Hybrid Prototyping of Multicore Embedded Systems

Ehsan Saboori

A thesis
in
The Department
of
Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Doctor of Philosophy
Concordia University
Montreal, Quebec, Canada

JULY 2016

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Mr. Ehsan Saboori**

Entitled: **Hybrid Prototyping of Multicore Embedded Systems**

and submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy (Electrical and Computer Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair

Dr. Wahid S. Ghaly

_____ External Examiner

Dr. Amirali Baniasadi

_____ External to Program

Dr. Lingyu Wang

_____ Examiner

Dr. Otmane Ait Mohamed

_____ Examiner

Dr. Yan Liu

_____ Supervisor

Dr. Samar Abdi

Approved by: _____

Dr. Wei-Ping Zhu, Graduate Program Director

July 22, 2016

_____

Dr. Amir Asif, Dean

Faculty of Engineering and Computer Science

Abstract

Hybrid Prototyping of Multicore Embedded Systems

Ehsan Saboori, Ph.D.
Concordia University, 2016

Multicore platforms are becoming increasingly pervasive in modern embedded systems. System level modeling techniques have enabled creation of fast software models of multicore platforms, commonly known as Virtual Prototypes, for early functional validation of embedded software, before the hardware is available. On the other hand, for accurate performance validation, the complete multicore platform can be implemented as a physical prototype on FPGA. Both virtual platforms and FPGA prototypes have their respective pros and cons. Virtual platforms have the advantage of high speed functional simulation and, typically, scale well with the number of cores. However, the accuracy of performance estimation is sacrificed. FPGA prototypes provide cycle-accurate performance estimation, because the software executes directly on an FPGA implementation of the target cores. However, it takes a significant amount of time to design, implement and test the inter-core communication architecture on the FPGA.

In this thesis we propose to design a novel system-level modeling framework, called Hybrid Prototyping. Our goal is to provide the benefits of both virtual platforms and FPGA prototypes. It aims to provide early, fast, and scalable models, similar to virtual platforms, along with the cycle-accuracy of FPGA prototypes. Using hybrid prototyping, embedded software designers will be able to create concurrent applications and accurately analyze the performance implication of their optimizations before the chip is delivered. At the same time, multicore architects will be able to modify the platform model without

having to do full system prototyping. Therefore, hybrid prototyping will enable early and reliable multicore embedded system design, resulting in huge productivity gains for both embedded software designers and multicore chip architects.

Acknowledgments

I would like to take this opportunity to thank all the people who have contributed to the fruition of this thesis. First of all, I would like to thank my advisor, Professor Samar Abdi for excellent guidance during my PhD studies. He has taught me how to think about problems, how to approach the solution and when to commit to a solution. I will forever be indebted for the time and the effort he has spent in my education. I would also like to thank many friends in ECE, including Shafigh Parsazad, Richard Lee, Partha Ravishankar, Paul Leons, Ali Barzegar, Karim Al-Khalek and Aryan Yaghoubian for their great company and all their help.

This thesis would not have been possible without the unflinching support of my wonderful wife, Golnaz. She celebrated my success and lifted my spirits whenever I faced rejection. She is by far the single most important reason why I maintained my emotional balance through the ups and downs of graduate life.

Finally, I would like to thank my parents, to whom this thesis is dedicated. My dear mother and father who have made many personal sacrifices to provide me the best possible education and a healthy atmosphere at home. It is impossible to put down in words their contribution to my personal growth. I would also like to thank my brother, sister and their families for always cheering me on.

Contents

List of Figures

List of Tables

Code Listings

Publications and Workshops

1. Ehsan Saboori, Samar Abdi, “Fast and cycle-accurate simulation of multi-threaded applications on SMP architectures using hybrid prototyping”, International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Pittsburgh, USA, 2016

2. Ehsan Saboori, Samar Abdi, “Rapid design space exploration of multi-clock domain MPSoCs with Hybrid Prototyping”, Electrical & Computer Engineering (CCECE), 2016 29th IEEE Canadian Conference on, Vancouver, BC, 2016

3. A. Barzegar, E. Saboori and S. Abdi, "DRAC: a dynamically reconfigurable active L1 cache model for hybrid prototyping of multicore embedded systems," 2014 25nd IEEE International Symposium on Rapid System Prototyping, New Delhi, 2014, pp. 86-92.

4. Ehsan Saboori, Samar Abdi, “Hybrid Prototyping of Multicore Embedded Systems”, qualified presentation at TEXPO, student competition and exhibition, Ottawa, Canada, Oct, 2014

5. S. Abdi, E. Saboori, “Hybrid Prototyping of Many-core Embedded Systems,” Many-Core Embedded Systems Workshop (MCES), Montreal, Canada, October, 2013.

6. E. Saboori, S. Abdi, “Hybrid Prototyping of MPSoCs,” In Proceedings of the International Forum on MPSoC and Embedded Multicore, Otsu City, Japan, June, 2013 (invited paper).

7. Saboori, Ehsan; Abdi, Samar, "Hybrid Prototyping of multicore embedded systems," Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013, vol., no., pp.1627,1630, 18-22 March 2013

- The source code is available at: **https://github.com/ehsab/hybrid_prototyping**

Glossary

DES	Design Space Exploration
DRAC	Dynamically Reconfigurable Active Cache
FPGA	Field Programmable Gate Arrays
FPGA Prototyping	Using FPGAs as a platform for SoC development and verification
Hybrid Prototyping	The proposed prototyping framework in this thesis
ISS	Instruction Set Simulator
MEK	Multicore Emulation Kernel
MPSoC	Multi Processors System on Chip
RTL	Register-Transfer Level
RTOS	Real-time Operating System
SMP	Symmetric Multiprocessing
SoC	System on Chip
Thread Scheduling	deciding which thread runs at a certain point in time
Virtual Prototyping	Software-based modelling for SoC development and verification

Chapter 1

Introduction

1.1 Motivation

Multicore platforms are becoming increasingly pervasive in modern embedded systems because of the potential computation speedups resulting from concurrent application execution on independent cores. However, both the multicore hardware platform and the embedded software that runs on it must be carefully designed for functional correctness and optimal performance. System level modeling techniques have enabled creation of fast models of multicore platforms, commonly known as virtual platforms, for early functional validation of embedded software. Virtual platforms enable early functional validation of embedded software, before the chip is delivered. For accurate performance validation, the complete multicore platform can be prototyped on *F*ield *P*rogrammable *G*ate *A*rrays (FPGA). The FPGA prototype serves as a cycle-accurate hardware model of the chip and can be used for embedded software design using in-circuit emulation tools.

Both virtual and FPGA prototypes have their respective pros and cons. Virtual prototypes have the advantage of high speed functional simulation and, typically, scale

well with the number of cores. However, the accuracy of performance estimation is sacrificed because the processor simulation models used are very abstract. Cycle-accurate models may be used in virtual prototypes, but they drastically slow down simulation speed, thereby defeating the purpose of fast and early software validation. FPGA prototypes provide cycle-accurate performance estimation because the software executes directly on an FPGA implementation of the target cores. However, it takes a significant amount of time to design, implement and test the inter-core communication architecture on the FPGA. Furthermore, if several cores are being used, the amount of reconfigurable logic required for implementing the cores, the communication fabric, and the on-chip memory for the full multicore system becomes too large to fit on a single FPGA. Using multiple FPGA chips adds another dimension of complexity to implementing the prototype. Therefore, the scalability and design time of full system FPGA prototypes are serious issues.

In this thesis, we present a new technique called Hybrid Prototyping framework that offers the scalability benefits of virtual prototypes, as well as the cycle-accuracy of FPGA prototypes. The system provides a high-speed model of a multicore platform that will enable embedded software designers to accurately analyze and debug their applications before the hardware is available. Application designers can also use these models to influence the multicore architecture design early in the design process. The fundamental idea of hybrid prototyping is to create an emulation kernel in software that executes on a single target core. The target core is physically implemented in FPGA. The emulation kernel simulates the execution of concurrent tasks on independent emulated cores by dynamically scheduling the tasks on the physical target core. The emulation kernel manages the state of the individual emulated cores and the logical times until which they have been simulated.

1.2 Literature Review

Pre-silicon performance validation of multicore embedded systems is a serious challenge. Both virtual prototyping and FPGA-based physical prototyping have been a topic of intense research with the growing adoption of multicore architectures and the corresponding need to provide early simulation models to embedded software designers.

Virtual prototype is a set of functional models of *S*ystem on *C*hip (SoC) hardware such as processors, peripherals and buses, in software form. Processor model is often implemented using *I*nstruction *S*et *S*imulator (ISS) which provides binary compatibility with embedded processor (called Target) instruction set. ISS converts target instruction set to the instruction set of the general-purpose computer by running the simulation (called Host) to allow execution of un-modified embedded software. Bus and peripheral are typically modeled using a high-level language such as ANSI C or SystemC with focusing on pin-accurate software visible interfaces such as register, bus protocol and peripheral functionality.

FPGA-based prototyping is another widely-used pre-silicon SoC evaluation method using FPGA as the platform. FPGA can be used to implement any logic function that an *A*pplication-*S*pecific *I*ntegrated *C*ircuit (ASIC) chip could perform, which makes it a good platform for rapid system prototyping. In FPGA-based prototype the application and the system software for a design is executed directly on FPGA. Figure 1 shows virtual prototyping vs. FPGA prototyping.

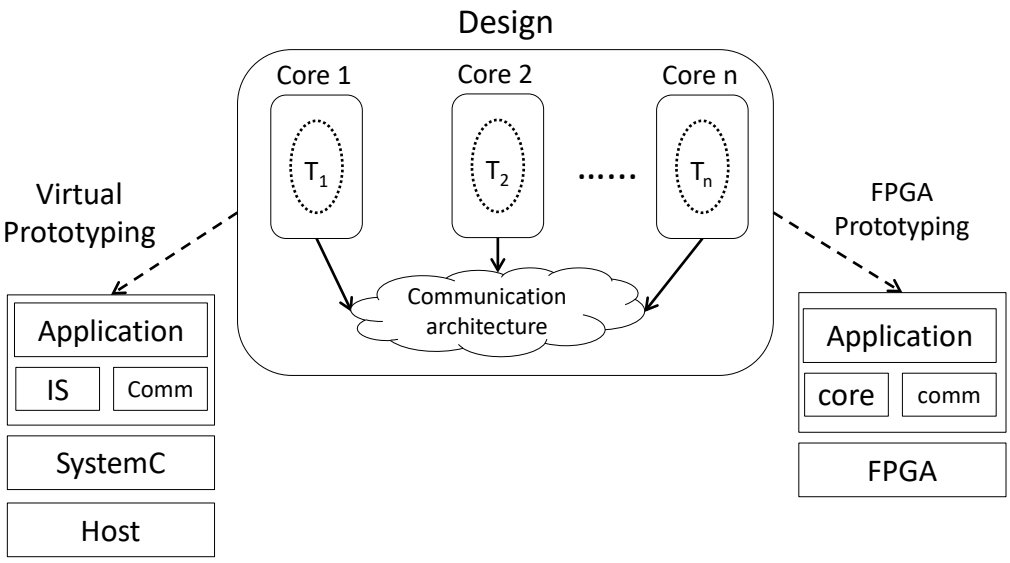


Figure 1: Virtual Prototyping vs. FPGA Prototyping

Conventional technologies such as virtual prototyping and FPGA prototyping have several limitations. Virtual prototypes, based on host-compiled ISS, can provide high simulation speed, but at the expense of limited or no timing accuracy. FPGA prototypes, based on instantiation and integration of processor cores in FPGA, provide cycle accuracy, but with the disadvantage of high development cost and lack of scalability. In addition, there is no flexibility of abstracting the inter-core communication in FPGA prototypes, since it is fixed in hardware. Furthermore, software debugging on multiple processors in FPGA can be quite challenging.

1.2.1 Virtual Prototyping

In recent years multicore virtualization has become an important research subject in computer architecture and embedded systems fields. Virtual prototyping involves the use of highly abstracted target architecture model. Many tools and frameworks have been developed for virtual prototyping. The SimpleScalar [1] tool set provides an infrastructure for simulation and architectural modeling. This tool is an interpreter which executes all program instructions and can model a variety of platforms ranging from simple unpipelined processors to detailed dynamically scheduled micro-architectures with multiple-level memory hierarchies. Quick EMUlator (QEMU) [2] is an open source machine emulator which relies on dynamic binary translation. It allows applications compiled for one architecture to be run on another. The proposed system provides performance estimation for *D*esign *S*pace *E*xploration (DSE). Mambo [3] is a full-system simulator for modeling PowerPC-based systems. It provides building blocks for creating simulators that range from purely functional to timing-accurate. ASIM [4] is a decoupled simulation framework. This framework provides modularity which helps break down the performance-modeling problem into individual pieces that can be modeled separately, while its reusability allows using a software component repeatedly in different contexts. PTLsim [5] is a cycle accurate full system x86-64 microprocessor simulator and virtual machine. This framework provides cycle accurate simulation with sacrificing the speed.

These frameworks not only enable earlier software development but also can give a feedback where the hardware needs to be adapted or to be changed prior the implementation. The enabler for virtual prototyping is a virtual platform, which is an executable model of the target core architecture, including processors, memories, buses and peripheral [6]. Virtual prototyping can provide flexibility, scalability and ease of

debugging for the designer, but one most compromise either simulation speed or accuracy. Virtual prototypes cannot provide highly accurate results due to abstract software implementation of models. Conversely, if accurate models are used in virtual prototyping, the simulation slows down significantly.

Software cycle-accurate simulation has been the primary tool to allow collaborative hardware and software [7]. Cycle-accurate *R*egister-*T*ransfer *L*evel (RTL) simulations accurately model hardware behaviors down to register transfer level, suitable for hardware verification and profiling. They provide very accurate timing with sacrificing the simulation time. ModelSim [8], Synopsys VCS [9] and Cadence Incisive Enterprise Simulator [10] are some example of these kind of simulators. Simics [11] is a full-system functional simulators. It provides the level of accuracy necessary to execute fairly complex binaries on the simulated machine. GEMS [12], timing multiprocessor simulator, and SimWattch simulation tool [13], used for microprocessor performance and power estimation, are built on top of the Simics library. Such simulators can be used for large designs with range of single-digit hertz which is not reasonable for regular software code to be run on it [6].

Amongst the software-based methods, the most successful developments have been virtual platform technologies based on binary translation, as commercialized by Windriver [14], Coware [15], and Xilinx XVP [16]. In most virtual platforms, host-compiled ISS have replaced or complemented traditional cycle-accurate micro-architecture simulators [17] [18]. HISCS [19] is a technique for generation of fast instruction-set simulators that combines the benefit of both compiled and interpretive simulation. A major challenge in this technique is the compilation time overhead that makes usage of compiler optimizations impractical, especially for large applications. DynamoSim [20] is a suite of techniques inspired by recent advances in dynamic compilers to construct a hybrid simulation framework. In this framework any

instruction can be interpreted and only frequently executed instructions are translated on-the-fly into native code for direct execution. SoClib [21] is an open platform for virtual prototyping of *M*ulti-*P*rocessors *S*ystem on *C*hip (MPSoC). Its core is based on a library of SystemC [22] simulation models to facilitate architecture exploration of MPSoC. Open Virtual Platform (OVP) [23] is an open source virtual platform which uses OVPsim to simulate different designs. OVPsim is an instruction accurate simulator which provides infrastructure for describing platforms with one or more processors containing shared memory and busses in arbitrary topologies and peripheral models [24]. Wang et al. [25] have shown OVP and its interoperability with the existing Transaction-Level Modeling (TLM) based SystemC platforms shows that OVP is faster than other existing solutions. Such simulators can provide significant speedups (reaching simulation speeds of several hundred MIPS), but often focus on functionality and speed at the expense of limited or no timing accuracy.

Host-compiled software simulation technique is based on source level static delay annotation in the application [26] [27]. The delays are derived by analyzing the execution of applications on an abstract model of the core. Although source-level annotation techniques promise high simulation speed, they require the full application source, including sources of libraries. These techniques also use an abstract core model, leading to estimation inaccuracies.

Another popular method to simulate a multiprocessor system is to integrate multiple ISSs into a SystemC based simulation backbone. MPARM [7] is well known academic simulation platforms based on this solution. Such simulators are able to execute target binary and provide cycle-accurate simulation. However, they are extremely slow and very complicated [28].

1.2.2 FPGA Prototyping

FPGAs are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing. This feature distinguishes FPGAs from ASICs, which are custom manufactured for specific design tasks. FPGAs has become a natural choice for building system prototypes of ASIC and SoC designs. It allows hardware designers to develop and test their systems, and it provides software developers early access to a fully functioning hardware platform.

FPGA Prototyping is a technique to verify the functionality and performance by implementing the design on a FPGA. FAST [29] is a methodology that enables a single FPGA to accelerate the performance of cycle-accurate computer system simulators modeling modem, realistic SoCs and embedded systems. ProtoFlex [30] simulation architecture is an FPGA-based, full-system functional simulator for a symmetric multiprocessor server, hosted on a single FPGA and achieves a significant speedup over comparable software simulation. Chiou et al. improve FAST simulator by supporting work in parallelized computer system simulators [31]. Taeweon et al. [32] show the possibility of using FPGA in architecture research to enhance the simulation time. They introduce a new hardware/software co-simulation method that performs execution-driven microarchitecture simulation. Based on an off-the-shelf Pentium-III system that communicates with an FPGA via the Front-Side Bus. RAMP gold [33] is a FPGA-based architecture simulator for multiprocessors provides a high-throughput, cycle-accurate full-system simulator that runs on a single Xilinx Virtex-5 FPGA board, and which simulates a 64-core shared-memory target machine capable of booting real operating systems.

FPGA prototypes are highly accurate and fast. So, designers do not have to rely only on software simulations to verify a design. The application and system software for a design is executed directly on FPGA prototype to ensure that it is functionally correct before implementation. In contrast to virtual prototyping, FPGA prototyping does not provide scalability and flexibility is too costly. It is impractical for designers to implement different hardware platform on FPGA, given the vast amount of design choices.

FPGA prototyping can be much more complicated for multicore architectures because FPGA logic on a chip is limited. Prototyping multicore may require multiple FPGA chips, which can compromise the accuracy of the FPGA prototype. In addition, debugging can be cumbersome and time consuming on FPGA prototypes. ChipScope [34] and SignalTAP [35] are standard debugging tools offered by FPGA vendors. However, they are difficult to use with multicore. Several prototyping frameworks using FPGA have been proposed. RAMP [36] provides the prototyping platform for implementing the full-system under test on FPGA. This platform puts together a large array of FPGAs in order to support the instantiation and integration of hundreds of cores [37] [36]. Unfortunately, the cost and design time of such full system prototypes is very high [30]. In addition, there is no flexibility of abstracting the inter-core communication in RAMP, since it is fixed in hardware by the inter-FPGA communication architecture.

Another FPGA-based modeling approach implements the SystemC simulation kernel in FPGA to support standard hardware I/O during simulation [38]. Yet, another type of FPGA-assisted simulation, called virtual in-circuit emulation, runs software-on-host and application-specific hardware on FPGA to avoid slow RTL simulation in software [39]. The above techniques are incremental improvements to cycle-accurate simulation and have not been shown to scale to large multicore designs.

1.3 General Problem Statement

The objective of this work is to design a novel system-level modeling framework, called hybrid prototyping that can provide the benefits of both virtual platforms and FPGA prototypes. Virtual prototypes have the advantage of scalability and high speed functional simulation by sacrificing the accuracy of performance estimation. Using cycle-accurate models are drastically slow down simulation speed. FPGA prototypes provide cycle-accurate performance. However, the scalability and design time of full system FPGA prototypes are serious issues.

The proposed framework targets the typical *S*ymmetric *M*ulti*P*rocessing (SMP) architecture consisting of multiple cores, each with a dedicated L1 cache and shared off-chip main memory. It also introduces emulation kernel and the modeling of dynamic *R*eal-*T*ime *O*perating *S*ystem (RTOS) scheduler as well as hardware interrupts on top of the emulation kernel, in order to support the simulation of unmodified multi-threaded applications. The L1 caches of the cores are emulated by a dynamically reconfigurable on-chip memory module to support dynamic thread scheduling in SMP designs. In order to meet the above objective, we need to address some key technical challenges in hybrid prototyping design. First, the core, the emulation kernel and the cache model must all fit on one FPGA chip for optimal performance. Second, the emulation kernel and the cache model must be highly optimized for performance so that the core-context switching overhead is minimized. Finally, the emulation system must be completely transparent to the user, similar to virtual platforms.

Hybrid prototyping aims to provide early, fast, and scalable models similar to virtual prototypes along with the cycle-accuracy of FPGA prototypes. Using hybrid prototyping, embedded software designers can create concurrent applications and accurately analyze the performance implication of their optimizations before

implementation. At the same time, hardware architects can modify the platform model without having to do full FPGA prototyping. Therefore, hybrid prototyping will enable early and reliable multicore embedded system design, resulting in huge productivity gains for both embedded software designers and multicore chip architects.

1.4 Thesis Contribution

The main contributions of this thesis are presented as follows:

1. **Multicore Emulation Kernel (MEK).** The fundamental idea of hybrid prototyping is to create multicore emulation kernel that executes on a single target core, which is physically implemented in FPGA. The MEK emulates the execution of concurrent tasks on independent cores by dynamically scheduling them on the physical target core. It provides simulation primitives and the modeling of inter-core communication. The emulation kernel, MEK, implements primitives for the management of discrete events and logical times for the tasks. It also provides simulation primitives and services to instantiate emulated cores and modeling of inter-core communication for SMP architecture. SMP is a system that has multiple, identical processors all sharing memory and devices.
2. **Multi-clock domains MPSoCs.** Most embedded processors support several operating frequencies, which allows us to create a mixture of cores, each running at a different operating point. So, a multicore embedded system might have multiple clock domains. Hybrid prototyping can be applied to realistic multi-clock MPSoC designs.

3. **Simulation of Hardware peripherals (interrupts).** An important aspect of RTOS design is the mechanism for servicing the interrupt-driven devices such as hardware peripherals. Hybrid prototyping is extended to simulate interrupts issued by external hardware.
4. **Memory hierarchy simulation.** Caches are widely used in embedded system to increase efficiency. So it is important to consider memory hierarchy in hybrid prototyping. A novel dynamically reconfigurable active L1 cache (instruction and

data) model called DRAC proposed for hybrid prototyping. It is an on-chip hardware peripheral connected to the local bus of the core. DRAC is responsible for swapping the cache context when the MEK switches simulation context from one emulated core to another. Utilizing DRAC model, embedded designers are able to analyze, verify, and optimize their multicore design with cache design without the need for full system prototyping.

5. **SMP designs with dynamic RTOS scheduler model.** In SMP designs, the number of threads that can run concurrently (simultaneously) is limited by the number of processors. Since each processor can execute only one thread at a time, with multiple processors, multiple threads can execute simultaneously. A single kernel manages all cores simultaneously. The hybrid prototyping supports SMP architectures and introduces the modeling of dynamic RTOS scheduler on top of the emulation kernel, in order to support the simulation of unmodified multi-threaded applications. Therefore, in the hybrid prototyping, the RTOS scheduler can dynamically schedule any thread on any emulated core to achieve full utilization of all emulated cores.
6. **Automatic/Semi-Automatic Design Space Exploration.** Using hybrid prototypes, multicore embedded system designers can create concurrent applications and accurately analyze the power and performance implication of their optimizations before the hardware is available. As such, the hybrid prototyping is capable of fast and early multicore design space exploration. It can provide huge productivity gains for multicore chip architects as they can optimize the hardware architecture without having to do full system prototyping.

1.5 Thesis Outline

The remainder of this thesis is organized as follows. In Chapter 2 the hybrid prototyping methodology is presented. We start with describing the main idea behind the hybrid prototyping technique followed by explaining its different layers. In chapter 3, we describe the MEK and primitives provided by this layer. Chapter 4 explains the hardware model layer and its primitives. Memory hierarchy and cache model supported by the hybrid prototyping are also discussed in details. Chapter 5 covers software model layer including thread management and RTOS model scheduler. Chapter 6 includes the experimental results for evaluating the hybrid prototyping in terms of accuracy, speed and scalability. Finally, conclusions and suggestions for future work are provided in Chapter 7.

Chapter 2

Hybrid Prototyping Methodology

Multicore platforms deliver greater computing power through concurrency, offer greater system density, and run at lower clock speeds than uniprocessor chips resulting lower power consumption and thermal dissipation. Multiprocessing includes several operating modes such as SMP and *A*symmetric *M*ulti*P*rocessing (AMP). An AMP system has multiple cores (may be either heterogeneous or homogeneous multicore). A separate operating system or a separate copy of the same operating system, manages each core. Typically, each application's process is locked to a single core. It provides an execution environment similar to that of uniprocessor systems. It allows simple migration of legacy code and facilitates management of each core independently. However, it can result in underutilization of processor cores. For instance, if one core becomes busy, applications running on that core cannot, in most cases, migrate to a core that has more CPU cycles available. Though such dynamic migration is possible, it typically involves complex check pointing of the application's state and can result in a service interruption while the application is stopped on one core and restarted on

another. SMP is a computer architecture that provides fast performance by using two or more homogeneous processors to complete individual processes simultaneously under a single operating system. Unlike asymmetrical processing, any idle processor can be assigned any task, and additional processors can be added to improve performance and handle increased loads. Specific applications can benefit the most if the code allows multithreading. SMP systems can easily move tasks between processors to balance the workload efficiently. Figure 2 shows different configurations of a multicore design.

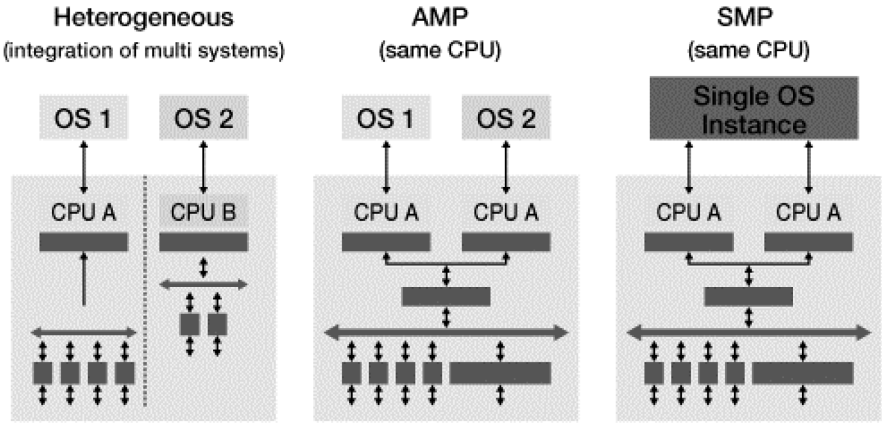


Figure 2: SMP vs. AMP configurations of multicore designs [40]

Hybrid Prototyping [41] is a modeling framework that aims to provide early, fast, and cycle-accurate models of SMP designs which are widely used in modern embedded and networking SoCs. The fundamental idea is to simulate a design with multiple processor cores by creating a *M*ulticore *E*mulation *K*ernel (MEK) in software on top of a single physical instance of the processor. The MEK switches between cores and manages the logical simulation times of the individual processor cores. Since the application executes on exactly the same core as it is targeted for, the simulation is cycle-accurate. As a result, we can achieve fast and cycle-accurate simulation of multicores, thereby overcoming the accuracy concerns of virtual prototyping and the scalability issues of FPGA prototyping. Using hybrid prototypes, multicore system

designers can create concurrent applications and accurately analyze the power and performance implication of their optimizations before the hardware is available.

2.1 Methodology

Hybrid prototyping time-multiplexes several emulated cores on a single physical target core. The principal idea is to simulate a design with multiple processor cores by creating an emulation kernel in software on top of a single physical instance of the processor core. Since the application executes on exactly the same core as it is targeted for, the simulation is cycle-accurate. The core and the additional simulation infrastructure can fit on a single FPGA chip, making it very cost effective in contrast to full system prototyping in FPGA. It supports the execution of any multi-tasking ANSI C/C++ application.

Since the application executes on exactly the same core as it is targeted for, the estimation accuracy is 100%, in contrast to binary translation. As opposed to source-level annotation techniques, there is no need for availability of source code or knowledge of the core datapath, since the application binary runs directly on the target core. Finally, the at-speed execution of application tasks in our technique provides significant speedup over cycle-accurate software simulation.

Figure 3 uses a simple example to illustrate the concept of multi-core simulation on a hybrid prototype. We assume that the design consists of multiple cores, communicating using inter-core communication primitives, such as simplex channels. The synchronization in the channels between the threads (mapped to different cores) is modeled using events. We assume a classical discrete event model, in which an event is consumed by a waiting thread, or lost if no thread is waiting at the logical time of notification.

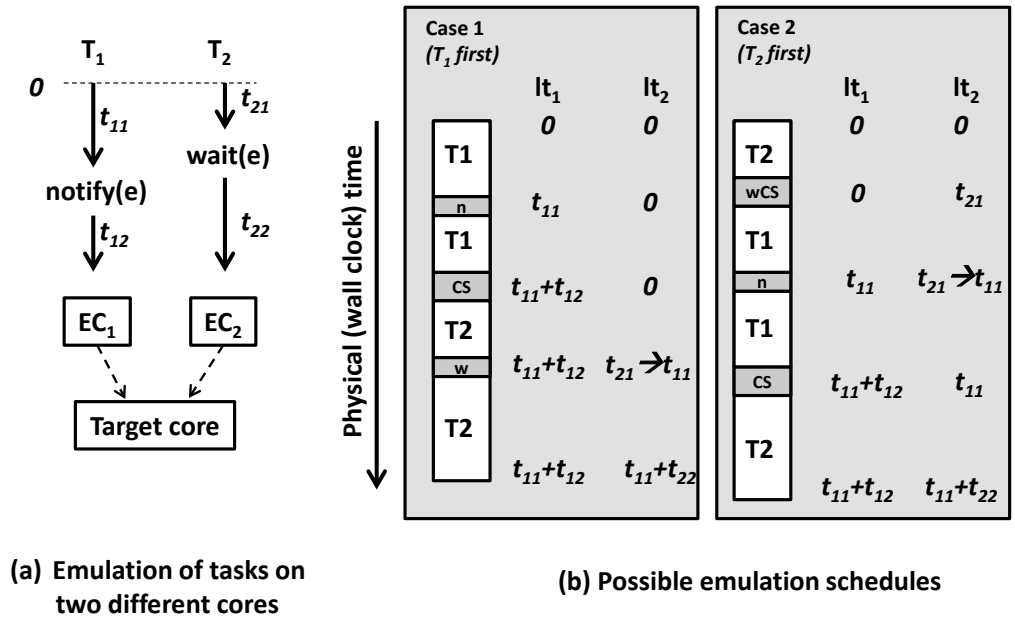


Figure 3: Simple example of simulation with a hybrid prototype

Figure 3(a) shows the design with two cores, each executing a single thread. Thread T_1 executes on core EC_1 for time t_{11} and notifies a global event *e*. After notification, it executes for another t_{12} units and terminates. Thread T_2 executes on core EC_2 (of the same type as EC_1) for time t_{21} ($< t_{11}$) and waits for the global event *e*. After *e* is notified (by T_1), it executes for another t_{22} units and terminates. Both tasks are assumed to start at the same time. The cores, EC_1 and EC_2, are simulated on the target core, which is of the same type as EC_1 and EC_2, and hosts the MEK.

Figure 3(b) shows two possible simulation schedules on the target core. A thread may be in four possible states: RUNNING, READY, BLOCKED or TERMINATED. The MEK maintains the logical times, lt_1 and lt_2, of the two emulated cores EC_1 and EC_2, respectively. The logical time for an emulated core is the time until which the core has been simulated since the beginning of system simulation. At logical time 0, the MEK may pick either EC_1 or EC_2 to simulate first. If the MEK schedules EC_1 to be simulated first, it runs T_1 on EC_1 until *e* is notified. The MEK saves the event's notification and its logical timestamp t_{11}. Since event notification is non-blocking in a

discrete event model, the MEK allows T_1 to execute until it is terminated. Once T_1 is terminated, the MEK does a context switch (CS) and runs T_2 from its logical time 0 until it reaches *wait(e)* at logical time t_{21}. At this point the MEK checks for any notifications of *e* that were made after logical time t_{21}. Indeed, since $t_{11}>t_{21}$, the MEK finds that *e* was notified by T_1 before T_2 started waiting for it. As such the MEK updates the logical time of EC_2 to t_{11} to model T_2 being blocked on the wait from t_{21} to t_{11}. Finally, T_2 is resumed and runs to completion.

If the MEK schedules EC_2 to be simulated first (Case 2), it runs T_2 on EC_2 from EC_2's logical time 0 until it reaches *wait(e)* at EC_2's logical time t_{21}. Since no notifications of *e* are found, the MEK stores the wait on event *e* with timestamp equal to t_{21} and blocks T_2. It then does a core context switch from EC_2 to EC_1. To emulate EC_1, the MEK runs T_1 from EC_1's logical time 0 until the notification of *e* at EC_1's logical time t_{11}. Upon notification, the MEK checks if there are any pending waits on event *e* at or before logical time t_{11}. Indeed, task T_2 is blocked since EC_2's logical time t_{21} ($< t_{11}$) on *e*. Therefore, the MEK unblocks T_2 and updates EC_2's logical time to t_{11} in order to account for the blocking time. The MEK continues simulating EC_1 until termination of T_1, followed by a context switch to EC_2 and its simulation until termination of T_2.

2.2 Modeling Framework

Figure 4 illustrates the modeling framework of the hybrid prototyping. A hybrid prototype is a combination of software and hardware components. The hardware component is the target core which is physically implemented in FPGA. And the software component consists of three layers: the MEK, software and hardware models.

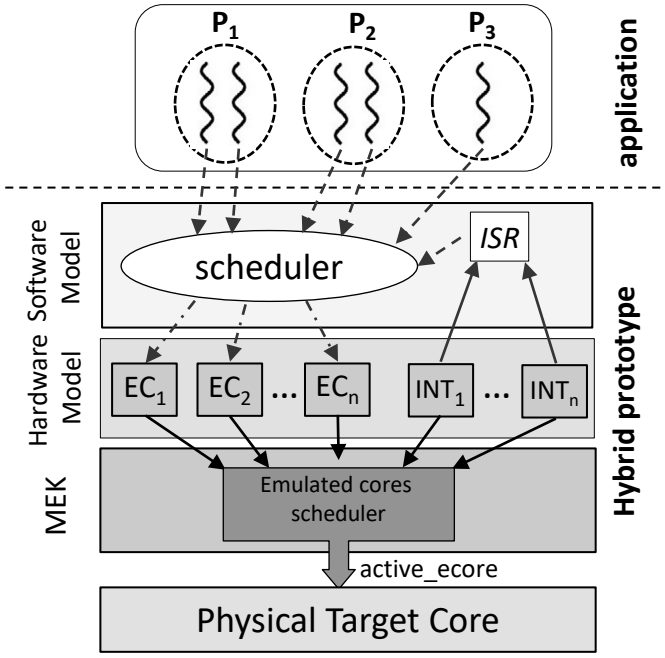


Figure 4: The hybrid prototyping framework

The MEK implements primitives for the management of discrete events and logical times of the emulated cores. It provides services to instantiate emulated cores and communication channels. This layer consists of discrete event model, shared resource (SR), hardware timer and emulated cores scheduler. The MEK defines primitives for event management using basic notify/wait concept. The context switching between emulated cores during simulation is done by the emulated core scheduler. It stores the context (stack, registers and state) of the running thread and loading the context of

the next ready thread. Timer primitives is also defined to provide a simple API to control the execution time.

The hardware layer models emulated SMP cores that are responsible for executing the user application threads, memory hierarchy and hardware interrupt sources. It provides services to allocate/deallocate a thread to a specific emulated core. The hardware design such as the number of processors, hardware interrupts, processors frequencies, Fast Simplex Link (FSL) (for modeling of simplex channels for point-to-point communication between the emulated cores) and etc. are implemented as an API on top of the MEK primitives in this layer

The software model layer implements OS primitives for scheduling and communication on top of the hardware model. It provides models of priority-based preemptive scheduler - which is responsible for scheduling threads on the emulated cores -, *I*nter-*P*rocess *C*ommunication (IPC) services, and *I*nterrupt *S*ervice *R*outines (ISR). The software model is the layer of the system which interacts directly with the user application. It defines thread management primitives (e.g. pthread_create, sleep, etc.), message queue, conditional variables and other essential services needed by the user application.

The hybrid prototyping simulates the execution of concurrent tasks on independent cores by dynamically scheduling the processes on the emulated cores. As it is shown in Figure 4, only one thread can be run on the physical processor at a time.

2.3 Summary

In this chapter we introduced the hybrid prototyping methodology and the idea behind it. We then explained the modeling framework and described different layers of the hybrid prototype. Hybrid prototype consists of three layers: software model layer which is responsible for thread scheduling, hardware model layer which is providing primitives for instantiating the emulated cores and the MEK which defines primitives for the management of discrete events and provides timer API required for managing the logical times for the emulated cores. In the next chapter we will talk about the MEK and its role in the hybrid prototyping.

Chapter 3

Multicore Emulation Kernel

Figure 5 shows the MEK structure. The MEK provides simulation primitives for the management of events and shared resources. The most important part of the MEK is the emulated cores scheduler which is responsible to switch context between emulated cores. The timer primitive is also defined to provide a simple API to control the execution time.

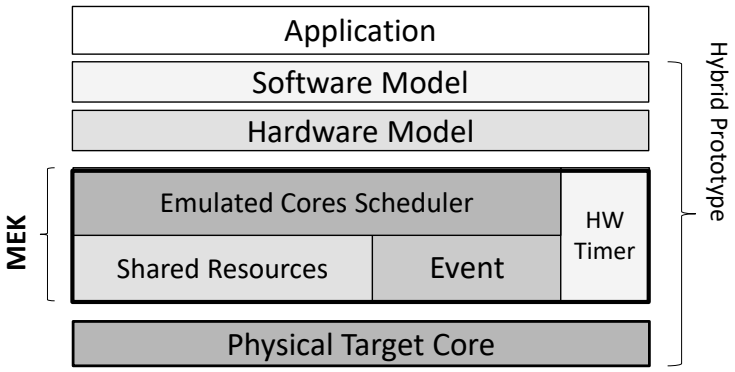


Figure 5: The MEK structure in the

3.4 Hardware Timer Controller

The MEK uses a hardware timer (XPS Timer [16]) controller to measure the execution time in CPU cycles. The hardware timer driver provides a simple API to control the timer. The XPS Timer is organized as two identical timer modules. Each timer module has an associated load register that is used to hold the value for the counter. The MEK defines timer class to provide a simple API to control the execution time. Listing 1 shows the timer class which is providing essential methods to work with both timer modules in the hardware timer.

```
Class Timer{
   public:
      Timer(u16 deviceId);
      void start(u16 timerId);
      void stop(u16 timerId);
      u32 getValue(u16 timerId);
   private:
      XTmrCtr XPS_Timer;
      int controlTime;
}
```

Listing 1: Timer class

For measuring the execution time of a block of code, the MEK starts the timer before the block by calling *start* method. At the end of the block, the MEK calls *stop* method to stop the timer and reads the timer's value by calling *getValue* method. To have accurate time measurement, the MEK must account for control time, which is the CPU time consumed for starting and stopping the timer, without any operations between them. The Listing 2 shows calculates the control time.

```
u32 get_control_time() {
   start(0);
   stop(0);
   return XTmrCtr_GetValue(&XPS_Timer, 0);
}
```

Listing 2: Control time

3.5 Event

We have implemented a classical discrete event model, where an event is consumed by a waiting thread, or is lost if no thread is waiting at the logical time of notification. Each event maintains a waitlist which is sorted by the logical time of wait calls. The item type in the list is a pair of id and timestamp. As the name implies, the waitlist is the list of all emulated cores that their running threads are waiting on the event. Listing 3 shows the event class in the MEK.

```
class Event {
   public:
      Event();
      ~Event();
      void wait();
      void notify();
   private:
      Boost::List< pair<int,int> >* waitList;
      void insertWait(int id, int timeStamp);
}
```

Listing 3: Event class

The MEK defines notify/wait methods for event management. Listing 4 shows the pseudo code for event's wait method. Each kernel call is surrounded by KERNEL_CALL_START and KERNEL_CALL_END functions. The first function, stops the timer to mark the end of user code and the start of execution of the kernel call. It also uses the timer value to update the logical time of the running emulated core. While the second function starts the timer before the kernel call returns to the user core. Wait operation puts the emulated core in suspended state (line 2), adds the wait to the event's waitlist (line 3) and gives the control of the physical processor to the emulated core on the top of the busy queue (next busy emulated core) (line 4). *Active_ecore* is the emulated core which is actually running on the physical processor and consume CPU cycles.

```
void event::wait() {
   1: KERNEL_CALL_START();
   2: suspend(active_ecore);
   3: this->waitlist.insert(active_ecore);
   4: run_next_ready_ecore();
   5: KERNEL_CALL_END();
}
```

Listing 4: Event's wait pseudo code

Listing 5 shows the pseudo code for event notification. An event cannot be notified unless all other emulated cores have been simulated at least until the current notifying emulated core's logical time. Therefore, the notification is committed when the logical time of the notifier is equal to MIN_SIM_TIME which is the minimum logical time among none suspended emulated cores. If the logical time of the notifier is not equal to MIN_SIM_TIME, the emulated core yields its execution turn (line 3) to make sure all other emulated cores will be simulated at least until its current logical time. It ensures all waits and all notifications of an event (from different emulated cores) are being processed in order. Notifying an event brings the first emulated core in the waitlist to the ready state (lines 4-6). The waiting emulated core is inserted back into the idle queue and the wait is deleted from the event's waitlist. It then updates the logical time of the waiting emulated core to current logical time only if the logical time of the waiting emulated core is less than the notify time (line 8). It implies that the waiting emulated core had waited on the event at a logical time before the current logical time of the notifying emulated core (active_ecore). In such a scenario the idle time of the waiting emulated core is incremented by the difference between the notifying emulated core's logical time (active_ecore) and the waiting emulated core's idle time (line 7). If no thread is waiting on the event, the notification will be lost.

```
void event::notify() {
   1: KERNEL_CALL_START();
   2: while(active_ecore.logicalTime != MIN_SIM_TIME)
   3:    yield();
   4: c = this->waitlist.first();
   5: if(c!=null) {
   6:  wakeup(c);
   6:  if(active_ecore.logicalTime > c.logicalTime){
   7:    c.idleTime+=active_ecore.logicalTime-c.logicalTime;
   8:    c.logicalTime = active_ecore.logicalTime;
   9:  }
   10:}
   11: KERNEL_CALL_END();
}
```

Listing 5: Event's notify pseudo code

It is important to note that on every update of the logical time after blocking, the difference between the new logical time and the task's logical time indicates the idle time for the corresponding core. Figure 6 shows the busy/idle time for tasks T_1 and T_2. In this case, when T_1 notifies event *e*, the MEK updates T_2's timestamp to t_{11} and increases T_2's idle time by $t_{11} - t_{21}$.

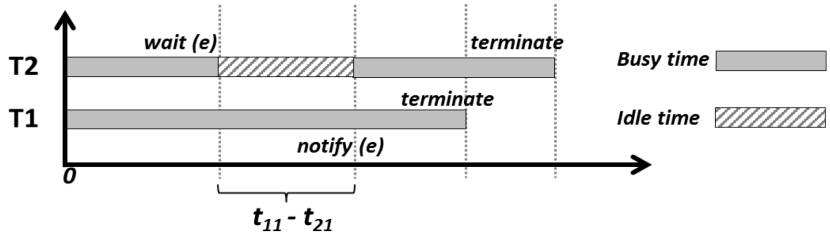


Figure 6: The busy/idle time

Figure 7 shows a design with two emulated cores, each executing a single thread. Both threads are assumed to start at the same time. Thread T_1 executes on emulated core EC_1 for time t_{11} and notifies a global event *e*. After notification, it executes for another t_{12} units and terminates. Thread T_2 executes on emulated core EC_2 for time t_{21} ($< t_{11}$) and waits for the global event *e*. After *e* is notified (by T_1), it executes for another t_{22} units and terminates. For simplicity, we assume that T_1 and T_2 are bound to EC_1 and EC_2, respectively.

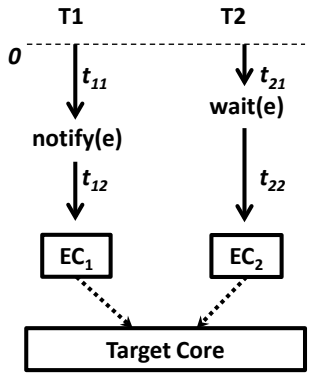


Figure 7: Simulation of two tasks on two emulated cores with hybrid prototyping

Figure 8 shows two possible simulation schedules on emulated cores. The MEK maintains the logical times, lt_1 and lt_2, and idle times, $idle_1$ and $idle_2$, of the two cores EC_1 and EC_2 respectively. At logical time 0, either EC_1 or EC_2 can be simulated first. If EC_1 is picked first to be simulated (Case 1), it executes thread T_1 for time t_{11} until *e* is notified. At this logical time the MIN_SIM_TIME is equal to zero because the thread T_2 has not yet been simulated and EC_2's logical time is zero. Therefore, EC_1 yields the execution to EC_2. The MEK switches the context and simulates EC_2. Thread T_2 then executes on emulated core EC_2 for time t_{21} and waits for *e*. Wait on event *e* puts EC_2 in suspended state and the next ready emulated core (EC_1) takes control of the processor. Based on the definition, MIN_SIM_TIME is the minimum of the logical time of all none suspended emulated cores. Therefore, now MIN_SIM_TIME is equal to the logical time of the EC_1. The MEK does the context switch and simulates the EC_1. EC_1 executes thread T_1 and it notifies the event *e*. After *e* is notified, T_1 executes for another t_{12} units and terminates. Notifying event *e* by EC_1 puts EC_2 in ready state. As EC_2 has waited before it gets notified, the MEK updates the logical time of EC_2 to t_{11} to model T_2 being blocked on the wait from t_{21} to t_{11}. It also updates the EC_2's idle time to $t_{11} - t_{21}$. Finally, T_2 is resumed and runs to completion.

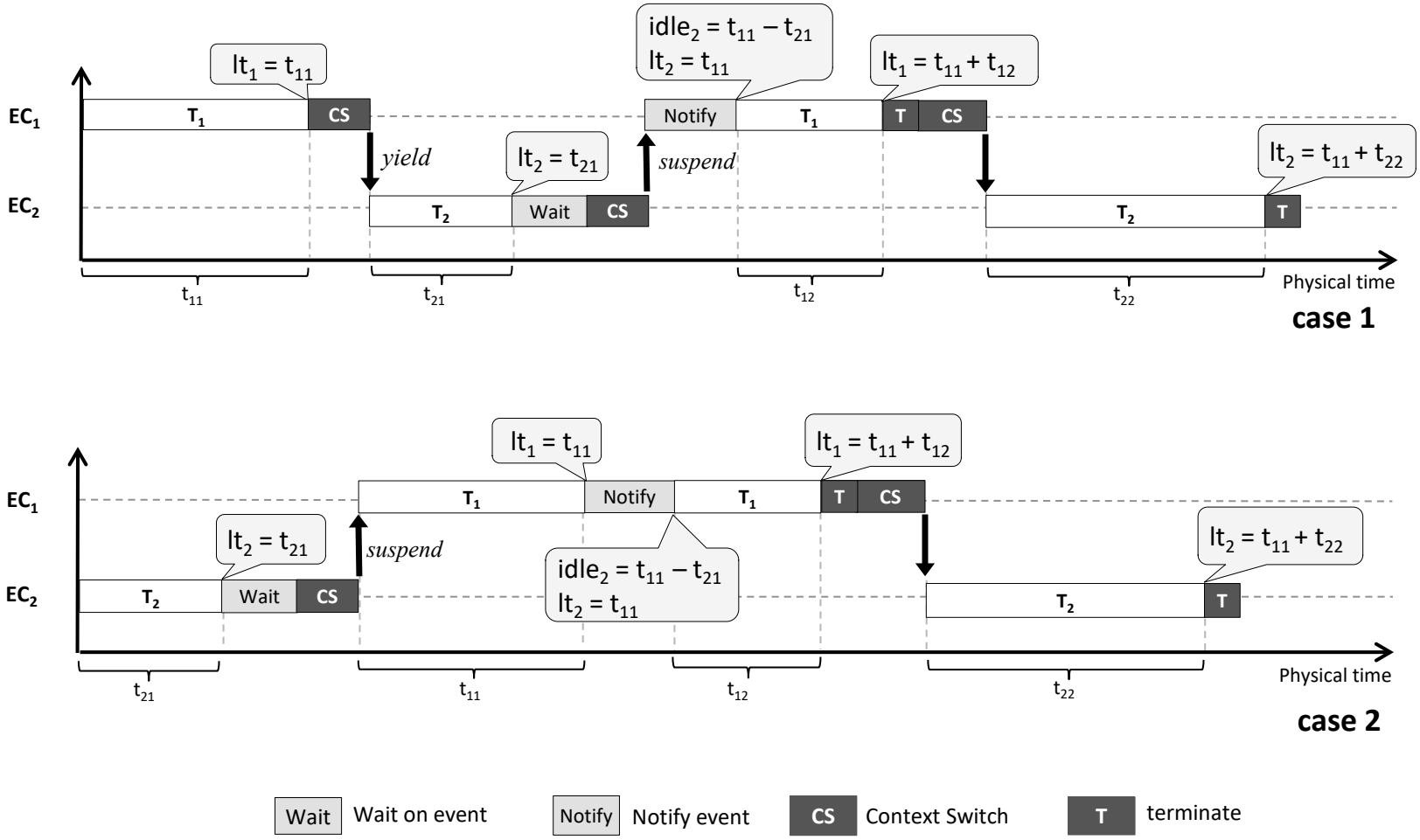


Figure 8: Possible emulation schedules

If EC_2 is scheduled to be simulated first (Case 2), it runs T_2 from EC_2's logical time 0 until it reaches wait on event *e* at EC_2's logical time t_{21}. The MEK suspends the EC_2 and switches the context to run EC_1. EC_1 executes thread T_1 from EC_1's logical time 0 until the notification of event *e* at EC_1's logical time t_{11}. Upon notification, since the MIN_SIM_TIME is equal to EC_1's logical time (EC_1 is the only none suspended emulated core) the MEK removes EC_2 from the event's waitlist and updates the logical time of the waiting emulated core (EC_2) to t_{11} and EC_2's idle time to $t_{11} - t_{21}$ since $t_{21} < t_{11}$. The thread T_1 is simulating until it terminates, followed by a context switch to EC_2 and its simulation until the termination of T_2. By the end of the simulation (in both cases) the hybrid prototype reports EC_1's logical time as $t_{11} + t_{12}$, EC_1's idle time is equal to 0, EC_2's logical time as $t_{11}+ t_{22}$ and finally EC_2's idle time as t_{11} - t_{21}. The busy time for each emulated core is the difference of the logical time and the idle time of the emulated core. Therefore, the MEK reports the EC_1's busy time as $t_{11} + t_{12}$ and EC_2's busy time as $t_{21} + t_{22}$. As we will see later in next chapters, the busy time is needed for estimating the energy consumption of cores.

3.6 Shared Resources

The MEK ensures that accesses to a shared resource from different emulated cores is processed in order. It means an emulated core cannot read or update a shared resource unless all other emulated cores have been simulated at least until its current logical time. The MEK layer provides primitives for shared resources. Listing 6 and Listing 7 describe write and read methods respectively. The emulated core can access the shared resource if the emulated core's logical time is equal to MIN_SIM_TIME (line 2). Otherwise, it yields (line 3) to the next ready emulated core to be simulated.

```
void SR::write(T newvalue) {
   1: KERNEL_CALL_START();
   2: while(active_ecore.logicalTime != MIN_SIM_TIME)
   3:    yield();
   4: this->value = newvalue;
   5: KERNEL_CALL_END();
}
```

Listing 6: Shared resource write pseudo code

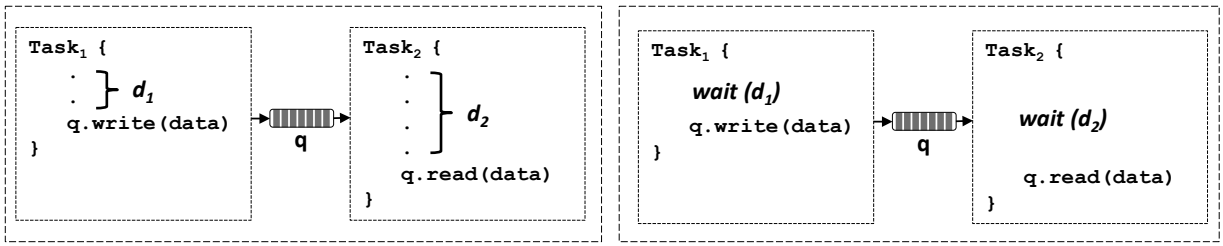
```
T SR::read() {
   1: KERNEL_CALL_START();
   2: while(active_ecore.logicalTime != MIN_SIM_TIME)
   3:    yield();
   4: KERNEL_CALL_END();
   5: return this->value;
}
```

Listing 7: Shared resource read pseudo code

It is important to note that the shared resource model only ensures that all accesses to a shared resource are being processed in order of logical time. It cannot be used to solve mutual exclusion in critical sections. Synchronization is required at the entry and exit of the critical section to ensure exclusive use.

3.7 Emulated Core Scheduler

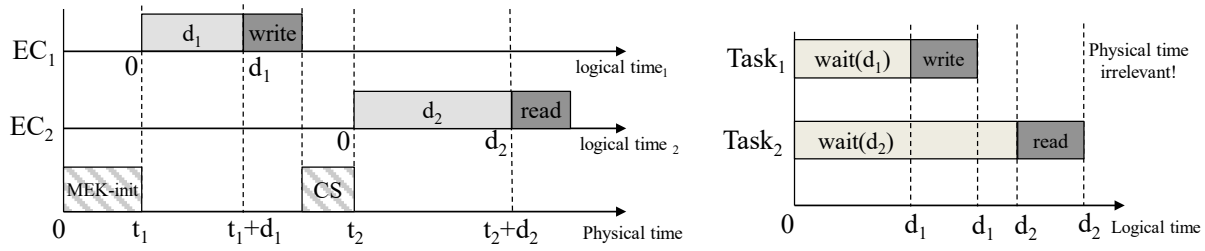
The implementation of discrete event and logical time in the hybrid prototyping are different from a pure software discrete event simulator. A discrete event simulator like SystemC provides primitives of logical time (wait) and events (wait/notify). But in order to model execution time, the user has to advance logical time. In hybrid prototyping, there is the notion of a single global logical time, but multiple logical timelines (one for each emulated core). The time on these logical timelines is advanced when the emulated core is executing its context. Logical time is completely different from the physical time (wall clock). It represents the time by which actions happen on the system.



(a) The MEK discrete event simulation (b) Classical discrete event simulation

Figure 9: Classical vs the MEK discrete event simulation

Figure 9 shows the difference between a classical discrete event and the MEK discrete event simulation. As it shows $task_1$ writes into message queue after d_1 unit of time. $Task_2$ reads from the message queue after d_2 unit of time. In the MEK discrete event simulation the user doesn't need to advance the logical time as both $task_1$ and $task_2$ execute on the emulated cores. In contrast, in the classical discrete event simulation the user needs to apply time primitive (wait) to advance the logical time.



(a) The MEK discrete event simulation (b) Classical discrete event simulation

Figure 10: Classical vs MEK discrete event simulation

Figure 10 shows how discrete events are simulated with the MEK. After the MEK initialization phase, $task_1$ is scheduled to run on EC_1. It executes for d_1 unit of time and then writes into message queue and terminates. The MEK then switches to EC_2. After $task_2$ executes for d_2 unit of time it reads data from message queue and terminates. Therefore, the user doesn't need to take care of logical time and the MEK measures the logical time for each emulated core. On the other hand, in the classical discrete event simulator, the pending event set is typically organized as a priority

queue, sorted by event time. Following that, the simulator simulates the events and uses time primitives to handle logical times.

The emulated core scheduler is the most important part of the MEK. It is responsible for switching between available emulated cores. It uses two queues to keep track of busy and idle emulated cores. At the beginning of the simulation, all emulated cores are initialized and placed in the idle queue. The thread scheduler in the software model dispatches the ready threads on the idle emulated cores. When an emulated core gets assigned to a thread, it becomes busy and is placed at the end of the busy queue.

The MEK uses a *F*irst-*I*n-*F*irst-*O*ut (FIFO) scheduling policy to schedule emulated cores on the physical target core. In FIFO scheduling algorithm, an emulated core is *simulated* on the target core as long as the task mapped to the given emulated core is running. However, if the running thread on the emulated core terminates, blocks or voluntarily yields the emulated core, then the MEK switches to the next ready emulated core. Emulated cores don't switch instantaneously. After a thread blocks or terminates, the MEK must save the running thread's state before simulating another emulated core. The operation to save this state and restore another is known as context-switch. To perform the context-switch the scheduler stores stack, CPU registers and state for each thread. It must be noted that the MEK scheduler and the RTOS scheduler model are theoretically orthogonal entities.

3.8 Summary

In this chapter we explained the MEK layer. It is the fundamental idea of the hybrid prototyping. We introduced discrete event model, shared resource (SR), hardware timer and emulated cores scheduler. As we have seen, the emulated core scheduler is responsible for switching between emulated cores. In the next chapter we will see how emulated cores are modeled in the hardware model layer and are managed by the scheduler.

Chapter 4

Hardware Model Layer

Figure 11 shows the hardware model layer structure in the hybrid prototyping. The hardware model layer instantiates the emulated cores, channels and interrupt sources. It provides services to allocate/deallocate a thread to a specific emulated core. This layer also models the cache behavioral by using an on-chip hardware peripheral.

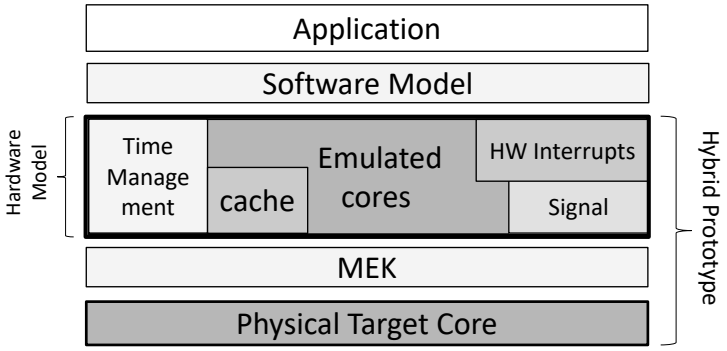


Figure 11: Hardware model layer structure

The hardware architecture includes the number and configuration of processors, hardware interrupt sources, and inter-core communication channels. The architecture is modeled on top of the MEK primitives in the hardware model. Listing 8 shows how

a simple hardware model described in a hybrid prototype with two emulated cores and a hardware interrupt.

```
void initialize() {
   1: hybrid_init(NUM_CORES,
         SCHED_TYPE,INIT_ROUTINE,CLOCK_DOMAINS);
   2: u16 Id = hw_model->create_int_source(PERIOD);
   3: hw_model->connect(Id, ISR_ROUTINE);
   4: hybrid_run();
}
```

Listing 8: Hardware description with hybrid prototyping

To initialize a hardware model of the design, *hybrid_init* method is used (line 1), where NUM_CORES is the number of emulated cores and SCHED_TYPE is the scheduling type which can be static or dynamic. INIT_ROUTINE is the start of the routine responsible to initialize all the application's thread and will be invoked by the prototype. Therefore, INIT_ROUTINE is the first thread which will be run by the MEK. If static scheduling is used, INIT_ROUTINE will be locked to the first emulated core. CLOCK_DOMAINS is an array of the frequencies which are going to be assigned (in order) to emulated cores.

The hardware model provides a *create_int_source* method to initialize an external hardware interrupt source where PERIOD determines the interval time between the interrupts (line 2). *Connect* method in hardware model connects the external device to ISRs routine where ISR_ROUTINE defines the ISRs function. At the end, the simulation will be run by calling hybrid_run method (line 3).

4.1 Emulated Cores

The hardware model provides simulation primitives for the management of the emulated cores. Each emulated core is responsible for executing the thread which has been assigned to it. The emulated core scheduler switches between different emulated cores and manages the logical simulation times and idle time of the individual emulated core. An emulated core is parameterized with:

1. *State* which can be IDLE, BUSY, RUNNING or SUSPEND
2. *Logical time*, the time until which the emulated core has been simulated.
3. *Idle time*, the time until which the emulated core has been idle.
4. *Clock Frequency*, the operating clock frequency which the emulated core works with.
5. *Running thread*, the thread which has been assigned to the emulated core to get executed.
6. *Awake* event, the event which is used to wake up the emulated core. As we will see later in chapter 5, the MEK uses this event to manage emulated cores' idle time. When an emulated core becomes idle it will run idle thread. By running the idle thread, the emulated core will wait on *awake* event and as a result it will be suspended. When the event is notified, the MEK wakes up the suspended emulated core and updates its logical time and idle time if needed.

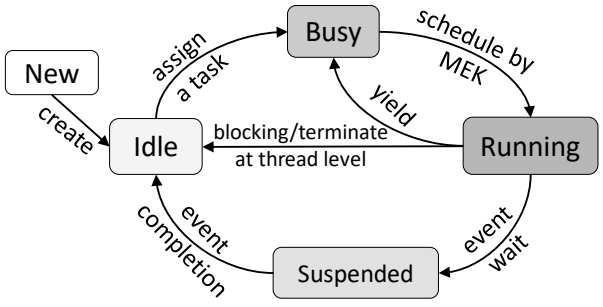


Figure 12: Emulated core life-cycle state diagram

The life-cycle of an emulated core is shown in Figure 12. An emulated core is initialized in the idle state; the emulated core scheduler puts it at the end of the idle queue. Idle emulated cores are waiting for threads to get assigned to them. After the RTOS model scheduler assigns a thread to an emulated core, it becomes *busy* and is placed at the end of the *busy queue*. When the emulated core is selected to run by the MEK, it is removed from the busy queue, changes its state to running and begins executing the assigned thread on the physical core. When the running thread on the emulated core blocks or terminates, the emulated core becomes idle and is placed at the end of the idle queue.

If the emulated core yields its execution (for example at event notification), it is placed at the end of the busy queue and the next ready emulated core, in the busy queue is moved to the running state. An emulated core is *suspended* if it waits on an event which has not yet been notified. The suspended emulated core is not scheduled to execute on the target core until it is unblocked by a notification. When the suspended emulated core is subsequently notified, it is placed back at the end of the idle queue and waits for the next ready threads.

4.2 Communication Models

There are three different ways of communication in a hybrid prototype. FSL is used to model communication between emulated cores. Signals are used to model hardware interrupt by connecting external hardware device to emulated cores. In SMP designs message queue is an efficient way of passing data between processes which is created in the shared memory. One program will create a protected memory portion which other processes can access. Figure 13 shows different types of communication in the hybrid prototyping.

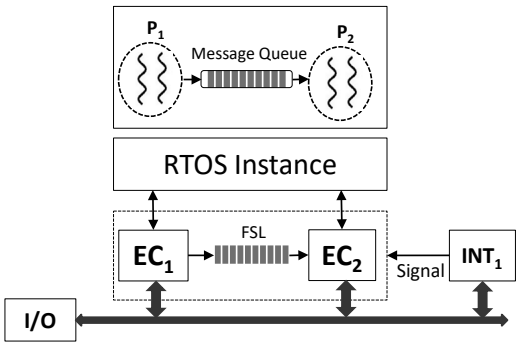


Figure 13: Communication models in the hybrid prototyping

4.2.1 Statically Scheduled MPSoCs

Static MPSoCs architecture can be modeled by a hybrid prototype. In such an architecture, there is no dynamic scheduling and each task is locked to a specific core and they can use inter-core communication channel such as FSL to communicate. FSL is a unidirectional point-to-point FIFO-based communication channel bus used to perform fast communication between Xilinx MicroBlaze [16] soft processor.

The basic simulation primitives of notify, wait, update and yield are powerful enough to build a complex communication models. In this section, we will describe modeling of simplex channels for point-to-point communication between the emulated cores called FSL. FSL is implemented as a circular buffer. Listing 9 shows the FSL class.

```
class FSL {
public:
  FSL(int len);
  void bwrite(const T value);
  T bread();
  bool isEmpty();
  bool isFull();
private:
  Event* ev_is_not_full;
  Event* ev_is_not_empty;
  SharedResource<bool> flag_not_empty;
  SharedResource<bool> flag_not_full;
  Cbuffer<SharedResource<T>>* cbuffer;
}
```

Listing 9: FSL class

The channel buffer is modeled as an array of shared resources of items which the user defined them. The head and tail of the circular buffer is maintained. Readers read from the head and writers write into the tail. The channel has boolean SR variables to indicate a full or empty state, as well as respective events that are notified whenever the buffer is read or written.

```
void bwrite(const T value) {
   1: KERNEL_CALL_START();
   2: while (!flag_not_full.Read())
   3:        this->ev_is_not_full->wait();
   4: SharedResource<T> sr(value);
   5: cbuffer->enqueue(sr);
   6: if (cbuffer->isFull())
   7:    flag_not_full = false;
   8: flag_not_empty = true;
   9: this->ev_is_not_empty->notify();
   10:KERNEL_CALL_END();
}
```

Listing 10: FSL's blocking write method

Listing 10 illustrates the pseudo code for a blocking write *(bwrite)* into the channel. At line 1, the timer is first stopped to mark the end of user code and the start of execution of the communication model. The timer value is then used to update the logical time of the caller emulated core. Since this is a blocking write, the writing task must wait as long as the channel is full. If the shared resource *flag_not_full* can be accessed by the task, it guarantees that the logical time of the emulated core is equal to MIN_SIM_TIME and all other emulated cores (including the reader of this channel) have been simulated at least until this time (line 2). If channel is full, the writer must block on the channel ev_is_not_full event (line 3). Wait on the event puts current emulated core in suspended state and the next ready emulated core takes control of the processor. The actual writing is subsequently done by copying over the data into the buffer's tail (line 5), updating the buffer full flag, if needed, (lines 6-8) and *ev_is_not_empty* event is then notified (line 9). Upon this notification, the

emulated cores which has been waiting on the channel will be waken up and the MEK will update the waiting emulated core's logical time and idle time if needed.

The blocking read method *(bread)* is the exact dual of *bwrite* as shown in Listing 11. After stopping the timer and updating its logical time (line 1), the reader checks if the channel is empty by reading the *flag_not_empty* shared resource (line 2). Similar to *bwrite*, if the shared resource *flag_not_empty* can be accessed by the task, it guarantees that the logical time of the emulated core is equal to MIN_SIM_TIME and all other emulated cores (including the reader of this channel) have been simulated at least until this time (line 2). If the channel is empty, the reader must wait on the *ev_is_not_empty* event (line 3). Otherwise, it proceeds to perform the actual data read by reading the tail of the circular buffer (line 4). As the circular buffer is an array of shared resources, the reader's logical time must be the same as MIN_SIM_TIME that the reader can read data from the buffer. After reading data, the MEK then updates the buffer full flag, if needed, (lines 5-7) and notifies *ev_is_not_full* (line 9). By notifying the event, the emulated core which has been waiting on the channel will be waken up and the MEK will update the waiting emulated core's logical time and idle time if needed.

```
T bread() {
   1: KERNEL_CALL_START();
   2: while (!flag_not_empty.Read())
   3:         this->ev_is_notEmpty->wait();
   4: T item = cbuffer->dequeue().Read();
   5: if (cbuffer->isEmpty())
   6:         flag_not_empty = false;
   7: flag_not_full = true;
   8: this->ev_is_not_full->notify();
   9: KERNEL_CALL_END();
   10:return item;
}
```

Listing 11: FSL's blocking read method

Figure 14 shows how the MEK manages tasks when they use communication models. We use this simple example to explain communication models in details. There are two possible scheduling for this example. It is assumed that the length of the channel is only 1 data item. The MEK can run either T_1 or T_2 first.

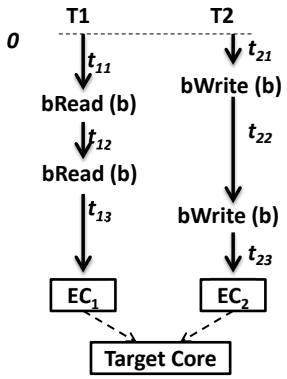


Figure 14: Simple example of using communication model by two emulated cores in the hybrid prototyping

Figure 15 illustrates the case when the MEK runs T_1 first. T_1 runs till reaches *bread* at time t_{11}. Since no data items are found, the MEK blocks T_1 and switches to task T_2. T_2 runs and at time t_{21} writes the data item into the channel and notifies the *ch→ev_ write*. So the MEK unblocks T_1. After this notification, T_2 executes for another t_{22}. Then, it wants to write for the second time. As there is no space in the channel, the MEK blocks T_2, set the T_2's timestamp to $t_{21} + t_{22}$ and switches back to T_1. T_1 reads the data item from the channel and notifies the *ch→ev_read*. Considering this notification, the MEK unblocks T_2. T_1 can execute for t_{12} till reaches the second read while it is empty. So the MEK sets T_1's timestamp to $t_{11} + t_{12}$ and switches to T_2. When T_2 writes in the channel, the MEK updates the T_1's timestamp to T_2's timestamp ($t_{21} + t_{22}$) because T_2's timestamp is bigger than T_1's. T_2, then, executes for t_{23} and terminates. The MEK switches back to T_1. It reads from the channels, executes for t_{13} units and then terminates. At the end of the simulation, the MEK reports T_1's timestamp as $t_{21} + t_{22} + t_{13}$ and T_2's timestamp as $t_{21} + t_{22} + t_{23}$.

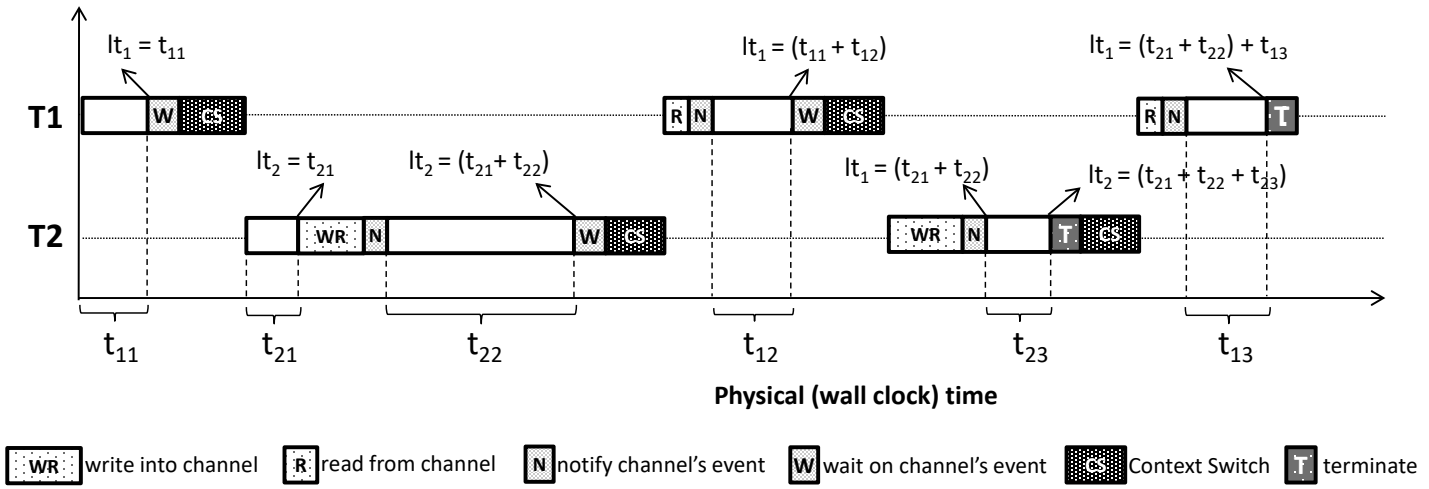


Figure 15: Simple example of using communication model by two tasks

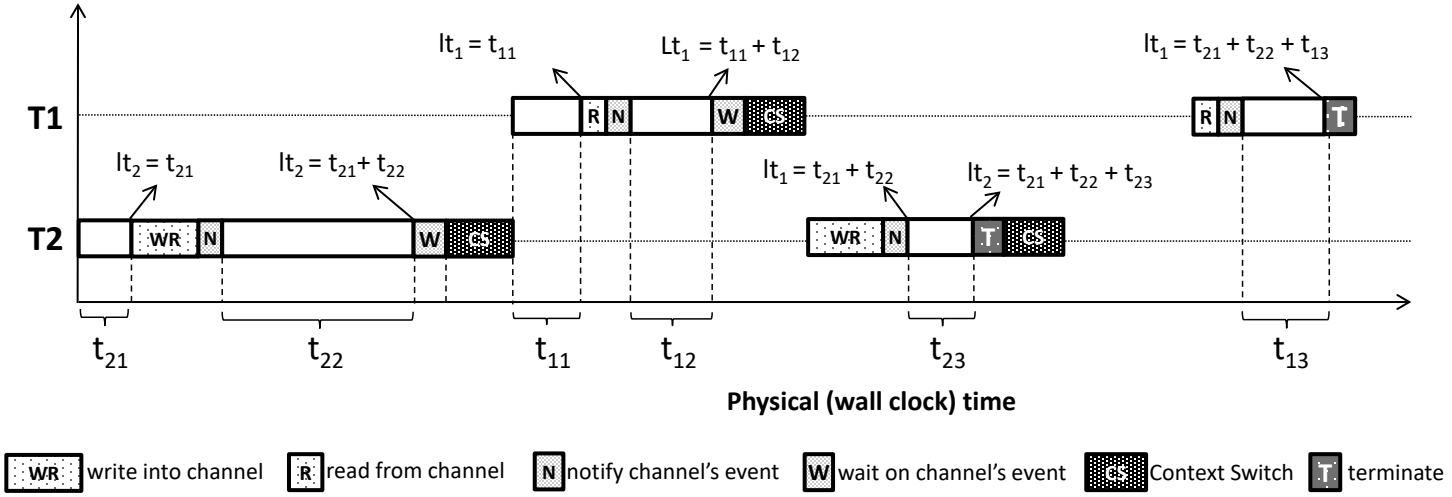


Figure 16: Simple example of using communication model by two tasks

Figure 16 shows the case when the MEK executes task T_2 first. When the MEK schedules T_2 to be simulated first, it runs T_2 until it reaches *bwrite* at logical time t_{21}. Since the channel is empty, T_2 can writes the data item into the channel. So the MEK writes the data item and updates T_2 timestamp to t_{21}, store the t_{21} as block's wire time and notifies *ch→ ev_ write*. By this notification, the MEK changes the state to ready for all the tasks that are blocked on this channel and have been blocked. T_1 then runs until it reaches the second write. As there is no space in the channel (channel size is 1) data item, the MEK blocks T_1, updates T_1's timestamp to $t_{21} + t_{22}$ and does a context switch.

The MEK then runs T_2 until reading data items from the channel. Before reading the data item, the MEK updates the T_2's time stamp to t_{11}. As the channel is not empty, T_2 can read data item. After reading operation, the *ch→ ev_ read* will be fired. Therefore, the MEK unblocks T_1. In this case, the MEK does not update the T_1's timestamp because T_1's timestamp is greater than data item's write time ($t_{11} > t_{21}$).

The MEK continues simulating T_1 until the second read from the channel. Before any operations, the MEK updates T_1's timestamp to $t_{11} + t_{12}$. T_1 reads the data item from the channel and as there are no more data items in the channel, the MEK blocks T_1 and switches the context to T_2. The channel has empty space and T_2 can write the data item.

Upon *ch→ ev_ write* notification, the MEK unblocks the T_1 because T_1 has been blocked on this channel. Also the MEK updates the T_1's timestamp to $t_{21} + t_{22}$ since the T_2 logical time is greater than T_1 logical time ($t_{21} + t_{22} > t_{11} + t_{12}$) on notify *ch→ ev_ write*. The MEK continues simulating T_2 until it terminates and updates its timestamp to $t_{21} + t_{22} + t_{23}$. The MEK then switches to T_1 and simulates it until termination of T_1. At the termination point, the T_1's timestamp is updated to $t_{21} +$

$t_{22} + t_{13}$. So, regardless which task is scheduled first by the MEK, the final results are identical.

4.2.2 SMP Architecture

Processes can communicate with each other using inter-process communication primitives provided by the operating system. One process can create a protected memory portion which other processes can access it. Message queue is an efficient means of passing data between processes which is provided by software model layer in the hybrid prototyping. A message queue is a way for applications to send messages between one another in order to reliably communicate. We will describe message queue in the next chapter.

4.2.3 Interrupt to Processor

Signals are a useful synchronization mechanism to connect a hardware interrupt sources to a core. It will be used to model hardware interrupts in the hybrid prototyping. As opposed to events, notification on signals won't be lost when no waiting thread is found on the signal. The signal structure consists of two lists: *pendinglist* and *waitlist*. As the name suggests, pendinglist is the list of logical time when the signal was initiated. Waitlist is the list of all threads that are waiting on the signal. The MEK defines signal/wait methods for signal management.

Listing 12 illustrates the pseudo code for a wait on signal. The MEK first tries to find a notification for the signal that has occurred at a logical time before the current logical time of the core executing the wait (line 2). A notifying thread may have been simulated before the waiting thread and the notification may be present in the pendinglist. If the notification is found, it will be removed from the signal's pendinglist and the caller proceeds (lines 2-3). If a notification is not found, the RTOS must allow

other thread to be run, so that a potential notify on the signal is executed. The wait is added to the signal's waitlist and the waiting thread is BLOCKED by the RTOS (lines 4-7). The RTOS reschedules the ready threads on the emulated cores and changing the context (lines 7-8) to the next emulated core.

```
void signal::wait() {
   1: KERNEL_CALL_START();
   2: if(∃ n ∈ pendinglist, n->lt < active_ecore.lt)
   3:    delete (pendinglist, n);
   4: else {
   5:    Thread *t = active_ecore.running_thread;
   6:    add (waitlist, t);
   7:    suspend(t);
   8:    run_next_ready_ecore();
   9: }
   10: KERNEL_CALL_END();
}
```

Listing 12: Signal's wait method pseudo code

Listing 13 shows the pseudo code for signal notification. A thread cannot notify a signal unless all other threads have been simulated at least until the current logical time. Therefore, the notification is committed when the logical time of the notifying core is equal to MIN_SIM_TIME (lines 2-3). The thread yield its simulation turn to make sure all threads have been simulated at least until the current logical time and ensure all waits and notifications have been processed in order. If the current time is equal to MIN_SIM_TIM, the RTOS then looks for the first thread that has been waiting on the signal (line 7). If such a wait is found, the RTOS wakes the thread up and removes the wait from the waitlist (lines 8-9). If no waiting thread is found, it is possible that the thread which might call the wait at a later logical time has not yet been simulated till the wait call. Therefore, the notification is added to the pendinglist of signal (lines 4-5).

```
void signal::signal() {
   1: KERNEL_CALL_START();
   2: while(active_ecore.logicalTime != MIN_SIM_TIME)
   3:    yield();
   4: if(waitlist.isEmpty())
   5:    pendinglist->add(active_ecore.logicalTime);
   6: else {
   7:    w = waitlist.first();
   8:    wakeup(w);
   9:    delete(waitlist, w);
   10: }
   11: KERNEL_CALL_END();
}
```

Listing 13: Signal's signal method pseudo code

4.3 Hardware Interrupt Handling

Hardware interrupts are issued by external peripherals, leading to execution of an ISR. ISRs are treated as special threads that have the highest priority. The signaling of an interrupt event notifies the ISR; the RTOS will run the ISR on the first available emulated core (either idle or busy with lowest priority task). Upon the ISRs execution the corresponding interrupt handler is called.

Signal delivery is not instantaneous. When a signal is posted to ISR from a peripheral, the signal flagged as pending and added in signal's pendinglist and the RTOS schedules the ISR to run on the first available emulated core. When ISR is next scheduled to be run, pending signals are checked and appropriate action is taken.

An external device which is generating hardware interrupt (HW_INT) is modeled as a special emulated core in the hybrid prototyping. Like a regular emulated core, it is scheduled by emulated cores scheduler and it is capable to run any thread. The difference is that it only runs a thread which is locked to it. This thread describes the HW_INT's behavior and defines in which circumstances the interrupt must occur. The other difference is that the MEK only update the HW_INT's logical time when it sleeps for a given length of time. The MEK simulates the sleep operation which

means the thread is not really blocked on the sleep. The MEK just advances the logical time of the emulated core as much as the given length of time. Therefore, the HW_INT's is always busy and its idle time is zero. For instance, Listing 14 shows a thread running on a HW_INT which is generating interrupts at a fixed time intervals (t_{INT}).

```
void interrupt() {
   1: while (true) {
   2:    pthread_sleep(tINT);
   3:    int_sig->signal();
   4:    yield();
   5: }
}
```

Listing 14: Simple interrupt thread pseudo code

As Listing 14 illustrates, the peripheral waits for t_{INT} milliseconds and then sends a signal on *int_sig* signal. At the end, it yields its execution turn to another thread to ensure all threads will be simulated until the signal call.

4.4 Multi-Clock Domains

Multiple clock domains are often used in power-efficient designs. These designs might run different cores at different clock domains. The hybrid prototyping provides multiple clock domains by running each emulated core with different clock frequencies [42]. As the MEK calculates the execution time in CPU cycles, the real execution time can be easily obtained by multiplying the number of cycles with the clock period of each core.

The MEK uses a hardware timer to measure the execution time in CPU cycles. The timer's value is used to manage the emulated core's logical time. It can be measured either in CPU cycles or real execution time (in milliseconds). To obtain execution time in millisecond, a clock frequency (f) should be assigned to each

emulated core. Equation 1 shows how the real execution time can be easily obtained by multiplying the number of cycles with the clock period of each emulated core.

$$execution\ time_{ms} = CPU\ Cycles \times \frac{1}{emulated\ core's\ frequency} \quad (1)$$

Figure 17(a) shows how the MEK maintains the logical times lt_1 and lt_2 on emulated cores EC_1 and EC_2 respectively. It is assumed that the length of the channel is only 1 item and EC_1 and EC_2 are running with frequencies f_1 and f_2 respectively. The MEK may pick either EC_1 or EC_2 to simulate first. Figure 17(b) illustrates the case when EC_1 is picket by the MEK. The X-axis shows the physical time measured by the hardware timer in CPU cycles. As this figure shows, the MEK runs T_1 on emulated core EC_1 until it reaches *bread* at time t_{11}. Since no data is found in the channel, the MEK update the lt_1 to $t_{11} \times f_1$, blocks T_1 and switches to task T_2. T_2 runs for t_{21} unit, writes data into the channel then notifies event *ch→ev_ write*. As a result, the MEK unblocks T_1. Right after this notification, T_2 executes for another t_{22} unit. Then, it tends to write for the second time. However, as there is no space in the channel, the MEK blocks T_2, set the lt_2 to $(t_{21} + t_{22}) \times f_2$ and switches back to T_1. T_1 reads data from the channel and notifies event *ch→ev_ read*. Considering this notification, the MEK unblocks T_2. As T_1 can execute for t_{12} unit till reaches the second read while the channel is empty, the MEK sets lt_2 to $(t_{11} + t_{12}) \times f_1$ and switches to T_2. When T_2 writes into the channel, the MEK updates the lt_1 to lt_2's value. This happens because lt_2 is bigger than lt_1. T_2, then, executes for t_{23} unit and terminates. The MEK switches back to T_1. It reads from the channels, executes for t_{13} units and then terminates. At the end of the simulation, the MEK reports lt_1 as $(t_{21} + t_{22}) \times f_2 + t_{13} \times f_1$ and lt_2 as $(t_{21} + t_{22} + t_{23}) \times f_2$.

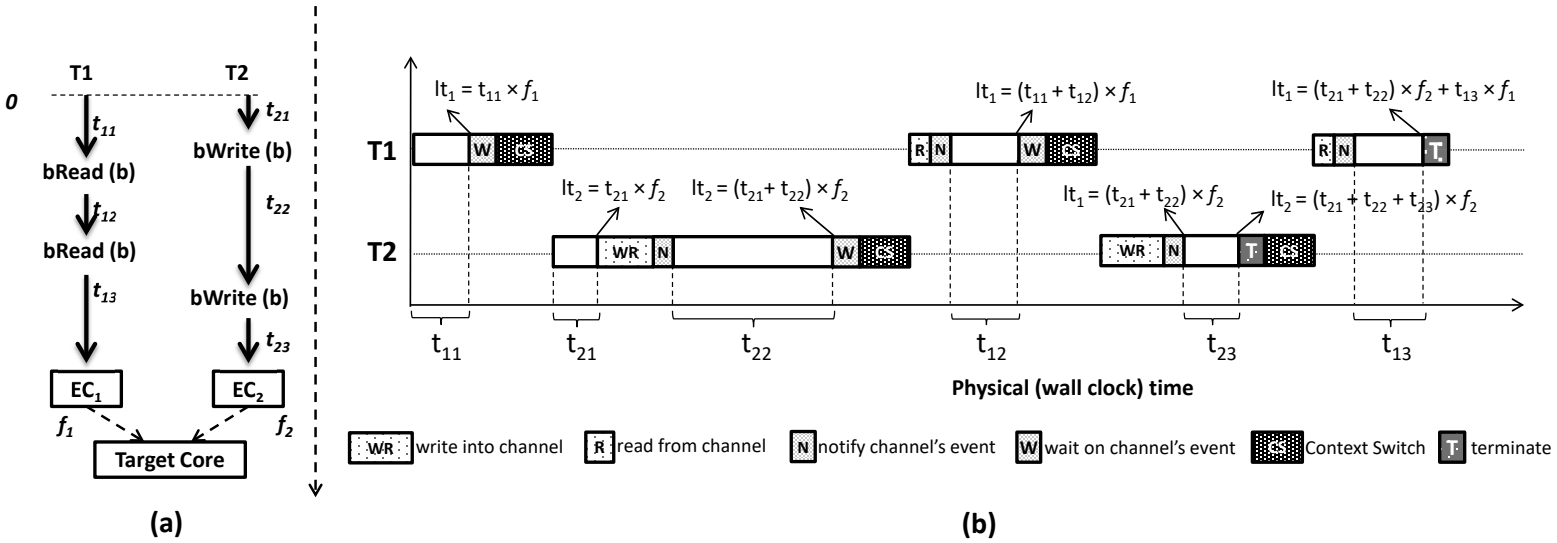


Figure 17: Multi-clock domain simulation example

4.5 Memory Hierarchy

Caches are widely used in embedded system to increase efficiency. So it is important to consider memory hierarchy in hybrid prototyping. L1 caches add a new degree of complexity to multicore emulation because the L1 context must be switched along with the core context during emulation. Fortunately, the typical L1 cache size in embedded multicore systems that we are targeting is relatively small. Therefore, the delay in replacing the cache data during context switches don't slow down the emulation drastically.

Cache modeling can be broadly divided into software-based and hardware-based modeling techniques. Software models allow a high degree of configurability; however, software-based simulators compromise accuracy for simulation speed. Trace driven simulators [43] [44] can be quite accurate, but at the cost of high simulation time. Furthermore, they require repeated simulation runs to reach acceptable accuracy levels. Source-level simulations proposed in [45] estimate only the worst case scenarios; they cannot generate memory transaction statistics over different designs, and are, therefore, unsuitable for design space exploration. Transaction Level Modeling (TLM) techniques [46] [47] increase the level of modeling abstraction to speed up simulation. Software debugging also becomes more efficient with TLMs. Nonetheless, TLMs compromise timing accuracy for greater simulation speed [48].

In order to support fast and accurate simulation, hardware cache emulators have been introduced. Hardware cache models can be classified into passive and active emulators. Passive emulators are hardware monitors that probe memory transactions over the processor bus. Passive cache models like P-cache [49] and RACFCS [50] are L1 cache models that provide cycle-accuracy, configurability and observability into

the cache state at run-time. However, they are not suitable for utilization in multicore emulation systems since they cannot support multiple cache contexts. MemorIES is a passive L3 cache emulator that is designed for multiprocessor servers, such as IBM S70 class RS/6000 or AS/400 servers. The main drawback of this model is the complexity of the hardware design by using several FPGA boards, and the inaccuracy when modeling large caches.

Active cache emulators provide the modeling accuracy and observability of passive models, while behaving as a built-in cache. Prototyping systems such as RMP [51], and RAMP [36] emulate the entire multiprocessor system and the memory hierarchy. Their key drawback is the scalability and the ease of debug. ACE [52] is an FPGA-based active emulator that models L3 cache. It actively interacts with the target system, and provides the same functionality as the built-in cache, but with larger latencies. ACE provides cycle-accuracy and observability. However, like other cache emulators, this model does not support run-time re-configurability for switching between cache contexts, so it cannot be used for multicore emulation. DRAC provides the emulation speed of active models, the observability of passive models, and the run-time configurability to support multiple cache contexts.

We have designed a *D*ynamically *R*econfigurable *A*ctive L1 *C*ache (DRAC) [53] module that emulates the local L1 caches of the cores. The DRAC is an on-chip hardware peripheral connected to the local bus of the core. All program and data memory transactions are made by the core on the local bus.

We have focused on modeling a direct-mapped L1 write-through cache because of the following reasons: (i) L1 has the maximum impact on performance, (ii) direct-mapped has the lowest energy footprint and is therefore popular in embedded systems, and (iii) write-through policy brings greater predictability to global memory state and is desirable for multicore embedded systems. Nevertheless, the ideas presented in this

paper can easily be extended for other types of caches using common cache modeling methods. L1 cache modeling for hybrid prototyping is more involved than implementing a simple cache model in FPGA. The cache model should be capable of simulating different caches of the emulated cores by dynamically changing its context. Therefore, DRAC is designed as a run-time configurable cache that enables the MEK to change the cache context as it switches from one emulated core to the next.

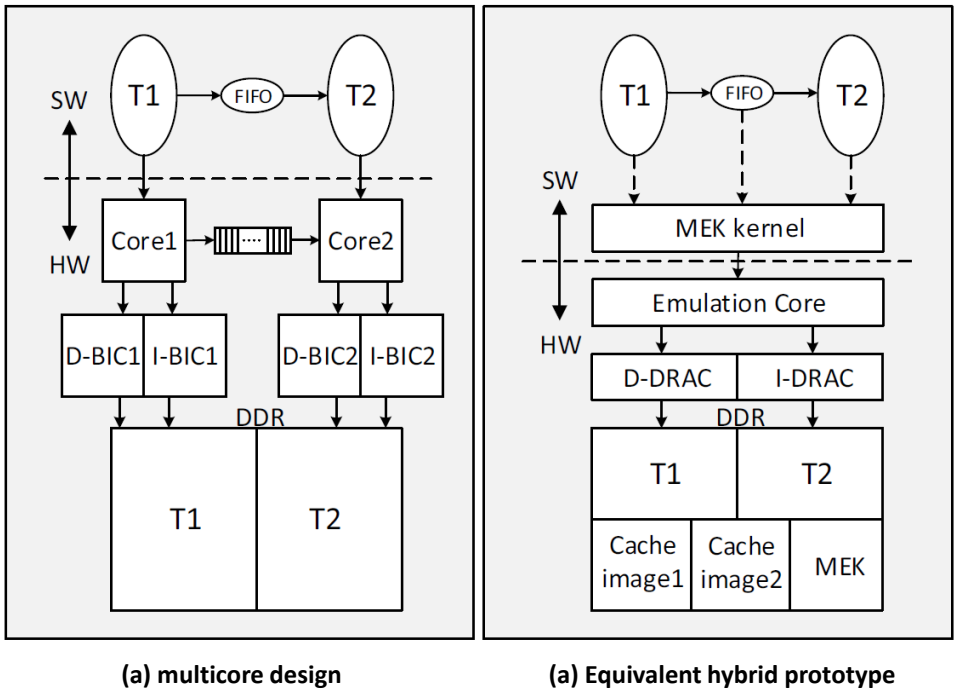


Figure 18: A multicore design with its equivalent hybrid prototype

Figure 18 presents the layered structured of a multicore design and its hybrid prototype equivalent. In the target design, which is shown in Figure 18(a), T_1 and T_2 are tasks running on separate cores. Each core has its own L1 cache and separate memory space on DDR. The communication between the cores is performed using FIFO-based communication channels. Hybrid prototyping introduces an additional software layer on top of an emulation core. Figure 18(b) illustrates the hybrid prototype that incorporates the MEK. The emulation core and the main memory in the hybrid prototype are of the same type as that used in the multicore design.

However, the built-in caches have been replaced by DRAC models. DRAC is customized to support different cache contexts for the two cores. For each cache context, a separate space on DDR is dedicated as cache image. Before the MEK starts emulating a core, it loads the corresponding cache image from DDR to DRAC. Similarly, after the MEK stops emulating a core, it saves the corresponding cache image to DDR. Hence the cache images are swapped in DRAC, when the MEK switches from one core to another.

4.5.1 Dynamically Reconfigurable Active Cache

The DRAC model includes the functionality of a standalone cycle-accurate data and instruction cache, and additional logic to support multicore hybrid prototypes. As discussed earlier, the hybrid prototype simulates several virtual cores on a single core. Thus, a single cache that is capable of switching its context over different virtual cores is needed. To realize this concept, an extra module has been implemented in the cache that can swap the cache contents across different virtual cores. Each virtual core's cache can be configured independently; however, this requires DRAC to change its configuration during run-time. The run-time configurability of DRAC provides the MEK to change the configuration of the cache.

DARC is implemented as an interface between the target processor and the main memory. It receives memory access instructions from the target, processes the requests from the processor, and delivers the instructions and data, as needed. Therefore, it actively interacts with the target system. In order to support multiple cache design choices, DRAC is designed as a parameterized cache model.

4.5.2 DRAC Design

The top level design of DRAC consists of Bridge/Cache Arbitrator, Bus Bridge module, and Cache/Swap module as shown in Figure 19.

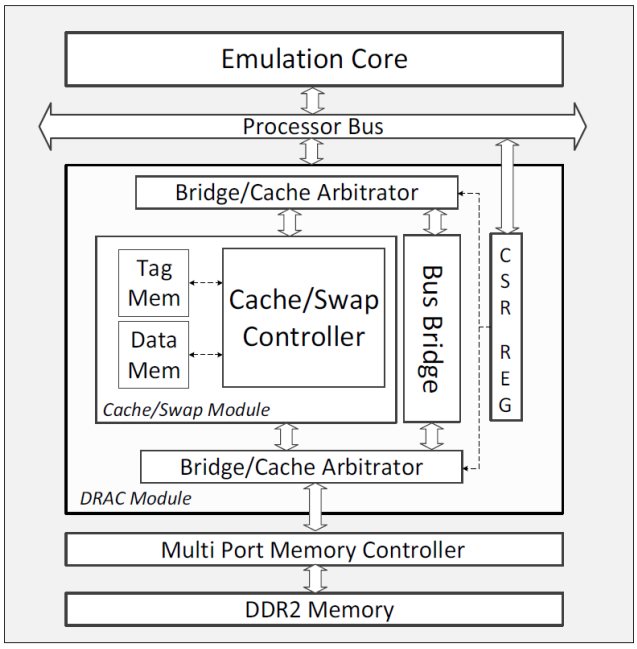


Figure 19: Top level design of DRAC

4.5.3 Bridge/Cache Arbitrator & Bus Bridge

DRAC is placed between the processor and the DDR memory controller; therefore, all memory transactions go through DRAC. The active behavior of DRAC requires it to have an extra module beside the cache, which is called Bus Bridge. This module is responsible for establishing the connection between the processor and the off-chip DDR memory when the cache is inactive. In this case, Cache/Swap module is bypassed by the Bridge/Cache Arbitrator, and the Bus Bridge provides processor's direct access to the DDR memory controller. The arbitrator multiplexes the address bus, data bus, and control signals between the Cache/Swap module and the Bus Bridge. The arbitrator is controlled by the Control Status Register's (CSR). CSR is a 32 bit control

register that resets, enables/disables, sets the size of the cache, and switches DRAC between bridge mode and swap mode.

4.5.4 Cache Module

This module is composed of a controller and two block RAMs used for data and tag memory. The cache size configuration is set by this module. The CSR is used to set the cache size in DRAC. To support large cache sizes, we dedicate a large amount of Block RAMs (BRAMs) for data and tag memory. We, then, utilize a part of the BRAMs as per the cache size requirement.

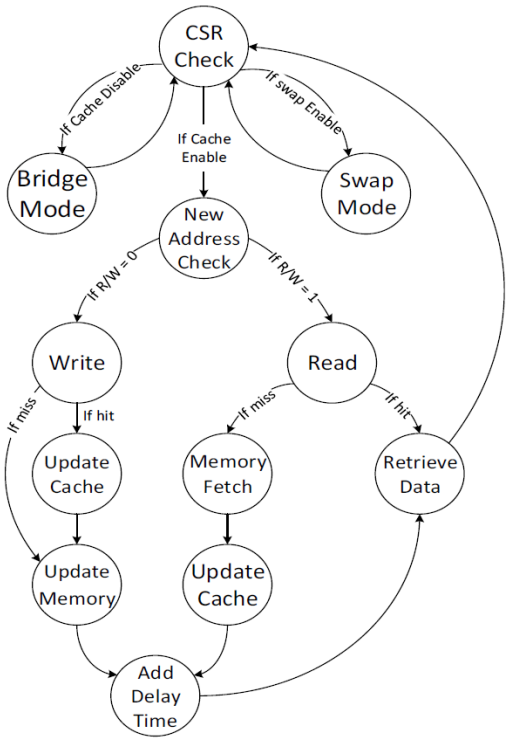


Figure 20: Finite State Machine of cache controller

The cache controller is key module of DRAC, which is used in both data and instruction cache models. The only difference between the data and the instruction cache is read-only. In order to simplify DRAC design, we used the same controller for

both instruction and data. Figure 20 demonstrates the cache controller's finite state machine (FSM).

Cache Controller always checks the CSR value before any memory transaction. If CSR is set to cache enable, the FSM in cache controller is triggered and the state is changed to *address check*. In this state, the module checks the address valid Bit (Addr_valid) signal on the bus in every clock cycle; if it is detected, then the controller checks R/W signal and goes to *Read* or *Write* state. In both *Read* and *Write* states, the cache module first checks the tag memory in order to locate the memory block in the cache. In the *read* state, if the data is found in the cache, it is a hit case, and the cache retrieves the data from its own data memory to the processor; otherwise, it is a miss case and the cache should fetch the regarding memory block from the main memory. The controller's last state is *Add Delay Time*. This state inserts delays depending on our timing model. The algorithm will be discussed in next section. DRAC is assumed to be a write-through cache. Hence, in the case of write, it updates both the cache and the main memory. At the end of each transaction, DRAC sets the acknowledgment signal in the processor's bus, to inform the processor the memory transaction is done.

4.5.5 Swap Module

The cache swap feature is the ability of the cache to save a copy of itself on the off-chip DDR memory and to load it later automatically. The swap module is responsible for switching the cache context from one core to another during run-time. Whenever a swap is triggered by the MEK, DRAC stalls the processor and saves the current cache context to the main memory, line by line. The context of the next core to be simulated is, subsequently, loaded. Figure 21 illustrates the finite state machine of the

swap controller. Similar to the cache controller, the swap controller has a *CSR Check* state as an initial state. The swap trigger is detected in this state.

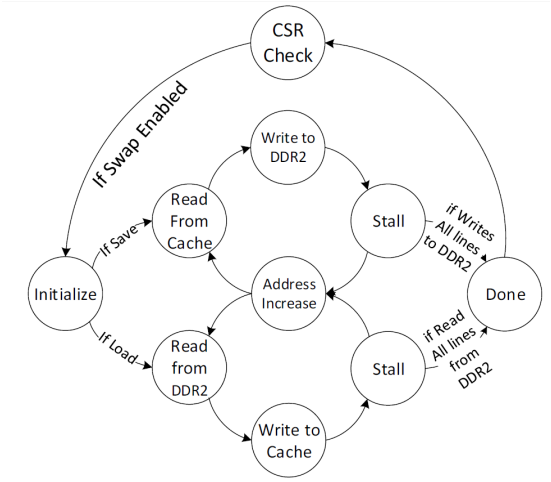


Figure 21: FSM of swap controller (Swap Mode)

Depending on the CSR value, the controller will save or load the cache state. As explained earlier, space in the DDR memory has been allocated for each core, depending on the cache size. The. *Initialize* state determines the starting and the ending address locations of each core's cache. If the processor issues the save cache state command, the controller goes to *Read from Cache* state, reads the first line of the cache, and writes it into main memory. It continues this process until all cache lines are written to the main memory. On the other hand, if the processor's command is load, the controller goes to *Read from DDR2* state, reads all previously saved contents of the cache from main memory, and writes them into the cache. There is a *stall* state in swap's FSM that ensures the data is safely resided in the cache or the main memory.

4.5.6 Timing Model

Designing DRAC as an active cache model and utilizing on-chip BRAM memory as data and tag memory, reduces the program's execution time, as compared to a passive

model. However, the DRAC delay is typically longer than that of the processor's built-in cache. In order to model the built-in cache, we add extra cycles to certain DRAC transactions such that all the DRAC delays are a multiple of corresponding built-in cache delays, by the same factor. As a result, the program's execution time, when using DRAC, will be a multiple of the execution time with built-in cache. For example, in equation 2, if a processor with built-in cache executes a program in *x Clock cycles*, the proposed model will execute the same program in $n \times x$ *Clock cycles*, where n is the linear scaling factor:

$$\text{Modeled Clock cycles} = n \times \text{Real Clock cycle} \quad (2)$$

In order to explain the timing model in more detail, we present the example of a MicroBlaze based system [16]. In this system, MicroBlaze is the core processor and CacheLink (XCL) is the communication bus between the main memory and the BIC. DRAC is designed as a master IP core that can be utilized as data or instruction cache. For hybrid prototypes, we disable the built-in caches, and use DRAC instead. Since DRAC is simply a peripheral to MicroBlaze, it is connected to processor local bus (PLB) which has a different protocol from XCL. XCL is an FSL based dedicated link, so DRAC cannot be connected to it. Clearly, replacing the XCL bus with PLB, disabling the built-in cache, and using DRAC will change the execution time of a program on MicroBlaze. However, it must be noted that our concern is not only the system speed performance, but also the timing estimation.

We used ChipScopePro bus analyzer [16] to obtain memory parameter values for DRAC. The hit time for the MicroBlaze built-in cache is 1 cycle, while this time is 12 cycles for DRAC over PLB. Therefore, we have defined our scaling factor as 12. It means every memory transaction in DRAC will incur 12 times the delay of the corresponding transactions with the MicroBlaze built-in cache. The other factor that

defines cache performance is read miss time. The average miss time latency for the MicroBlaze built-in cache is 29 cycles. This miss time is 149 cycles in DRAC. In order to model read miss time, the emulator inserts 199 cycles to make read miss latency 12×29 cycles.

The write operation is another factor that impacts the system performance. DRAC models a write-through cache. Hence, in every write transaction the main memory will be updated. Writing into on-chip BRAM is quite simple and predictable; however, the write operation to the main memory, which in our case is DDR2 RAM, is quite complex. The complexity comes from the buffers that are implemented in the DDR2 memory controller. Therefore, in order to model write operation in DRAC, we first need to model DDR2.

4.5.7 DRAM Modeling

The connection of DDR2 memory to the system is established by Multi-Port Memory Controller (MPMC). MPMC provides separate accesses to the main memory for different modules in the system. It shares single off-chip DDR2 memory between multiple devices. We have two kinds of memory transactions in the system: read and write. The effect of multiple reads from different ports of MPMC is negligible since reading from the memory does not affect the saved data. The write delays behave differently, though. For a write into the main memory, MPMC stalls other memory transactions to make sure that the memory is in a consistent state. Therefore, if there is a write into a port of MPMC, the read or write access time of other ports will increase.

MPMC uses the buffering technique in order to reduce the write time latency. In case of a single write, MPMC processes the write transactions in the background while it handles other read accesses. Buffering offers the system a better performance,

although it creates irregularity in successive or multiple memory transactions. In case of successive writes into a single port, the write operation time will be different depending on the number of consecutive writes in that port. The first write will take the least, and the last write will take the most operation time. The read time will also be affected by the successive writes of the other port. If the number of consecutive writes increases, the read access time of the other port will also increase.

Table 1: Effect of concurrent writes to DRAM

Number of concurrent writes to port 0	**0**	**1**	**2**	**3**	**>=4**
Number of cycles to write to MPMC port 0 in BIC	0	2	4	5	11
Number of cycles to write to MPMC port 0 in DRAC	0	$2\times12\times f_m$	$4\times12\times f_m$	$5\times12\times f_m$	$11\times12\times f_m$
Number of cycles to read from MPMC port 1 in BIC	29	42	53	65	79
Number of cycles to read from MPMC port 1 in DRAC	$29\times12\times f_m$	$42\times12\times f_m$	$53\times12\times f_m$	$65\times12\times f_m$	$79\times12\times f_m$

Previously, it was mentioned the DRAC model scales its delays to be a multiple of built-in cache delays. Besides hit and miss time latencies, DRAC also models the successive and multiple write delays. Table 1 presents the write and read access parameters of the built-n cache, and the modeled parameter values of DRAC. In the single core design, the instruction and the data cache are utilizing separate ports of MPMC. Since there is no write into MPMC in the instruction cache, the read access time of the instruction cache is only effected by data cache writes. In the multicore design, there are more than one data caches that write into MPMC. Hence, the effect of multiple writes will be more severe in higher number of cores.

In multicore emulation with hybrid prototyping, only one core is simulated at a time. Hence, it is not possible to predict the exact behavior of the other cores during simulation. This effect causes the predicted execution time to be less than what is expected. In order to decrease this effect, we introduce a multiplication factor f_m, which

models the multiple write effect. To determine f_m, we tested different multicore designs with all the cores running in parallel. We observed that the multiple write effect depends on the number, density, and distribution of writes over different cores. As a result, we executed a sample software code with different write distributions on multiple cores running in parallel. In each experiment, we kept the write density of the first core constant, and changed the write density of the other cores. We test different write densities for different number of cores ranging from the best case, in which there is no write in the second core, to the worst case, that the write density is almost 100%. We found an average f_m value for each core. Table 2 presents the values of f_m for different number of cores.

Table 2: Multiple write factor for different number of cores

Number of cores	**2**	**3**	**4**
f_m	4	9	12

4.5.8 Cache Modeling Limitation in Hybrid Prototyping

There are several limitations of cache modeling in hybrid prototyping. The L1 images of all the cores are maintained in a dedicated memory, which may be on-chip if there is enough space. During the core context switch, the MEK instructs the DRAC to store the current L1 image and load the appropriate L1 image for the next core to be emulated. By increasing the number of cores we need more space to store the L1 images which may not be available. Furthermore, increasing the number of cores will impact on prototyping speed as the L1 image must swap for each context switch. Therefore, increasing the number of cores could affect the space and speed of the hybrid prototype.

The other limitation of using cache model in the hybrid prototyping is modeling multiple cache hierarchy. Cache pollution is a serious problem for modeling multiple cache hierarchy in multicore designs. For instance, a core may replace the blocks fetched by another core into shared L2 cache. In the hybrid prototyping, running emulation kernel between application's thread is also resulting the cache pollution. Therefore, running a design in FPGA prototype with shared L2 cache reports different results comparing to the hybrid prototype due to unpredictable misses and hits that occurs in shared L2 cache.

Inability of modeling scratchpads is another limitation of cache modeling in the hybrid prototyping. Scratchpad is a high-speed internal memory used for temporary storage of calculations, data, and other work in progress. As it was explained earlier, to model the cache behavior, an on-chip hardware peripheral can be connected to the local bus of the core. It then can model the cache by observing all memory transactions. In contrast, scratchpads are transparent to the system. They cannot be accessed and cannot be observed as they sit close to the CPU. As a result, modeling scratchpads without the ability of observing or accessing them is another issue for modeling memory hierarchy in the hybrid prototyping.

Cache model heavily depends on hardware architecture. Current DRAC cache model can only be used in designs with MicroBlaze as the target core. To use DRAC cache model for other architectures such as PowerPC or ARM we need to investigate the new architectures and change DRAC to support them which is a very time consuming and difficult work.

4.6 Summary

In this chapter we presented the hardware model layer of the hybrid prototyping. We explained how the processors in a SMP design were emulated by emulated cores. Different communications models which are supported in the framework were also described. We then talked about the modeling of the hardware interrupts in the framework. We have also seen that the hybrid prototype can support multi-clock domain frequencies. Finally, we explained memory hierarchy model in details. In the next chapter we will talk about the software model layer which provides RTOS model scheduler to manage threads on top of the emulated cores.

Chapter 5

Software Model layer

Software model layer provides simulation primitives for the management of threads. It defines model of RTOS scheduler for dynamic task scheduling on the emulated cores. Models of RTOS scheduler developed for system level design languages have been proposed, but are non-trivial to port to hybrid prototypes, given the absence of a single logical time [54].

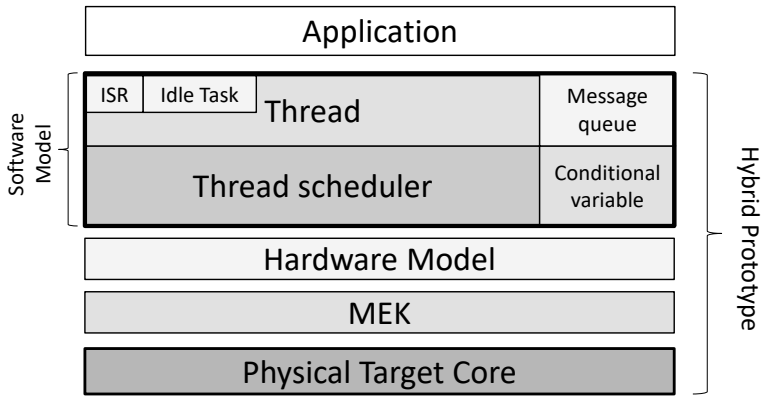


Figure 22: Software model layer structure

This layer defines message queue primitives which models the inter-process communication services implemented as an API on top of the conditional variable simulation primitives. Conditional variable is an important synchronization primitive beyond locks and allows threads to sleep when some program state is not as desired. The idle task provided by this layer is a special thread that has the lowest priority and is always ready to be run on the idle emulated cores. Finally, ISRs is defined in this layer to handle hardware interrupts in the design [55]. Figure 22 shows the software mode layer structure in the hybrid prototyping.

5.1 Thread

Threads (sometimes called lightweight processes) are the basic unit of processor use. Each is a separate control path through the code, and the ones within a process are essentially independent. Threads can access all the address space by the process, and they have no protection against each other. Software model layer in the hybrid prototyping provides all primitives and services for thread management. In a hybrid prototype, each thread comprises a thread id, program counter, register set and a stack.

Every thread is assigned a priority. The priority can be set to a level from 0 to 10 (10 is the highest priority which can be varied by the designer). As we will see later in this section, the thread scheduler selects the next threads to be run by looking at the priority assigned to every thread that is READY (i.e., capable of using the processor). The thread with the highest priority is selected to be run.

The created thread becomes the tail of the ready queue for that priority. The ready queue is an array of the queues. Each entry of the array consists of a queue of the threads that are READY at that priority. Any threads that aren't READY aren't

in any of the queues, however they will be when they become READY. The RTOS uses ready queue to decide who to schedule next.

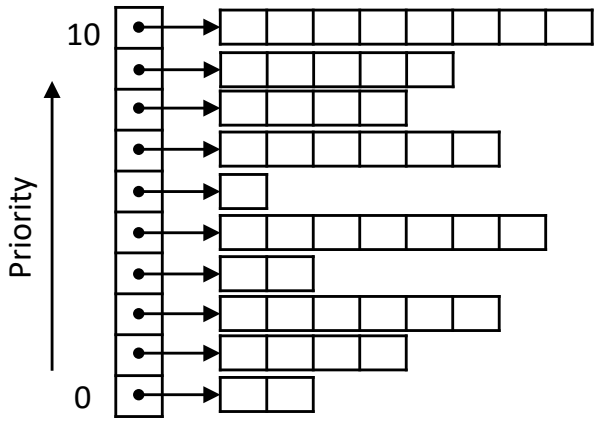


Figure 23: Threads ready queue

Figure 23 shows the ready queue data structure. Listing 15 shows the thread class in the software model layer.

```
class Thread {
   public:
      Thread(functionPtr, int, int, int);
      void setState(THREAD_STATE);
      void setPrio(int);
      void setId(int);
      THREAD_STATE getState() const;
      int getId() const;
      int getPrio() const;
      int getCoreId() const;
      bool hasContext() const;
      Context context;
      functionPtr start_routine;
private:
      THREAD_STATE thread_state;
      int thread_id;
      int thread_prio;
      int core_id;
}
```

Listing 15: Thread class in the software model

In our RTOS scheduler model, a thread may be in four possible states: “running”, “ready”, “blocked” or “terminated”. *Running* state means that the thread is now actively

consuming the physical target core. The *ready* state means that a thread is ready to be run right now but all emulated cores are being used by other threads at that time. *Terminated* simply means that the thread is terminated and no longer needs to get executed. The terminated threads are removed from the ready queue. *Blocked* states means a thread must wait for some event to occur (e.g. response to a signal, event, etc.). The blocked thread is also removed from the ready queue until the blocking will be completed. Figure 24 illustrates the thread life cycle. It is important to mention that a running thread can voluntarily yield its execution turn. By yielding the execution, the thread will be placed at the end of the ready queue for that priority. Then the highest priority thread will be run.

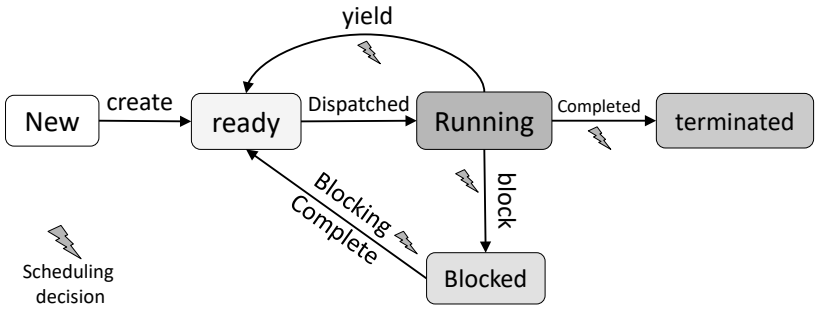


Figure 24: Thread life cycle in the software model scheduler

5.2 Thread Scheduler

Thread scheduling refers to the assignment of idle emulated cores to ready threads. The RTOS thread scheduler in our software model supports the FIFO scheduling policy. In FIFO scheduling algorithm, a thread is allowed to consume emulated core for as long as it wants. If the running thread terminates, blocks or voluntarily gives up the emulated core (yield), the RTOS looks for another ready thread in the same priority, and if there are no such threads, the RTOS looks for lower-priority threads capable of using the emulated core. Therefore, the highest-priority threads will be run.

If there's another thread that is ready to be run and if there's an available emulated core, the thread will be run on it. If there aren't enough threads to go around, the idle emulated cores will run the idle thread. Idle thread is a special thread that has the lowest priority and is always ready to be run. An emulated core is considered to be idle when the idle thread is scheduled to be run on it. If there aren't enough emulated cores to go around, then only the N-highest-priority ready threads will be run, where N is the number of available emulated cores. The scheduling decisions may take place when a thread:

1. switches from the running to the blocked state
2. switches from the running to the ready state
3. switches from the blocked to the ready state
4. terminates

Under condition 1, the running thread waits for some events to occur and gets blocked. The blocked thread is removed from the ready queue and the highest-priority ready thread is then run. When the blocked thread is subsequently unblocked, it's usually placed at the end of the ready queue for that priority level. In condition 2, the running thread may yield the execution. The RTOS scheduler then inserts it at the end of the ready queue for that priority and does the scheduling decision. When a thread becomes unblocked (condition 3), the RTOS scheduler puts it back at the end of the ready queue and decides which threads should be run at this particular time. Finally, if a thread is terminated the RTOS scheduler removes it from the ready queue and reschedule the ready threads on the emulated cores.

The simulation exits successfully if all threads are in the terminated state. A scheduling event may also result from a hardware interrupt. If an external interrupt occurs, a signal is posted to ISR. As we will see later in this section, ISR is a special

thread with the highest priority which is responsible to respond to external interrupt and performs the appropriate action based on the interrupt.

5.3 Processor Affinity

Processor affinity is the ability of binding or unbinding a process or a thread to a particular processor. Our software model supports static scheduling by specifying strict core affinity. During the application initialization in the hybrid prototyping, a setting determined by the system designer forces all of an application's threads to execute only on a specified emulated core. It offers the benefits of SMP's transparent resource management, but gives designers the ability to lock any application (and all of its threads) to a specific core to help migrate uniprocessor code to a multicore environment. It allows legacy applications written for uniprocessor environments to be run correctly in a concurrent multicore environment, without modifications.

5.4 Condition Variable

Conditional variable is an important synchronization primitive typically used in implementing deterministic producer/consumer behavior. Our software model layer provides conditional variable primitives using basic wait/signal methods. The wait method is executed when a thread wants to put itself to sleep until a condition is satisfied, and the signal method is executed when a thread wants to wake sleeping threads waiting on the given condition.

Conditional variable class has a list called waitlist. Waitlist is a list of all threads that are waiting on the conditional variable. Upon a wait call, the RTOS scheduler puts the caller thread into the blocked state, removes the thread from the ready queue

and inserts the caller thread into the conditional variable's waitlist. On a signal call, all threads which have been waiting on the conditional variable are notified and the RTOS scheduler changes their state to ready, puts them back in the ready queue and reschedules the ready threads on the emulated cores. If there is no sleeping thread on the conditional variable, the signal will be lost.

5.5 Message Queue

Message queue (mqueue) is an asynchronous communication mechanism between discrete components of an application. It facilitates message passing by connecting producers which create messages and consumers which then process them. The mqueue is modeled as a variable size circular buffer. The mqueue has two boolean variables to indicate a full or empty state, as well as respective conditional variables that are signaled whenever the buffer is read or written.

```
void msqueue::send(T newvalue) {
   1: KERNEL_CALL_START();
   2: while (this->isFull())
   3:    this->con_var_is_not_full->wait();
   4: this->cbuffer->enqueue(newvalue);
   5: if (this->cbuffer->isFull())
   6:    this->flag_not_full = false;
   7: this->flag_not_empty_ = true;
   8: this->con_var_is_not_empty->signal();
   9: KERNEL_CALL_END();
}
```

Listing 16: Message queue send method pseudo code

Listing 16 shows the pseudo code for sending data over the mqueue. Since this is a blocking operation, the sending thread must wait as long as the mqueue is full (lines 2-3). If the queue is not empty, the item is placed into the tail of the circular buffer (line 4) and *con_var_is_not_empty* is signaled (line 8). The condition variable wakes up all threads which have been waiting on the mqueue for receiving data.

The receiving method is the exact dual of sending as shown in Listing 17. The consumer checks if the mqueue is empty or not (line 2). If the mqueue is empty the consumer waits on the condition variable (line 3). Otherwise, the item is read from the buffer (line 4) and the *con_var_is_not_full* is notified (line 8) to wake up all threads which have been waiting on the mqueue for sending data.

```
T msqueue::receive() {
   1: KERNEL_CALL_START();
   2: while (this->isEmpty())
   3:    this->con_var_is_not_empty->wait();
   4: T item = this->cbuffer->dequeue();
   5: if (this->cbuffer->isEmpty())
   6:    this->flag_not_empty = false;
   7: this->flag_not_full = true;
   8: this->con_var_is_not_full->signal();
   9: KERNEL_CALL_END();
   10: return item;
}
```

Listing 17: Message queue receive method pseudo code

5.6 Idle Task

Idle task is a special thread with the lowest priority and always ready to run. It is scheduled on the emulated cores when there aren't enough threads to go around. The primary purpose of the idle thread is to measure the idle time for each emulated core by waiting on awake event of the emulated core. When idle thread executes on an emulated core, the core is considered as an idle and the amount of time that the idle thread spend on running on the core is considered as idle time. We will see later in the next section how the hybrid prototyping calculates the idle and busy time for each individual emulate core.

The idle thread waits on emulated core's event (*awake*) as shown in Listing 18 the wait call puts the emulated core in the suspended state. When the event *awake* is

notified, the MEK wakes up the suspended emulated core and updates its logical time and idle time if needed.

```
void idle_thread() {
  1: while (true) {
  2:    active_ecore->awake->wait();
  3: }
}
```

Listing 18: Idle thread pseudo code

5.7 Dynamic Scheduling Example

Dynamic scheduling enables the execution of unmodified multi-threaded applications on top of a SMP-based hybrid prototype. In this section we describe how a hybrid prototype emulates a SMP design using a simple example. Figure 25 uses a simple example to illustrate dynamic scheduling simulation on a hybrid prototype. We assume that the design consists of two emulated cores and one hardware interrupt generator (HW_INT). The application has two threads which are communicating to each other with message queue primitives. The HW_INT generates interrupt at fixed time intervals ($t_{INT} > t_{11} > t_{21}$).

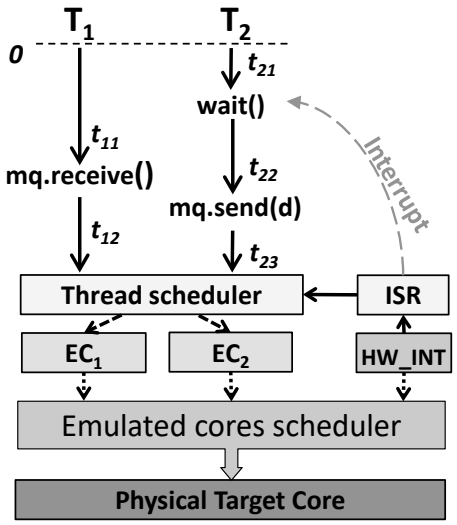


Figure 25: Simple example of dynamic scheduling on two emulated cores with a hardware interrupt

Figure 26 shows how the hybrid prototype maintains the logical times, lt_1 and lt_2, and idle times, $idle_1$ and $idle_2$, for emulated cores EC_1, EC_2 respectively. To avoid complexity of the example we just show one interrupt signal simulation from the HW_INT. The ISRs thread is also is not shown in the figure. We assume that the interrupt is sent to T_2 directly. After the hybrid prototype is instantiated, EC_1 and EC_2 are placed at the end of the idle queue which means they are capable of executing

threads. The RTOS then schedules T_1 and T_2 on EC_1 and EC_2 respectively. By starting the simulation, the RTOS schedules EC_1 to execute on the physical target core. Thread T_1 executes on EC_1 from EC_1's logical time 0 until it reaches *mq.receive* at EC_1's logical time t_{12}. At this time, there is no data in the message queue because T_2 has not yet been simulated. Therefore, T_1 must be blocked until some data is written in the message queue. T_1's state becomes blocked, the EC_1's logical time is set to t_{11} and the MEK removes it from EC_1.

The MEK then puts EC_1 at the end of the idle queue, does a context switch and schedules EC_2 to execute on the target core. At this point of time, EC_1 is the only idle emulated core. As there is no ready thread available, the RTOS schedules idle thread on EC_1 and inserts it back into the busy queue.

EC_2 then executes on the target core and runs T_2 from its logical time 0 until it reaches wait on the interrupt at EC_2's logical time t_{21}. At this point the MEK checks if there are any pending signals on the interrupt at or before the current logical time t_{21}. Since no notifications for interrupt are found, the MEK stores the wait on the interrupt in the signal's waitlist and the RTOS blocks T_2 and removes it from EC_2. The MEK updates EC_2's logical time to t_{21} and puts EC_2 at the end of the idle queue, does a context switch and schedules HW_INT to get control on the target core.

HW_INT task sleeps for t_{INT} and then posts a signal on the interrupt. Upon notification, the RTOS checks if there are any pending waits on the signal at or before logical time t_{INT}. As thread T_2 is blocked on the signal at EC_2's logical time t_{21} ($t_{21} < t_{INT}$), the RTOS unblocks T_2 and updates EC_2's logical time to t_{INT} and EC_2's idle time to $t_{INT} - t_{21}$ in order to calculate the blocking time. The HW_INT's logical time is also set to t_{INT}.

The MEK then schedules EC_1 to execute on the target core. EC_1 executes the idle thread until it reaches wait on the *awake* event. The MEK puts EC_1 in suspended

state and removes it from the busy queue. The MEK switches the context and schedules EC_2 to run on the target core. It runs idle thread and waits on its awake event as well. EC_2 becomes suspended and the MEK removes it from the busy queue and switches the context to the HW_INT task. HW_INT runs again and now its logical time is equal to MIN_SIM_TIME (the only non-suspended emulated core is HW_INT, therefore, MIN_SIM_TIME will be equal to HW_INT's logical time). HW_INT then notifies EC_1's awake event and EC_2's awake event.

Upon this notification, the MEK updates EC_1' logical time to HW_INT's logical time ($t_{11} < t_{INT}$) and updates EC_1's idle time to $t_{INT} - t_{11}$. Now the HW_INT yields its exaction turn to let other emulated cores to run and may consume the interrupt. Both EC_1 and EC_2 are now capable of getting assigned to the next available threads. Since at this point of time T_2 is the only ready thread that can be run, the RTOS schedules it to be run on EC_1 (the first available emulated core) and schedules idle thread on EC_2. EC_1 executes T_2 from EC_2's logical time t_{INT} until it sends the item on the message queue. The MEK then updates the EC_1's logical time to $t_{INT} + t_{22}$, changes the state of T_1 to ready and puts it back at the end of the ready queue.

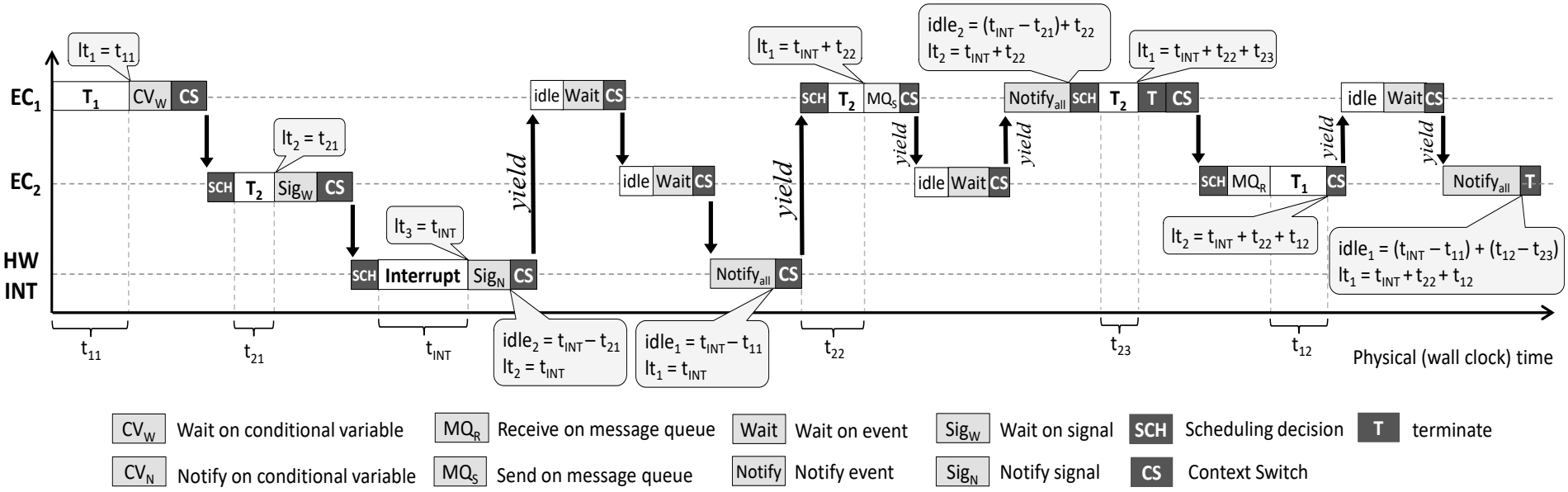


Figure 26: Timing estimation example with two threads running on a design with two emulated cores and a hardware interrupt

At this point, the MEK sends a notification to all other emulated cores which have been waiting on their awake event. Since, EC_2's logical time is equal to MIN_SIM_TIME, EC_1 yields its execution turn and EC_2 executes on the target core. EC_2 runs idle thread and waits on its *awake* event. The MEK puts EC_2 into suspended state and removes it from the busy queue, switches the context and runs EC_1. At this time, the notification is committed and as a result the MEK updates EC_2's logical time to $t_{INT} + t_{22}$ and EC_2's idle time to $t_{INT} - t_{21} + t_{22}$.

The RTOS makes scheduling decision and schedules T_1 to run on EC_2. T_2 executes on EC_1 for another t_{23} unit of time until it terminates. The RTOS changes T_1's state to "*terminated*" and removes it from the ready queue. The MEK updates EC_1's logical time to $t_{INT} + t_{22} + t_{23}$. It then switches the context to simulate EC_2. Since a thread has been terminated, the RTOS does a scheduling decision and schedules idle thread on EC_1. EC_2 executes T_1 from its logical time $t_{INT} + t_{22}$ until it receives data from the message queue. After receiving data from the message queue, EC_2 continues simulating T_1 until it terminates. Then the MEK updates EC_2's logical time to $t_{INT} + t_{22} + t_{12}$, the RTOS changes the T_1's state to "*terminated*", removes it from the ready queue and the MEK then notifies the *awake* events of all emulated cores (in this case EC_1). Upon this notification the MEK updates EC_1's logical time to $t_{INT} + t_{22} + t_{12}$ and EC_1's idle time to $(t_{INT} - t_{11}) + (t_{12} - t_{23})$ (since EC_2's logical time $>$ EC_1's logical time). The simulation has been done because there aren't any available ready threads in ready queue and all threads are terminated successfully. By the end of the simulation, the hybrid prototype reports EC_1's logical time as $t_{INT} + t_{22} + t_{12}$, EC_1's idle time as $(t_{INT} - t_{11}) + (t_{12} - t_{23})$, EC_2's logical time is equal to $t_{INT} + t_{22} + t_{12}$ and finally EC_2's idle time as $(t_{INT} - t_{21}) + t_{22}$. The busy time for each emulated core is the difference of the logical time and the idle time of the core. Therefore, the hybrid prototype reports the EC_1's busy time as $t_{11} + t_{12} + t_{23}$ and EC_2's busy time as $t_{21} +$

t_{12}. Table 3 shows the emulated cores states and the actions each thread takes, as well as its scheduler state over time.

Table 3: Threads and emulated cores trace

EC1	**EC2**	**T_1**	**T_2**	**Comment**
T1	T2	Running	Ready	Simulation get started
idle	T2	Blocked	Running	T_1 blocked on message queue
idle	idle	Blocked	Blocked	T_2 waits for interrupt
T2	idle	Blocked	Ready	T_2 get notified by the interrupt
T2	idle	Blocked	Running	
idle	T1	Ready	Terminated	T_2 sends data on the message queue
idle	T1	Running	Terminated	T_1 receives data
idle	idle	Terminated	Terminated	

5.8 Summary

In this chapter we presented the software model layer of the hybrid prototyping and its model of RTOS scheduler. We first talked about threads and thread scheduling in this layer. Then we explained the inter-processor communication model in the hybrid prototyping. We also introduced conditional variables as a synchronization mechanism. Finally, we went through an example in order to explain how RTOS schedules different threads on the emulated cores and handles hardware interrupts.

Chapter 6

Evaluation

6.1 Use cases

To evaluate the speed and accuracy of the hybrid prototypes, we used the JPEG encoder, the MP3 decoder and a simple packet forwarding applications. The JPEG encoder is a simple pipelined multicore application with IO file. The MP3 decoder is a larger and more complex application with real-time constraints and hardware IO. The packet forwarding application can be massively parallel to demonstrate scalability of the hybrid prototyping. There are many alternative design options to implement these applications. These design options can be considered as benchmarks in our experimental results. Therefore, using these applications can help us to evaluate the hybrid prototypes in different aspects. We chose the MicroBlaze [16] core from Xilinx for the target multicore architectures because of easy integration with FPGA and the ability to instantiate multiple Microblazes (soft processor) to create reference FPGA

prototypes for accuracy evaluation. The FIFO communication between the tasks is performed using the FSL buses supported by MicroBlaze.

6.1.1 MP3 Decoder

The MP3 decoder application reads and decodes data from the media file. The MP3 data is fetched from a file, and after being decoded it is written into a serial buffer. The buffered data can be played on the handset speaker. This application has 5 separate tasks which can be run on different cores: *isrPulser* which is responsible for sending pulse in proper time to task *isr*. Task *isr* is the interrupt handler that notifies the decoding task if more data is needed by the serial buffer for the speakers. Task *audiosal* which reads and decodes data from the media file. Task *mixerctrl* is in charge of the channel and task *dspaudio* converts the rate, playback, mix, etc. on the data. Figure 27 shows the MP3 decoder application.

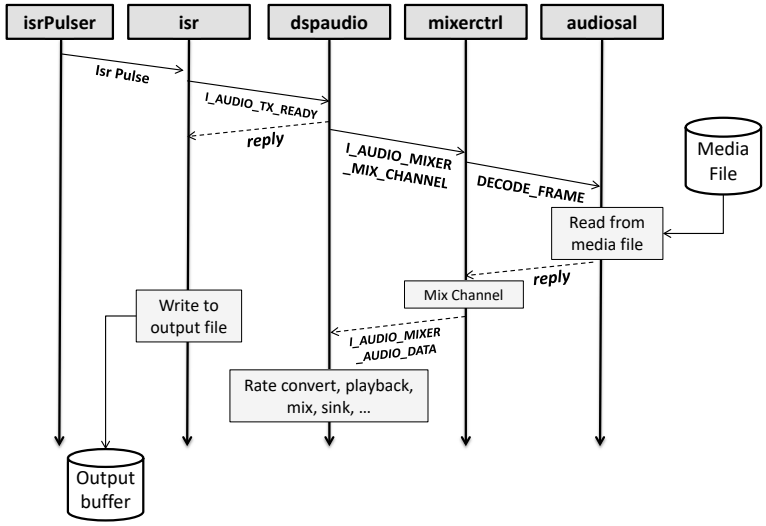


Figure 27: The MP3 decoder application

6.1.2 Jpeg Encoder

The JPEG encoder consists of 5 tasks: Read the bitmap (*Read*), Discrete Cosine Transform (*DCT*), Quantization of values (*Quant*), ZigZag transform (*ZigZag*) and

Huffman encoding (*Huff*). As Figure 28 shows, each task consumes a frame, which is an 8×8 block of integers, processes it and passes the block to the next task. Given the application structure, it can be easily pipelined and the concurrent tasks can be mapped to different cores.

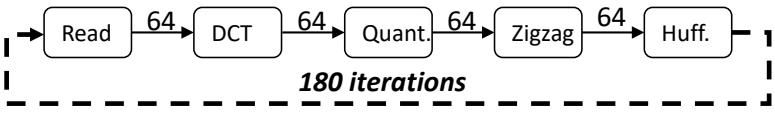


Figure 28: JPEG encoder application

Table 4 shows 15 different multicore designs (ranging from 2 core to 5 cores) of the JPEG encoder application. As it shows, each design has different mappings from tasks to cores.

Table 4: Task mappings for the JPEG encoder multicore designs

Design	**#Cores**	**Mapping**
2a	2	Read → mb1; DCT, Quant, ZigZag, Huff → mb2
2b	2	Read, DCT → mb1; Quant, ZigZag, Huff → mb2
2c	2	Read, DCT, Quant → mb1; ZigZag, Huff → mb2
2d	2	Read, DCT, Quant, ZigZag → mb1; Huff → mb2
3a	3	Read → mb1; DCT → mb2, Quant, ZigZag, Huff → mb3
3b	3	Read → mb1; DCT, Quant → mb2; ZigZag, Huff → mb3
3c	3	Read → mb1; DCT, Quant, ZigZag → mb2; Huff → mb3
3d	3	Read, DCT → mb1; Quant, ZigZag → mb2; Huff → mb3
3e	3	Read, DCT → mb1; Quant → mb2; ZigZag, Huff → mb3
3f	3	Read, DCT, Quant → mb1; ZigZag → mb2; Huff → mb3
4a	4	Read → mb1; DCT → mb2; Quant → mb3; ZigZag, Huff → mb4
4b	4	Read → mb1; DCT → mb2; Quant, ZigZag → mb3; Huff → mb4
4c	4	Read → mb1; DCT, Quant → mb2; ZigZag → mb3; Huff → mb4
4d	4	Read, Quant → mb1; DCT → mb2; ZigZag → mb3; Huff → mb4
5	5	Read → mb1; Quant → mb2; DCT → mb3; ZigZag → mb4; Huff → mb5

6.1.3 Packet Forwarding Application

The JPEG encoder and the MP3 decoder can be run on a design with maximum number of 5 cores. However, to evaluate the overhead of the hybrid prototype and also to show its scalability, we need an application that can be run on a large number of cores simultaneously. A packet forwarding application would be an ideal choice for this purpose. Therefore, a simple application has been implemented in order to process packets. The application has a dispatcher responsible for reading packets and distributing them among the inner-cores. The inner-cores execute packet processing tasks and send the processed packets to the collector. The collector receives all packets and puts them in a proper order. This application can be implemented with a large number of inner-cores that can each be implemented as a MicroBlaze in FPGA prototype. The cores are connected using FSL as shown in Figure 29.

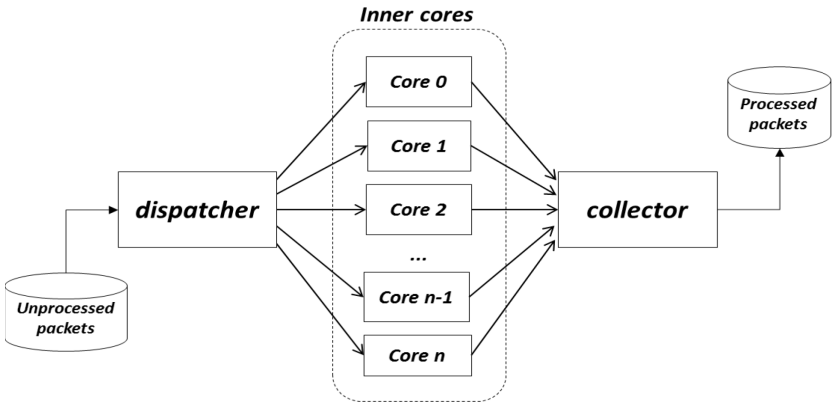


Figure 29: Simple Packet forwarding application

6.2 Experimental Results

We created a FPGA prototype, a hybrid prototype and a virtual prototype for different designs for JPEG encoder, MP3 decoder and packet forwarding applications.

All the used MicroBlaze cores are clocked at 125 MHz. Each MicroBlaze core in the FPGA prototypes has 64 KB of dedicated BRAM for program and data. The hybrid prototypes use a single MicroBlaze core with 64 KB of BRAM since all the tasks and the MEK fit in a single BRAM. For larger programs, one may create multiple instances of BRAMs with contiguous address space assignment. OVP is used to create the virtual prototypes. As OVP is an instruction accurate simulator, it only calculates the number of instructions and cannot measure the idle time. Therefore, the busy time for each core is the sole result that can be provided by the OVP.

6.2.1 Accuracy

We used static binding to lock each task to a specific emulate core in hybrid prototypes to be able to compare them with the PFGA and virtual prototypes. FSL provided by the hardware model was used for the inter-process communication. Static binding and dynamic binding in the hybrid prototype use exactly the same approach to deal with time estimation, therefore, the hybrid prototypes can be evaluated using static binding.

Figure 30 shows the busy time reported by FPGA, Hybrid and virtual prototypes for each core for all different designs mentioned in Table 4. The X-axis shows the designs and Y-axis shows the execution time in million cycles for each design.

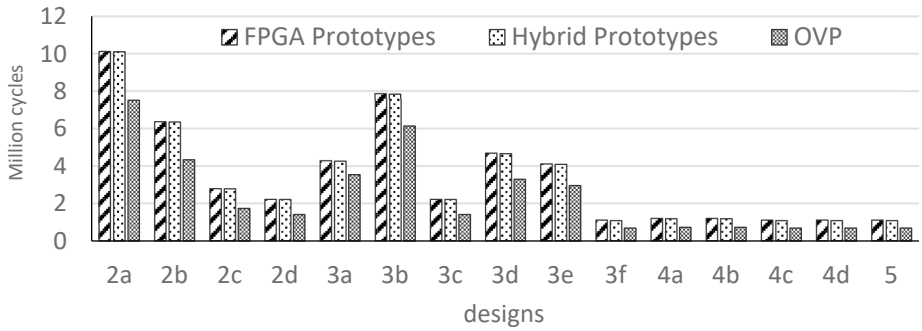


Figure 30: The busy times for FPGA, hybrid and Virtual prototypes for the JPEG encoder

Figure 31 shows the number of cycles needed to execute the JPEG encoder for a given image reported by FPGA and hybrid prototypes for each design mentioned in Table 4. It does not contain OVP results because OVP reports busy time instead of total execution time (busy time + idle time). In a given multicore design, the longest execution time amongst all tasks (mapped to different cores) can be considered as the design's total execution time. The X-axis shows the designs and Y-axis shows the execution time in CPU cycles.

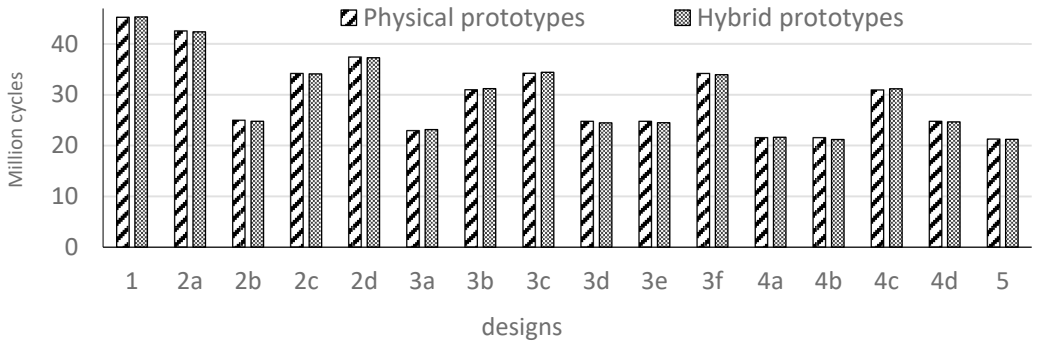


Figure 31: The execution time reported by FPGA and hybrid prototypes for the JPEG encoder

Table 5 contains all the results for the JPEG encoder application in CPU cycles. The first column indicates the designs which are explained in Table 4. The second column shows the hybrid prototyping emulation time. The fourth and fifth columns contain the execution time for each core. Accuracy column shows how hybrid prototypes are accurate. The two last columns contain the busy time ratio for each core in a design. Busy time ration can be easily calculated by equation 3.

$$busytime_{ratio} = \frac{busy_{time}}{total_{time}} \quad (3)$$

Table 5: The JPEG encoder execution time in CPU cycles for all possible design

Design	Hybrid prototyping total emulation time	Core	FPGA prototype execution time	Hybrid prototype execution time	Accuracy	Busy time ratio		
						FPGA	Hybrid	OVP
1	47872930	1	4329834	4289109	99.06%	28%	28%	17%
		2	4326674	4335045	99.80%	99%	98%	82%
		3	4397971	4356360	99.05%	94%	94%	68%
		4	4402759	4360950	99.05%	25%	25%	16%
		5	4413335	4371449	99.05%	50%	51%	32%
2a	10728896	1	10059354	10069038	99.90%	12%	12%	7%
		2	10168752	10178315	99.90%	100%	99%	74%
2b	10730450	1	6371169	6380750	99.85%	78%	77%	61%
		2	6438799	6448354	99.85%	99%	99%	67%
2c	10751181	1	8535527	8512615	99.73%	100%	100%	77%
		2	8549169	8526277	99.73%	33%	33%	20%
2d	10751181	1	9104767	9081955	99.75%	100%	100%	75%
		2	9115258	9092454	99.75%	24%	24%	16%
3a	20507035	1	6301868	6311253	99.85%	19%	19%	11%
		2	6323034	6382167	99.06%	68%	67%	56%
		3	6440517	6449781	99.85%	99%	99%	67%
3b	20507043	1	7889652	7827749	99.21%	15%	15%	9%
		2	7926390	7913477	99.83%	99%	99%	78%
		3	7989885	7927139	99.21%	35%	35%	22%
3c	20507043	1	8452546	8390763	99.27%	14%	14%	9%
		2	8495609	8482817	99.85%	99%	99%	76%
		3	8555930	8493316	99.27%	26%	26%	17%
3d	20523589	1	4958290	4934183	99.50%	100%	100%	79%
		2	4947656	4970043	99.55%	95%	94%	66%
		3	5007977	4980542	99.45%	44%	44%	28%
3e	20562359	1	4957187	4934183	99.54%	100%	100%	79%
		2	4929016	4955498	99.46%	83%	83%	60%
		3	4992511	4969160	99.53%	56%	56%	35%
3f	20573659	1	8535472	8512615	99.73%	100%	100%	77%
		2	8535472	8517205	99.78%	13%	13%	8%
		3	8550902	8527704	99.73%	26%	26%	17%
4a	32887985	1	4329783	4289109	99.06%	28%	28%	17%
		2	4326617	4335045	99.80%	99%	98%	82%
		3	4348274	4356360	99.85%	95%	94%	68%
		4	4411776	4370022	99.05%	63%	64%	40%
4b	32888142	1	4705107	4665371	99.15%	25%	25%	16%
		2	4706371	4715728	99.80%	91%	90%	75%
		3	4757431	4766119	99.82%	99%	98%	69%
		4	4817769	4776618	99.15%	46%	46%	30%
4c	32888102	1	7868474	7827749	99.48%	15%	15%	9%
		2	7905091	7913477	99.90%	100%	99%	78%
		3	7910030	7918067	99.90%	14%	14%	9%
		4	7970382	7928566	99.48%	28%	28%	18%
4d	30515938	1	4947440	4934183	99.73%	100%	100%	79%
		2	4919245	4955498	99.26%	84%	83%	60%
		3	4924184	4960088	99.27%	22%	22%	14%
		4	4984536	4970587	99.72%	45%	44%	28%

The MP3 decoder has only one design. The FPGA prototype was created with 5 MicroBlazes which are connected to each other through FSLs. The hybrid prototype has five emulated cores running on the MEK, and the OVP was used to create the virtual prototype. Table 6 shows the accuracy and busy time percentages for each core for FPGA, hybrid and virtual prototypes of the MP3 decoder.

Table 6: The MP3 decoder execution time

Core	FPGA prototype execution time	Hybrid prototype execution time	Accuracy	Busy time ratio		
				FPGA	Hybrid	OVP
1	31246757[1]	31250217	99.98 %	100%	100%	96%
2	32890978	32719520	99.47 %	22%	23%	0%
3	33058711	32908133	99.54 %	28%	28%	26%
4	32972069	32839963	99.59 %	17%	17%	5%
5	32962442	32780419	99.44 %	14%	14%	45%

[1] The time unit is nanoseconds.

Both FPGA and hybrid prototypes were created for packet forwarding application. MicroBlaze was used to implement the dispatcher, the inner cores and the collector while the channels were implemented by FSL in FPGA prototypes. Up to 8 MicroBlazes can be used on the FPGA due to MicroBlaze Debug Module (MDM) restriction. MDM can be connected to the maximum of eight MicroBlazes at the same time. Therefore, FPGA prototypes can be implemented with only up to eight cores. There is no such limitation in hybrid prototypes, as they use a single MicroBlaze. Therefore, the hybrid prototypes can be easily implemented for designs with more than 8 cores. Taking all these in to consideration, we created hybrid prototypes for multicore designs ranging from 1 to 22 inner cores for packet forwarding application. Table 7 contains the results for designs with up to 8 cores.

Table 7: Packet forwarding application execution time for all designs with up to 8 cores

# of cores	Hybrid prototyping total emulation time	Core	FPGA prototype execution time	Hybrid prototype execution time	Accuracy	Busy time ratio	
						FPGA	Hybrid
3	16543797[1]	1	8215201	8197827	99.78%	31%	31%
		2	8110358	8095852	99.82%	99%	99%
		3	8179239	8160706	99.77%	25%	25%
4	16321045	1	4253865	4243561	99.75%	30%	30%
		2	4185051	4179134	99.85%	98%	98%
		3	4147267	4117228	99.27%	97%	97%
		4	4191826	4163493	99.34%	25%	25%
5	16290722	1	2698869	2679012	99.26%	32%	32%
		2	2696830	2682205	99.45%	100%	100%
		3	2695860	2682673	99.51%	100%	100%
		4	2695925	2677983	99.33%	99%	99%
		5	2658783	2647032	99.55%	26%	26%
6	16275717	1	2060713	2047099	99.33%	32%	32%
		2	2085449	2074646	99.48%	100%	100%
		3	2032422	2024721	99.62%	99%	98%
		4	2064680	2059527	99.75%	97%	97%
		5	2059318	2048999	99.49%	97%	96%
		6	2025314	2019111	99.69%	26%	26%
7	16329727	1	1692604	1681916	99.36%	31%	31%
		2	1652163	1640601	99.30%	99%	99%
		3	1716628	1701179	99.10%	97%	97%
		4	1649340	1641103	99.50%	93%	92%
		5	1709723	1697389	99.27%	97%	97%
		6	1655231	1644970	99.38%	98%	97%
		7	1644068	1635644	99.48%	26%	26%
8	16448208	1	1383061	1387820	99.65%	32%	31%
		2	1411095	1400584	99.25%	100%	100%
		3	1419597	1411888	99.45%	100%	100%
		4	1410275	1399360	99.22%	99%	99%
		5	1416298	1407350	99.36%	92%	91%
		6	1423591	1415296	99.41%	90%	89%
		7	1379922	1371985	99.42%	93%	92%
		8	1340852	1329655	99.16%	26%	27%

[1] The time unit is CPU cycles.

Figure 32 illustrates the results for designs with up to 8 cores. The X-axis shows the number of cores and Y-axis shows the execution time in million cycles for each design.

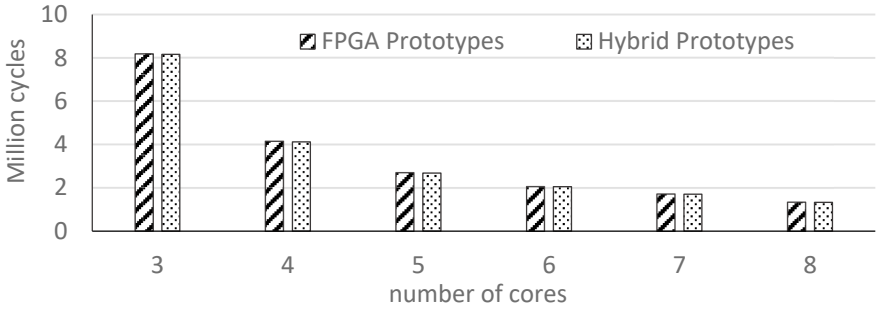


Figure 32: Packet forwarding application execution time for all designs with up to 8 cores

We created both hybrid and FPGA prototypes for the MP3 decoder and the JPEG encoder by using multi-clock domains (60, 90, 25, 45 and 55 MHz) in which each core is run with different clock frequencies. Table 8 contains the results for the MP3 decoder.

Table 8: MP3 decoder results with multiple clock domains

Core	Clock	FPGA prototype execution time	Hybrid prototype execution time	Accuracy	Busy time ratio	
					FPGA	Hybrid
1	60 MHz	518693760[1]	520836936	99.58 %	100%	100%
2	90 MHz	559167104	560874804	99.69 %	14%	14%
3	25 MHz	560718592	566038909	99.06 %	43%	43%
4	45 MHz	560128320	563309406	99.43 %	22%	22%
5	55 MHz	560088448	561985784	99.66 %	15%	15%

[1] The time unit is nanoseconds.

Figure 33 shows the results for all 15 designs mentioned in Table 4 for the JPEG encoder with multi-clock domains. The X-axis shows the designs and Y-axis shows the execution time in million cycles for each design.

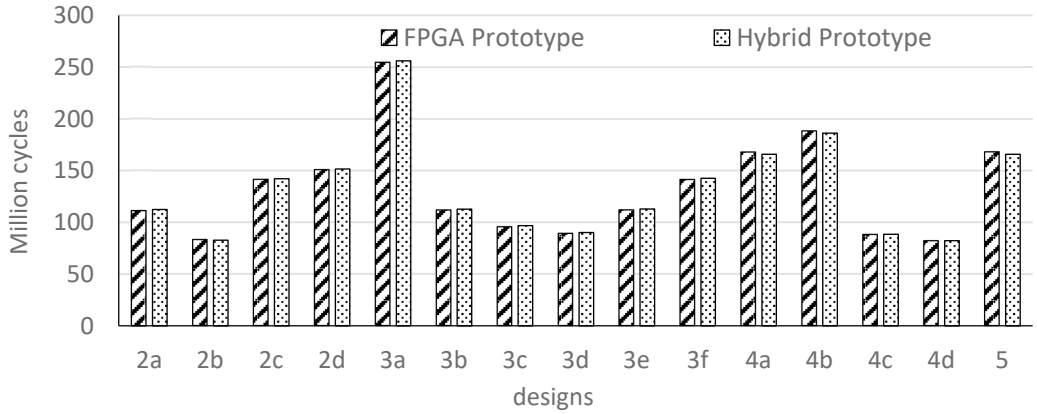


Figure 33: The execution times for FPGA and hybrid prototypes for the JPEG encoder with multiple clock domains

Table 9 shows the results for all designs of JPEG encoder application. These results show that the hybrid prototypes are accurate with multiple clock domains and that they report the same number of cycles for each task as measured by the FPGA prototypes. As it was described earlier, all the inner cores in packet forwarding application are doing same processing, therefore, there would be no point to use multiple clock domains for it.

The hybrid prototype reported exactly the same number of cycles for each task as measured by the FPGA prototype. This is because we execute the tasks on the same core as in the FPGA prototype. In contrast, because of the high abstraction level of the underlying ISS, OVP simulation had an error of over 25% in the number of cycles reported. Furthermore, OVP can only report busy time for each core because it is an instruction accurate simulator**.** Therefore, hybrid prototype was shown to be more reliable than abstract virtual prototypes.

Table 9: The JPEG encoder all possible design results with multiple clock domains

Design	Hybrid prototyping total emulation time	Core	Clock	FPGA prototype execution time	Hybrid prototype execution time	Accuracy	Busy time ratio	
							FPGA	Hybrid
1	629268678[1]	1	60 MHz	163592080	161934608	98.98%	12%	12%
		2	90 MHz	165787840	163735376	98.76%	29%	29%
		3	25 MHz	164717088	165544016	99.49%	99%	99%
		4	45 MHz	166585072	165646112	99.43%	14%	15%
		5	55 MHz	168085792	165836912	98.66%	24%	24%
2a	142173208	1	60 MHz	111329376	112436944	99.00%	18%	18%
		2	90 MHz	113249352	113645104	99.65%	99%	99%
2b	143218269	1	60 MHz	82245400	82379056	99.83%	99%	100%
		2	90 MHz	83217800	82752432	99.44%	85%	86%
2c	143387489	1	60 MHz	141520480	142068624	99.61%	100%	100%
		2	90 MHz	142504016	142220304	99.80%	22%	22%
2d	143371408	1	60 MHz	150971520	151547664	99.61%	100%	100%
		2	90 MHz	151956928	151664208	99.80%	16%	16%
3a	276169513	1	60 MHz	249373888	250510384	99.54%	8%	8%
		2	90 MHz	252952912	253386448	99.82%	19%	19%
		3	25 MHz	254799392	256137024	99.47%	100%	99%
3b	276544872	1	60 MHz	110199088	110328288	99.88%	18%	18%
		2	90 MHz	111642624	111607512	99.96%	78%	78%
		3	25 MHz	111883280	112760784	99.21%	99%	99%
3c	276587332	1	60 MHz	94964488	95294840	99.65%	21%	21%
		2	90 MHz	96082808	96312088	99.76%	97%	97%
		3	25 MHz	95658920	96772440	99.83%	92%	91%
3d	277076094	1	60 MHz	88173024	88211656	99.95%	93%	93%
		2	90 MHz	89276248	89237840	99.95%	58%	58%
		3	25 MHz	89261640	90138072	99.01%	99%	98%
3e	276562294	1	60 MHz	110316464	110444032	99.88%	74%	74%
		2	90 MHz	111759408	111723256	99.96%	41%	41%
		3	25 MHz	112000080	112876528	99.21%	99%	98%
3f	277978006	1	60 MHz	141520480	142068624	99.61%	100%	100%
		2	90 MHz	141853152	142119232	99.81%	8%	8%
		3	25 MHz	141433840	142580096	99.18%	62%	62%
4a	452180251	1	60 MHz	163592080	161934608	98.98%	12%	12%
		2	90 MHz	165787840	163735376	98.76%	29%	29%
		3	25 MHz	164717088	165544016	99.49%	99%	99%
		4	45 MHz	167886784	165847264	98.78%	37%	37%
4b	452077382	1	60 MHz	185770512	184201264	99.15%	11%	11%
		2	90 MHz	188306144	186255024	99.91%	25%	25%
		3	25 MHz	187490688	188316672	99.55%	99%	99%
		4	45 MHz	190587840	188549632	98.93%	26%	26%
4c	434297936	1	60 MHz	87211288	87537448	99.62%	22%	23%
		2	90 MHz	88228984	88484408	99.71%	99%	99%
		3	25 MHz	85587480	85708400	99.85%	50%	50%
		4	45 MHz	88694840	88941360	99.72%	55%	55%
4d	414815049	1	60 MHz	82118512	82243696	99.84%	100%	100%
		2	90 MHz	82404392	82481352	99.90%	55%	55%
		3	25 MHz	79762280	79705344	99.92%	54%	54%
		4	45 MHz	82870224	82938304	99.91%	59%	59%

[1] The time unit is nanoseconds.

6.2.2 Speed

Figure 34 shows speed comparison between hybrid, FPGA and virtual prototypes to execute the JPEG encoder for a given image. The X-axis is the number of cores and the Y-axis is the simulation time in milliseconds. The real execution time can easily be obtained by multiplying the number of cycles with the clock period.

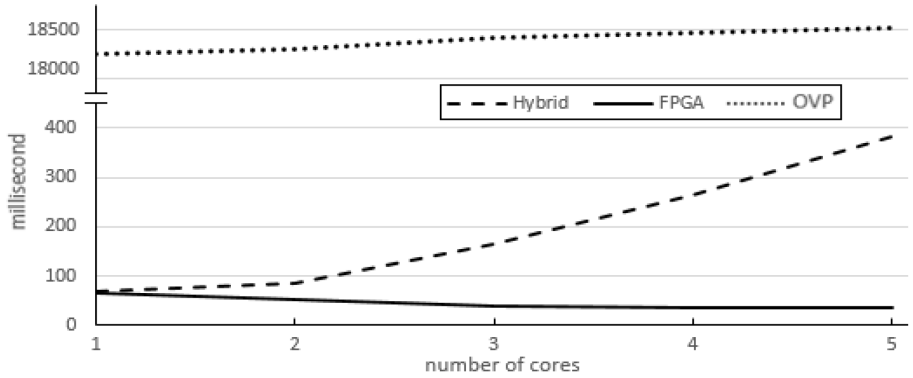


Figure 34: Prototyping speed comparison between FPGA, hybrid and OVP prototypes for the JPEG encoder

Figure 35 shows speed and overhead comparison between hybrid and FPGA prototypes using the number of cycles needed to execute the JPEG encoder for a given image. The X-axis is the design and the Y-axis is the number of cycles in millions. The overhead is defined as the difference between the cycles for simulating JPEG on the hybrid prototype and those on the FPGA prototype. As we can see, the hybrid prototype takes approximately the same time for all mappings with a given number of cores. This is because the total inter-core data communication is the same for different mappings of JPEG. The small variations are due to different absolute communication times for each channel, and the variations in task scheduling in the RTOS scheduler model. We can also see that the worst case overhead for a given number of cores scales well with the number of cores and the total amount of data communication.

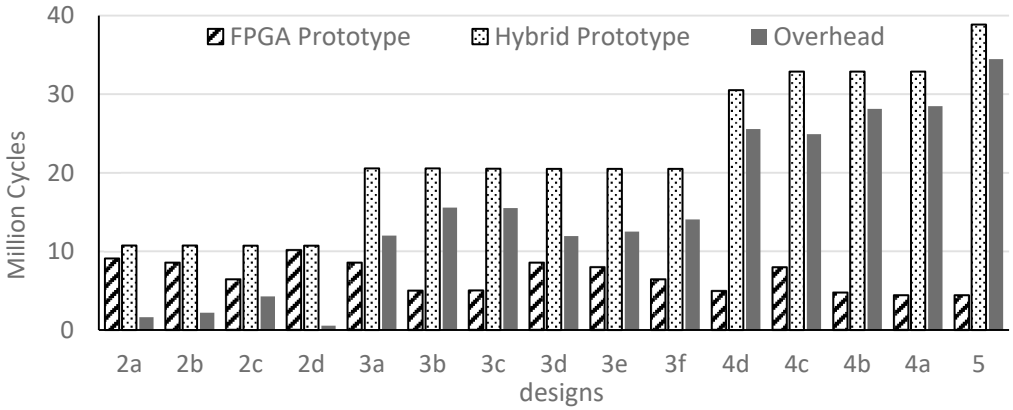


Figure 35: The simulation times for FPGA and hybrid design for the JPEG encoder application

The overhead of the hybrid prototype itself can be observed as the difference between the hybrid prototyping simulation times and the 1-core JPEG FPGA prototype execution time, since the total computation on the core stays constant. The hybrid prototype overhead consists of the wall clock time used for task/event management, scheduling and channel calls. As we can see, the hybrid prototype overhead also scales well with the number of cores and the amount of channel communication.

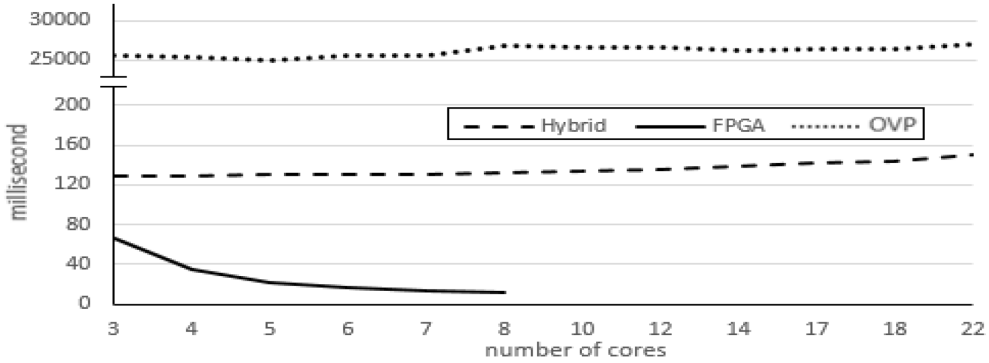


Figure 36: Prototyping speed comparison between FPGA, hybrid and OVP prototypes for Packet forwarding application

Figure 36 shows the speed comparison between hybrid, FPGA and OVP prototypes for packet forwarding application. The X-axis is the number of cores used in each design and the Y-axis is the simulation time in milliseconds. Up to eight

MicroBlazes can be used on the FPGA due to MDM restriction. MDM can be connected to the maximum of eight MicroBlazes at the same time. Therefore, FPGA prototypes can be implemented with only up to eight cores.

In the most complex design with 5 cores, the hybrid prototype took about 40 M cycles (or 400 ms) to simulate JPEG. On the other hand, the FPGA prototype took 4 M cycles (or 40 ms). In contrast, the virtual prototyping using OVP took over 20 seconds on a 2GHz Pentium host with 8GB of RAM and the behavioral RTL simulation of the 5-core design took over 3 hours on a 2GHz Pentium host with 8GB of RAM. We were unable to create a 5-core virtual prototype, because the Xilinx Virtual Platform (XVP) simulator supports only a single instance of MicroBlaze [16]. For the 1-core design, the XVP took 3 minutes to simulate JPEG on the same host as the one used for RTL simulation. Based on the above results, we can conclude that hybrid prototyping outperforms both cycle-accurate RTL software simulation and virtual prototypes. The FPGA prototype took 33M cycles (330 ms) to execute the MP3 decoder while the hybrid prototype took 47.25M cycles (470 ms) to emulate it and OVP took about 28 second to simulate the design. The FPGA prototype with eight cores took 11 ms to execute packet forwarding application, while the hybrid prototypes took 131 ms to emulate the application. In contrast, the OVP took 26 seconds to simulate the design.

Different multicore designs (ranging from 1 to 5 cores) also created to dynamically schedule JPEG encoder's threads on top of them. Figure 37 shows the simulation time needed to execute the JPEG encoder for the given image reported by hybrid prototypes. The Y-axis shows the simulation time in seconds and the X-axis shows the number of cores used in different multicore designs.

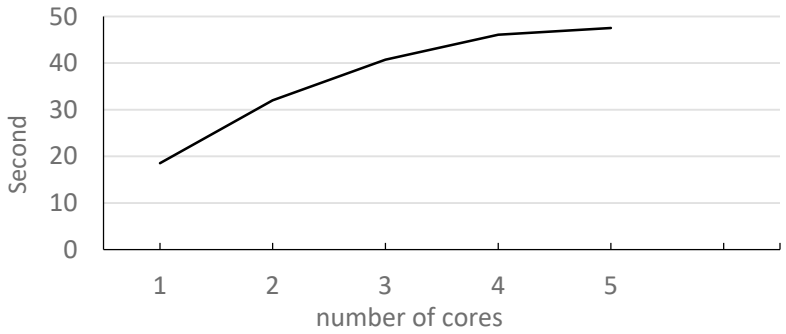


Figure 37: Simulation time (second) reported by the hybrid prototype with dynamic scheduling with different number of cores for JPEG encoder

To have dynamic scheduling model of the MP3 decoder, different multicore designs with up to 4 cores are created. Figure 38 shows the simulation time needed to execute the MP3 decoder application reported by hybrid prototypes for different designs with different number of core. The Y-axis shows the simulation time in second and the X-axis shows the number of cores used in different multicore designs.

Simulation took longer with dynamic scheduling in hybrid prototypes. This is because of the RTOS scheduling and more kernel calls. However, as Figure 37 and Figure 38 show the simulation time increases linearly with the number of emulated cores.

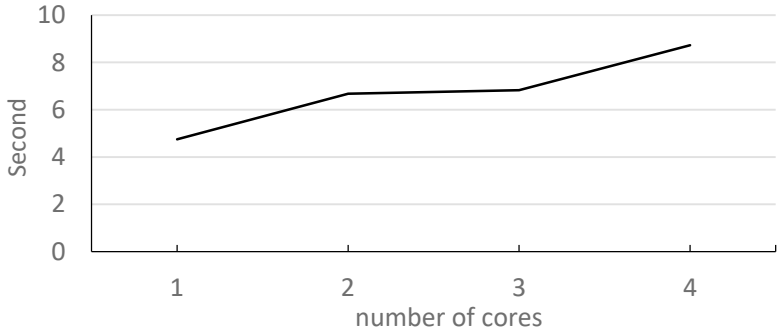


Figure 38: Simulation time (second) reported by the hybrid prototype with dynamic scheduling with different number of cores for MP3 decoder

As we have seen, a hybrid prototype can provide highly cycle-accurate and fast simulation similar to FPGA-based prototypes. The simulation time also increases linearly with the number of emulated cores.

6.2.3 Scalability

As it was mentioned earlier, due to MDM limitation, the number of MicroBlazes are limited to 8 in FPGA prototypes. To overcome this problem, multiple FPGAs can be used, but with lots of effort. In contrast, in hybrid prototypes, there is no limitation to have designs with more than 8 cores. Furthermore, as it was described before, the experimental results for the JPEG encoder and the packet forwarding applications show that the hybrid prototype simulation time increases linearly when the number of cores are being increased. Therefore, hybrid prototyping provides scalable models of multicore embedded system design.

6.2.4 Modeling Effort

Modeling effort is a difficult metric to measure because of the human element. In creating our experimental setup, we found out it was very difficult to debug the FPGA prototypes as the number of cores increased. We used a JTAG based debug module provided in the Xilinx Embedded Development Kit. The I/O from the different cores was sent to the hyper-terminal on the host. In the case of multiple cores, it was difficult to sort them through the debug messages from the different cores. Figure 39 shows the difference between the hybrid prototype and FPGA prototype hyper-terminal output. As it shows, hybrid prototype output is more clear and readable comparing to FPGA prototype output. FPGA prototype output is hard to understand because all cores write into terminal simultaneously and they override other cores' output. To

solve this problem in FPGA prototypes we need to use synchronization mechanism like mutex which make FPGA prototype much more complicated.

```
23  playbackProc
24  I_AUDIO_TX_READY received
25  SSRdinead A_AUD_M_XER_eIX__fifo
26  L
27  W:AtDIO_A.XIR_MI fCHm DSP_AeDeOvISR:enttempDECg EoFwrit
28   ...DEyFRs Eo sceial
29  uffiaDecod
30  rinnnn .d. byten to
31  cSR: Wrfting.. i bytei to
32  ISR:nnendSngdinA DIOUDXO_EIDE
33  ISRDIOrDAT
34  _eUD I_AX_fIOoMXXERreplyyfiffo
35  SP_AUDIO: I_AUDIO_MIXER_AUDIO_DATA received
```

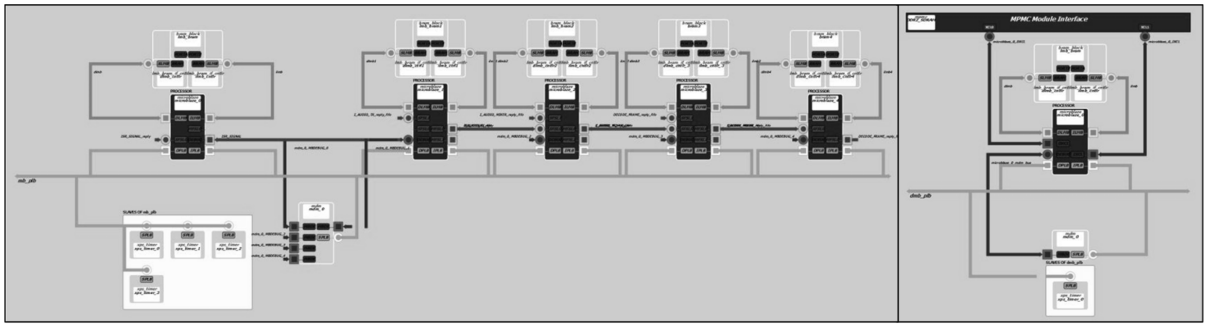
(a) FPGA prototype output

```
23  core4 : Write I_AUDIO_MIXER_reply_fifo
24  core2 : ISR: Write I_AUDIO_TX_fifo
25  core2 : ISR: Read I_AUDIO_TX_reply_fifo
26  core2 : ISR: Got ... bytes from DSP_AUDIO
27  core2 : ISR: Attempting to write ... bytes to serial buff
28  core2 : ISR: Writing ... bytes to
29  core2 : ISR: Writing ... bytes to
30  core2 : ISR: Sending I_AUDIO_TX_READY
31  core3 : I_AUDIO_TX_READY received
32  core3 : Sending I_AUDIO_MIXER_MIX_CHANNEL
33  core4 : I_AUDIO_MIXER_MIX_CHANNEL received
34  core4 : Sending DECODE_FRAME
35  core5 : DECODE_FRAME received
```

(b) Hybrid prototype output

Figure 39: Hybrid prototype vs. FPGA prototype hyper-terminal output

Furthermore, it takes a significant amount of time to design, implement and test the inter-core communication architecture on the FPGA comparing to hybrid prototypes. Because in the hybrid prototype, we have to interface with only one core, and the state of the core being emulated was easily observed at any given time. Figure 40 shows the complexity of hybrid prototype vs FPGA prototype for running MP3 decoder application. In summary, we found it much more challenging to implement and validate the FPGA prototypes than the hybrid ones.



(a) FPGA prototype (b) Hybrid prototype

Figure 40: Hybrid prototype vs. FPGA prototype hardware design for MP3 decoder application

6.3 Design Space Exploration

Design Space Exploration (DSE) is the process of analyzing and modeling several possible design alternatives prior to implementation. By using DSE, designers can discover and evaluate their designs during system development. DSE is critical for design optimization before silicon is available. Rapid prototyping is often used to implement a set of prototypes for different design choices. By analyzing these prototypes, designers can improve their understanding of the impact of design decisions. The set of prototypes can be compared using well defined metrics such as execution time, cost and power consumption. As such, DSE can be used to discover the optimization possibilities before implementation [56]. Our primary goal with hybrid prototyping is to make DSE fast, early and reliable [42].

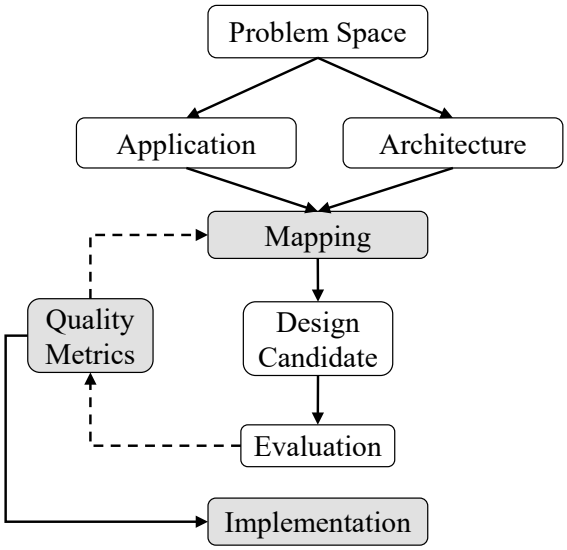


Figure 41: Design Space Exploration

Figure 41 explains how DSE is performed using hybrid prototyping. For a given application and architecture, there are several possible mappings. Different mappings are created and evaluated as per the chosen quality metrics. Eventually, the designer can select the best design amongst the evaluated mappings. For each design, the hybrid

prototype provides a simple energy consumption model and a highly accurate estimation of the application's execution time. We consider execution time (speed) and energy consumption as quality metrics for DSE.

6.3.1 Speed

As it was mentioned earlier, a hybrid prototype provides a highly accurate estimation of the application's execution time for a given design. The timing estimates are generated for both total execution time and busy time for each core. In a given multicore design, the longest execution time amongst all tasks (mapped to different cores) can be considered as the design's total execution time. So by comparing the total task execution times on all cores, we can determine the speed of a multi-core design.

6.3.2 Energy Estimation

Energy consumption is one of the most important quality metrics in embedded system design. The power consumption of a core is directly related to its frequency. Most embedded processors support several operating frequencies, which allows us to create a mixture of cores, each running at a different operating point. The busy power consumption is a measure of the power which is consumed by the core when it executes the instructions. The idle power consumption is a measure of the, largely static, power consumed by the core while it waits on external events, and does not execute any instruction. We used the Xilinx XPower analyzer [16] to measure both the busy and idle power consumptions. The idle dynamic power is zero for the MicroBlaze when it waits on the FSL communication channels. The Static power, consumed at all times irrespective of whether the core is busy or idle, is the same for all cores with different clock domains and is measured to be 1.48 mw. If the clock frequency is increased, the

power consumption will increase as well. As CPU and memory are the most power consuming parts in our designs, we consider the busy power as sum of CPU power and memory power. Table 10 shows the average busy power for MicroBlaze and BRAM.

Table 10: The busy power consumption for different clock domains

Frequency	MicroBlaze	BRAM
25 MHz	07.14 mw	14.57 mw
45 MHz	12.23 mw	25.68 mw
55 MHz	14.65 mw	30.80 mw
60 MHz	16.00 mw	34.01 mw
90 MHz	23.24 mw	50.65 mw
125 MHz	31.91 mw	68.91 mw

A simplistic, yet reasonably accurate, power model of a processor assigns a single power consumption number to each operating point. Clearly, the processor is only consuming dynamic power when it is busy. Since different mappings may result in different busy times for the cores, we can change the mapping in order to obtain the best energy consumption by the design. Using a hybrid prototype, the designer can quickly obtain the busy times for the design with different operating frequencies and mappings. The estimated energy consumption for each emulated core can be calculated by the following equations.

$$Energy = (Idle_{time} \times Idle_{power}) + (busy_{time} \times busy_{power}) \quad (4)$$

$$busy_{power} = CPU_{busy_{power}} + Memory_{busy_{power}} \quad (5)$$

6.3.3 Automatic Design Space Exploration

Hybrid prototyping is extended to support automatic design space exploration. In most multicore embedded systems, there are several possible design options depending

on the number of cores, their frequencies and the mapping of application tasks to the cores. Each of the design options may consume different energies, may have different execution times and different chip area. Higher core frequency results in greater power consumption. Less power design, using lower operating frequencies, increases the execution time. There may be chip area constraints as well, which limit the number of cores that can be implemented on the chip. Moreover, limited parallelism in the application may limit the speed advantage of adding more cores to the design. Therefore, the right multicore design for a given application is not obvious until accurate models of possible design options have been evaluated.

Clearly, implementing all possible designs is often impractical. So, the designer needs a mechanism to evaluate most of the promising design options before implementation. Hybrid prototyping provides such mechanism for designer. As mentioned earlier, the hybrid prototype can calculate energy and execution time for each design, for a given input. So, the designer can evaluate any designs in terms of energy, speed and area.

As we described before, the JPEG encoder application has five tasks which can be mapped on 1 to 5 cores. Each core can have a different CPU clock frequency. Also we can have different mapping of tasks on the cores. The equation 6 shows the number of all possible designs.

$$Number\ of\ Design = \sum_{1}^{Max}(M \times C^{D}) \quad (6)$$

Where max is the maximum number of cores, M is the number of possible mapping, C is the number of cores and D is the number of clock domains. Table 11 shows all possible numbers of designs with different clock domains and maximum five different cores. For instance, the third column shows that there are 16 possible designs with only 1 clock domain which were described in Table 4. By increasing the number of

clocks, the number of possible designs increases dramatically. There are 14406 different possible designs for JPEG encoding application with six clock frequencies. Therefore, it is impractical to implement all these designs and choose the best one.

Table 11: Number of all possible design with multiple clock domains

# cores	# mapping	Number of possible design with different clock domains					
		1	**2**	**3**	**4**	**5**	**6**
1	1	1	2	3	4	5	6
2	4	4	16	36	64	100	144
3	6	6	48	162	384	750	1296
4	4	4	64	324	1024	2500	5184
5	1	1	32	243	1024	3125	7776
Total		**16**	**162**	**768**	**2500**	**6480**	**14406**

To overcome this problem, we can create a script for the hybrid prototype that takes the following inputs: clock domains, the maximum number of the cores and possible mappings to generate all possible models. In the most complex design the hybrid prototype takes 165ms to complete the simulation. Therefore, the total time for all 14406 design can take around 40 minutes to be done in the hybrid prototyping framework. The result is a log which contains the design mapping, total execution time, the busy time and energy consumption, for each core as well as the entire design. Designers can use this output to analyze all designs and choose the best one. For instance, we can use the hybrid prototype to log the results for all possible designs with two different clock domains (60 MHz and 125 MHz).

Our experiments with the hybrid prototyping demonstrate its applicability to fast multicore design space exploration. We have modeled the 162 possible designs of the JPEG encoder which is being run with 2 clock domains (60 and 125 MHz). Xilinx Virtual Platform (XVP) simulation shows errors of over 40% in the number of cycles reported because of its high abstraction level. The FPGA prototype takes 35ms to

execute and 15 minutes to synthesize every design choice. Therefore, full FPGA prototyping takes almost 40 hours for all 162 possible designs without considering the effort of creating the FPGA prototypes. In contrast to the above techniques, it takes only 15 minutes to synthesize the hybrid platform's target core, which is a one-time effort. The hybrid prototype takes 382ms (in worst case) to emulate each design, thereby enabling extremely fast, early and reliable design space exploration. Figure 42 plots speed vs. energy consumption reported by the hybrid prototypes for all 162 designs which each spot presents a design. The circle highlights the best designs that consume minimal energy and shortest execution time.

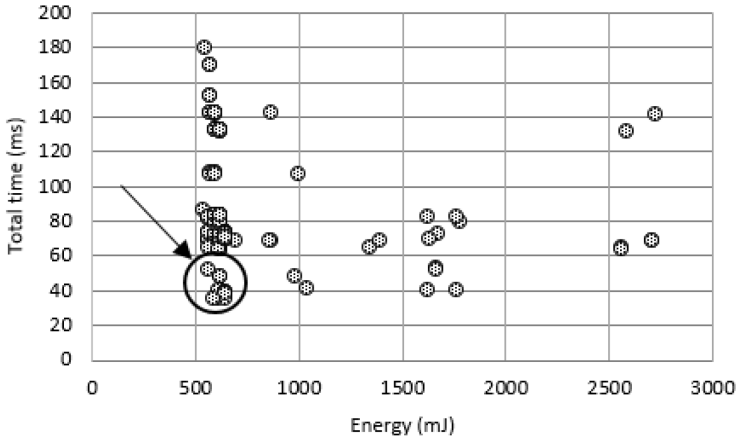


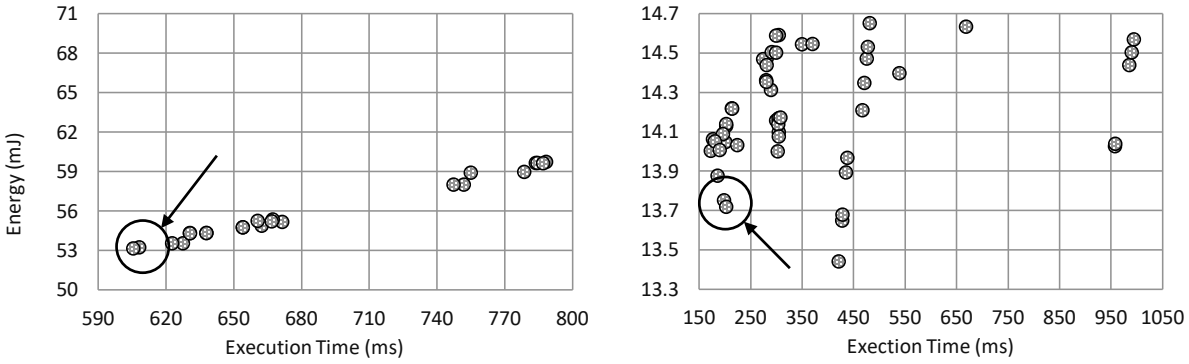
Figure 42: Scatter chart for design exploration with two different clock domains

6.3.4 Dynamic scheduling

Each design can have different number of cores clocked at different speeds. It is also possible to have threads running with different priorities which may affect the total execution time and energy consumption for the application. For each design, the hybrid prototype provides a simple energy consumption model and a highly accurate estimation of the application's execution time. So, by comparing the execution times

and energy consumption for each design, we can reliably analyze the power and performance implication of their optimizations before the hardware is available.

Our experiments with the hybrid prototyping also demonstrate its fast design space exploration for SMP designs. We have modeled 50 different designs of the JPEG encoder and 40 different designs for the MP3 decoder with different threads' priorities which are being run on up to 5 cores with different clock frequencies (55, 60, 90, 25, 45 and 125 MHz). The FPGA prototype takes 400ms (in average) to execute and 15 minutes to synthesize every design choice. Therefore, full FPGA prototyping takes almost 12.5 hours for all the 50 designs without considering the effort of creating the FPGA prototypes. In contrast to FPGA prototyping, it takes only 15 minutes to synthesize the hybrid platform's target core, which is a one-time effort. The hybrid prototype takes 48 second (in worst case) to emulate each design, thereby enabling fast, early and reliable design space exploration. Figure 43 plots speed vs. energy consumption reported by the hybrid prototypes for all different designs for JPEG encoder and MP3 decoder applications in which each spot presents a design. The circle highlights the best designs that consume minimal energy and shortest execution time.



(a) MP3 decoder application **(b) JPEG encoder application**

Figure 43: Speed vs. Energy consumption for different SMP designs with multi-clock domains and different threads' priorities for the JPEG encoder and MP3 decoder applications

6.4 Dynamically Reconfigurable Active Cache

DRAC was implemented in VHDL, and synthesized using Xilinx ISE toolset [16] on ML507 evaluation board using Vertex5 FPGA. The soft-core MicroBlaze processor, running at 125MHz, was chosen as the target core for all the experimental cases.

6.4.1 Standalone Accuracy

Prior to using DRAC in hybrid prototyping, we evaluated the standalone model in a single core design. In order to check the functionality and timing accuracy of the standalone instruction and data DRAC model, we ran JPEG Encoder, Quicksort, and Dhrystone benchmarks for different cache sizes in a single core design. The MicroBlaze built-in cache configuration was set to direct map, 4-word line size, with write through policy. The result for different cases is shown in Table 12. We observed an average error of 3% and the worst-case error is only 5%, thereby demonstrating the accuracy of DRAC as a standalone cache model.

Table 12: Estimation accuracy of standalone DRAC

Benchmark	**Cache Size**	**T_{BIC}** (Million Cycles)	**T_{DRAC}** (Million Cycles)	**Error** %
JPEG	256B	48.63	48.05	-1.18
	1KB	23.19	23.31	0.49
	2KB	18.11	17.91	-1.10
	4KB	13.72	13.45	-1.98
	8KB	12.55	12.18	-2.90
Quicksort	256B	13.83	13.13	-5.06
	1KB	12.27	11.72	-4.48
	2KB	9.76	9.32	-4.59
	4KB	6.28	5.99	-4.61
	8KB	6.28	5.99	-4.61
Dhrystone	256B	22.25	22.79	2.41
	1KB	8.79	9.02	2.63
	2KB	7.90	8.05	1.90
	4KB	7.90	8.05	1.90
	8KB	7.90	8.05	1.90

6.4.2 Accuracy in the Hybrid Prototype

We created 15 different multicore designs for JPEG Encoder in the full FPGA design and the hybrid prototype, ranging from 1 to 4 cores. Each core is running different tasks of JPEG with different mappings to the cores. In the full FPGA design, cores are connected to each other with FIFOs. Reading/writing from/to FIFO's is blocking method. Cores stop executing unless there is a value on the FSL.

Table 13: Estimation accuracy of DRAC-based hybrid prototype

Number of cores	Mapping	Average Error	Worst-case	
			Error	Cache Size
2core	4-1	3.17%	6.76%	1 KB
	3-2	4.24%	8.56%	1 KB
	2-3	3.50%	12.3%	4 KB
	1-4	5.88%	10.2%	1 KB
3core	1-1-3	4.50%	7.24%	8 KB
	1-2-2	2.73%	4.34%	4 KB
	1-3-1	3.76%	7.81%	1 KB
	2-2-1	3.24%	6.73%	256 B
	2-1-2	4.76%	6.92%	256 B
	3-1-1	6.76%	10.8%	8 KB
4core	1-1-1-2	12.24%	12.98%	4 KB
	1-1-2-1	10.80%	12.55%	4 KB
	1-2-1-1	5.96%	9.78%	2 KB
	2-1-1-1	6.27%	9.09%	8 KB

There are two timers implemented on each core in full FPGA prototype. The first timer calculates the actual busy-time of a core regardless of that core's waiting time on blocking reads or writes. The second timer starts at the first of the program and measures the total execution time including program execution time and FSL waiting times. In hybrid design, there are also two timers. One timer is used by the MEK, to simulate the busy-time and the total execution time of each core; the other timer calculates the total simulation time, including the swap time, the total execution time of the tasks, and the MEK software. Table 13 presents the total execution time error

for different JPEG mappings and cache sizes. The mapping values represent number of JPEG tasks that have mapped to each core. For example, in the 2core design, mapping 4-1 means 4 tasks of JPEG have been mapped to the first core, and one task to the second core. As it is shown, the average error is 9.00% and the worst case error is 13% in the 4 core design.

6.4.3 Simulation Speed

As mentioned earlier, there is a timer for calculating the total simulation time. It starts at the first of the simulation, and stops at the end of the procedure. The total simulation time of the hybrid prototype can be seen in Figure 44.

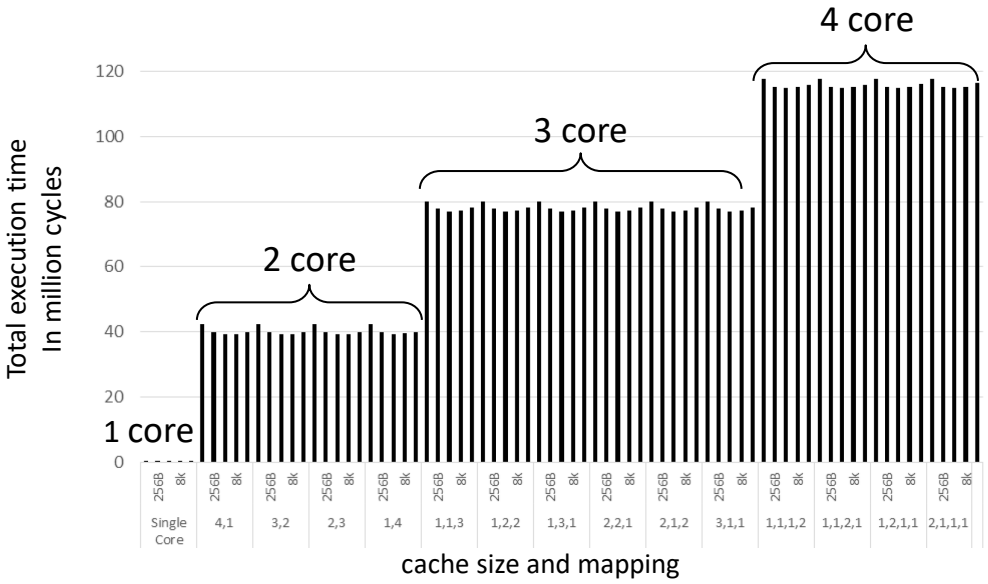


Figure 44: Simulation speed of hybrid prototypes with DRAC

The values are obtained for all task mappings and all 5 different cache sizes ranging from 256B to 8KB. Because the hybrid prototype platform is running on off-chip DDR2 SDRAM memory, and the swap is also running during the simulation, the timing is quite high in compare to hybrid prototype running on BRAM. The simulation time increases by increasing the number of cores, since number of cache swaps increase.

During simulation, both instruction and data cache is disabled, and the cache is enabled only when a task is running. Because of this, cache size increment effect is not significant in total simulation time. Even in some cases, the cache size increment results in higher simulation timing. The reason is that if the cache size increases, the swap time increases as well. Table 14 reports the time consumption for a load/save from/to DDR2 to/from the cache, and total swap (Load + Save).

Table 14: Swap time consumption for different L1 sizes

Cache Size	Save (Cycles)	Load (Cycles)	Total Swap Time (Cycles)
256B	2467	3831	6298
1k	8499	13725	22224
2k	16477	26957	43434
4k	32513	53343	85856
8k	64521	106123	170644

6.4.4 DRAC Resource Usage

Each design consumes a certain amount of time to be synthesized, and occupies a portion of FPGA area during implementation. For each full FPGA multicore and hybrid design, we have obtained synthesis time and resource usage. Table 15 presents resources usage of the hybrid and full FPGA multicore designs. As it can be seen, as much the number of cores in the full FPGA design increases, the synthesis time and the area consumption gets higher. The full FPGA design must be synthesized once, with any configuration change like cache size; however, the hybrid design can emulate any configuration with only one time synthesis. Furthermore, in the hybrid design the area consumption remains the same during emulation of all designs and configurations.

Table 15: Resource usage of hybrid vs FPGA prototype

Design	Synthesis Time	Host PC memory Usage	Resources Usage Percentage				
			LUT	BRAM	Reg.	Slice	Bonded IO
Real 2core	332s	737 MB	16%	23%	17%	36%	18%
Real 3core	424s	802 MB	21%	35%	23%	47%	18%
Real 3core	639s	925 MB	28%	57%	28%	59%	18%
MEK with DRAC	500s	656 MB	22%	33%	23%	53%	18%

6.4.5 Energy Analysis

Beside the speed of the system, power consumption is the other main factor for the designer to choose the best design in multicore processing. The main components that consume the most of the power are the processor, built-in cache, and off-chip main memory. Figure 45 demonstrates the total power consumption of the system for different number of cores and cache sizes. As can be seen, the cache size increment results in more BRAM utilization and more power consumption. On the other hand, adding more cores to the system and using more ports of MPMC increase the power consumption as well.

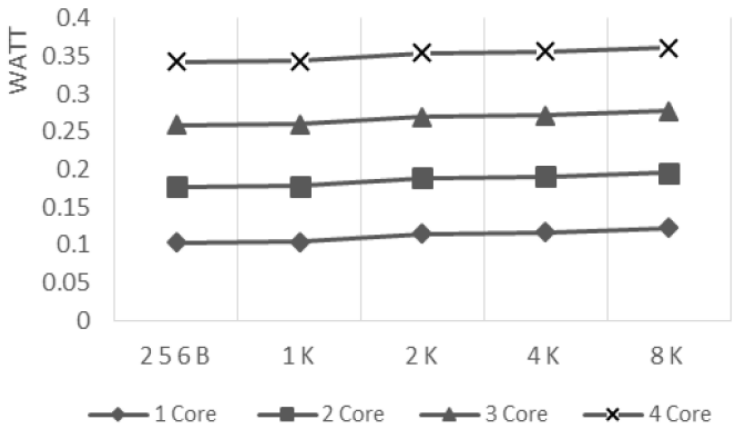


Figure 45: Power consumption for different L1 cache sizes

In multicore systems, power consumption is different core by core, depending on the task running on each core. Energy is the best way to measure the system performance in terms of power and time. The busy-time of a task is the time for a core to execute a task without considering blocking data transfer among different cores. The processor is on idle during blocking reads or writes, hence it consumes negligible amount of energy. Because of this fact, we multiplied the total power consumption of each core to the total busy-time of all cores and obtained the energy consumption for different task mappings.

6.4.6 Design Space Exploration

Two of the most important factors that define system efficiency, are the speed of the system and the energy consumption of the design. Figure 46 plots all the full FPGA multicore designs from 2 to 4 cores with all possible JPEG Encoder mappings, and five different cache sizes execution time versus energy consumption. Each point is a design with certain mapping and the cache size. As it is circled on the figure, the best designs are the one that consume less energy and execute the program in the shortest time. For example, the best design in JPEG Encoder is a 2 core design with 2k cache size and the mapping of 2 tasks in the first core and 3 tasks in the second core.

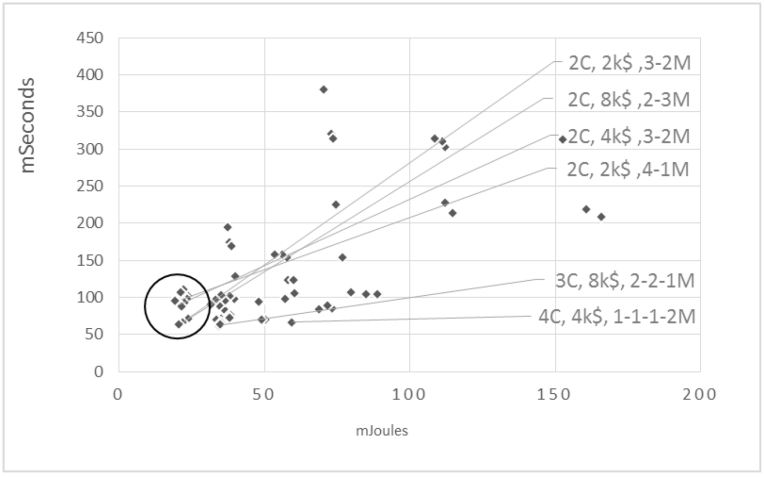


Figure 46: Design exploration using full FPGA prototype

The hybrid prototype provides a simple environment for the designer to choose the best design among the others, without having the full FPGA multicore prototype. The consistency of the results, 100% relative accuracy among different cache sizes and different task mapping, make the hybrid prototype a powerful tool to compare different designs. Figure 47 presents energy versus execution time for all JPEG Encoder possible mapping and the cache sizes, predicted by the hybrid prototype. The correlation of the hybrid prototype results and the full FPGA results is clear. In both Figure 46 and Figure 47, the best design is the 2core design with 2k instruction and data cache with 3-2 JPEG mapping. This confirms the accuracy and reliability of the hybrid prototype with cache.

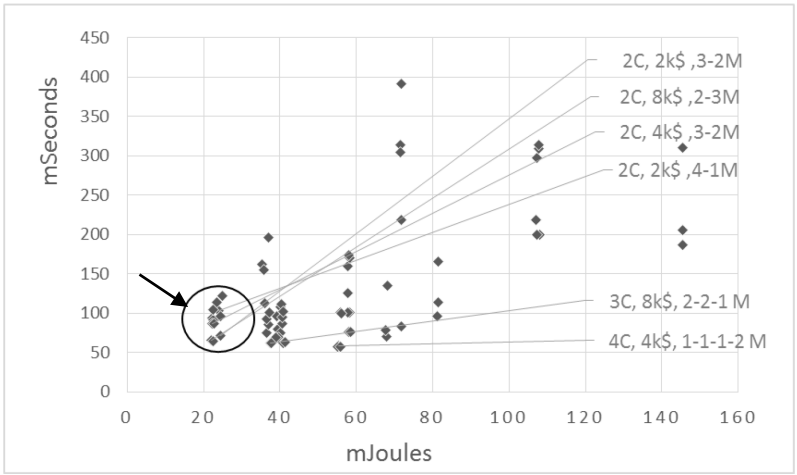


Figure 47: Design exploration using hybrid prototype

6.5 Summary

In this chapter we evaluated the hybrid prototypes and compared them with virtual and FPGA prototypes. We have seen, a hybrid prototype can provide highly cycle-accurate, scalable and fast simulation similar to FPGA-based prototypes. In contrast, virtual prototypes had an error of over 25% in the number of cycles reported. The simulation time also increases linearly with the number of emulated cores. We also found it much more challenging to implement and validate the FPGA prototypes than the hybrid ones. Furthermore, the experiments with the hybrid prototyping has demonstrated its applicability to fast multicore design space exploration.

Chapter 7

Conclusions and future work

In this thesis we have presented a novel modeling technique called hybrid prototyping that aims to provide early, fast, cycle-accurate and scalable models of multicore embedded systems. It also provides the modeling of a dynamic RTOS scheduler as well as hardware interrupts on top of the MEK, in order to support the simulation of unmodified multi-threaded applications.

Our experimental results demonstrate the high accuracy, simulation speed and scalability of our hybrid prototyping-based simulation models. The hybrid prototype reported exactly the same number of cycles for each task as measured by the FPGA prototype. This is because it executes the tasks on the same core as in the FPGA prototype. In contrast, because of the high abstraction level of the underlying ISS, OVP simulation had an error of over 25% in the number of cycles reported. Therefore, hybrid prototype was shown to be more reliable than abstract virtual prototypes.

Our experiments with the hybrid prototyping also demonstrate its applicability to fast multicore design space exploration. Multicore embedded system designers can

create concurrent applications and accurately analyze the power and performance implication of their optimizations before the hardware is available. As such, the hybrid prototyping was proven capable of fast and early multicore embedded design space exploration. Embedded system architects can optimize the hardware architecture without having to do full system prototyping. Therefore, hybrid prototypes can provide huge productivity gains for both embedded software designers and multicore chip architects.

7.1 Future work

Based on the work that has been done in this thesis and the obtained results, in the following some of the potential areas of study and suggestions for future work and research directions are presented:

1. **Processors with different instruction-set architectures.** Extending the hybrid prototyping to support different target core architecture such as PowerPC or ARM. It can be easily done by providing the new processor's architecture drivers in the MEK layer.
2. **Heterogeneous MPSoCs Architectures.** Heterogeneous MPSoCs refers to systems that use a variety of different types of cores with different architectures. In general, a heterogeneous computing platform consists of processors with different instruction set architectures (ISAs). Embedded appliances designers rely on them to provide better performance [57]. Due to the inherent complexity of this kind of platform, we need a mechanism to support heterogeneous design in hybrid prototyping. Investigating the heterogeneous cores may cause adding some additional kernel functions and it may need more than one target core to execute the kernel (the MEK, software

and hardware models). For instance, assume that a designer wants to simulate a design with multiple instances of MicroBlaze, PowerPC and ARM A7 cores. In this case, the hybrid prototype may require instantiation of 3 cores on the FPGA. The kernel will need to be distributed on the cores to provide a consistent simulation context for the design.

3. **Asymmetric multiprocessing (AMP)**. An AMP system has multiple cores (may be either heterogeneous or homogeneous multicore). A separate operating system or a separate copy of the same operating system, manages each core. Typically, each application's process is locked to a single core. It provides an execution environment similar to that of uniprocessor systems. It allows simple migration of legacy code and facilitates management of each core independently. However, it can result in underutilization of processor cores.
4. **Debugging.** It needs to enable basic run control debug, where all emulated cores can be halted and ensured all emulated cores have been simulated until the debugging time.
5. **Complex inter-core communications and synchronization.** The hybrid prototyping provides FIFO channel for inter-core communications. It can support many designs with FIFO communication. However, more complex communication models are needed when more complex communication architectures, such as shared buses and Networks-on-Chip (NoCs), are used. Complex communication models require additional hardware peripherals and additional kernel functions in the framework, to support them. One of the primary challenges is to efficiently model synchronization mechanisms that are used to control access to shared resources by the multiple cores. For instance, we need a mechanism to synchronize two or more cores to grant access to shared memory or I/O device. Lock, Mutex and Semaphore are such

mechanisms to ensure that no two threads, running on separate cores, are in their critical section at the same time. The hybrid prototyping can be extended to provide such synchronization mechanisms and complex communication architectures.

6. **Reference SMP designs for accuracy comparison.** There is no equivalent Microblaze reference SMP design for accuracy measurement. Therefore, we need to find out a way to compare the accuracy of hybrid prototype of SMP design with the FPGA SMP reference design.

Bibliography

[1] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: an infrastructure for computer system modeling," *Computer,* vol. 35, pp. 59-67, 2002.

[2] F. Bellard, "QEMU, a fast and portable dynamic translator," presented at the Proceedings of the annual conference on USENIX Annual Technical Conference, Anaheim, CA, 2005.

[3] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold*, et al.*, "Mambo: a full system simulator for the PowerPC architecture," *SIGMETRICS Perform. Eval. Rev.,* vol. 31, pp. 8-12, 2004.

[4] J. Emer, P. Ahuja, E. Borch, A. Klauser, L. Chi-Keung, S. Manne*, et al.*, "Asim: a performance model framework," *Computer,* vol. 35, pp. 68-76, 2002.

[5] M. T. Yourst, "PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator," in *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, 2007, pp. 23-34.

[6] N. Yi, M. Wai Sum, and Z. Jianwen, "On virtual prototyping of embedded system-on-chips," in *ASIC (ASICON), 2011 IEEE 9th International Conference on*, 2011, pp. 1106-1109.

[7] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC," *J. VLSI Signal Process. Syst.,* vol. 41, pp. 169-182, 2005.

[8] *ModelSim*. Available: https://www.mentor.com/products/fv/modelsim/

[9] *Functional Verification Choice of Leading SoC Design Teams*. Available: http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/VCS.aspx

[10] *Incisive Enterprise Simulator* Available: http://www.cadence.com/products/fv/enterprise_simulator/pages/default.aspx

[11] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg*, et al.*, "Simics: A full system simulation platform," *Computer,* vol. 35, pp. 50-58, 2002.

[12] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, *et al.*, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Computer Architecture News,* vol. 33, pp. 92-99, 2005.

[13] C. Jianwei, M. Dubois, and P. Stenstrom, "Integrating complete-system and user-level performance/power simulators: the SimWattch approach," in *Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on*, 2003, pp. 1-10.

[14] *Wind River Simics*. Available: http://www.windriver.com/products/simics/

[15] *Coware Platform Studio*. Available: http://www.synopsys.com/Tools/SLD

[16] *Xilinx Embedded Development Kit*. Available: http://www.xilinx.com/edk

[17] A. Gerstlauer, "Host-compiled simulation of multi-core platforms," in *Proceedings of 2010 21st IEEE International Symposium on Rapid System Protyping*, 2010, pp. 1-6.

[18] G. Braun, A. Nohl, A. Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr, "A universal technique for fast and flexible instruction-set architecture simulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 23, pp. 1625-1639, 2004.

[19] M. Reshadi, P. Mishra, and N. Dutt, "Hybrid-compiled simulation: An efficient technique for instruction-set architecture simulation," *ACM Trans. Embed. Comput. Syst.,* vol. 8, pp. 1-27, 2009.

[20] M. Wai Sum and Z. Jianwen, "DynamoSim: a trace-based dynamically compiled instruction set simulator," in *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, 2004, pp. 131-136.

[21] *System on Chip library (SoClib)*. Available: http://www.systematic-paris-region.org/fr/projets/soclib

[22] *SystemC*. Available: http://www.accellera.org/downloads/standards/systemc

[23] *Open Virtual Platforms*. Available: http://www.ovpworld.org

[24] I. Nita, V. Lazarescu, and R. Constantinescu, "A new Hw/Sw co-design method for multiprocessor system on chip applications," in *Signals, Circuits and Systems, 2009. ISSCS 2009. International Symposium on*, 2009, pp. 1-4.

[25] C. L. Wang, B. Yao, Y. Yang, and Z. Zhu, "A survey of embedded operating system," *Techical Report, University of California, San Diego, USA,* 2001.

[26] Y. Hwang, S. Abdi, and D. Gajski, "Cycle-approximate Retargetable Performance Estimation at the Transaction Level," in *2008 Design, Automation and Test in Europe*, 2008, pp. 3-8.

[27] Z. Wang and A. Herkersdorf, "An efficient approach for system-level timing simulation of compiler-optimized embedded software," in *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, 2009, pp. 220-225.

[28] Z. Wang, K. Lu, and A. Herkersdorf, "An approach to improve accuracy of source-level TLMs of embedded software," in *2011 Design, Automation & Test in Europe*, 2011, pp. 1-6.

[29] D. Chiou, S. Dam, K. Joonsoo, P. Nikhil, W. H. Reinhart, D. E. Johnson*, et al.*, "The FAST methodology for high-speed SoC/computer simulation," in *2007 IEEE/ACM International Conference on Computer-Aided Design*, 2007, pp. 295-302.

[30] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi, "ProtoFlex: Towards Scalable, Full-System Multiprocessor Simulations Using FPGAs," *ACM Trans. Reconfigurable Technol. Syst.,* vol. 2, pp. 1-32, 2009.

[31] D. Chiou, D. Sunwoo, H. Angepat, J. Kim, N. A. Patil, W. Reinhart*, et al.*, "Parallelizing computer system simulators," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 2008, pp. 1-5.

[32] T. S. H.-H. S. Lee and S.-L. L. J. Shen, "Initial Observations of Hardware/Software Co-Simulation using FPGA in Architecture Research," 2006.

[33] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson*, et al.*, "RAMP gold: An FPGA-based architecture simulator for multiprocessors," in *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, 2010, pp. 463-468.

[34] *ChipScope* Available: http://www-mtl.mit.edu/Courses/6.111/labkit/chipscope.shtml

[35] C. Yajun, C. Qingshan, Z. Lianqing, G. Yangkuan, and P. Zhikang, "Signal Tap-II Based Debugging Approach for the Data Acquisition System of Multi-joint Coordinate Measuring Machine," in *Instrumentation, Measurement, Computer, Communication and Control (IMCCC), 2012 Second International Conference on*, 2012, pp. 1182-1184.

[36] J. Wawrzynek, D. Patterson, M. Oskin, S. L. Lu, C. Kozyrakis, J. C. Hoe, *et al.*, "RAMP: Research Accelerator for Multiple Processors," *IEEE Micro,* vol. 27, pp. 46-57, 2007.

[37] C. Chang, J. Wawrzynek, and R. W. Brodersen, "BEE2: a high-end reconfigurable computing system," *IEEE Design & Test of Computers,* vol. 22, pp. 114-125, 2005.

[38] S. S. Sirowy, B. Miller, and F. Vahid, "Portable SystemC-on-a-chip," presented at the Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis, Grenoble, France, 2009.

[39] L. Benini, D. Bruni, N. Drago, F. Fummi, and M. Poncino, "Virtual in-circuit emulation for timing accurate system prototyping," in *ASIC/SOC Conference, 2002. 15th Annual IEEE International*, 2002, pp. 49-53.

[40] *Renesas' Multi-Core Technology*. Available: http://www.renesas.com/products/mpumcu/multi_core/child/multicore.jsp

[41] E. Saboori and S. Abdi, "Hybrid Prototyping of multicore embedded systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, 2013, pp. 1627-1630.

[42] E. Saboori and S. Abdi, "Rapid design space exploration of multi-clock domain MPSoCs with Hybrid Prototyping," in *Electrical & Computer Engineering (CCECE)*, Vancouver, Canada, 2016.

[43] *Dinero IV Trace-Driven Uniprocessor Cache Simulator*. Available: http://pages.cs.wisc.edu/~markhill/DineroIV/

[44] B. Atanasovski, S. Ristov, M. Gusev, and N. Anchev, "MMCacheSim: A Highly Configurable Matrix Multiplication Cache Simulator," *ICT Innovations 2012, Web Proceedings ISSN 1857-7288,* p. 185, 2012.

[45] S. Stattelmann, G. Gebhard, C. Cullmann, O. Bringmann, and W. Rosenstiel, "Hybrid source-level simulation of data caches using abstract cache models," presented at the Proceedings of the Conference on Design, Automation and Test in Europe, Dresden, Germany, 2012.

[46] R. Dömer, "Transaction level modeling of computation," *Center for Embedded Computer Systems, Technical Report,* 2006.

[47] D. Araki, N. Ito, T. Shinsha, and Y. Mori, "High speed hardware/software coverification with cpu model generator from software code," in *5th NASCUG (North American SystemC User's Group) meeting Co-located with DAC*, 2006.

[48] G. Schirner, A. Gerstlauer, and R. Doemer, "Abstract, multifaceted modeling of embedded processors for system level design," in *Design Automation Conference, 2007. ASP-DAC'07. Asia and South Pacific*, 2007, pp. 384-389.

[49] P. Ravishankar, "An Observable Data Cache Model for FPGA Prototyping," Concordia University, 2013.

[50] H.-M. Yoon, G.-H. Park, K.-W. Lee, T.-D. Han, S.-D. Kim, and S.-B. Yang, "Reconfigurable address collector and flying cache simulator," in *High Performance Computing on the Information Superhighway, 1997. HPC Asia'97*, 1997, pp. 552-556.

[51] L. A. Barroso, M. Dubois, and K. Ramamurthy, "RPM: A rapid prototyping engine for multiprocessor systems," *Computer,* vol. 28, pp. 26-34, 1995.

[52] J. Hong, E. Nurvitadhi, and S.-L. L. Lu, "Design, implementation, and verification of active cache emulator (ACE)," in *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, 2006, pp. 63-72.

[53] A. Barzegar, E. Saboori, and S. Abdi, "DRAC: a dynamically reconfigurable active L1 cache model for hybrid prototyping of multicore embedded systems," in *2014 25nd IEEE International Symposium on Rapid System Prototyping*, 2014, pp. 86-92.

[54] A. Gerstlauer, Y. Haobo, and D. D. Gajski, "RTOS modeling for system level design," in *Design, Automation and Test in Europe Conference and Exhibition, 2003*, 2003, pp. 130-135.

[55] E. Saboori and S. Abdi, "Fast and cycle-accurate simulation of multi-threaded applications on SMP architectures using hybrid prototyping " in *International Conference on Hardware/Software Codesign and System Synthesis (CODES/ISSS)*, Pittsburgh, USA, 2016.

[56] E. Kang, E. Jackson, and W. Schulte, "An approach for effective design space exploration," presented at the Proceedings of the 16th Monterey conference on Foundations of computer software: modeling, development, and verification of adaptive systems, Redmond, WA, 2011.

[57] X. Guerin and F. Petrot, "A System Framework for the Design of Embedded Software Targeting Heterogeneous Multi-core SoCs," in *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2009, pp. 153-160.