evaluation environment,

the ELA can access all the scan chains of the DUT (without capturing data) in 145 ns. The simulation cycle of the emulated design in Section 7.1.3 is 32 ns, where the critical path is in the testbench code and not in the DUT. Assuming that we activate the ELA once in every cycle, the new simulation cycle will be 180 ns or about 5 times longer. This means that the simulation will be about 5 times slower for a design of 786 FFs.

Extrapolating from these results, Figure 7.9 shows how much slower the simulation is when the ELA is activated in every simulation cycle as a function of the number of registers in the DUT. The number of registers in the scan chains affects the emulation speed linearly. This is because the scan period is proportional to the length of the longest scan chain which is proportional to the number of registers in the DUT as derived from equation (1) in Section 7.2.2.

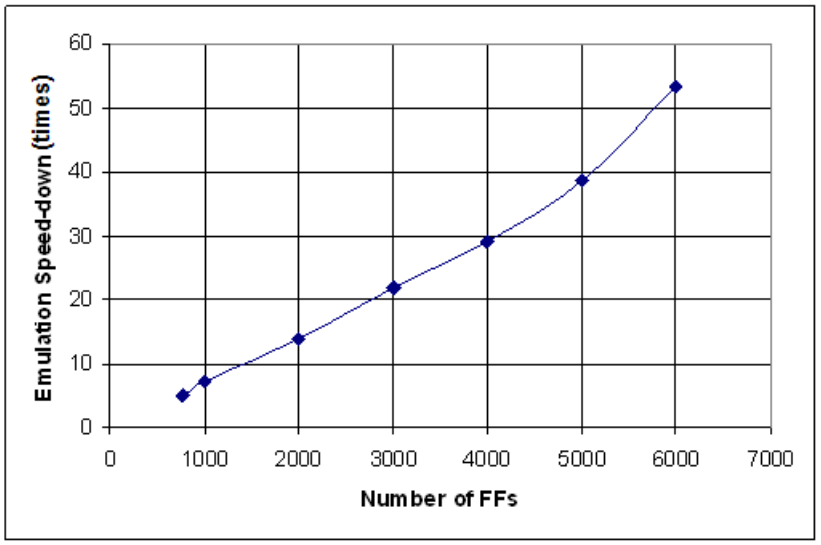

Figure 7.9. Speed Degradation

In order to decide whether the design should be recompiled or use the ELA (no recompilation) instead, the designer should consider how much time it takes to compile the design, when the trigger condition becomes true and how often the ELA should evaluate the trigger condition. For example, if we want to emulate a big design with 5K registers and the trigger condition becomes true in 10 seconds when we emulate its behavior without using scan chains, then it takes 10 secs x 40 = 6.6 minutes with the scan chains enabled. Recompiling (synthesis, place and route) a design with 5K registers will probably take more than 6.6 minutes.

7.3 Hardware/Software Partitioning

Firstly, we evaluated our approach when implementing the three real world designs described in Sections 6.5, 6.6 and 6.7 . However, in all three cases our toolset ended up with the optimal solution so we could not get, from those embedded systems, any additional information (except of the fact that it works very well for those designs) regarding our novel scheme's characteristics.

In order to demonstrate the efficiency of our approach, the two stages of the partitioning algorithm were extensively analyzed and evaluated using a very large set of possible graphs. In order to create this large set of graphs we varied the following parameters: (a) the number of nodes, (b) the average input and output degrees of the nodes, (c) the MIPS, (d) the MTPS and (e) the Size. We used both random graphs and geometric graphs since they can represent complex embedded systems [YC95]. Random geometric graphs result from taking uniformly distributed points in a cube and connecting two points if their Euclidean distance is less than a prescribed distance.

The example platform to which we mapped these designs utilizes different embedded CPUs as their software system entities, while their MIPS were derived from the Dhrystone performance results [DPR]. We also used the specifications of these CPUs to derive the MTPS values of their external busses. Additionally, for simplicity reasons, we assumed a 5% OS overhead in all CPUs, which seems realistic for computationally intensive workloads according to [RE00]. The hardware system entities of our example systems need not be simulated since our algorithm is evaluated in terms of how many design components it is able to fit into the software system entities; we assume that the more design components we are able to map into software, the better the results, as it also supported in many different papers in this area.

In order to evaluate the results and the effectiveness of our partitioning algorithm, we compare them against the optimum solution which is derived through an exhaustive search of the complete design space. We define as the *software percentage* of any partitioning the total Size of the threads implemented in software over the total Size of all the threads. We then use the *software percentage difference* between the partitioning that our algorithm triggers and the optimum partitioning, in order to accurately evaluate the effectiveness of our algorithm.

Figure 7.10 shows this percentage difference of the Mapping algorithm (see Section 6.4.1) for random geometric graphs with various average MTPS values on their arcs. The exhaustive

search scheme, which is used as a reference of the optimal solution, is prohibitively slow to partition graphs with more than 30 nodes so we demonstrate our results for up to 30 nodes. A SW entity that sustains 500 MIPS and 40 MTPS was used and the ranges of the node Sizes, MIPS and MTPS values were selected in such a way that the optimum solution partitions the system roughly in the middle. Each point in the graph is the average result from multiple test cases.

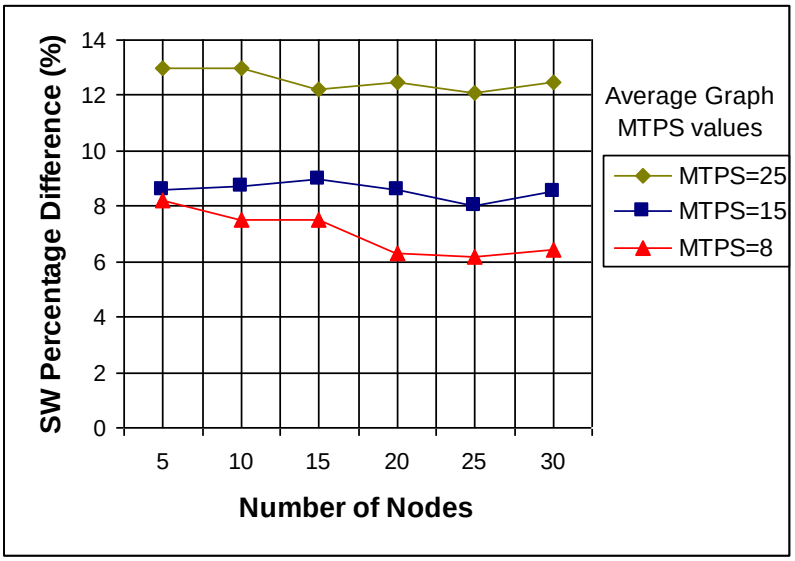

Figure 7.10. Evaluation of Mapping Algorithm

While the Mapping algorithm seems to operate efficiently in most graphs, we realized that the results deviate significantly from the optimum ones when the MTPS values of the design components are getting closer to the sustainable MTPS of the software entity, as also shown in Figure 7.10 for the case of MTPS=25. The reason is that the Mapping algorithm is not flexible enough to select, in a single step, nodes connected with arcs that have high MTPS values. For instance, assuming that in the example graph of Figure 7.11 the Mapping algorithm has already selected node N1 as the first node of the sub-graph, the best alternative is to select nodes N3 and N4 together. However, it will select node N2 because the arc between nodes N3 and N4 has a high MTPS value and as a result the F_{SW} metrics of nodes N3 and N4 are high. In order to solve this problem, we can set an upper threshold to the MTPS values in the graph, at the Grouping stage before the Mapping stage, merging nodes interconnected with high MTPS values, such as N3 and N4.

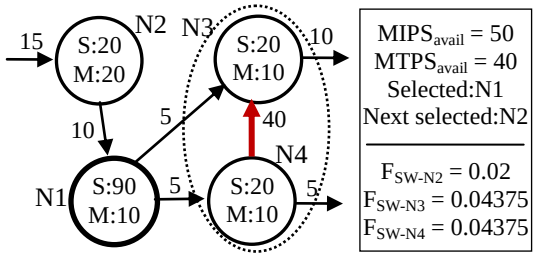

Figure 7.11. Problem with Mapping Algorithm

Figure 7.12 shows the results from three test cases when both Grouping and Mapping algorithms are applied. The MTPS values of the software entities that were used are 40, 70 and 100 while the average MTPS values of the arcs in the graphs are 25, 40 and 60 respectively. In the first test case ($MTPS_{entity}$=40 and $MTPS_{graph}$=25) when the MTPS threshold of the Grouping algorithm ranges between 20 and 30, groups with closely related nodes are formed providing better alternatives to the Mapping algorithm. When the MTPS threshold drops below 20, over-grouping deteriorates the partitioning results. When the MTPS threshold is above 30 the Grouping algorithm is essentially inactive. We derived similar results from all the test cases.

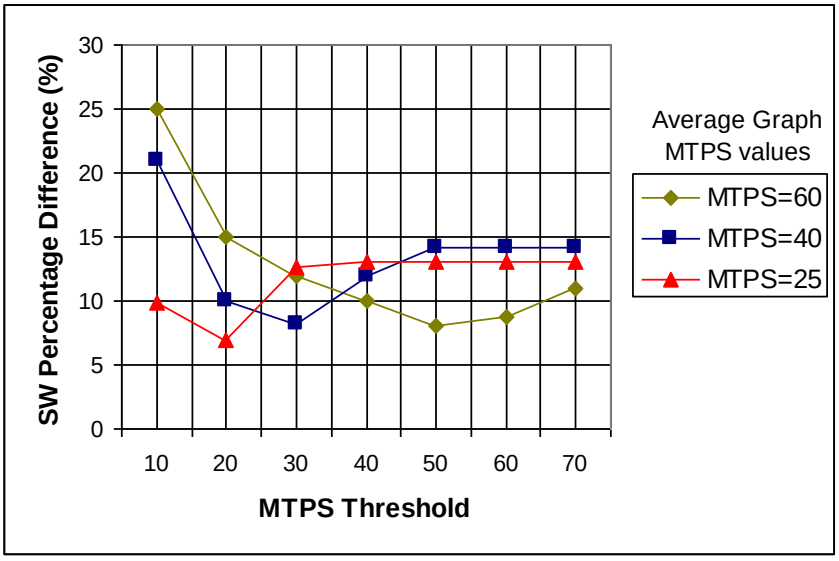

Figure 7.12. Problem tackled with Grouping Algorithm

If the total MTPS value of the interconnections between two nodes is larger than the MTPS value of the software entity, the bus of the software entity cannot sustain the transactions between the two nodes and therefore the Grouping algorithm should group such nodes together. By analyzing more test cases we concluded that an MTPS threshold close to half of the MTPS

value of the software entity derives the optimum solutions. So if such a threshold is utilized our combined Grouping and Mapping algorithm reports partitioning results that differ by less than 10% from the optimal ones which are denoted by an exhaustive search of the complete search-space.

To further evaluate our novel GMP algorithm we compare it against the SA algorithm, which, as it is demonstrated in [WC02], produces efficient results. Figure 7.13 shows the *software percentage* difference between the SA algorithm and the GMP algorithm using random geometric graphs with various sizes. Each point in the graph is the average result from multiple test cases. The software partitioning percentage of the SA algorithm is about 2.5% better than the GMP algorithm which shows that the GMP algorithm provides very efficient results. In several test cases the GMP algorithm and the SA algorithm resulted in exactly the same partitioning. Moreover, Figure 7.13 shows that the GMP algorithm derives efficient partitioning solutions for any graph size.

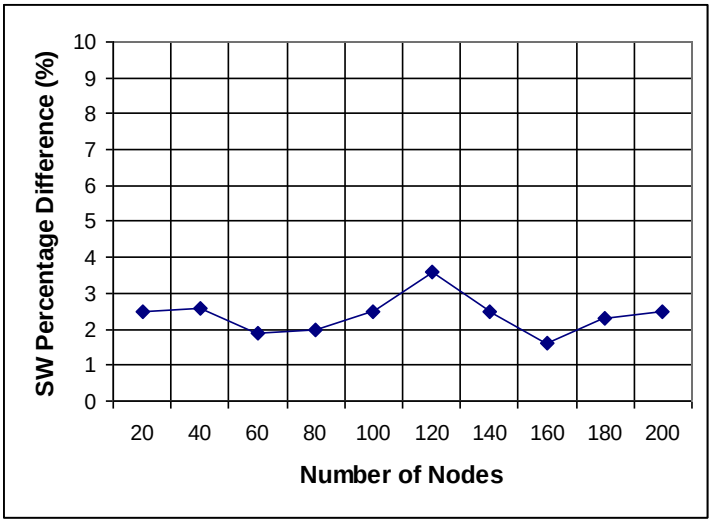

Figure 7.13. Comparison between GMP and SA

Figure 7.14 shows the time consumed by the two algorithms. Notice that the plots use different time units. A Pentium 4 at 2.8Hz was used for our measurements.

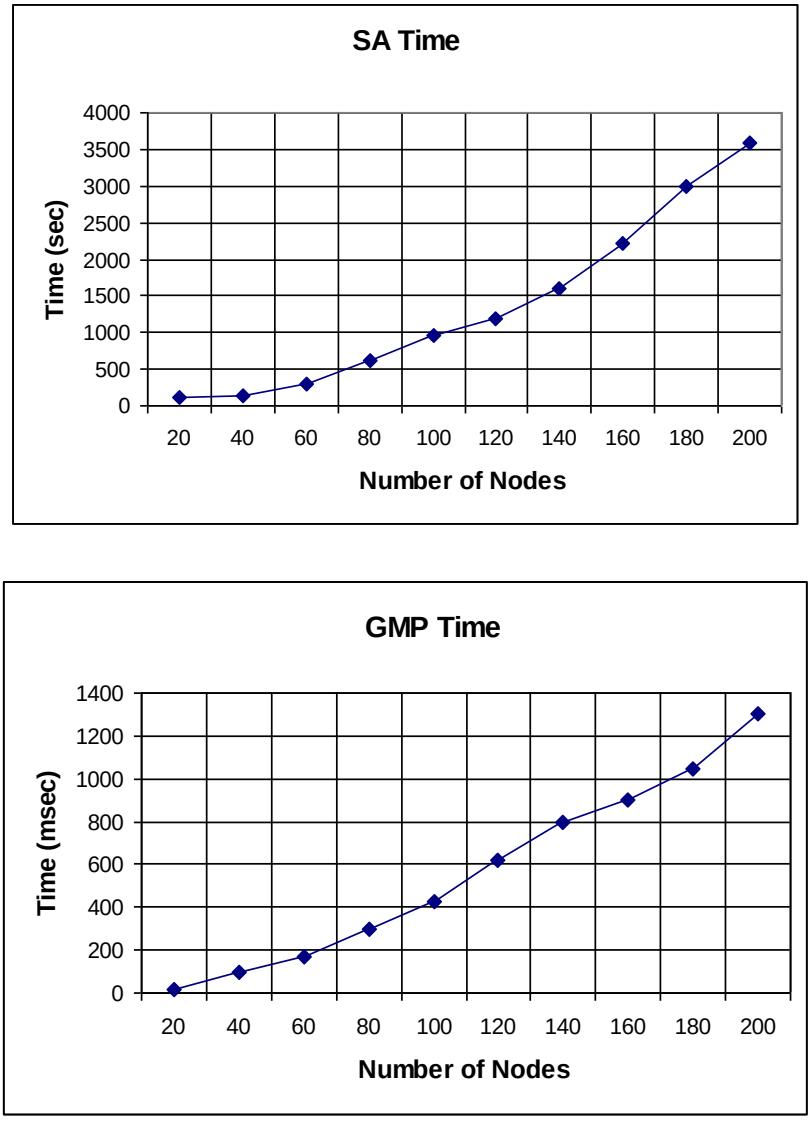

Figure 7.14. Time of SA and GMP algorithms

As those results clearly demonstrate the SA algorithm is about 2500 times slower than our GMP. The GMP greedy algorithm is very fast since the time complexity of the algorithm is O(N). On the other hand, the more time we give to the SA by adjusting its parameters, the more efficient results we derive. The effectiveness as well as the time of the SA algorithm depend on the cooling ratio parameter, *a*. This parameter should be close to 1 in order to derive efficient results prolonging in this way the cooling process.

So, while the SA algorithm supported by our tool derives slightly better results than the GMP algorithm, the SA algorithm is more than three orders of magnitude slower than the GMP algorithm making it impractical for large designs.

7.4 Summary

We analyzed the proposed methodologies through simulations of random and real world test cases. In summary, the evaluation results we derived are the following:

- We could overcome the testbench-DUT communication bottleneck of existing emulators and therefore increase the capabilities of today's hardware emulators by up to 1500% when applied to real-world systems.
- We managed to provide full chip observability and controllability using the scan chain methodology. Our measurements showed that using between 15 and 25 scan chains offers the best tradeoffs in terms of performance and area.
- We proposed the GMP hardware/software partitioning algorithm that provides cost-efficient solutions in a much faster way than traditional partitioning algorithms/tools. In particular the algorithm is about 2500 faster than the SA while the results of the GMP are always less than 3% worse than those triggered by the SA.

Chapter 8. Conclusions and Future Directions

"The real danger is not that computers will begin to think like men, but that men will begin to think like computers."
Sidney J. Harris

Due to the complexity involved in today's embedded systems, designing them requires a lot of man power and the support of advanced CAD tools. Over the years, several algorithms, methodologies and platforms have been developed in order to speed up this process; hardware simulation accelerators, hardware emulators and automatic hardware/software partitioning are some of them. However, there are still several limitations in all the proposed approaches, such as:

- Complex systems demand high communication throughput between the devices running the testbench and those emulating the synthesizable DUT. This communication can easily become the bottleneck and eventually limit the performance of the hardware emulation. Most of the proposed solutions for reducing the communication require manually rewriting the testbench in a different language/manner.
- Circuit observability of emulated systems is not efficient. Modification to the signals being captured or to the size of the data capture buffer often requires a time-consuming full recompilation of the design. In addition, no circuit controllability is provided.

- Although automatic hardware/software partitioning has been extensively studied in the last twenty years, there are still many important open issues, which is probably the reason why commercial CAD tools do not support this functionality yet. There is no standard and satisfactory methodology that can automatically split any design into design components and allocate them to the available hardware and software system entities in a cost-effective manner.

This dissertation described cost-efficient techniques to attack each of the aforementioned limitations. In particular we proposed the following:

- A methodology that greatly reduces or completely eliminates the aforementioned emulator bottleneck. The idea is to transform the part of the testbench that communicates the most with the DUT to synthesizable code, which allows us to emulate it close to the DUT, thus avoiding costly off-chip communication. In order to evaluate our methodology, we built a tool that provides a way to synthesize a behavioral VHDL code in a hardware simulation environment. Our real world experiment demonstrate that we can overcome the testbench-DUT communication bottleneck and therefore increase the capabilities of today's hardware emulators by up to 1500% when applied to real-world systems.

- Next, we attack the problem of circuit observability and controllability during emulation. Towards this end, we first introduce scan chains that can quickly access any register of the emulated DUT, and then add a novel Embedded Logic Analyzer, along with a software toolset. The resulting emulation environment is able to support full on-the-fly circuit observability and controllability. Our real-world experiments show that using between 15 and 25 scan chains offers the best tradeoffs in terms of performance and area.

- Finally, we propose a methodology to attack the hardware/software partitioning problem. A graph is constructed out of the design components, and novel cost metrics are used to annotate this graph. Based on these annotations, as well as on certain capacity metrics for the system entities, our novel GMP algorithm is able to group closely-related design components, and decide which should be implemented in hardware and which should be executed on the embedded CPUs. A large number of randomly generated realistic graphs are used to compare our algorithm against the widely used Simulated Annealing (SA) algorithm. Our experiments show that, even though SA produces slightly better results, it is more than

three orders of magnitude slower than our innovative approach making it impractical for large designs.

8.1 Future Directions

Regarding the hardware/software partitioning we plan to investigate a hybrid approach using both the GMP and the SA algorithms. The GMP algorithm will provide a fast solution to the SA, which will be farther optimized in order to derive a better partitioning. In this case, the cooling process of the SA algorithm will be faster than running the SA alone, since the SA will start from a reasonable system partitioning instead of a random one. The hybrid approach is shown in Figure 8.1.

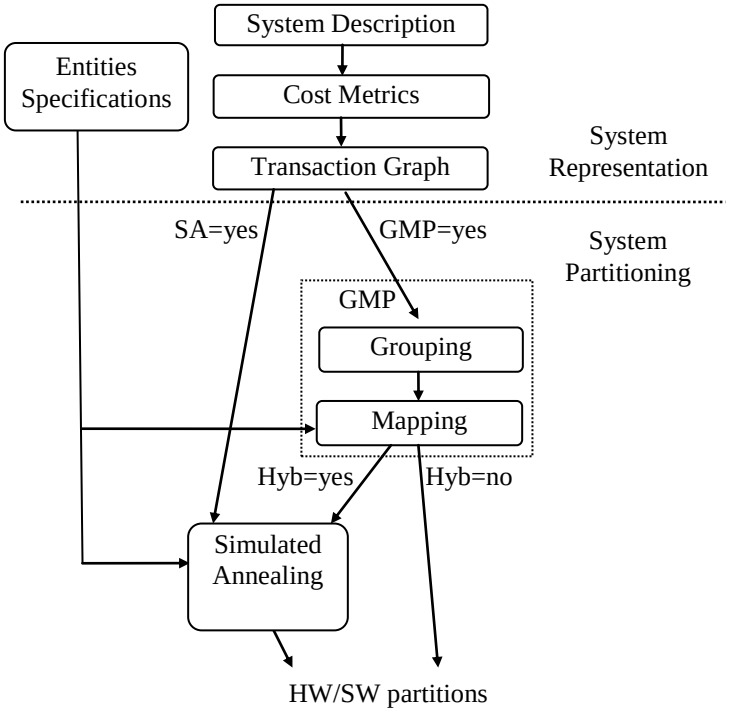

Figure 8.1. Hybrid GMP-SA partitioning

Regarding the emulation platform, one thing that this dissertation does not take into account is that big designs usually do not fit on a single FPGA. The proposed hardware emulator has to decide where to implement the testbench-DUT communication channel in a multi-FPGA environment. One potential approach would be to keep the communication channel on the same FPGA as shown in Figure 8.2, while other approach may favor to split the channel in order to increase the parallelism. In a hardware emulator consisting of multiple FPGAs, the utilization of

the FPGA logic resource is usually very low due to the limitation on the number of I/O pins. Therefore, there is a lot of available free FPGA space that can be used in order to have multiple HDL TB Simulator blocks in the system.

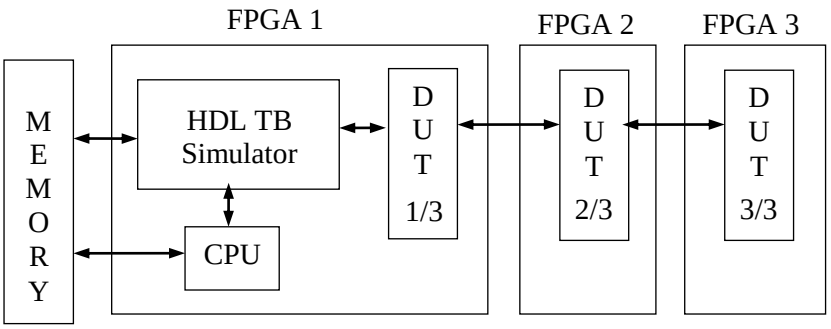

Figure 8.2. Hardware Emulator using three FPGAs

Moreover, we can next investigate the inter-FPGA communication. *Virtual wire* technology not only increases the inter-FPGA communication capability, but it also increases the logic resource utilization by means of time division multiplexing (TDM). TDM allows one physical wire to be shared by multiple logical wires. For TDM to be effective, each transportation of an inter-FPGA signal must be carefully assigned to a slot of the time division. Essentially, there are two things we could investigate:

1. The inter-FPGA requirements in a hardware emulator where the DUT runs several times slower than the speed of the final design and therefore the internal signal transactions are much slower than the real system.

2. How we could use the pause mechanism provided by our proposed hardware emulator (see Section 4.7) in order to pause periodically (probably in every simulation cycle) the emulation and transfer the values of the signals between the FPGAs. In this way, the system can essentially operate with any number of I/O pins for the inter-FPGA communication.

Finally, we plan to investigate the use of General Purpose Graphics Processor Units (GPGPUs) in the emulation environment. GPGPU is a new parallel technology that can provide cost-efficient solutions for Single Instruction Multiple Data (SIMD) applications. Such a parallel application could be the emulation of a design. Towards this end, the authors in [NP10] provide

an approach that parallelizes SystemC's discrete-event simulation (DES) on GPGPUs by transforming the model of computation of DES into a model of concurrent threads that synchronize as and when necessary. The proposed threading model is capable of executing in parallel on the large number of simple processing units available on GPUs.

Because the GPUs are stuck on the relatively slow PCIe bus, the communication overhead between a GPU and an external CPU can slow down significantly the emulation performance. Therefore, the GPU-CPU communication channel can become the bottleneck of the system emulation if the testbench runs on an external CPU and the DUT is emulated on the GPU.

State-of-the-art technologies, such as multi-core CPUs, FPGAs and GPUs, can become powerful platforms for advanced CAD tools in order to develop high quality systems satisfying all the needs of the demanding market. Further investigation is required in order to combine the advantages of these technologies and derive optimum solutions.

Bibliography

[AC07] Standard Co-Emulation Modeling Interface Reference Manual, Version 2.0, Accellera, 2007, http://www.accellera.org/activities/itc/Release200.pdf

[AGC] Agility Compiler, http://agilityds.com/support/download_agility.aspx

[ALR] ALDEC, Riviera, http://www.aldec.com/products/riviera/

[AS93] P. Athanas and H. F. Silverman, "Processor reconfiguration through instruction-set metamorphosis," IEEE Computer Mag., pp. 11–18, Mar.1993.

[AST] Altera, SignalTap, http://www.altera.com/literature/hb/qts/qts_qii53009.pdf

[AXS] Axis, SEmulation, http://www.design-reuse.com/news/1702/axis-tool-blends-emulation-simulation.html

[CAP] Cadence, Palladium Accelerator/Emulator, http://www.cadence.com/products/sd/pages/default.aspx

[CA96] C. Carreras et. al., "A Co--Design Methodology Based on formal Specification and High--Level Estimation", Fourth International Workshop on Hardware/Software Codesign, Pittsburgh, Pennsylvania, 1996

[CA02] K Ben Chehida, M Auguin, "HW/SW partitioning approach for reconfigurable system design", CASES 2002

[CH94] M. Chiodo, et. al, "Hardware- Software Codesign of Embedded Systems", *IEEE Micro*, Vol. 14, No. 4, pp.26-36, Aug. 1994

[DCB] DAFCA, ClearBlue, http://www.dafca.com/products/clearblue.html

[DPR] Dhrystone performance results: http://performance.netlib.org/performance/html/dhrystone.data.col0.html

[EH93] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers", IEEE Design and Test,V-10, Dec 1993.

[EP97] P. Eles, Z. Peng, K. Kuchcinski, A. Doboli, "System level hardware/software partitioning based on simulated annealing and tabu search", Design Automation of Embedded Systems, 1997, 2(1). pp.5-32.

[EVZ] EVE, Zebu hardware emulator, http://www.eve-team.com/products/index.php

[FZ05] M. Finc, A. Zemvra "Profiling soft-core processor applications for hardware/software partitioning", Journal of Systems Architecture: the EUROMICRO Journal, 2005

[GM93] R.K. Gupta and G.D. Micheli, “HW-SW Cosynthesis for Digital Systems”, IEEE Design & Test of Computers, Sep 1993, pp. 29-41

[GRR] GateRocket**,** RocketDrive, http://www.gaterocket.com/

[HE01] J. Henkel, R. Ernst, “An approach to Automated Hardware/Software Partitioning Using a Flexible Granularity that is Driver by High-Level Estimation Techniques”, IEE Transactions on Very Large Scale Ingration (VLSI) Systems, 9(2):273–289, April 2001.

[HE98] J. Henkel, R. Ernst, “High-Level Estimation Techniques for Usage in Hardware/Software Co-Design”, ASP-DAC 1998.

[HGP] HiTech Global, PALMiCE, http://www.hitechglobal.com/designtools/computex/palmicefpga.htm

[HP99] I. Hamzaoglu and J. H. Patel, "Reducing Test Application Time for Full Scan Embedded Cores", Int. Symp. on Fault-Tolerant Computing, pp. 260-267, June 1999.

[HS06] Ho-seok Choi, Seung-beom Lee, Sin-chong Park , “Instruction Based Synthesizable testbench Architecture”, IEICE Transactions on Electronics Vol. E89C No.5 pp.653-657, 2006

[INV] Intel® VTune™ Performance Analyzer, http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/

[IS05] IEEE, "IEEE Standard Testability Method for Embedded Core-based Integrated Circuits", IEEE Std 1500-2005, 2005, pp. 0_1-117.

[JC05] Jiann-Chyi Rau, Chih-Lung Chien, and Jia-Shing Ma, “Reconfigurable Multiple Scan-Chains for Reducing Test Application Time of SOCs”, Circuits and Systems, 2005. ISCAS 2005, 23-26 May 2005 Page(s):5846 - 5849 Vol. 6.

[JE99] A.A. Jerraya, et al., “Multilanguage Specification for System Design and Codesign” chapter on “ System-Level Synthesis”, *NATO ASI 1998*, A. Jerraya and J. Mermet eds., Kluwer Academic, 1999

[KB03] Karthikeyan Bhasyam, et. al, “HW/SW Codesign Incorporating Edge Delays Using Dynamic Programming”, Euromicro Symposium on Digital System Design, IEEE Computer Society, Sept. 2003

[KK04] Ryan Kastner, Adam Kaplan, Majid Sarrafzadeh, “Synthesis Techniques and Optimizations for Reconfigurable Systems”, ISBN:978-1-4020-7698-5, 2004

[KL94] A. Kalavade, E. A. Lee, "A Global Criticality/Local Phase Driven Algorithm for the Constrained Hardware/Software Partitioning Problem”, Workshop on Hardware/Software Codesign, 1994, pp. 42-48.

[LH04] Il-soo Lee , Yong Min Hur , Tony Ambler, "The Efficient Multiple Scan Chain Architecture Reducing Power Dissipation and Test Time", pp.94-97, 13th Asian Test Symposium (ATS'04), 2004

[LL09] Yang Liu, Qing Cheng Li, "Hardware Software Partitioning Using Immune Algorithm Based on Pareto", vol. 2, pp.176-180, International Conference on Artificial Intelligence and Computational Intelligence, 2009

[LS07] Lattice Semiconductor Corporation, "FreedomChip - A Cost Reduction Methodology Lattice SC/M Devices", Lattice Semiconductor Corporation, 2007.

[MB99] M. Bauer et al., "A Method for Accelerating Test Environments", Proc. 25th Euromicro Conf., vol. 1, IEEE Press, 1999. pp. 477-480

[MGV] Mentor Graphics, Veloce, http://www.mentor.com/products/fv/emulation-systems/

[MP07] I. Mavroidis, I. Papaefstathiou, "Efficient Testbench Code Synthesis for a Hardware Emulator System", Proc. DATE 2007

[MP08] I. Mavroidis, I. Papaefstathiou, "Accelerating Hardware Simulation: Testbench Code Emulation", FPT 2008, pp. 129-136

[MP09] I. Mavroidis, I. Papaefstathiou, "Accelerating Emulation and Providing Full Chip Observability and Controllability", IEEE Design & Test, vol. 26, 2009, pp. 84-94

[MP10] I. Mavroidis, I. Papaefstathiou, A. Garbo, S. Nocco, J. Kim, G. Cabodi, L. Lavagno, "An Open-Source, Fast, Cost-Efficient Hardware/Software Partitioning Tool", Poster Session 3, FCCM 2010

[MTI] Micron Technology, http://www.micron.com/

[MZ06] Pierre-Andre Mudy, Guillaume Zufferey, Gianluca Tempesti, "A Dynamically Constrained Genetic Algorithm For Hardware-software Partitioning", GECCO, 2006

[NM96] Ralf Niemann, Peter Marwedel, "Hardware/Software Partitioning using Integer Programming", European Design and Test Conference (ED&TC '96)

[NP10] M. Nanjundappa, H.D. Patel, B.A. Jose, S.K. Shukla, "SCGPSim: A Fast SystemC Simulator on GPUs", pp. 149-154, Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific

[OP10] OSMOSIS partitioning tool, http://sourceforge.net/projects/osmosispart

[OS10] OSMOSIS synthesis tool, http://sourceforge.net/projects/osmosissynth

[PA04] Pradeep Adhipathi "Model based approach to Hardware/Software Partitioning of SOC Designs", MS, Blacksburg, Virginia, 2004

[RE00] J. Redstone, S. J. Eggers, and H. M. Levy, "An analysis of operating system behavior on a simultaneous multithreaded architecture", Architectural Support for Programming Languages and Operating Systems, 2000, pp. 245.256.

[RH03] R. Henftling et al., "Re-use-centric Architecture for a Fully Accelerated testbench Environment", Proc. 49th Design Automation Conf. (DAC 03), ACM Press, 2003. pp. 372-375

[RM03] A. Ramani, I. Markov, ”Combining two local search approaches to hypergraph partitioning”, International Joint, Conference on Artificial Intelligence, AAAI, 2003.

[SM97] M. J. S. Smith. “Application Specific Integrated Circuits”, chapter 14, page 764, Addison-Wesley, Reading, Mass., 1997

[SN04] Sudarshan Banerjee, Nikil Dutt, “Efficient Search Space Exploration for HW-SW Partitioning”, International Conference on Hardware Software Codesign, 2004

[SR98] A. Srinivasan and et. al, “Accurate area and delay estimation from RTL descriptions”, IEEE Transactions on VLSI Systems, 6(1):168--172, Mar. 1998

[TA06] Tan et. Al. “Testing of UltraSPARC T1 Microprocessor and its Challenges” ITC '06. Oct. 2006 Page(s):1 - 10

[TT04] Anurag Tiwari and Karen A. Tomko, “Scan-chain Based Watch-points for Efficient Run-Time Debugging and Verification of FPGA Designs”, ASPDAC 2004

[VEE] Verisity, eCeleretor testbench Acceleration, http://www.verisity.com/products/ecelerator.html

[VL97] F Vahid, T D Le, ”Extending the Kernighan-Lin heuristic for Hardware and Software functional partitioning”, Jrnl Design Automation for Embedded Systems, V-2, 1997

[WC02] T. Wiangtong, P. Cheung, and W. Luk, ”Comparing three heuristic search methods for functional partitioning in hardware-software codesign”, Journal of Design Automation for Embedded Systems, 2002, pp.425-449.

[WG01] T. Wheeler, P. Graham, B. Nelson, and B. Hutchings, **“**Using Design-Level Scan to Improve FPGA Design Observability and Controllability for Functional Verification”, Proceedings of the 11th International Conference on Field-Programmable Logic and Applications, pages: 483-492, 2001.

[WL08] Dawei Wang, Sikun Li, Yong Dou, “Collaborative hardware/software partition of coarse-grained reconfigurable system using evolutionary ant colony optimization”, ASP-DAC, 2008, pp. 679-684

[WP82] T. W. Williams and K. P. Parker, “Design for testability - a survey”, IEEE Transactions on Computers, C-31(1):2–15, January 1982.

[XCS] Xilinx, ChipScope Pro, http://www.xilinx.com/literature/literature-chipscope.htm

[XW09] Xiao-zhang LU, Wei LIU, Yao-dong TAO, “Method of HW/SW partitioning based on NSGA-II”, Journal of Computer Applications, 2009, 29(1), pp. 238-241

[YC04] Young-Il Kim, Chong-Min Kyung, "TPartition: testbench Partitioning for Hardware-Accelerated Functional Verification", IEEE Design and Test of Computers, vol. 21, no. 6, pp. 484-493, Nov/Dec, 2004.

[YC95] C. W. Yeh, C. K. Cheng, T. T. Y. Lin, “Optimization by Iterative Improvement: An Experimental Evaluation on Two-Way Partitioning”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 14, no. 2, 1995, pp. 145-153.

[YM07] Young-Il Kim, Moo-Kyoung Chung, Ando Ki, and Chong-Min Kyung, "Reducing Transaction-Level Modeling Effort while Retaining Low Communication Overhead for HW/SW Co-Emulation System", IEEE International Symposium on VLSI Design, Automation, and Test (VLSI-DAT), Hsinchu, Taiwan, 2007

[YW04] Young-Il Kim, Wooseung Yang, Young-Su Kwon, Chong-Min Kyung, "Communication-Efficient Hardware Acceleration for Fast Functional Simulation", pp. 293-298, Design Automation Conference, 41st Conference on (DAC'04), 2004.