T2R2 東京科学大学 リサーチリポジトリ
Science Tokyo Research Repository

論文 / 著書情報
Article / Book Information

題目(和文)	
Title(English)	Multi-FPGA based Prototyping Framework for Emerging Manycores
著者(和文)	高前田（山崎）伸也
Author(English)	Shinya Takamaeda-Yamazaki
出典(和文)	学位:博士(工学), 学位授与機関:東京工業大学, 報告番号:甲第9554号, 授与年月日:2014年3月26日, 学位の種別:課程博士, 審査員:吉瀬　謙二,横田　治夫,宮崎　純,金子　晴彦,渡部　卓雄
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:甲第9554号, Conferred date:2014/3/26, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Powered by T2R2 (Science Tokyo Research Repository)

Tokyo Institute of Technology
Department of Computer Science

Multi-FPGA based Prototyping Framework for Emerging Manycores

(次世代メニーコアのための マルチ FPGA ベースプロトタイピングフレームワーク)

A dissertation submitted in partial fulfillment of
the requirements for the degree of Doctor of Engineering

Shinya Takamaeda-Yamazaki

February 2014

Graduate School of Information Science and Engineering
Department of Computer Science
Tokyo Institute of Technology

Abstract

In modern microprocessor architecture research, hardware-based prototyping using FPGAs has been used in order to accelerate simulations of processor behaviors. The faster simulation speed strongly enables system-level evaluations of not only the processor architecture, but also applications and system softwares. This thesis presents a sophisticated prototyping framework for future manycores. The framework comprises of the acceleration method of many-core processor simulations and the design methodology to decrease the development complexity.

The first contribution of the framework is to propose an FPGA-based simulation method which achieves the scalable simulation speed against the increasing core count of simulated processors with the cycle-level accuracy of simulation results. In order to accelerate simulations of many-core processors, I propose a system architecture of fast and cycle-accurate processor simulator employing multiple FPGAs. I developed a test bed platform of multiple FPGAs to evaluate the viability of the proposed method. I evaluated the proposed method by using the test bed system in point of simulation speed. The evaluation result shows that the proposed method achieves effective scalability of the simulation speed to simulate a large scale many-core processor with keeping the cycle-accuracy of the simulation consequences. As the case studies, I applied the test bed system for two innovative researches of task allocation schemes on many-core processors. By employing the test bed system, the evaluation phases are dramatically accelerated, so that it enables effective evaluations of computer systems.

The second contribution of this framework is to propose a design methodology under the resource abstraction of FPGA platforms. In order to mitigate the development complexity of FPGA-based simulators, I propose a novel design methodology under the abstraction of various resources FPGA platforms have, such as memory systems and inter-FPGA interconnections. I developed the Python-based design tool-chain that automatically synthesizes ready-to-implement RTL designs for actual FPGA platforms from target RTL descriptions under the abstraction. This methodology enables designers to model a prototyping target processor without concern for actual platform resources. I evaluated simulation speed under the abstraction using a standard FPGA platform with large capacity of logic and memory. The evaluation result shows that the simulation speed degradation under the abstraction is not critical so that the abstraction tool-chain offers the helpful support to develop a high-speed processor simulator rapidly.

Finally I evaluated the integrated framework of the two contributions, the scalable simulation accelerator and the abstraction methodology. The evaluation result shows that the simulation system automatically synthesized by the abstraction tool-chain archives almost equivalent performance to manual-tuned multi-FPGA based simulator. The integrated framework aggressively improves the prototyping efficiency for emerging many-core processors by providing the sufficient simulation speed and the effective abstraction reducing the development complexity.

Contents

1 Introduction 1

1.1 Motivation . . . 1
1.2 Contribution . . . 3
1.3 Outline of This Thesis . . . 4

2 Background 6

2.1 Simulation for Processor Architecture Evaluation . . . 6
2.1.1 Purpose . . . 6
2.1.2 Processor Evaluation Flow . . . 6
2.2 Software Simulator . . . 8
2.2.1 Functional-Accurate Simulator . . . 9
2.2.2 Cycle-Accurate Simulator . . . 9
2.2.3 Simulator not for General Purpose Processors . . . 10
2.2.4 Simulator not for Performance Modeling . . . 11
2.2.5 Register Transfer Level (RTL) and Circuit Simulation . . . 11
2.2.6 Discussion . . . 11
2.3 FPGA-based Processor Prototyping . . . 12
2.3.1 Direct Implementation . . . 12
2.3.2 Simulation-Oriented Implementation . . . 15
2.4 Goal of This Research . . . 17
2.5 Baseline Many-core Architecture as a Prototyping Target . . . 19
2.6 Summary . . . 20

3 ScalableCore System: A Multi-FPGA based Processor Simulation Platform 22

3.1 Motivation . . . 23
3.2 Overall Concept and Architecture . . . 23
3.3 Design . . . 25
3.3.1 Architecture of ScalableCore Unit . . . 25
3.3.2 Architecture of Memory Unit . . . 26
3.3.3 System Level Function for Cycyle-Accurate Simulation . . . 27
3.3.4 RTL Design Rule . . . 29
3.4 Hardware Platform Implementation . . . 30

3.5 Evaluation 31
3.5.1 Simulated Processor Setup 31
3.5.2 Simulation Speed 32
3.5.3 Resource Utilization 36
3.6 Discussion 36
3.6.1 Comparison with Prior Software Simulators 36
3.6.2 Comparison with Prior FPGA-based Prototyping Systems 37
3.7 Case Study 1: Evaluation of Task Allocation Method for Multiple Parallel Applications 39
3.7.1 RMAP: Contention-Aware Task Allocation for Manycores 40
3.7.2 Evaluation 41
3.7.3 Summary of Case Study 1 41
3.8 Case Study 2: Evaluation of Task Allocation Method for NoC-based DMR Execution Manycores 42
3.8.1 SmartCore System: An NoC-based DMR Execution Mechanism for Manycores 42
3.8.2 Task Allocation for DMR Executions 46
3.8.3 Evaluation 48
3.8.4 Summary of Case Study 2 54
3.9 Summary 54

4 flipSyrup: An FPGA-based Flexible Prototyping Methodology 56
4.1 Motivation 57
4.2 Methodology Overview 58
4.3 Design Flow 58
4.3.1 Flow Overview 58
4.3.2 Step 1: Design Partitioning 60
4.3.3 Step 2: Object Replacement 61
4.3.4 Step 3: flipSyrup Synthesis 61
4.3.5 Step 4: Bitstream Synthesis on Standard EDA 61
4.4 RTL Modeling under flipSyrup Abstractions 61
4.4.1 Syrup Memory 62
4.4.2 Syrup Channel 64
4.5 flipSyrup in Detail 65
4.5.1 Abstract Object Conversion 65
4.5.2 Architecture of Cycle-Accuracy Manager 66
4.5.3 Throttling by Cycle-Accuracy Manager 67
4.5.4 RTL Design Conversion 69
4.6 Pyverilog: A Python-based Hardware Design Processing Toolkit for Verilog HDL . . 71
4.6.1 Overview 71
4.6.2 Design and Implementation 72

4.6.3 Summary of Pyverilog 74
4.7 System Generation by flipSyrup 76
4.7.1 Memory Specification Reader 76
4.7.2 Control Circuit Synthesis 76
4.8 Multi-core Processor Simulation on a Single FPGA Platform 77
4.8.1 Evaluation Methodology 77
4.8.2 Simulation Speed 78
4.8.3 Resource Utilization 81
4.8.4 Clock Frequency 81
4.8.5 Summary of Evaluation on a Signle FPGA Platform 82
4.9 Many-core Processor Simulation on a Multi-FPGA Platform 82
4.9.1 Evaluation Methodology 83
4.9.2 Simulation Speed 83
4.9.3 Resource Utilization 84
4.9.4 Summary of Evaluation on a Multi-FPGA Platform 85
4.10 Comparison with Other Methodology 85
4.11 Summary 86

5 Conclusion **88**
5.1 Concluding Remarks 88
5.2 Open Research Topics 89

Acknowledgement **90**

Bibliography **94**

Publication **104**
5.3 Journal Paper 104
5.4 International Conference Paper 104
5.5 Domestic Conference Paper (with Review) 105
5.6 Technical Report 105
5.7 Other Presentation and Poster 107

List of Figures

2.1 Processor Evaluation Flow 7
2.2 Object Duplication vs. Multithread Simulation 17
2.3 M-Core Architecture 19

3.1 Concept of ScalableCore System 24
3.2 Function Stack of ScalableCore System 25
3.3 ScalableCore Unit 26
3.4 ScalableCore Memory Unit 27
3.5 Virtual Cycle 28
3.6 Local Barrier Synchronization 29
3.7 RTL Coding Rule for Virtual Cycle 30
3.8 Snapshot of ScalableCore System with 100 FPGA Units 31
3.9 Snapshot of ScalableCore Unit (Left) and Memory Unit (Right) 32
3.10 Simulation Speed 33
3.11 Relative Speed 34
3.12 Floorplan of ScalableCore Unit on Spartan-6 XC6SLX16 35
3.13 Task Allocation of 4 Parallel Applications 40
3.14 Task Allocation of 2 Parallel Ppplications (for This Case Study) 40
3.15 Performance comparison of RMAP allocation and the normal allocation 41
3.16 SmartCore System: NoC-based DMR Execution Mechanism for Many-core Processors 43
3.17 Standard On-chip Router Architecture 44
3.18 DMR On-chip Router Architecture 45
3.19 Task Allocation (Separate, Master-Slave Distance = 1) 46
3.20 Task Allocation (Interleave, Master-Slave Distance = 1) 47
3.21 Task Allocation (RMAP, Master-Slave Distance = 1) 48
3.22 Task Allocation (Block, Master-Slave Distance = 4) 49
3.23 Task Allocation (Interleave ROT, Master-Slave Distance = 5) 50
3.24 Task Allocation (Interleave ROTx2, Master-Slave Distance = 8) 51
3.25 Task Allocation (RMAP ROT, Master-Slave Distance = 8) 52
3.26 Performance (without DMR mode) 52

3.27 Relationship between Master-Slave Distance and Task Allocation Pattern (Bitonic Sort) 53
3.28 Relationship between Master-Slave Distance and Task Allocation Pattern (Matrix Multiply) 54

4.1 Development Flow of Prototyping System with flipSyrup 59
4.2 Base Processor Design 59
4.3 Partitioned Processor Design for flipSyrup 60
4.4 Generated Simulation System with flipSyrup 62
4.5 Abastract Objects of flipSyrup 63
4.6 Cycle-Accuracy Manager 66
4.7 Timing Chart Example of Cycle-Accuracy Manager 68
4.8 Inserting Control Ports of Abstract Objects 69
4.9 RTL Conversion Rules for Inserting Throttling Ports 70
4.10 Pyverilog Toolkit Overview 72
4.11 Example Verilog HDL Code (Vector-Add) 75
4.12 Graphical Output of Dataflow Graph 76
4.13 Graphical Output of Finite State Machine 77
4.14 Simulation Speed of Single FPGA Simulator with flipSyrup 79
4.15 Resource Utilization Breakdown of Single FPGA Simulator with flipSyrup 80
4.16 Maximum Frequency of Each Module 81
4.17 Simulation Speed of ScalableCore System with flipSyrup 84
4.18 Resource Utilization Breakdown of ScalableCore System with flipSyrup 85

List of Tables

3.1 Microarchitecture of Simulation Target (M-Core) 32
3.2 Resource Utilization 35
3.3 Simulation Speed Comparison 37
3.4 Evaluation Setup 49

4.1 Evaluation Setup 78
4.2 Evaluation Setup 83

Chapter 1

Introduction

1.1 Motivation

Computer systems are essential for our daily lives. In Japan, most people have some mobile systems, such as smart phones and mobile gaming machines. Automotive consists of a lot of microprocessors in order to control the drive system and the peripherals, such as air conditioner and navigation system. Supercomputers with high-capability of computing power are also used for weather forecasting so that it improves the prediction accuracy of a forecast. In areas of engineering and drug development, super computers are used for scientific simulations.

One of the important parts of a computing system is a processor. A processor adopts various operations to the given data being dependent on the given instructions. Processor is also called as CPU (Central Processing Unit), MPU (Main Processing Unit) or microprocessor. The first microprocessor had consisted of multiple electric boards. After Intel 4004 released in 1974, microprocessors have been furnished as integrated circuits. Now most of the microprocessors are given in unified LSI packages of the memory system and peripheral circuits, by advancing the semiconductor technology.

In the past, the performance of microprocessors had been enhanced by (1) the increase of operation frequency thanks to the semiconductor processes refinement and (2) the employment of inherent instruction level parallelisms (ILP) by complex hardware components. However, the limitations of capable energy consumption and amount of heat generation have made it difficult to increase the operation frequency more. Additionally, it has been also challenging to obtain corresponding performance improvements by additional hardware resources to employ the more inherent ILPs. Therefore, the microprocessor development strategy has moved to utilize another parallelisms, such as thread level parallelism and data level parallelism, for more computation power and efficiency.

The trend of processor architecture has shifted to multicore architecture that integrates multiple cores into a single chip. Processors employing multicore architecture are called also as CMP (chip multi processor), multicore processor or merely multicore. Of the growing semiconductor process, the number of cores integrated on a single chip is still increasing. Most microprocessors in these days have two

or more cores on a single chip. They are employed in wide range areas, including high-performance computing systems and low-energy embedded systems.

In modern microprocessor architecture research and development, software-based simulators of microprocessors are essential to evaluate and verify an architectural idea. By using a software-based simulator with detailed models of processor cores and memory systems, we can realize and evaluate the proposed idea in an easy way without actual LSI fabrication.

The most critical issue of software-based simulations is simulation speed. Unfortunately, regardless of simulation accuracy, however, simulation speeds of software-based simulators are considerably slower than actual processor LSIs. The simulation speeds of software-based processor simulators are about 100 KHz in the case of a detailed model. For instance, MARSSx86, a modern cycle-accurate CMP simulator, can achieve the speed of about 200 KIPS (kilo instructions per second) when an 8-core CMP is simulated[1].

These software-based simulators have massive inherent fine-grain parallelisms corresponding to each circuit component. Parallelization of simulators can improve the simulation performance in some moderate degree on contemporary multicore processors. However, it is extremely difficult to fully exploit these parallelisms in order to accelerate the simulations, because there are huge amount of synchronizations and communications with tiny messages for every modeled clock cycle in the simulated processor to keep the cycle level accuracy of simulation results. This property is not appropriate to run on standard based computing environments very well. Otherwise, relaxing these synchronizations eliminates accuracy from the simulation result.

In order to accelerate such processor simulations, the hardware-based prototyping with FPGAs (Field Programmable Gate Arrays) have been commonly used. The fine-grain parallelisms of processor components can be naturally utilized by employing internal fabrics of FPGAs. Moreover fine-grain synchronization for cycle-accuracy is much faster on FPGAs than the software simulators. The faster simulation speed strongly supports system-level evaluations of not only the processor architecture but also the application and system software.

A problem of FPGA-based prototyping is the lack of any scalable simulation methods. In standard prototyping way, a large high-end and expensive FPGA is required for simulation of future many-core processor with over 100 cores. Additionally, the synthesis time that is the elapsed time to create an FPGA circuit image (bitstream) from HDL source codes is very long in large FPGAs. Previous FPGA-based simulators projects proposed some clever techniques to reduce resource consumption of FPGAs. They, however, decrease also the simulation speed if a processor with a large number of cores is simulated.

Another problem is the absence of any suitable abstraction methods to handle bare-metal sea-of-resources of FPGAs. Since there is a gap of resource characteristics between simulated processors and

FPGAs, the structure of microprocessors is not necessarily suitable for FPGA characteristics. In respect of the memory system, processors have on-chip cache memory systems. Usually these cache memory systems are implemented by using on-chip fast memory fabrics (block RAM) of FPGAs. Unfortunately, the amount of these on-chip memory fabrics of FPGA is limited and small. If the simulated processor requires larger capacity of on-chip FPGA memory, the simulator developer should consider combining an external memory component (DRAM) for cache system implementation. Development of such a hierarchical memory system with keeping the cycle level accuracy of simulation result is highly error-prone and time-consuming.

This thesis presents a sophisticated prototyping framework for future manycores. The framework comprises of the acceleration method of many-core processor simulations and the design methodology to decrease the development complexity. In order to accelerate simulations of many-core processors, I propose a development method of fast and cycle-accurate many-core processor simulator by employing multiple FPGAs. In order to mitigate the development complexity of FPGA-based simulators, I propose a novel design methodology with the resource abstraction of FPGAs. The integrated framework aggressively improves the prototyping efficiency for emerging many-core processors.

1.2 Contribution

The contributions of this thesis are as follows:

- to propose an FPGA-based simulation method which achieves the scalable simulation speed against the increasing core count of simulated processors with the cycle-level accuracy of simulation results;
- to show that the proposed simulation method is feasible by designing and developing an actual multi-FPGA based acceleration system;
- to propose a novel design methodology under the physical resource abstraction of FPGAs, with its software tool-chain to improve the efficiency of prototyping system development; and
- to present and evaluate the integrated framework that wraps up the multi-FPGA based simulation method and the prototype design methodology.

I refer to the respective contributions below.

As the first and second contributions, I propose a system architecture of fast and cycle-accurate processor simulator employing multiple FPGAs, focused on the structure of the on-chip-network and the memory systems on a many-core processor for future manycores with over 100 cores. I developed a test bed platform of multiple FPGAs to evaluate the viability of the proposed method. I evaluated the proposed method by using the test bed system in point of simulation speed. The evaluation result shows that the proposed method achieves effective scalability of the simulation speed to simulate a

large scale many-core processor with keeping the cycle-accuracy of the simulation consequences. As the case studies, I applied the test bed system for two innovative researches of task allocation schemes on many-core processors. By employing the test bed system, the evaluation phases are dramatically accelerated, so that it enables effective evaluation of computer systems.

As the third and forth contributions, I present a novel design methodology under the abstraction of various resources FPGA platforms have, such as memory systems and inter-FPGA interconnections. I developed the Python-based design tool-chain that automatically synthesizes ready-to-implement RTL designs for actual FPGA platforms from target RTL descriptions under the abstraction. This methodology enables designers to model a prototyping target processor without concern for actual platform resources. I evaluated simulation speed under the abstraction using a standard FPGA platform with large capacity of logic and memory. The evaluation result shows that the simulation speed degradation under the abstraction is not critical so that the abstraction tool-chain offers the helpful support to develop a high-speed processor simulator rapidly.

Finally I evaluated the integrated framework of the two contributions, scalable simulation accelerator and the abstraction methodology. The evaluation result shows that the simulation system automatically synthesized by the abstraction tool-chain archives almost equivalent performance to manual-tuned multi-FPGA based simulator. The integrated framework aggressively improves the prototyping efficiency for emerging many-core processors by providing the sufficient simulation speed and the effective abstraction reducing the development complexity.

1.3 Outline of This Thesis

The outline of the subsequent chapters is as follows.

In Chapter 2, I describe the background of my work. At first, I present various prior software-based processor simulators. Software-based simulators are fundamental weapons for comprehensive evaluation in processor architecture research. As the trend of microprocessor shifted to multicore architecture, the purpose and structure of software simulators have been drastically changed. Then I present prior researches of simulation acceleration using FPGAs. I roughly classify the FPGA-based processor simulation into two categories based on the implementation ways. I summarize the motivation of this work based on the discussion about the previous researches. Additionally, I briefly introduce the example many-core architecture that I used as a simulation target in this research.

In Chapter 3, I describe the first main contribution of this work that accelerates the processor simulation by employing FPGAs. I present a system architecture of fast and cycle-accurate processor simulator employing multiple FPGAs. I describe the implementation features of the developed test bed system built on the proposed method. As the case studies, I applied the test bed system for two innovative researches of task allocation schemes on many-core processors. By employing the test bed system,

the evaluation phases are dramatically accelerated, so that it enables effective evaluation of computer systems.

In Chapter 4, I propose a novel design methodology under the abstraction of various resources on FPGAs. Hardware components within FPGAs, such as memory block and communication interfaces, are abstracted and given to simulator designers. I describe the implementation features of the Python-based software tool-chain that realizes the proposed methodology. I evaluated simulation speed with the resource abstraction on the single large FPGA platform. Then I evaluated the integrated framework of the two main contributions of this thesis, scalable simulation accelerator and the abstraction methodology.

Finally, in Chapter 5, I wrap up this thesis with the discussion of open-research areas and the conclusion.

Chapter 2

Background

This chapter provides the fundamental knowledges of various evaluation ways on processor architecture researches. First I briefly introduce the entire evaluation flow of microprocessors using several identical weapons. Then I describe and examine the pros and cons of each evaluation way, in points of simulation speed and development complexity. Especially I present noteworthy problems on the aforementioned FPGA-based processor prototyping researches. I summarize the motivation of this work taken into consideration these discussions. Finally I briefly introduce the baseline many-core processor architecture for my research.

2.1 Simulation for Processor Architecture Evaluation

2.1.1 Purpose

Performance modeling is a critical matter in computer system development and research. As increasing in the number of cores integrated in a single processor, complexity of microprocessor is also increasing. At the same time, the required simulation amount of a microprocessor is also increasing to validate cross cutting issues among microarchitecture, system software and application. To develop a further useful computer system, evaluation method will become more important.

The main purpose of simulation is to estimate the impact of an idea to the computer performance. Performance is just an indicative value that represents speed of the processor, in this thesis. We can obtain not only values of absolute performance but also various associated values of the performance through processor simulations, such as hit/miss rate of cache memory and branch prediction. Architects use the performance value and associated values to optimize the architecture and its software systems.

2.1.2 Processor Evaluation Flow

Figure 2.1 shows a standard evaluation flow of processor architecture research. Each simulation technology has a unique characteristic. Architects should choose an appropriate method for each evaluation purpose and stage. In early stage of research and development, the customary evaluation way is to use

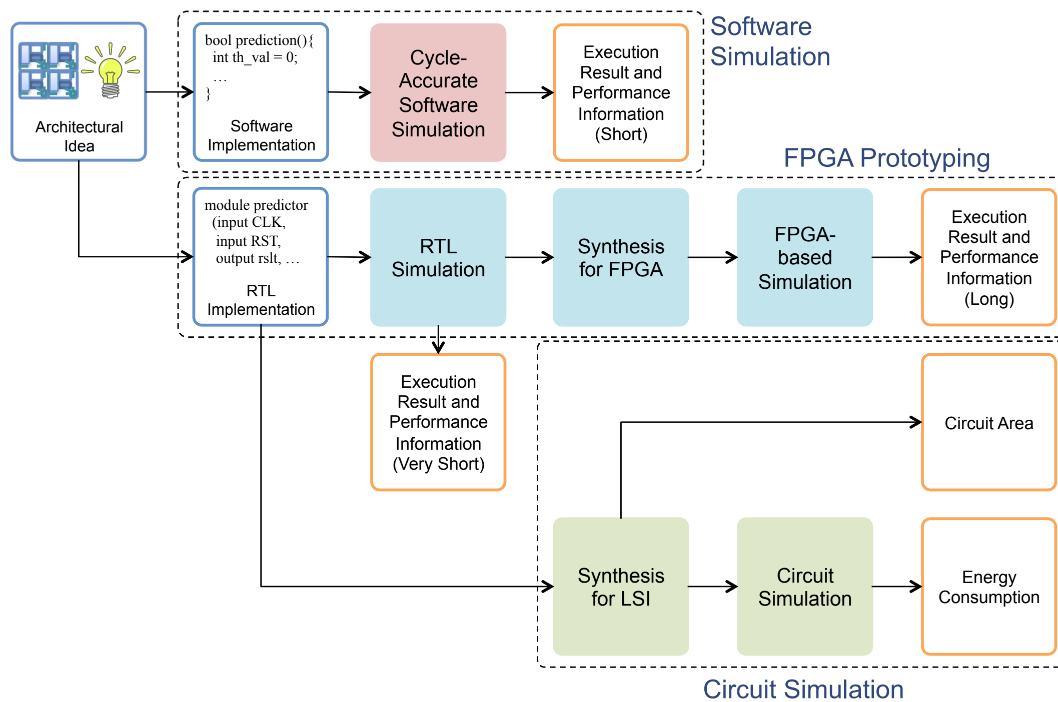


Figure 2.1 Processor Evaluation Flow

software simulators. Simulation is a means to validate and evaluate a microprocessor without any actual LSI fabrication. To evaluate an idea, the architect implements the idea on a software simulator with the target architecture model. Then the architect can evaluate the implemented idea by using appropriate benchmark applications, such as SPEC[2] and PARSEC[3].

Software Simulation

As an important point of software simulators, we can identify the structure of everything in microprocessors in separated pieces of functional model and timing model. The timing model corresponds to a definition of the conditions that the hardware component is enabled. The functional model is a definition of the behavior or result that the hardware component generates when the component is enabled. In both parts, designers can develop a simulator with standard software development techniques to clarify the software structure. This separation simplifies the implementation of the target processor and improves the portability for another design of processors.

As a critical problem of software simulators, simulation speed of software simulations is slow. While the absolute simulation speed depends on the accuracy, typical simulation speed of software-based processor simulators is about some 100 KHz. For instance, MARSSx86, a modern cycle-accurate CMP simulator, can achieve the speed of about 200 KIPS (kilo instructions per second) when an 8-core CMP is simulated, which corresponds to about 25 KHz[1].

FPGA-based Prototyping

Acceleration of processor simulation enables much longer simulation using more realistic and complicated applications. FPGA-based simulation is often used to accelerate the simulation. As the standard development flow of FPGA system, it requires a detailed RTL (Register Transfer Level) design of target hardware in low level HDL (Hardware Description Language), such as Verilog HDL and VHDL. The circuit image (bitstream) for FPGA is available from the given RTL design by using a vendor-provided EDA tool[4, 5]. Circuit synthesis for FPGAs usually takes critically longer time than software compilation, about hours to days for large high-end FPGAs. In order to prevent rollbacks of circuit synthesis, RTL simulation is often performed before the FPGA implementation for the RTL design verification. As same as software simulation, the architect then can simulate the behavior of the target processor using the generated bitstream for FPGA system with adequate benchmarks.

Note that the RTL design for FPGA prototyping might differ from the RTL design for LSI manufactures, due to the characteristic differences between LSI and FPGA. The structure of microprocessors is not necessarily suitable for FPGA characteristics. In this thesis, I offer a counterplot against the gap of resource characteristics introducing complications of prototyping system development.

Circuit Simulation

In order to estimate the impact of an idea to the circuit area, architects create a physical LSI design by using an LSI compiler from the RTL designs. While actual circuit area strongly depends on efforts for the optimization, the architects can only estimate the approximate impact to the area. In order to estimate the impact of the idea to the energy, circuit simulation is an effective way without LSI manufacturing. In general, circuit simulation is employed to estimation of energy consumption for each distinct event on the simulated hardware, but not for entire system simulation, because circuit simulation is absolutely slow. The total energy impact is calculated by using the cycle-accurate software simulators or FPGA-based simulators with the discrete estimated values.

2.2 Software Simulator

Software simulators are roughly classified into two categories: functional-accurate simulators and cycle-accurate simulators. Functional-accurate simulators generate execution results on the simulated processor, but does not produce performance information such as elapsed time and cache miss rates. Functional-accurate simulators are called also as instruction set simulator (ISS). In contrast, cycle-accurate simulators have an advanced capability to model performance of the simulated processor. In this thesis, I define the cycle-accurate simulation as a simulation that conforms to the cycle-by-cycle behavior of the target design definition, based on the definition used in literature[6].

2.2.1 Functional-Accurate Simulator

Various researchers have developed functional-accurate simulators for research and educational purpose. SimCore/Alpha[7] is a sophisticated and well-tuned Alpha ISA simulator used for evaluation of branch predictors. SPIM[8] is a famous instruction set simulator of MIPS used for education. WebMIPS[9] is a similar simulator for MIPS ISA but is provided as a web application with a visualizer of pipeline status. SimMips[10, 11] is a functional simulator supporting operating systems for MIPS ISA, which is used as a fundamental tool for my research.

QEMU[12] is commonly used for virtualization of physical computer and software development for embedded systems. As a key feature of QEMU, it employs binary translation technique to transform the simulated software into suitable binary of the host computer. It enables fast emulation of various instruction set architectures on various host computers. As an interesting approach to accelerate instruction set simulation, ISSGPU[13] employs a GPU as a massive parallel computing unit for future many-core consisting over 1000 cores on a single chip.

In general, functional-accurate simulators do not provide timing information in detail. These functional-accurate simulators are still useful to obtain traces of instructions and memory accesses, so that the generated trace results can be used in other software simulators with detailed models of the target hardware structure.

2.2.2 Cycle-Accurate Simulator

Until the paradigm shift to multicores, SimpleScalar[14] had been a widely used cycle-accurate processor simulator for microarchitectural simulations of uniprocessor. SimpleScalar models microarchitectural features of Out-of-Order processors very well. SimpleScalar had no essential functions to simulate the cross-cutting behavior with system software components.

As the growing of multicore processors and accelerators, various research groups developed full-system simulators, in order to simulate performance on the target processor with interactions between applications, operating systems and hardware. M5[15], Simics[16], GEMS[17], gem5[18], MARSSx86[1], SESC[19], Manifold[20] and CMP$IM[21] are major full-system simulators for modern multicore processor environments with operating system support, used in various researches. Each simulator is different in points of supported ISA, memory hierarchy, on-chip network, off-chip peripherals and so on. Most of these simulators are under a similar structure that a functional simulation unit and a timing simulation unit are tightly coupled. Transformer[22] employs a unique approach that decouples the function simulation unit and timing simulation unit, in order to improve the simulation speed and portability of simulator.

As I noted above, the decisive drawback of these software simulators is their simulation speed.

The typical simulation speed of these full-system simulators is about some 100 KHz, while the absolute value is dependent on the simulated processor size and the simulation accuracy. For example, MARSSx86 can achieve speed of about 200 KIPS (kilo instructions per second) when an 8-core CMP is simulated, which equals about 25 KHz[1].

The most major approach accelerating software simulators is parallelization. HORNET[23] is a software simulator of a tile many-core architecture with various network-on-chip options, which takes a parallelization approach with the accuracy relaxing option to mitigate synchronization overheads among multiple threads. In their evaluations, parallelizing a simulation of a 64-core target processor on the Intel Xeon (12-core, 2-way SMT) host machine with keeping the cycle-accuracy achieves 5 times faster in speed opposed to the sequential simulation. This simulation speed is limited by the speed of inter-core synchronization. They used a standard SMP environment with two 6-core processor dies to totally offer 12 cores. The best simulation speed is achieved when only 5 cores on the single identical processor die are activated. Striding across the two dies drastically decreases the simulation speed. Therefore employing actual modern many-core environments, such as Intel Xeon Phi, might more accelerate such software simulations, because the inter-core synchronization overhead on such environments will much smaller than that of inter-die on SMP environments.

ZSim[24] is the state-of-art software simualtor for future many-core processors of thousands cores, which uses the accuracy relaxation approach for highly prallelization simulations. They stated that ZSim models a 1024-core chip on a 16-core host at speeds of up to 300 MIPS using detailed Out-of-Order cores, 2-3 orders of magnitude faster than existing parallel simulators. Unfortunately, their simulation method might generate incorrect simulation results due to the accuracy losses. Therefore, there is a trade-off between the simulation speed and the accuracy.

2.2.3 Simulator not for General Purpose Processors

Numerous software simulators for non-general purpose processors have been developed. SimCell[25] is a functional simulator of subset of Cell/B.E.. GPGPU-sim[26] is a commonly used cycle-accurate simulator of NVIDIA Fermi-like GPUs for GPU architecture research. SimMc[27, 28] is a cycle-accurate simulator of a many-core accelerator with modern network-on-chip supports. SimMc is also used as a fundamental environment for my research.

Software simulators used not only for entire the processor simulation, but also for particular components of microprocessors. Booksim[29] is a major software simulator for network-on-chip (NoC) simulation. Orion[30] is also major NoC simulator with capability to evaluate power-performance. In NoC evaluation, standardized traffic patterns and packet traces of particular benchmarks generated by a full-system simulator are often used as an input to the simulator. NoC simulators are often used in NoC-centric research, because simulation speed is dramatically improved compared to full-system

simulation by using traces or standard traffic patterns. Additionally, in order to obtain more accurate simulation results with due consideration of exact dependencies across the on-chip traffics, Netrace[31] can handle the dependency-driven packet generation from the given application traces.

2.2.4 Simulator not for Performance Modeling

The architects' concern is not just the performance. They must examine the impact of a tested idea to the enegy-efficiency and the circuit area. Not only for the performance evaluation, software simulators are used also for power modeling and circuit area estimation. Wattch[32] and McPAT[33] are software simulators to estimate energy consumption of a simulated processor using event counter values obtained from the corresponding cycle-level software simulators. CACTI is an estimation tool of energy and circuit area for SRAM. GPGPUWattch[34] is an energy modeling software of GPU architecture used with GPGPU-sim. Using these software simulators for evaluations of energy and area, architects can avoid cost-inefficint and time-consuming steps for actual processor implementation as an LSI.

2.2.5 Register Transfer Level (RTL) and Circuit Simulation

Before the LSI fabrication and the development FPGA-based prototyping, the RTL (Register Transfer Level) design of the target processor should be validated. VCS[35], NC-Verilog[36] and ModelSim[37] are widely used commercial simulator for RTL simulation of Verilog HDL. Icarus Verilog[38], GPLCver[39] and Verilator[40] are famous open-source implementations of Verilog simulators. GHDL[41] is a common open-source VHDL simulator.

In order to improve productivity of hardware development using hardware description languages, modern hardware description languages and their processing environments have been proposed. Bluespec[42] is a commercial HDL with a sophisticated debugging tool and simulator. Chisel[43] is a domain specific language (DSL) on Scala for clear hardware description. It includes a fast simulation infrastructure of 8 times faster speeds compared to VCS. Many other researchers proposed various HDLs employing existing software programing languages as a DSL front-end[44, 45, 46]..

SPICE[47, 48] simulators are employed to verification of the circuit behavior in transistor level. SPICE simulation is absolutely very slower than the cycle-accurate architectural simulators. Therefore cycle-accurate architectural simulators are often used together to estimate the energy consumption of the simulated processor.

2.2.6 Discussion

There are many software simulators to strongly encourage computer architecture research. While the initial effort to develop a simulation system with an existing software simulator is relatively low, the critical issue of such software simulators is their simulation speed.

Major approach accelerating software simulations is parallelization. As a beneficial characteristic, software simulators have inherent fined-grain parallelisms corresponding to each circuit component. Parallelizing the simulators can improve the simulation performance in some moderate degree on contemporary multicore processors. Unfortunately, there are not only many fine-grain parallelisms but also very fine-grain synchronizations to keep the cycle level accuracy of the simulation result. This feature makes it difficult to accelerate simulations by running in parallel with keeping the cycle-accuracy.

While HORNET[23], as I mentioned, employs a parallelization approach, its simulation speed with the cycle accuracy of simulation results is limited by the inter-die synchronization overheads. Employing modern many-core accelerator integrating a lot of small independent cores on a single die, such as Intel Xeon Phi, will reduce synchronization overheads among the multiple threads independently running on each core. However, absolute simulation speed is still very low, up to some hundreds of KIPS or some Hz in emulation frequency. Like ZSim[24], I mentioned above, relaxing the simulation accuracy is effective to reduce the impact of synchronization overheads to the simulation speed. However, it also reduces the simulation accuracy. Additionally, the absolute speed of such software simulators is not still enough high.

While there is a trade-off between simulation speed and accuracy, my study in this thesis explores accurate approaches without any accuracy losses of simulation results.

2.3 FPGA-based Processor Prototyping

FPGA based prototyping is used for higher simulation speed than that of software simulators. The faster simulation speed strongly supports system-level evaluations of not only the processor architecture but also the application and system software.

According to literature[49], implementation ways of FPGA prototyping system are classified by three properties: (1) direct vs. decoupled, (2) full RTL vs. abstract machine and (3) single-threaded vs. multithreaded. In this thesis, I simply classify the implementation methods into two categories: *direct implementation* and *simulation-oriented implementation*. I present prior researches of each category as below.

2.3.1 Direct Implementation

Direct implementation is a natural way that the simulated processor is directly implemented on an FPGA platform with keeping almost structures of the simulated hardware.

Direct implementation will be adequate if any FPGA platform with enough capacity to implement the entire simulated hardware is available. A simulation system of the target hardware naturally is realized through standard FPGA system development flow from the RTL design.

Processor Prototyping on FPGAs

The typical use way of FPGAs is prototyping of LSIs, such as ASICs and general purpose processors. Depending on the increase of core count in a single microprocessor, several innovative schemes for rapid prototyping of multicores and many-cores have been proposed.

Heracles[50, 51] is an architectural exploration framework that provides RTL designs of the MIPS processor, scratchpad-based memory system and on-chip network, with corresponding compiler kit and software simulators. Heracles provides 3 kinds of processor RTLs of (1) single hardware-threaded MIPS core, (2) two-way hardware-threaded MIPS core and (3) Two-way hardware-threaded MIPS core. They evaluated the implemented system on a Xilinx Virtex-6 LX550T FPGA platform.

Unfortunately, available logic capacity of a single FPGA is limited. Additionally, a larger high-end FPGA is expensive. Some researchers have used multi-FPGA based platforms to extend utilizable logic capacity of FPGAs.

BeeFarm[52] is an FPGA-based multicore emulation system that supports run-time and compiler infrastructure by using open-source soft processor written in HDL, plasma, for evaluation of software transactional memory system.

ATLAS[53] is a prototyping system of 8-core PowerPC-based CMP with hardware transactional memory support, which implemented on BEE2 FPGA platform. Hu et al. developed a multi-FPGA based prototyping system for Godson-2G processor[54].

Formic[55] is a multi-FPGA based platform of 64 Xilinx Spartan-6 LX150T FPGAs, which can simulate a multi-processor system (not CMP) of 512 Microblaze soft processor cores without hardware coherence of memory. Xinyu et al. developed a many-core prototype of 48-core processors on 4 large-scale FPGAs[56].

As a practical prototyping system for the commercial system, IBM developed a large-scale prototype of Bluegene/Q processor from ASIC RTL designs on multi-FPGA platform[57]. The system operates at 4MHz that is 100,000 faster than the logic level software simulation.

SoC Prototyping on FPGAs

Numerous implementations of FPGA-based prototyping system are reported for rapid SoC (Systemp on Chip) verification. Banovic et al. developed a prototyping system of DSP (domain specific processor) to explore the future DSP systems[58]. Nakamura et al. proposed a prototyping framework of custom processors with a sophisticated debugging interface via PC[59, 60].

Kulmala et al. developed a multi-FPGA based prototyping system for GALS (Globally Asynchronous Locally Synchronous) SoC with 35 Nios II soft processors[61]. Meloni et al. proposed a cycle-accurate verification platform of ASIC (Application-Specified Integrated Circuit) evaluation[62, 63].

Logic Emulation on FPGAs

FPGA platforms are used to accelerate logic emulations with keeping cycle-accuracy of the result. Particularly, various efficient logic-partitioning techniques for multi-FPGA based emulation are proposed[64, 65, 66, 67, 68, 69, 70].

Babb et al. proposed virtual wire that is a technique to overcome pin limitations in multi FPGA-based logic emulators[71, 72]. Baviskar et al. proposed a pipelined approach to reduce the slow down of emulation speed due to cross cutting signals[73].

Discussion

The key advantage of direct implementation is RTL design compatibility to actual LSI design. If an FPGA platform has enough capacity of logic and memory to implement the target processor, the modification of RTL design for FPGA implementation is very small.

The main purpose of these prototyping systems is for efficient verification of hardware/software co-designs, instead of cycle-accurate architectural simulators. Therefore all of these prototyping systems employ FPGA-friendly microarchitecture for processor cores, instead of no complex microarchitectures such as Out-of-Order processor. In multi-FPGA platform of direct implementation, additionally, the cycle-accuracy depending inter-FPGA communication may not be necessarily kept.

The problem, however, is that a large FPGA of appropriate capacity for simulated processor is required. If multi-FPGA platforms are not considered, a high-end and expensive FPGA is required for simulations of large systems, such as a many-core processor with over 100 cores in this way. As a critical problem of rapid prototyping using large FPGA platforms, the synthesis time that is the elapsed time to create an FPGA circuit image (bitstream) from HDL source codes becomes critically long if large FPGAs are used.

If multi-FPGA platforms are available, architects should think about how to partition the simulated processor design into multiple parts and how to keep the cycle-accuracy of simulation results related to inter-FPGA communications. Those prior researches, as I mentioned above, did not adopt any techniques to satisfy the cycle-accuracy related to cross cutting signals among FPGAs.

The purpose of the logic emulation approaches is to verify the logics by longer simulation than the software RTL simulators. Therefore the cycle-accuracy is a most important concern. The cycle-accuracy should be satisfied even in multi-FPGA platforms. The critical problem of multi-FPGA based logic emulation is slow down of the emulation speeds due to signal dependencies among FPGAs.

They also lack any appropriate abstractions to emulate memory components of processors. All memory components should be perfectly mapped into adequate on-chip FPGA resources, even if a fat processor design with massive memory components is emulated. In general, structure of microprocessors is not necessarily suitable for FPGA characteristics. In respect of memory systems, a processor

has several on-chip cache memory systems. Usually these cache memory systems are implemented by using on-chip fast memory fabrics (called as block RAM) of FPGAs. Unfortunately, the amount of on-chip memory fabrics of FPGA is limited and few.

If the simulated processor requires larger capacity of on-chip FPGA memory than the available capacity of the FPGA, the simulator developer should consider combining external memory components (DRAM). Development of such a hierarchical memory system with keeping the cycle-accuracy of simulation results is highly error-prone and time-consuming.

Therefore appropriate abstractions of inter-FPGA communications and memory system are required for rapid development of useful FPGA-based prototyping systems.

2.3.2 Simulation-Oriented Implementation

The main purpose of simulation-oriented prototyping systems is just to accelerate processor simulations with keeping cycle-accuracy of processor behavior. Against the direct implementation approach, the simulation-oriented approach does not necessarily keep the hardware structures of the simulated processors on FPGA platforms, because the purpose of simulation-oriented systems is just to rapidly obtain execution results of the benchmark applications with some performance information, not to verify the logic property in register transfer level. Therefore simulation-oriented systems employ the specialized hardware structure to efficiently simulate the behavior of the target.

HW/SW Hybrid Approach

FAST[74] simulator is a hardware/software hybrid simulator that splits the simulation system into a QEMU-based functional emulator and an FPGA-based accurate timing model. Decoupling into functional model and timing model simplifies the simulator implementation and improves the customizability to other processor design with different property.

PROTOFLEX[75] is also HW/SW hybrid simulator to accelerate cache simulation of CMP systems. PROTOFLEX virtualizes the execution of many logical processors onto a consolidated number of multiple-context execution engines on the FPGA. Through virtualization, the number of engines can be scaled to deliver on necessary simulation performance at a substantial savings in complexity.

Pure HW Approach

RAMP Gold[76] is a simulation-oriented system of 64-core SPARC processor with L2 cache memory system which does not contain cache-coherence. RAMP Gold also employs the multithreaded simulation feature to improve hardware resource utilization as well as PROTOFLEX so that the system can be implemented on a common middle range FPGA board.

HAsim[77] is a state-of-art FPGA-based full-system simulator of a multicore processor with 16 cores and detailed on-chip network, with A-Ports[78] technique to reduce hardware resource pressures.

HAsim employs multithreaded feature not only for processor core simulation but also on-chip router simulation. On-chip network is a key feature to determine overall performance of modern microprocessors.

Arete[6] is a different sophisticated simulation platform with detailed core model, coherent cache memory and on-chip network. In Arete, the simulated architecture is represented in cycle-level specification, and is then transformed into simulation-oriented FPGA-optimized RTL using LI-BDN[79] technique that relaxes the connectivity amount each hardware unit and helps to improve the FPGA cycle time and to reduce the FPGA resource requirements by using multiple FPGA cycles to simulate one cycle of the target architecture.

Network on Chip Simulator on FPGAs

FPGA-based simulation accelerators are not just for full-system processor simulation but also for particular components of processors. DART[80] is FPGA-assisted NoC cycle-accurate simulators. DART employs the function-timing-decoupled style as well as RAMP Gold and ProtoFlex to support various NoC simulations without re-synthesis of FPGA.

FIST[81] takes another approach that abstractly models each router as a load-delay curve and sum of load-dependent delay at each visited router's load at runtime. Mostefaoui et al. developed a multi-FPGA based NoC simulation platform stratifying near cycle-accuracy of the simulation result[82, 83].

GPU Architecture Simulator on FPGAs

FastLanes[84] is an FPGA-based architectural simulator of GPU microarchitecture. FastLanes has a functional model and a timing model on a FPGA as well as the other simulation-oriented FPGA simulators. FastLanes outperforms its software equivalent by up to 2 orders of magnitude.

Discussion

The main advantage of these simulation-oriented systems is the fact that the hardware utilization is much smaller than that of the direct implementation approach. Decoupling functional model and timing model can avoid implementation of costly hardware components, such as forwarding unit.

These simulators also employ the multithreading technique. When the target system contains multiple instances of the identical component, such as processor cores in a many-core processors, the simulation engine can be modeled so that one physical hardware unit simulates multiple target components by interleaving the component models' execution with multithreading manner.

For example, the behavior of the target processor with four isolated cores is modeled using a single processor core hardware, as shown in Figure 2.2(b). The single physical core unit is reused for 4 target cores so that total hardware consumption can be dramatically reduced. If 4 target cores are simulated without multithreading, 4 physical hardware units are required, as shown in Figure 2.2(a). Note that, in either cases, the execution context, program counter and register file values, for each core should be

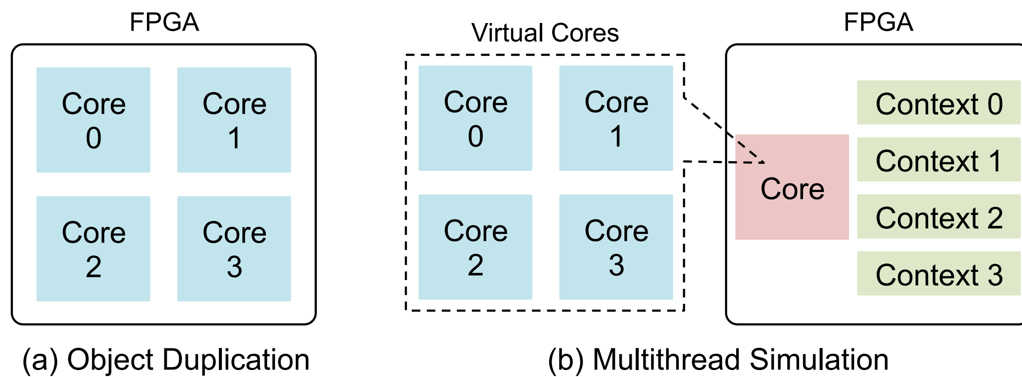


Figure 2.2 Object Duplication vs. Multithread Simulation

stored independently on the FPGA.

A critical drawback of these simulation-oriented approaches is that the RTL design for simulation-oriented system is much different from the RTL design for pure implementation. In order to evaluate energy and area of the simulated processor, a pure RTL design of the target architecture is required. Therefore the architect has to implement both pure RTL design of simulated processor and corresponding yet another RTL design just for simulation acceleration.

Additionally, employing multithreading technique to improve hardware utilization incurs a scalability problem with the increase of simulated core count. Since a single physical object is shared to simulate entire same kind objects, activation rate of each simulated object certainly degrades. Finally, with the increase of simulated core count, the simulation frequency degrades. The simulator developer should find a balanced point between multithreading simulation and duplicating physical object for efficient simulation.

2.4 Goal of This Research

Based on the discussions above, I summarize my goals of this research. In this thesis, I establish a sophisticated prototyping framework for future many-core processors. The framework consists of two key technological contributions as follows:

- the acceleration method of cycle-accurate processor simulations on multi-FPGA based platforms; and
- the design methodology under the resource abstraction of FPGA platforms, in order to reduce the development complexity of FPGA-based processor simulators.

The integrated framework aggressively improves prototyping efficiency using FPGAs for emerging many-core processors.

I describe as below each technology for the final goal of this thesis.

Multi-FPGA based Fast Processor Simulation Platform with Scalable Simulation Speed and Short Synthesis Time

In this study, I explore a cycle-accurate and scalable FPGA-based prototyping scheme for many-core processors. While the general FPGA-based prototyping requires the extensive synthesis time to generate a FPGA circuit image from RTL designs, FPGA-based prototyping system is still effective for efficient processor simulation because of its absolute simulation speed. In order to avoid additional design efforts of RTL designs for the simulation-oriented approach, this research explores a sophisticated *direct implementation* approach to satisfy the scalability of simulation speeds and the cycle-accuracy of simulation results.

As the first technological contribution of this thesis, I propose a multi-FPGA based simulation method that offers the fast simulation speed, the scalability against the core count, short synthesis time with keeping the cycle-accuracy of simulation results.

In order to archive scalable simulation speed with increasing the core count, my method employs a tile structure using multiple FPGAs. With increasing the core count of the simulated processor, my method can increase the used FPGA count in order to keep the same simulation speed. It enables that the simulation system achieves the perfect weak-scaling in simulation speed.

Another beneficial point of this strategy is the reuse of FPGA circuit image for synthesis time reduction. The simulated design is portioned into multiple regions so that they are mapped into for each small FPGA. With the aid of module level redundancy of manycores, the identical circuit image is re-used for several FPGAs of the system. It dramatically reduces the circuit synthesis time.

Design Methodology of FPGA-based Prototyping Systems under Resource Abstractions of FPGA Platforms

The advantage of the direct implementation approach is that almost identical RTL designs can be used both for LSI fabrications and for FPGA-based prototyping. However, the simulator designer should manage the characteristic gap between the raw RTL design of simulated processors and the utilizable resources on FPGA platforms. If the required resource amount exceeds the on-chip capacity of logic or memory on a single FPGA, simulator designers should consider employments of multiple FPGAs or off-chip memory components. It needs several time-consuming and error-prone steps, especially for cycle-accurate simulation results. Therefore appropriate abstractions of inter-FPGA communications and memory system are required for rapid development of useful FPGA-based prototyping systems.

As the second technological contribution of this thesis, I propose an abstraction methodology of various resources on FPGA platforms, in order to mitigate the critical gap between ideal LSI-oriented

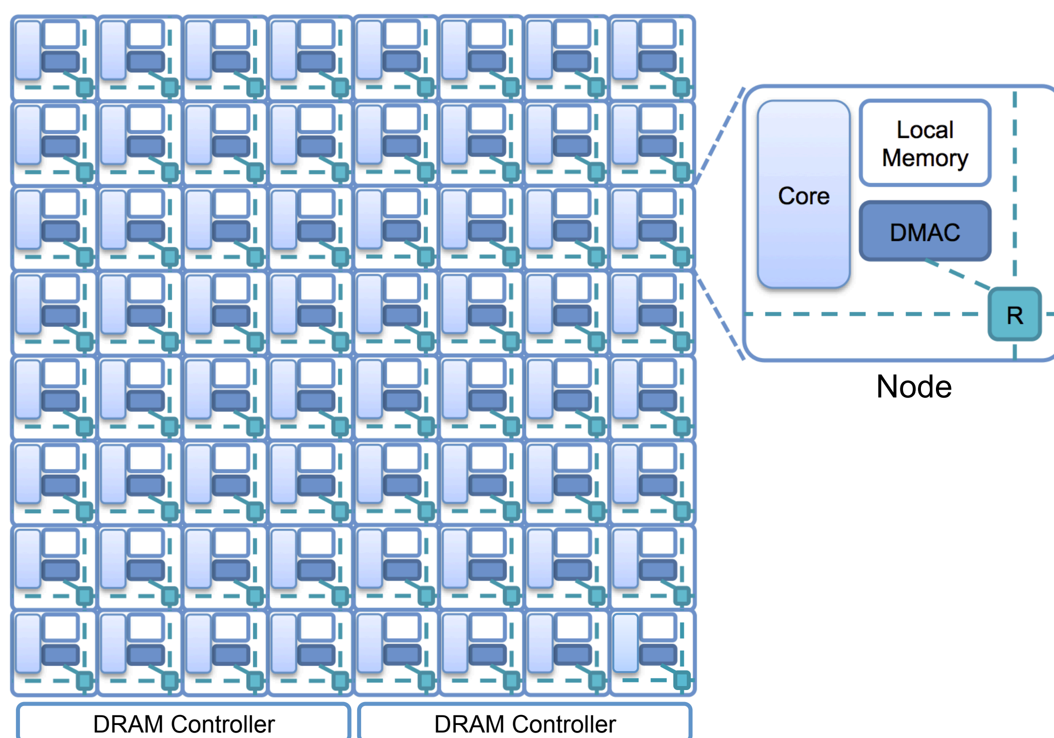


Figure 2.3 M-Core Architecture

RTL designs and FPGA-prototyping-oriented RTL designs. In order to prevent FPGA-specific implementations of simulated processors, the design methodology offers two abstract objects corresponding to memory resources of FPGA and inter-FPGA communications on multi-FPGA platforms.

RTL designs using the provided abstractions should be translated into actual FPGA-oriented RTL designs for FPGA implementation. I developed a IP-core synthesis tool-chain that generates a FPGA-oriented RTL design package of simulated processors as an IP-core, for my methodology with resource abstractions. The tool-chain automatically synthesizes an AMBA AXI4 (a popular on-chip interconnect standard) IP-core from the input RTL design under the resource abstractions. In order to change the hierarchy of input RTL designs, I also developed an open-source hardware design processing toolkit for Verilog HDL RTL.

2.5 Baseline Many-core Architecture as a Prototyping Target

This section presents a baseline many-core architecture used as a prototyping target in this research. I use M-Core architecture[28] as a prototyping target for evaluations of my methods. Figure 2.3 shows the architectural overview.

M-Core is a many-core accelerator with multiple computation nodes and a 2D mesh interconnection

connecting the nodes. In this thesis, the computation node is represented as *Node*. Each Node has a processor core, a local memory, a DMA controller and an on-chip router. The interconnection for inter-core communications is a modern packet-based on-chip network, instead of a legacy shared bus, for the high performance and the scalability. On-chip data sharing is represented by using DMA operations to another Node, similar to Cell/B.E. It simplifies the structure of hardware systems. Note that the number of cores is not identified in the architecture. The architect can arrange the appropriate count of cores.

2.6 Summary

In this chapter, I presented the background of my work. As the fundamental knowledges, I described the purpose of processor simulation and the standard evaluation flow of microprocessors using various simulation tools.

I described the prior researches of software-based processor simulations. Numerous software simulators have been developed to strongly encourage processor architecture researches. The purpose and structure of software simulators have been changed with the paradigm shift to multicores.

The software simulators are helpful for early stage evaluations, due to their eases on modifications. The critical drawback of the software simulators is that the simulation speed is very slow. Parallelization of software simulators can improve the simulation performance in some degree. However, the scalability of simulation speed on the parallelization is restricted by inter-core communication overheads. In order to more accelerate the simulations by parallelizations, the accuracy relaxation approach is highly effective. Unfortunately, it incurs incorrect simulation results. Therefore there is a trade-off between the speed and the accuracy.

I introduced the related researches of FPGA-based processor simulations. By using FPGAs, processor simulations are highly accelerated by employing inherent fine-grain parallelisms with frequent synchronizations. It enables larger and realistic simulations that are difficult on the software simulators. I classified the FPGA-based simulators into two categories based on their strategies: direct implementation and simulation-oriented implementation.

The direct implementation approach is a natural prototyping way to realize a simulation target on FPGAs. In order to implement a simulation target on FPGAs, the direct implementation approach does not usually require so many modifications of the RTL designs. FPGA-based simulators of direct implementation are efficient especially for verification of hardware/software co-designs. Unfortunately, capacity of logics and memory blocks on a single FPGA is limited. To expand the logic capacity, multi-FPGA based platforms are typically used. To expand the memory capacity, an off-chip memory component, such as DRAM, is also used together. In such situations, simulator designers should map the simulated processor with the adequate logic partitioning and hierarchical memory systems onto FPGA platforms. To this end, it takes highly error-prone and time-consuming steps in the system

development.

Simulation-oriented FPGA-based simulators can achieve good area efficiency so that the simulation system can be implemented on a single middle range FPGA. However, the main purpose of simulation-oriented prototyping systems is to accelerate cycle-accurate processor simulations. They employ some clever techniques of decoupling the simulated processor design into a functional model and a timing model, and multithreaded simulation to reduce the hardware resource overhead. However, these techniques bring a critical drawback that it requires yet another RTL design instead of pure RTL design of the simulation target for LSI-fabrications.

Based on these discussions, I presented the goal of this research. The goal of this research is to realize a sophisticated prototyping framework for future manycores. The framework consists of two key technologies: The acceleration method of cycle-accurate processor simulations on multi-FPGA based platforms; and the design methodology under the resource abstraction of FPGA platforms in order to reduce the development complexity of FPGA-based processor simulators. The framework aggressively improves the prototyping efficiency for emerging many-core processors.

Finally, I presented M-Core, a baseline many-core architecture used for evaluations of my research. M-Core is a many-core accelerator with simplified DMA-based on-chip memory systems and a 2D on-chip network.

Chapter 3

ScalableCore System: A Multi-FPGA based Processor Simulation Platform

In this chapter, I introduce the acceleration method of cycle-accurate processor simulations on multi-FPGA based platforms. To counter scalability issues of the simulation speed and the synthesis time, I propose *ScalableCore system*, a cycle-accurate FPGA-based simulator of a scalable platform structure using multiple FPGAs.

I introduce the overall concept and architecture of ScalableCore system. ScalableCore system is an extendable platform using numerous FPGA units corresponding to each processor core of a 2D mesh many-core processor. The core count of a simulated processor on ScalableCore system can be increased by changing the number of used FPGA units. The remarkable characteristic of ScalableCore system is that the system achieves the perfect weak-scaling at simulation speeds increasing the core count of the simulated processor, with keeping the cycle-accurate simulation results. In other words, even increasing the core count of a simulated processor does not decrease the rate of simulated cycle count per unit time, corresponding to Hertz (Hz).

Not just to propose the architecture of a simulation system, in order to evaluate the viability of the proposed method, I developed an actual test bed FPGA platform of ScalableCore system. The evaluation result shows that the method of ScalableCore system provides the good scalability of simulation speeds with low overheads in FPGA resource consumption.

I show informative case studies of many-core processor evaluations using the developed test bed system. The first case study is an evaluation of a task allocation scheme on a many-core processor. I evaluated a novel task allocation scheme on many-core processors to improve the chip-level performance. The second case study is an evaluation of a task allocation scheme on a dependable many-core processor with DMR (dual modular redundant) execution mode. I explored a balanced strategy of task allocations on many-core processors for both high performance and high dependability, by using the test bed system.

The main contributions of this chapter are as follows:

- to describe the overall concept and architecture that realizes the scalable simulation speed with keeping the cycle-accuracy;
- to describe the test bed implementation features;
- to describe the evaluation results using the test bed system, compared with a corresponding software-based simulator and prior FPGA-based prototyping platforms; and
- to describe the case studies of many-core platform evaluations using the test bed system.

3.1 Motivation

FPGA-based prototyping is effective way for efficient processor simulation, due to its absolute simulation speed. However, FPGA-based prototyping using a large high-end FPGA requires a very long time to synthesize an FPGA circuit image file (as known as *bitstream*). Especially future many-core processors will be greater in circuit sizes.

Fortunately, many-core processors have numerous redundant components, such as processing cores. With appropriate preparations, we can re-use the identical circuit data for multiple FPGAs. Additionally, by employing the tile architecture for the simulation system and by splitting the simulated processor design into multiple parts in advance, we can easily expand the simulated processor with keeping the fast simulation speed and the cycle-level accuracy. To this end, I choose a multi-FPGA based approach, in order to reduce the synthesis time and to obtain the scalable simulation speed against the core count of a simulated processor. In this work, I aim for hundreds fold speed up of processor simulations with keeping the cycle-accuracy with the multi-FPGA way.

3.2 Overall Concept and Architecture

I propose ScalableCore system, a scalable environment using multiple FPGAs for tile architecture simulation[85, 86]. In this study, I selected as a prototyping target M-Core architecture[28] described in the previous chapter, a DMA-based homogeneous many-core architecture with a mesh on-chip network.

Figure 3.1 shows an example structure of ScalableCore system to simulate a many-core processor with 16 processor cores and 4 off-chip memory interfaces. The system consists of 16 FPGA units corresponding to processor cores and 4 another FPGA units corresponding to off-chip memory interface. Whole the system emulates behavior of the target processor.

ScalableCore System is an implementation fashion that the simulated processor is partitioned into multiple regions, and then connects them to construct an entire processor. If identical partitioned regions exist, sharing the same FPGA circuit image can reduce the synthesis time of the circuit image. Finally it helps much quick FPGA-based simulation accelerator development.

FPGA unit corresponding to a processor core is named as ScalableCore Unit, and FPGA unit cor-

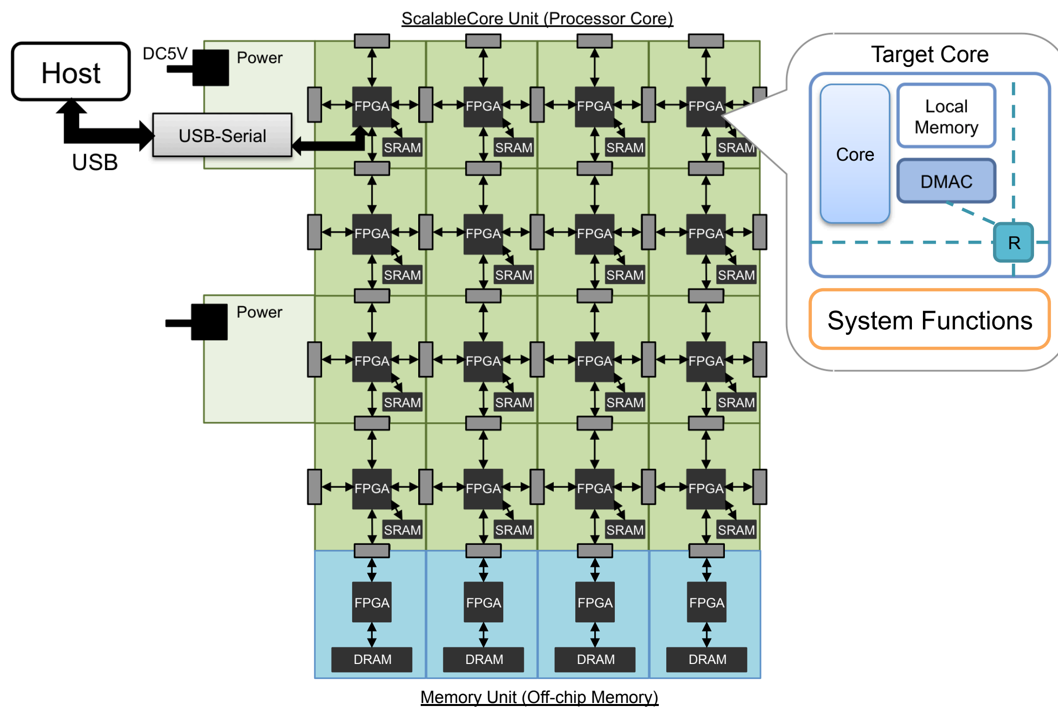


Figure 3.1 Concept of ScalableCore System

responding to a memory interface is named as Memory Unit, respectively in this work. ScalableCore Unit is a small circuit board with a single FPGA (Xilinx Spartan-6 XC6SLX16) and an independent SRAM IC. Memory Unit is also a small circuit board with a single FPGA (as same as ScalableCore Unit) and an independent DRAM IC.

As the important feature of ScalableCore system is that the simulated core count can be increased by increasing the amount of employed FPGA units. For instance, If a many-core processor of 64 (8×8) cores and 4 DRAM controllers is the simulation target, the system of 64 (8×8) ScalableCore Units and 4 Memory Units is the corresponding setup.

As another important feature of ScalableCore system is the connectivity. Every ScalableCore Unit is connected to only 4 neighbors of upside, downside, left side and right side. ScalableCore Unit has 4 bi-directional serial I/O ports to be connected and communicate to the neighbors. For the system stability in the case of an extended system, ScalableCore system has no global signals through whole the system, and each ScalableCore Unit works under its own clock signal by the clock oscillator. Therefore the system is extendable for the target core count by increasing the number of ScalableCore Units with keeping the stability of the system.

Power supply is also essential for stability of the system. DC 5V power to drive the system is supplied from the left edge of the system and is shared by the same rows of ScalableCore Units. The voltage

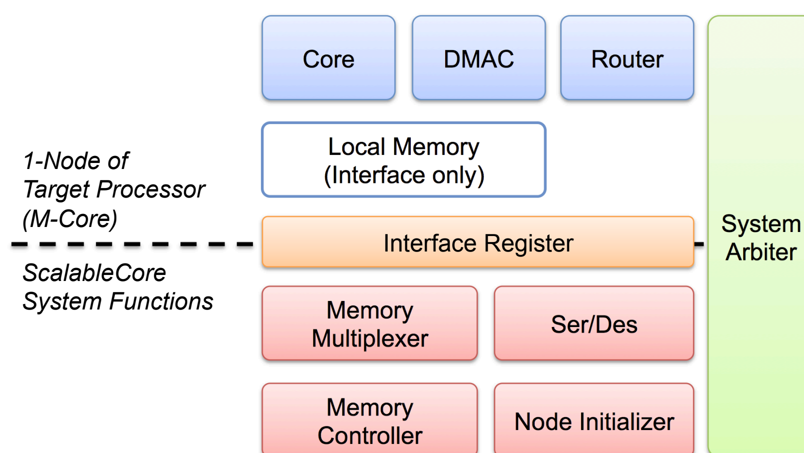


Figure 3.2 Function Stack of ScalableCore System

of applied power is changed to the appropriate voltage to drive each IC by the DC-DC regulator each ScalableCore Unit has.

In the current implementation, applications running on the simulated processor are loaded via the USB-serial controller from a host PC to the ScalableCore Unit in upper left in the figure. Simulation results are transferred to the host from the ScalableCore Unit via the USB-serial controller.

3.3 Design

3.3.1 Architecture of ScalableCore Unit

ScalableCore system consists of multiple ScalableCore Unit (FPGA node) with processor core functions and some system level functions to control the system. Figure 3.2 shows the function stack of ScalableCore Unit. Figure 3.3 shows the architecture of ScalableCore Unit.

One of principal motivations of this work is to reduce the synthesis time of FPGAs. In ScalableCore system, the simulated processor is manually partitioned into multiple regions corresponding to each ScalableCore Unit. Since our simulation target is M-Core, a pure and homogeneous many-core architecture, all ScalableCore Units use the identical FPGA circuit image (bitstream). It realizes short synthesis time for all FPGA circuit images, even if the core count of the simulated processor increased.

In the current system for M-Core, each ScalableCore Unit has 4 functions of M-Core Node: Core, Router, DMAC and Local Memory. System function in ScalableCore Unit is an essential component to support cycle-accurate emulation cooperated with all ScalableCore Unit.

Each FPGA emulates behavior of the target core by taking synchronizations of simulation status for emulated clock cycle. As I explain later, synchronization combinations of each ScalableCore Unit are restricted to the 4 neighbor ScalableCore Units. We named this bound-reduced synchronization as *local*

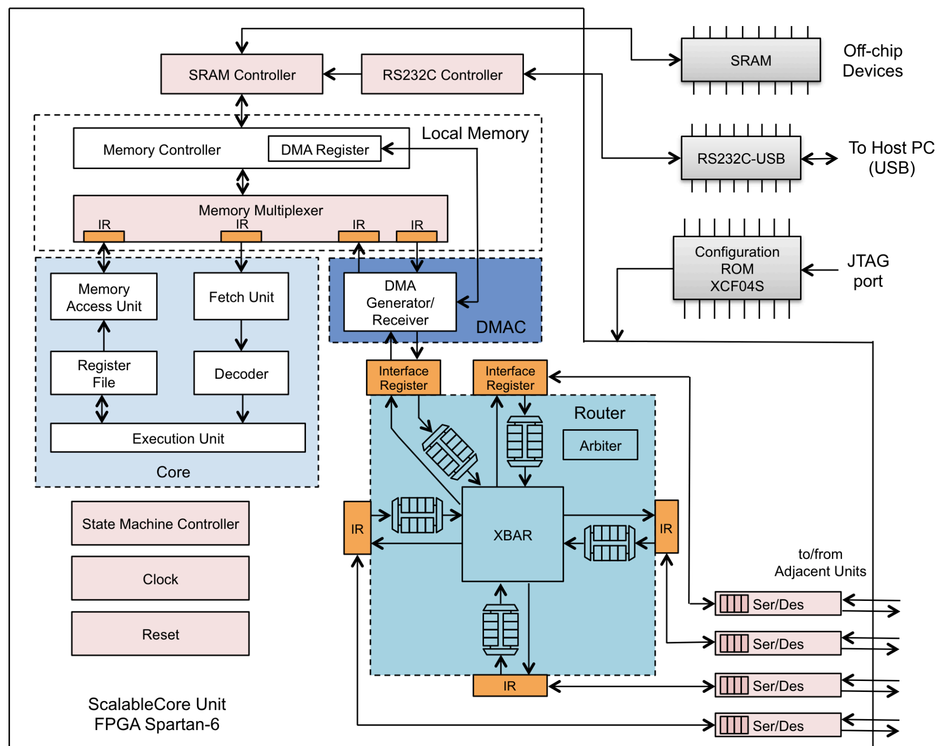


Figure 3.3 ScalableCore Unit

barrier synchronization. It serves the simulation speed scalability in any core count.

As the low level feature of ScalableCore Unit, FPGA-FPGA communications are treated via Ser/Des (serializer / de-serializer) components with NRZI encoding and parity code for reliability. Since Ser/Des runs faster than the other internal components, asynchronous FIFOs are employed in the boundaries of clock domains.

ScalableCore system adopts an abstraction technique to easily implement a target processor, similar to A-Ports[78]. In the figure, Memory multiplexer and state machine controller are important system functions to virtualize the limited FPGA resources in time-multiplexing manner. SRAM controller, Clock and Reset are basic functions of the system.

3.3.2 Architecture of Memory Unit

In order to emulate an off-chip memory, Memory Unit consisting of a DRAM emulation component instead of several processor core components in ScalableCore Unit is used. Figure 3.4 shows the architecture of the Memory Unit. Memory Unit has an off-chip DRAM on the board, and uses a primitive DRAM controller of Spartan-6 hard macro. In order to provide adequate access latency of DRAM in the simulation world, this primitive DRAM controller is wrapped with a DRAM timing

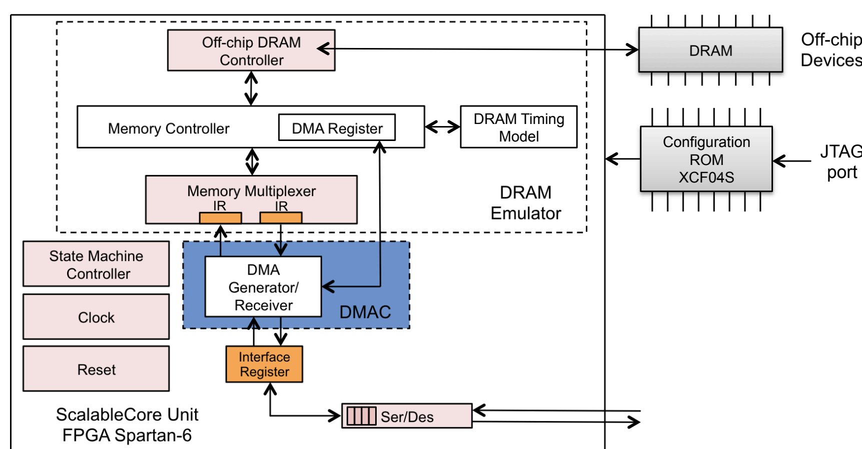


Figure 3.4 ScalableCore Memory Unit

model. DRAM timing model generates stall signal to control the Memory Controller for access latency emulation.

3.3.3 System Level Function for Cycyle-Accurate Simulation

In ScalaleCore system, two techniques are important to satisfy the scalability of simulation speed and the cycle-accuracy of simulation: local barrier synchronization and virtual cycle.

Virtual Cycle

Direct implementation of processor component is not easy due to FPGA resource characteristics and limitations. As prior works, a clock cycle on the FPGA should not necessarily correspond to a clock cycle on the simulated processor. In order to emulate complex hardware component, such like multi-port RAM, using simple FPGA-friendly hardware component, a reasonable way is to emulate a clock cycle of the simulated processor by using multiple clock cycle on FPGA.

In order to simulate the target processor with the cycle-accurate manner, each ScalableCore Unit takes a synchronization of simulation status for every simulated clock cycle. Synchronization via the Ser/Des units requires some physical traversal latency. We well scheduled multi-cycle circuit emulations and synchronizations via Ser/Des components so that these operations are overlapped to improve the simulation performance.

In this work, this method of well scheduling of time-multiplexed circuit simulation and synchronization communication is called Virtual Cycle. It makes easy to apply complex hardware components on FPGAs with keeping cycle accuracy and simulation speed.

Figure 3.5 shows a timing chart of virtual cycle in a ScalableCore Unit. In the beginning of every virtual cycle, all statuses of internal registers and output signals in the target processor are emulated.

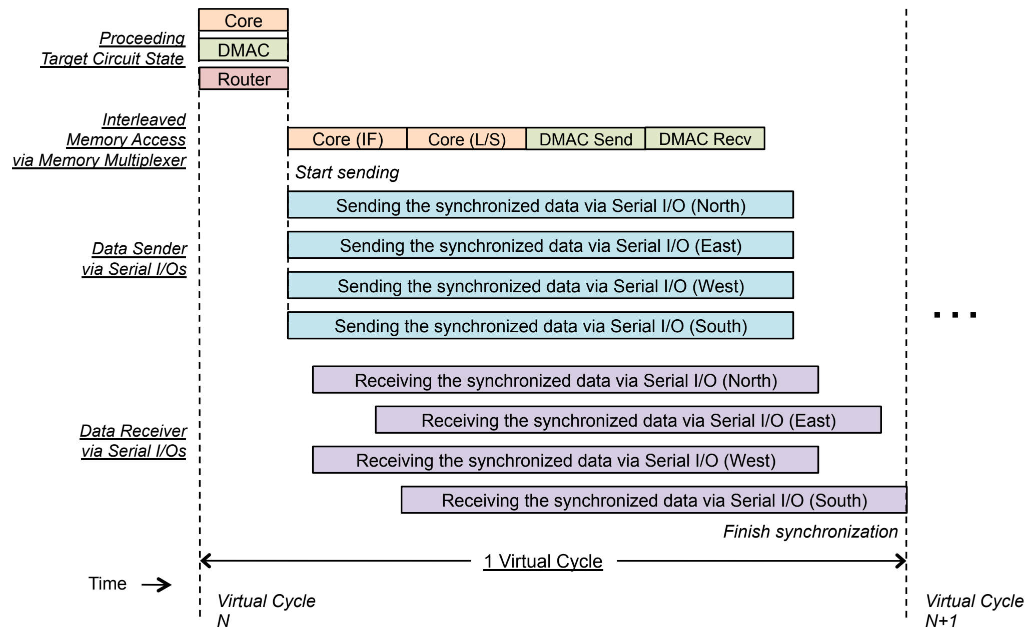


Figure 3.5 Virtual Cycle

Then, the memory multiplexer and memory controller emulate behavior of memory accesses using the latest status in the target in order to emulate memory operations.

Now we assume that the simulated component on each ScalableCore Unit requires 4 ports for memory accesses: (1) instruction fetches for a core, (2) load/stores for a core, (3) reads for a DMA controller and (4) writes for a DMA controller. In contrast, hardware unit for emulating local memory is 1-port memory controller with some DMA control registers and memory-mapped registers. This gap in the number of ports is resolved by the time-division multiplexing. Finally the memory units complete the emulation of multi-ported memory.

At the same time, the latest simulation data are arrived from the neighbors. Since every ScalableCore Unit runs under the control of the clock oscillator, the arrival timing of synchronization data is different for each ScalableCore Unit. To ensure the cycle-accuracy, state machine controller of ScalableCore Unit waits for all synchronization data transfers and memory emulation. After the synchronization, simulation step goes to the next cycle of the simulated processor.

Local Barrier Synchronization

In ScalableCore system, the simulated processor is partitioned and mapped into multiple FPGAs. In order to obtain a cycle-accurate simulation result for all FPGA units, each ScalableCore Unit needs to use the latest simulation states generated in other ScalableCore Units.

A simple way to make this is to take an all-to-all barrier operation with respect to each emulated

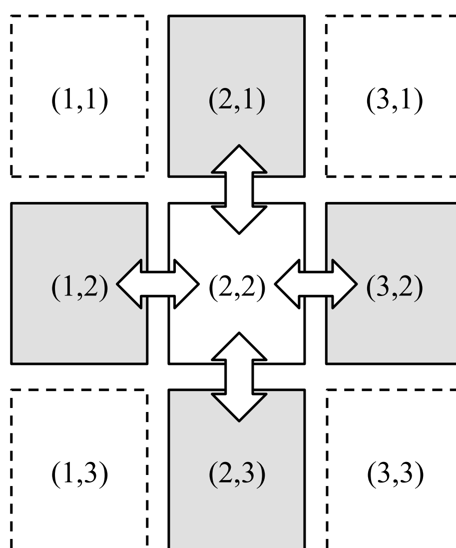


Figure 3.6 Local Barrier Synchronization

cycle is proceeded on each ScalableCore Unit. However, taking an all-to-all operation is not realistic because of its large overheads interfering the system scalability.

Now we again consider the simulation target architecture. The current ScalableCore system focuses on M-Core, a tile architectures with 2D mesh on-chip network. Therefore, to satisfy the cycle-accuracy, each ScalableCore Unit has to know the newest simulation status of only its 4 neighbor ScalableCore Units.

Newest signal state of a simulated component generated at cycle N can propagate up to only its neighbors till the next cycle, cycle N+1. Therefore, each ScalableCore Unit has to wait just its 4 neighbor ScalableCore Units to satisfy the cycle accuracy. I named local barrier synchronization for this minimum status synchronization technique. This technique enables to increase the simulated core count by increasing employed ScalableCore Units without synchronization overhead increase.

3.3.4 RTL Design Rule

Simulation state is defined as output of emulated hardware, such as flip-flops and wires. In order to update at certain timing using the correct value, ScalableCore system requires a special input signal for each register update statement in RTL designs. Figure 3.7 shows a sample code of emulated component written in Verilog HDL.

Update statements for flip-flops (always statement in Verilog HDL) have an *if(EN)* rule. *EN* is an input signal to drive the update statements by state machine controller at the beginning of a virtual cycle.

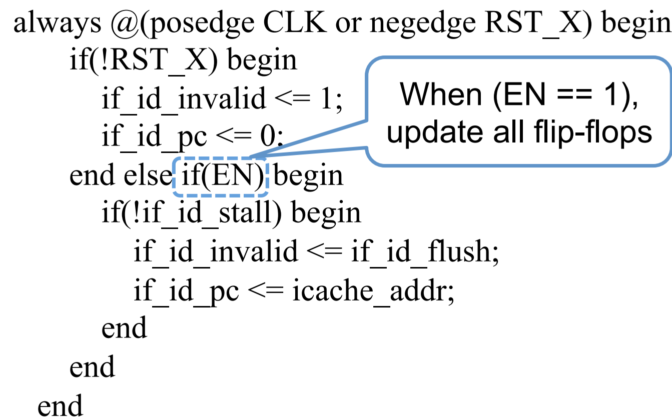


Figure 3.7 RTL Coding Rule for Virtual Cycle

Additionally, to avoid propagating newest output state of emulated hardware to the other hardware components before the next virtual cycle, interface registers (IR in Figure 3.3) are inserted in boundaries of hardware components. Then values of them are updated at the end of a simulated cycle. Insertion points of interface register represent the modularity of each emulated component. The system designer should manually insert interface registers into the design of ScalableCore Unit.

3.4 Hardware Platform Implementation

To evaluate the viability of my proposal, I developed an actual test bed of ScalableCore system. In this section, I explain implementation features of the actual ScalableCore system test bed. Figure 3.8 shows a snapshot of ScalableCore system consisting of 100 ScalableCore Units. The overall size of the system of 100 ScalableCore Units is very small, 46.7cm $\times$ 60cm.

As shown in left of Figure 3.9, a ScalableCore Unit is a small card-sized, 4.67cm $\times$ 6.0cm, FPGA board with Xilinx Spartan-6 XC6SLX16 (Speed Grade -2), a 512KB (1-port, 8 bit $\times$ 512K entry) SRAM and a configuration ROM Xilinx XCF04S.

A ScalableCore Unit also has a JTAG port to write circuit information for FPGA and configuration ROM. Several FPGA's I/O ports are assigned to external I/O pins in edge of the board to communicate the neighbors. To drive the system of 100 units requires DC 5V power supplies.

In the current implementation, clock frequency of the oscillator is 40 MHz. Clock frequency of the Ser/Des modules is 80 MHz. The data transfer rate of the Ser/Des modules is 80 Mbps.

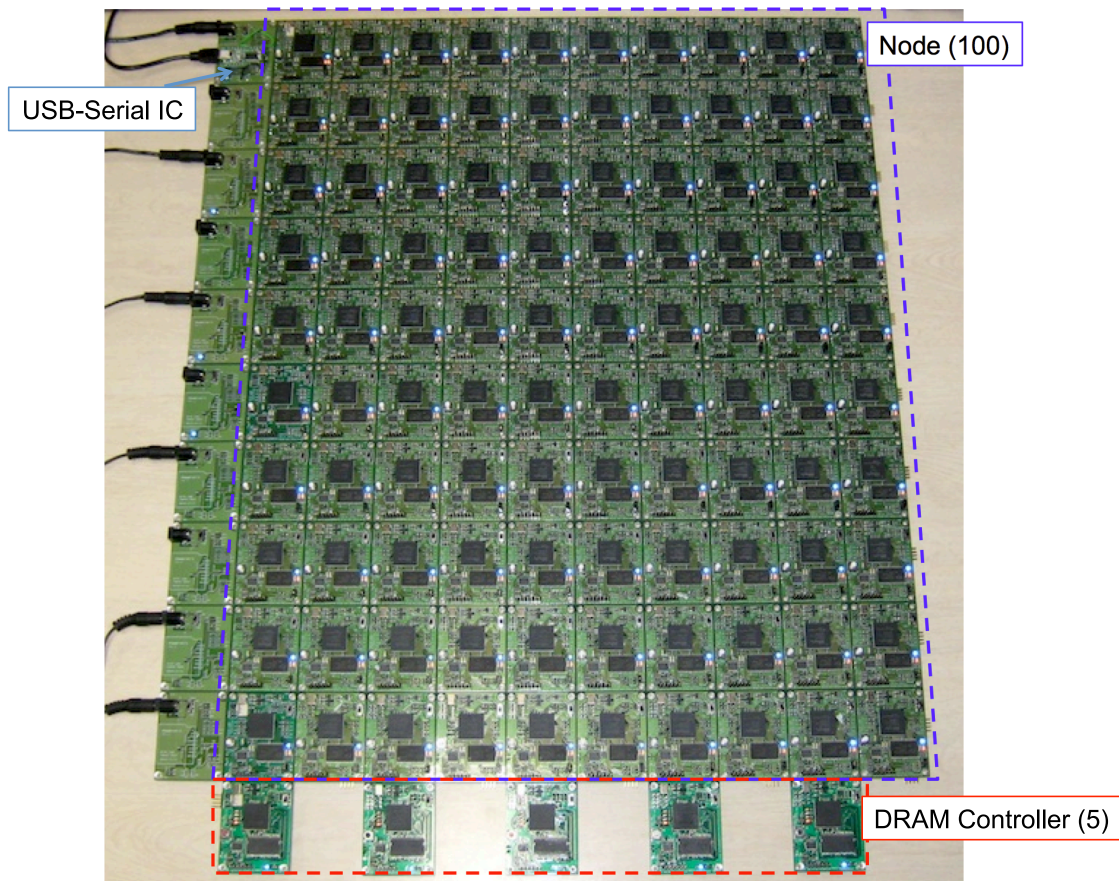


Figure 3.8 Snapshot of ScalableCore System with 100 FPGA Units

3.5 Evaluation

This section provides the evaluation results of the method of ScalableCore system on the actual test bed system. I evaluated the system in two points: (1) simulation speed and (2) FPGA resource usage. I used Xilinx ISE 12.4 for FPGA circuit synthesis.

3.5.1 Simulated Processor Setup

The configuration of the simulated processor for evaluations of ScalableCore system is listed in Table 3.1. As I noted above, I choose M-Core architecture, a homogeneous 2D mesh NoC-based many-core architecture presented in the background chapter, for the evaluations. The processor core is a simple single-issue pipeline core without floating point units. The router has an advanced micro architecture of 4-stage and 2-VC for use in NoC-oriented research. DMA Controller has 2-port for DMA read and DMA write to local memory. To realize a on-chip node memory of each Node, a SRAM chip on a

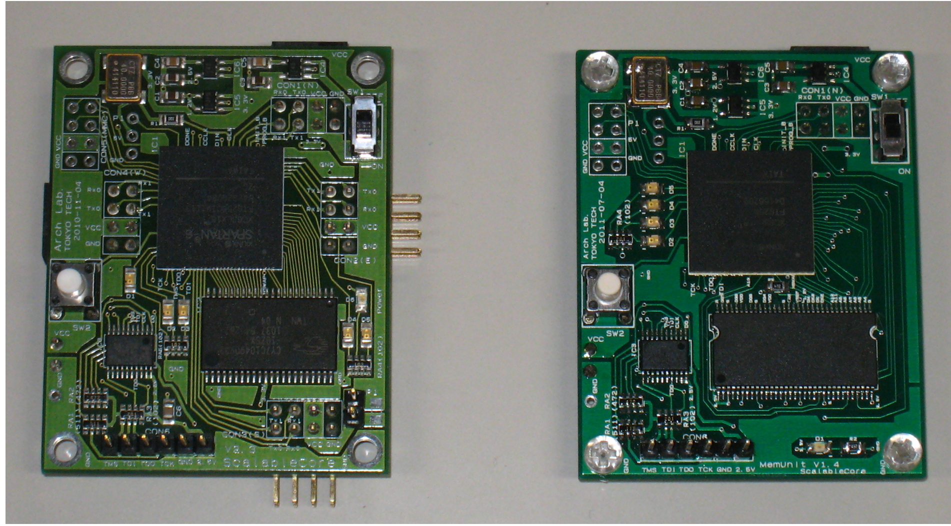


Figure 3.9 Snapshot of ScalableCore Unit (Left) and Memory Unit (Right)

Table 3.1 Microarchitecture of Simulation Target (M-Core)

Core	MIPS32 ISA, 5-stage, Single-issue,2-memory-port (Fetch, Load/Store)
DMA controller	2-memory-port (32-bit DMA Read and DMA Write)
Network topology	2-D mesh
On-chip router	5-input/output (North, East, West, South, Core), 4-stage (NRC and VA: Next Route Computation and Virtual Channel Allocation, SA: Switch Allocation, ST: Switch Traversal, LT: Link Traversal), 2-virtual-channel, FIFO depth: 4, Credit-base flow control, X-Y Dimension Order Routing
Node memory	512KB (per Node), 32-bit width, access latency: 1 4-port (Fetch, Load/Store, DMA Read, DMA Write)

ScalableCore Unit is used.

3.5.2 Simulation Speed

In order to demonstrate the speed scalability of ScalableCore system, I evaluated the simulation speed of actual test bed of ScalableCore system by comparing with the corresponding cycle-accurate software simulator, SimMc. The configuration in microarchitecture is listed in Table 3.1, but the flow-control of on-chip network in SimMc is Xon/Xoff. I measured the simulation speed of SimMc running on a standard computer with Intel Core i7 870 with 4GB memory. As the compiler, gcc 4.5.2 is used with -O3 option. The operating system is Ubuntu server 11.04.

I used two benchmark applications for the speed evaluation: N-Queen (NQ) and Matrix multiplication (MM). N-Queen is a computing intensive and massive parallel application of master-worker model,

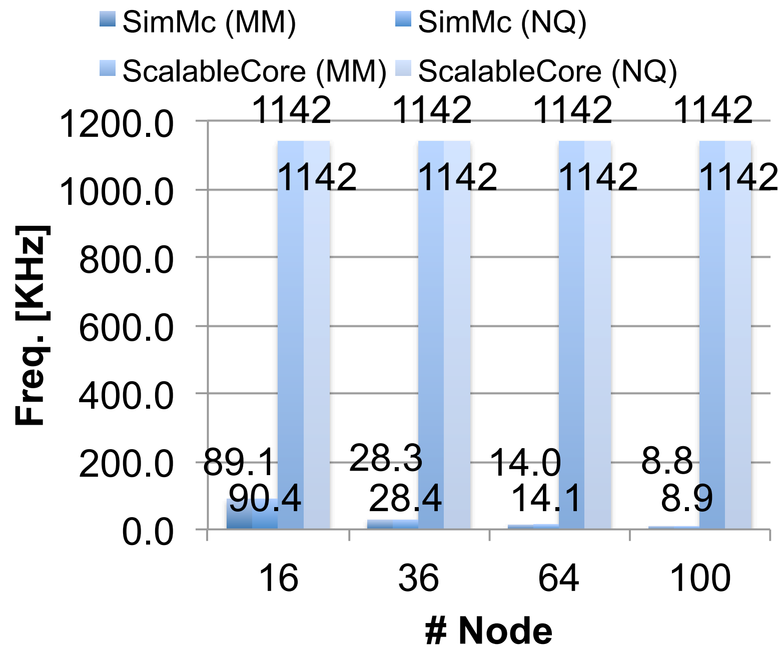


Figure 3.10 Simulation Speed

which contains limited DMA communications. The matrix multiplication I used contains much more DMA communications than the N-Queen.

Figure 3.10 shows the simulation speeds of SimMc and ScalableCore system, respectively. Simulation speed is defined as a simulation step count proceeded in a unit time (The unit is KHz). In SimMc, with increase of the Node count of the simulated processor, the simulation speed decreases. While SimMc is not parallelized for readability and customizability of the simulator code, speed up by the parallelization will be small due to synchronization overheads for cycle-accuracy. Finally, SimMc achieves the speed of 90 KHz in 16-Nodes simulation with N-Queen. Then the speed decreases to 8.9 KHz in 100-Nodes simulation with N-Queen.

In contrast, ScalableCore system achieves scalable simulation speed with increase of the Node count in the simulated processor. It achieves constant simulation speed of 1142 KHz in any cases. In ScalableCore system, a clock cycle in the simulated processor is emulated by using multiple FPGA-side clock cycles. To give emulation result of 1 cycle in the target processor, it takes about 35 FPGA clock cycles for updating of target circuit state, memory emulation and synchronization of simulation states. The evaluation result shows that it does not depend on the application behaviors and the core count, thanks to local barrier synchronization and virtual cycle.

Figure 3.11 indicates the relative speed of ScalableCore system to SimMc. As shown in Figure

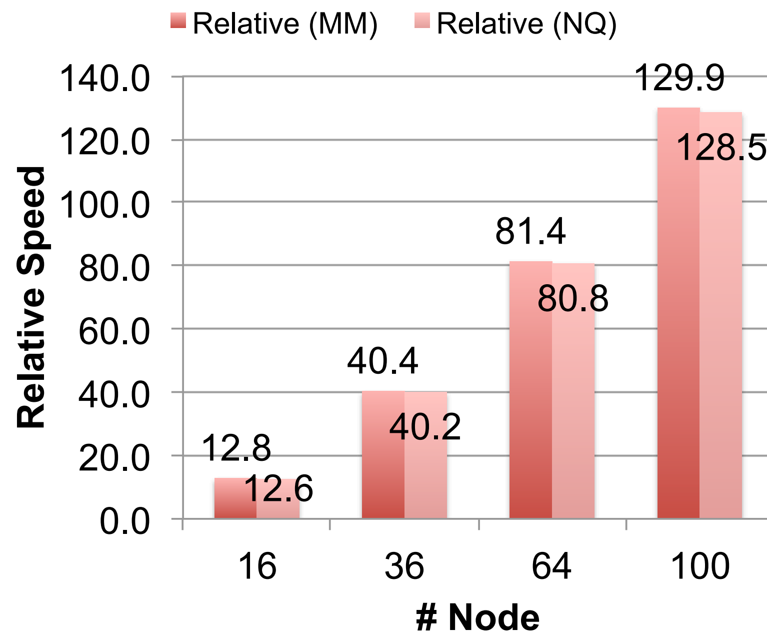


Figure 3.11 Relative Speed

3.10, increase of the core count of the simulated processor super-linearly decreases the simulation speed of SimMc. In contrast, simulation speeds of ScalableCore system are consistent. In 16 nodes, ScalableCore system runs at 12.6 times faster than SimMc. The relative speed to SimMc increases with the increase of the core count. In 100 nodes, the speed of ScalableCore system peaks at 129 times faster simulation speed of SimMc.

In respect of absolute simulation speed, ScalableCore system has an advantage over prior FPGA-based simulation-oriented prototyping systems. HAsim, an FPGA-based multicore simulator employing fine-grain time multiplexing for efficient resource usage, achieves up to 3.2 MHz at the maximum, 160 KHz at the minimum and 625 KHz on the average for simulation of 16 cores. While not accurate due to some differences in the architecture of the simulated processor, my evaluation result shows that ScalableCore system achieves excellent simulation rate on the average compared to the other systems.

In respect of cost performance, ScalableCore system achieves higher simulated frequency per price than software-based simulations on a standard computer. Price of each ScalableCore unit is about 8,000 yen. Since 100 of ScalableCore units are utilized to simulate a many-core processor with 100 cores, the total price of the FPGA system is 800,000 yen. Additionally, a standard computer is also required to manage ScalableCore system. The price of the computer is about 100,000 yen. The total price of ScalableCore system is 900,000 yen. Absolute simulation speed of simulating100 cores is 1142 KHz. Therefore the rate of simulated frequency per price is 1.4275 [Hz/Yen]. In contrast, to

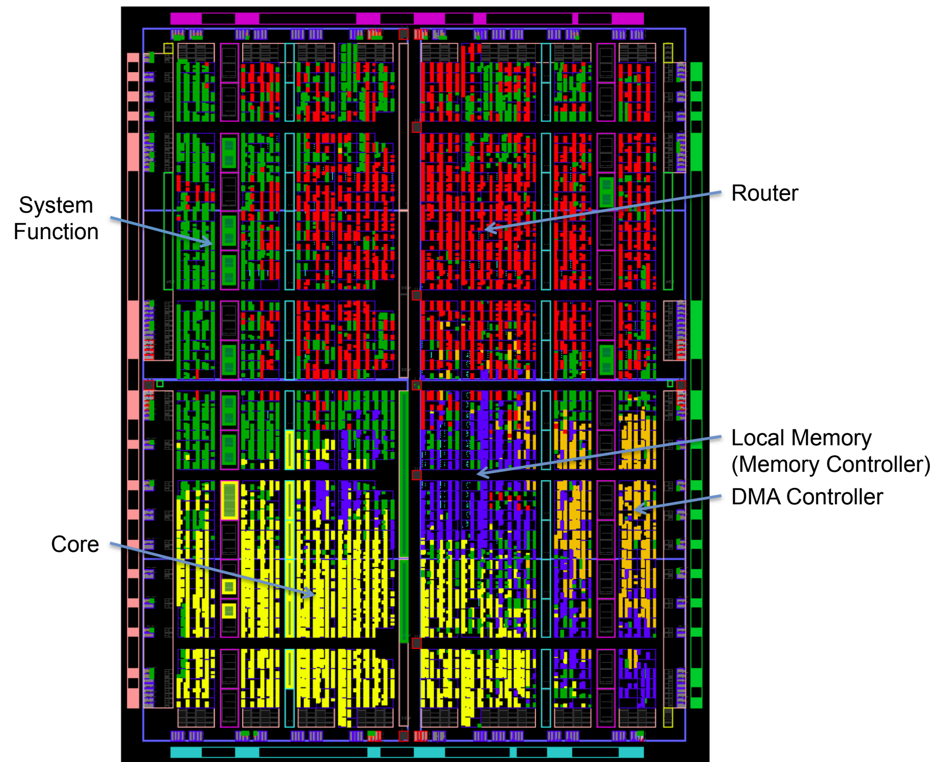


Figure 3.12 Floorplan of ScalableCore Unit on Spartan-6 XC6SLX16

Table 3.2 Resource Utilization

Module	LUT	Register	BRAM	LUTRAM	DSP
System Function	1700	2693	16	0	0
Core	1910	713	3	0	6
DMA Controller	444	378	0	0	0
Memory Controller	590	535	0	0	0
Router	2475	959	0	280	0
Target Total	5429	2585	3	280	6
Total	7129	5278	19	280	6
Percent Utilization	84%	29%	31%	Nan	6%

simulate the processor using SimMc on a standard computer, it requires only about 100,000 yen for the computer. However, absolute simulation speeds are relatively slow, which is 8.9 KHz. Therefore the rate of simulated frequency per price is 0.089 [Hz/Yen]. Finally, this comparison demonstrates that ScalableCore system achieves 16 times better cost performances.

3.5.3 Resource Utilization

Figure 3.12 depicts the actual floor plan on Spatan-6 XC6SLX16 FPGA of ScalableCore Unit with the simulated processor components (core, DMAC, router and local memory) and simulation functions after all synthesis steps (logic synthesis, map and place-and-route). The detailed breakdown of resource usage is listed in Table 3.2.

As the result, 84% of LUTs and 29% registers of Spartan-6 XC6SLX16 FPGA are utilized for the target implementation and system functions, respectively. Note that the resource usage of the local memory is comprised of the memory controller and the system functions, because the storage part of the local memory is mapped to the off-chip SRAM.

The target processor use 5429 LUTs and 2585 registers; LUTs are consumed than registers due to direct implementation of the target processor. An on-chip router consumes much more LUTs than the registers, because the router is a collection of a lot of multiplexers and decision logics for arbitrations.

The system functions for cycle-accurate simulations utilize 1700 LUTs and 2693 registers; Due to memory multiplexer and interface registers for separation of emulated components, the amount of consumed registers is much more than the amount of LUTs. Totally, 20% of LUTs and 15% registers of entire Spartan-6 XC6SLX16 FPGA are used for the system functions for simulation support. These resource overheads do not be too serious.

3.6 Discussion

This section provides several discussions compared with prior researches of software-based processor simulators and FPGA-based prototyping systems.

3.6.1 Comparison with Prior Software Simulators

As I mentioned above, software simulators have many fine-grain parallelisms corresponding to circuit components of the simulated processor. Parallelizations on modern multicore environments can enhance the simulation performance in some moderate degree. However, existence of much synchronizations to satisfy the cycle-accuracy of simulation results restricts the achievable simulation performance on parallel platforms. For instance, HORNET[23], I referred to in the background, supports cycle-accurate parallel simulations on the modern multicore SMP environments. A modern standard SMP environment consists of multiple multicore CPU dies on an identical mother board. The CPU dies are tightly connected via fast and high throughput inter-die links for each other. However, the typical communication latency across CPU dies is much bigger than that of on-die inter-core communications using cache coherent messages. Such many synchronizations with short messages dramatically decrease the

Table 3.3 Simulation Speed Comparison

Name	FPGA	Processor Configuration	Simulation Speed
Direct Implementation			
ScalableCore system	100 × Xilinx Spartan 6 LX16	100-core	1.1 MHz
BeeFarm[52]	Xilinx Virte 5 LX115T	8-core	25 MHz
Heracles[50]	Virtex 6 LX550T	36-core	100 MHz
Simulation-Oriented Implementation			
RAMP Gold[76]	Xilinx Virtex 6 LX240T	64-core	50 MIPS (Up to 800 KHz)
HAsim[77]	Xilinx Virtex 5 LX330T	16-core	625 KHz
HAsim with LEAP[87]	2 × Xilinx Virtex 5 LX330T	128-core	3 MIPS (Up to 23 KHz)
Arete[6]	4 × Xilinx Virtex 5 LX110T	8-core	55 MIPS (Up to 6.87 MHz)

simulation performance.

If each core on a multicore processor executes a larger simulation engine with numerous instances of simulated cores, the impact of inter-die synchronization can be reduced. It will improve just the scalability to the core count. However, absolute simulation speed of software simulators, even if just a single core is simulated, is much slower than that of FPGA-based simulators. Employing such scheme will make the absolute simulation speed much slower. Like ZSim[24], I mentioned above, relaxing the simulation accuracy is effective to reduce the impact of synchronization overheads to the simulation speed. However, it also reduces the simulation accuracy.

As an innovative approach, many-core processor simulations can be performed on an existing many-core processors, such as Intel Xeon Phi. Employing modern many-core accelerators with many small independent cores on a single die will reduce synchronization overheads among the multiple threads independently running on each core. It improves the scalability of simulation speed. Unfortunately the absolute simulation speed will be slow, due to their powerless thin cores.

In contrast, ScalableCore system provides a low latency communication module for fast inter-FPGA synchornizations by the pure hardware implementation. ScalableCore system employs a specialized structure for tile architecture simulations, so that it partitions and maps a target tile architecture in a natural way. Even if the core count of a simulated processor increases, the synchronization overheads do not increase, because synchornizations for each direction on each FPGA can use independent inter-FPGA communication channel for each. Therefore, when a tile architecture is simulated, ScalableCore system can provide constant simulation speeds with keeping the cycle-accuracy.

3.6.2 Comparison with Prior FPGA-based Prototyping Systems

The strategy of ScalableCore system is much similar to prior FPGA-based prototyping systems of the direct implementation approach, such as BeeFarm[52] and Heracles[50], than the simulation-oriented

systems, such as RAMP Gold[76] and HAsim[77]. The reported simulation speed of each FPGA-based simulation platform is listed in Table 3.3.

BeeFarm[52] and Heracles[50] are FPGA-based simulators on the direct implementation approach using a single FPGA. Each of them achieves faster simulation speed, 25 MHz of BeeFarm and 100 MHz of Heracles, respectively, than ScalableCore system. The reason is that each of them employs a very large FPGA environment without any consideration for multi-FPGA environments. Therefore they require a very long synthesis time for bitstream generations. In contrast, absolute simulation speed of ScalableCore system is not faster than them. However, the method of ScalableCore system can easily expand the simulatino system without additional bitstream synthesis steps. In addition to the advantage of circuit synthesis, since the method of ScalableCore system is originally designed for multi-FPGA environment, the system achieves scalable simulation speeds against the increasing core count.

RAMP Gold[76], HAsim[77] and Arete[6] employ the simulation-oriented approach for area-efficient simulation engine implementation for multiple identical cores. They can achieve good simulation performance with simulation-specific RTL designs with a separation of functional model and timing model. Therefore the RTL design structures for LSI implementation and simulation purpose are much different. Unfortunately, increasing the core count of a simulated processor decreases the simulation speed represented as a frequency value. Actually, ScalableCore system also requires certain modification of RTL designs. However, the most structures of RTL designs can be kept, because ScalableCore system requires only inserting interface registers into boundary of modules and inserting throttle signals for cycle-accurate register updates. ScalableCore system has an advantage to simplicity of RTL designs for simulation.

While the current implementation of ScalableCore system is for a DMA-based architecture, ScalableCore system can model modern cache-based architectures, if the architecture employs an approach of tile architecture. Since those architectures have RAM components, as well as M-Core, and cache controllers, instead of DMA controllers in M-Core, those architectures can be implemented in the same way.

Comparison with Prior Multi-FPGA based Platforms

As a popular multi-FPGA platform of many small FPGAs, CUBE[88] is a systolic array system of 512 Spartan-3 FPGAs. The system consists of 8 baseboards that implements 64 FPGAs. The difference with ScalableCore system is that the interconnection of inter-FPGAs is a unidirectional parallel bus, and that each FPGA has no corresponding off-chip memory such like SRAM on ScalableCore Unit.

Formic[55] employs more similar a FPGA system architecture to ScalableCore system. Actually Formic system simulates a multiprocessor system of 512 Microblaze soft processors. I think the FPGA platform of Formic is also useful to simulate a many-core processor by employing the same implemen-

tation scheme as well as ScalableCore system.

Additionally, there are various prior researches of multi-FPGA based accelerators. BEE[89] is a popular multi-FPGA platform used in various researches. BEE employs multiple large-scale high end FPGAs with costly interconnections among FPGAs, in contrast to ScalableCore system. Sano et al. proposed several multi-FPGA based platform and efficient computation techniques under the limited memory bandwidth for high performance CFD (computational fluid dynamics)[90, 91, 92]. FLOPS-2D[93] is also a multi-FPGA platform for CFD with optimized deep pipelines. Yokota et al. proposed a multi-FPGA based real-time medical diagnosis system with good scalability[94]. Lin et al. proposed a vocabulary speech recognizer on dual FPGA platform[95]. Kapre et al. proposed a VLIW-based accelerator for SPICE model execution on multiple FPGAs[96].

Multi-FPGA platforms are hopeful to accelerate certain applications, not limited to processor simulation. ScalableCore system is useful not only for fast processor simulations but also for high performance computing. We developed a scalable stencil computation accelerator using ScalableCore system as a computing platform[97]. The implementation result shows that the ideology of ScalableCore system that partitions an application into multiple parts works very well for HPC applications.

3.7 Case Study 1: Evaluation of Task Allocation Method for Multiple Parallel Applications

In many-core processors, task allocation (thread allocation) is one of distinct factors to affect application performance and whole processor throughput[98, 99]. However, to determine a task mapping for better performance on the many-core processor is not easy due to its large exploration field. As a novel lightweight task allocation method for manycores, RMAP[100, 101], has been proposed.

As a case study, I evaluated the impact of RMAP method to M-Core architecture by using ScalableCore system and larger applications that do not suit for slow software-based simulation. I used a test bed of ScalableCore system with 100 nodes as a simulation accelerator of M-Core architecture. The simulation takes about **20 minutes** for all test applications. If this evaluation is fully completed in software simulations, it takes about **43 hours**.

Note that the all inclusive evaluation time of FPGA bitstream syntheses and simulations is lower than that of software simulations. In this case, time to synthesize the bitstream for all FPGA units is about **15 minutes**. Additionally ScalableCore system requires a time to write the bitstream to every FPGA unit. It takes about **15 minutes**. Therefore the total time of the synthesis and the simulation is about **50 minutes**. In a situation that a lot of simulations are required, FPGA-based prototyping systems are very effective.

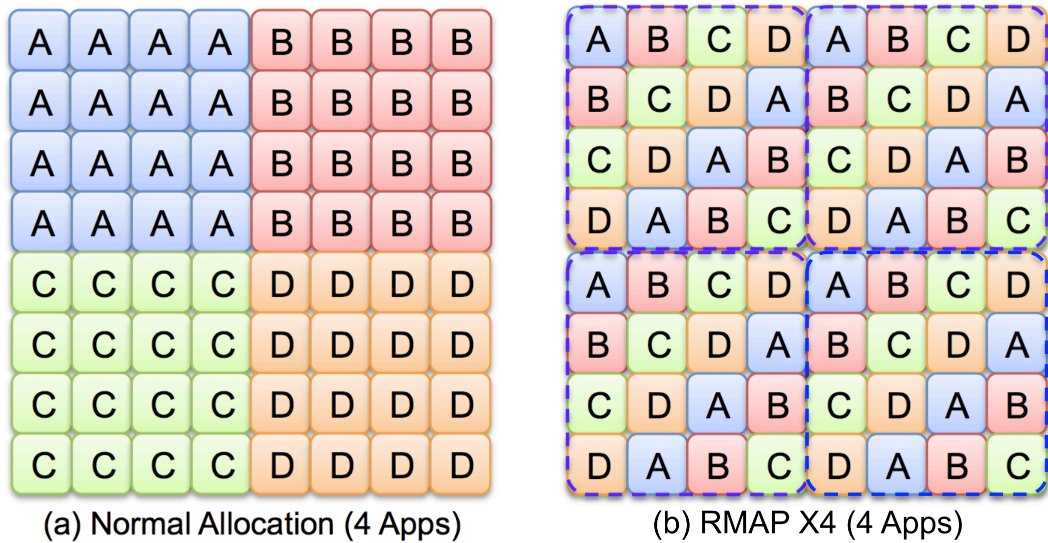


Figure 3.13 Task Allocation of 4 Parallel Applications

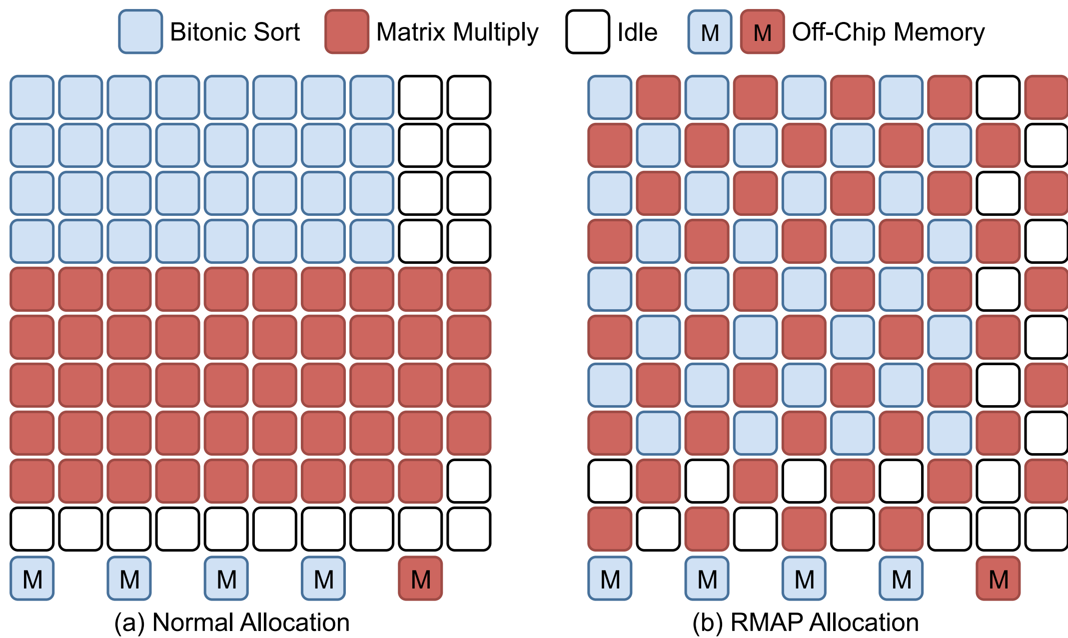
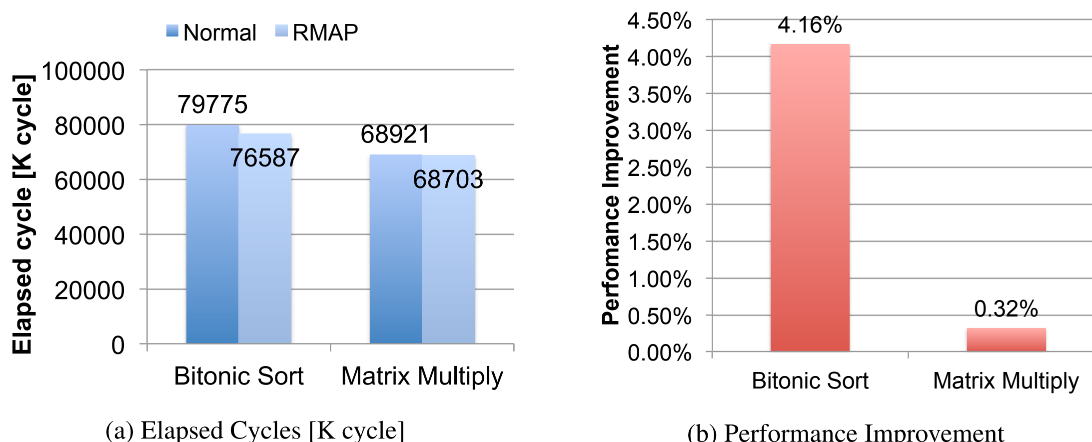


Figure 3.14 Task Allocation of 2 Parallel Ppplications (for This Case Study)

3.7.1 RMAP: Contention-Aware Task Allocation for Manycores

Parallel applications have some fragments of network traffic along with application behavior; at one phase, the application has low network usage, but at other phase, it has high network usage. Key idea of RMAP to improve the performance is to reduce self-contention of network (additional network con-



(a) Elapsed Cycles [K cycle]

(b) Performance Improvement

Figure 3.15 Performance comparison of RMAP allocation and the normal allocation

tention by communications traffic of the own application) by allocating together with other applications with different network behaviors.

Figure 3.13 shows a normal task allocation and a RMAP allocation for 4 applications. Figure 3.14 shows each allocation with 2 applications for this case study. I used two parallel applications: bitonic sort and matrix multiply. Bitonic sort is network-intensive workload, heavier than matrix multiply. In normal task allocation, threads of each application are mapped separately to each group of Nodes. In contrast, in RMAP task allocation, threads of two applications are mixed and mapped. Location of each application forms one solution of N-Rook problem. When one application consumes network bandwidth a lot and the other does not, the bi-section bandwidth for the network-consuming application is temporally increased by RMAP mapping.

3.7.2 Evaluation

I evaluated two task allocation strategies by using ScalableCore system. The configuration of the simulated M-Core processor is same as the previous evaluation, which is listed in Table 3.1.

Figure 3.15 shows the application performances of each application in two task allocations and the performance improvement by RMAP. The performance of bitonic sort is improved at 4% by RMAP because the serious self-contention of bitonic sort is eliminated in RMAP allocation. In contrast, the performance of matrix multiply is improved by 0.32% because matrix multiply does not originally have such serious self-contention.

3.7.3 Summary of Case Study 1

In this case study, I evaluated the task allocation method for multiple parallel applications on many-core processors. The evaluation result shows that the task allocation strategy for reducing the communica-

tion contention within each application improves the total performance of the processor. By utilising the ScalableCore system, simulation time is dramatically reduced from 43 hours to 20 minutes.

3.8 Case Study 2: Evaluation of Task Allocation Method for NoC-based DMR Execution Manycores

In order to develop a functional computing system, dependability of a microprocessor is a key issue. However, transistor scaling increases the soft-error rate of microprocessors. It is one of the critical problems to decrease the dependability.

I have proposed SmartCore system[102, 103], NoC-based DMR (Dual Modular Redundant) execution mechanism on many-core processors. Key feature of SmartCore system is to employ the inherent redundancy of PEs (Processing Element) constructing a DMR pair executing the identical thread by using multifunction routers.

In order to detect errors on PEs, a DMR pair of independent PEs executes a single identical thread. Eventually PEs of a DMR pair generates some packets to the network. Then all packets going out from the DMR couple are verified at the multifunction router.

In this case study, I used ScalableCore system to explore an effective task allocation strategy with good performance and dependability for NoC-based DMR many-core processors.

3.8.1 SmartCore System: An NoC-based DMR Execution Mechanism for Manycores

Overview

I provide an overview of the architecture of SmartCore system, a DMR execution mechanism with multifunction on-chip routers' support. Figure 3.16 shows a simple example of DMR execution on SmartCore system. Now we assume that a many-core architecture with multiple PEs (processing element) and a conventional 2D-mesh on-chip network. Each PE in the processor is attached to the others by the on-chip network. In the figure, a circle with R is a multifunction router.

The purpose of SmartCore system is to detect any soft-error on the PEs by simultaneously executing an identical thread on the different PEs. If any soft-errors are occurred on the PE, the contents of packets from the PE may be changed by the errors. SmartCore system detects these soft-errors by comparing the packets from two different PEs executing an identical thread.

In the figure, **Master A** and **Slave A** work as a pair of DMR execution, which executes identical program thread. **Master B** and **Slave B** also work as a pair of DMR execution. Master A and Master B are proper PEs executing a program, and Slave A and Slave B are additional PEs for DMR executions. In this paper, we refer to these proper PE as **Master Node**, and also refer to these additional PE as **Slave Node**.

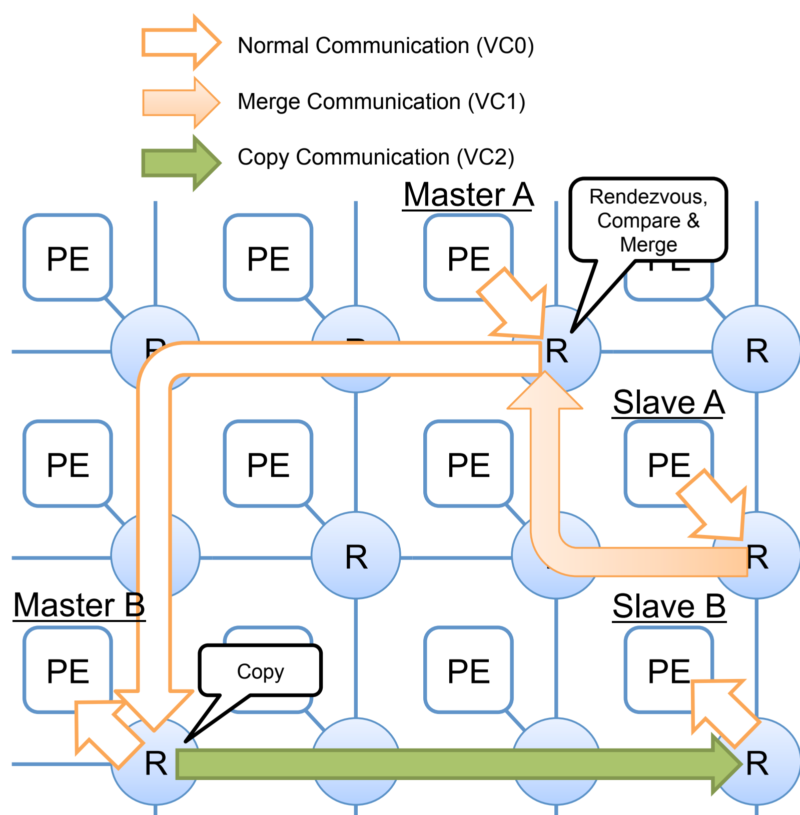


Figure 3.16 SmartCore System: NoC-based DMR Execution Mechanism for Many-core Processors

Now we consider that Master A and Slave A generate packets to the other PE (Master B). In order to verify the contents of the packet, the packet generated on Slave A is forwarded to the router of Master A, is not traversed to the original destination. The router on Master A waits the forwarded packet from Slave A. After the forwarding packet arrives at the router on Master A, the forwarded packet and the original packet generated on Master A are checked by comparing each flit of these packets in sequential order. When these packets are verified correctly by the comparison, these two packets are merged and forwarded to the original destination. If the verification result shows that the there are any mismatches of flits, the SmartCore system takes some adequate recovery processes, such as stopping the corresponding threads and re-executions.

When the merged packet arrives at the router of the destination, the packet is copied into two packets that a packet destined to the PE of the original destination and a packet destined to the Mirror Node of the destination. The pair of DMR execution must receive the same packet sequence to continue the program execution with the same control flow with the same data.

On-chip communications in DMR execution mode of SmartCore system are classified into 3 parts: (1) Normal communication from a Master Node (Master-A) to another Master Node (Master-B), (2)

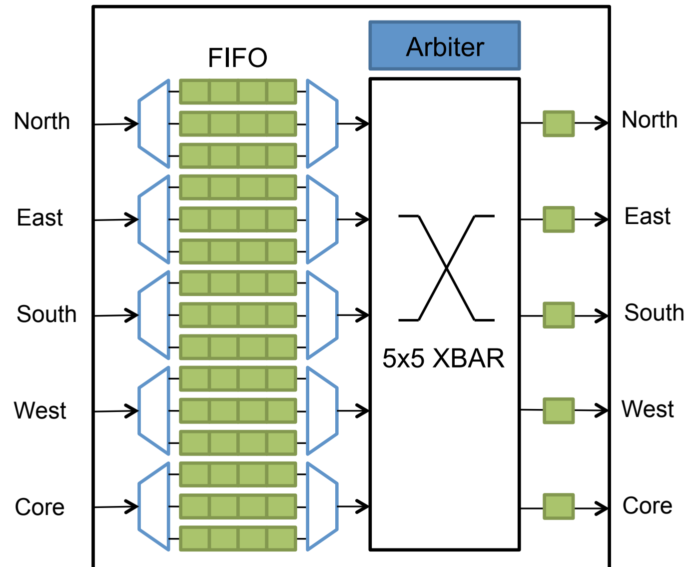


Figure 3.17 Standard On-chip Router Architecture

Merge communication from a Slave Node (Slave-A) to a Master Node of the pair (Master-A) and (3) Copy communication from a Master Node (Master-B) to a Slave Node of the pair (Slave-B). Without DMR executions, the merge communication and the copy communication are not used.

SmartCore system does not restrict the locations of DMR pairs so that the programmer or system software can allocate the pair for any places in the processor. In some allocations, irregular turns that are not used in the original X-Y dimension order routing are used to forward the packets in merge communication and copy communication. These irregular turns may incur some deadlocks of network. In order to avoid deadlocks due to these irregular turns, a different virtual channel is used for each kind of communication.

With DMR executions, amount of packets is increased due to the copy communication and the merge communication. It increases the packet contentions on the network. Additionally every packet from a PE on a Master Nodes has to wait for the corresponding packet from its Slave Node to verify the contents. It increases the traversal latency from the source PE to the destination PE.

DMR On-chip Router

Figure 3.18 shows the microarchitecture of multifunction router. Figure 3.17 shows the microarchitecture of standard on-chip router for comparison. Multifunction router has 3 special functions to support DMR executions: (1) Packet coping; (2) Packet rendezvousing; and (3) Packet comparison and merging. Hence multifunction router architecture includes some unusual hardware components that are shown in gray.

In order to sustain the same control flow between the DMR pair, a packet arrived at the router on the

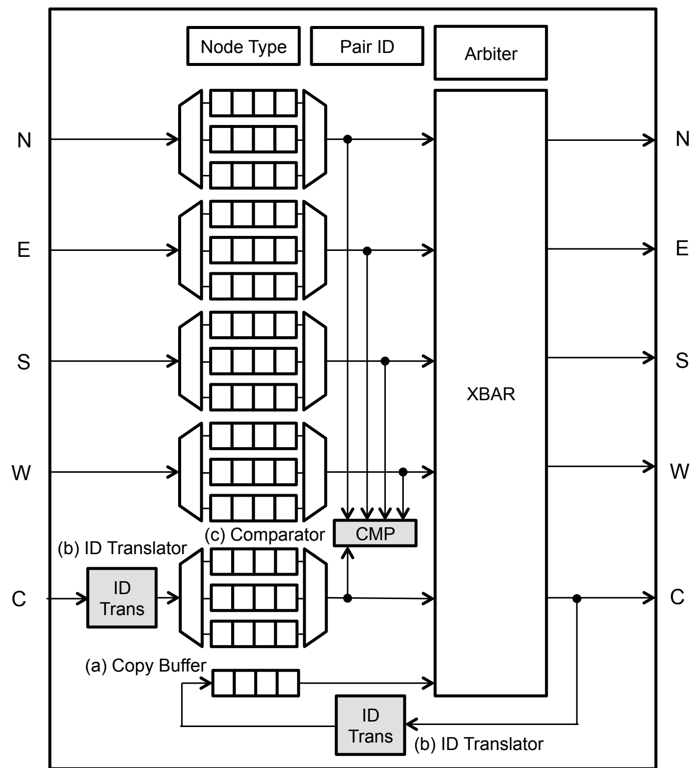


Figure 3.18 DMR On-chip Router Architecture

master node is copied and forwarded to its slave node. When a packet going to the PE of the master node from its router, the packet is copied and inserted into copy buffer ((a) in Figure 3.18). At this time, ID translator for the copy buffer ((b2) in Figure 3.18) changes the destination into its slave node, and it also changes the virtual channel number of packets from the normal communication channel to copy communication channel. By employing a copy buffer, an extended crossbar of 6-input 5-output is used.

A packet is checked by rendezvous and comparison of the packet contents. When a packet inserted from the PE on the master node, the packet waits for the corresponding packet from its slave node in the input buffer. At the same time, a packet inserted by the PE on the slave node to the router forwarded to its master node by changing the destination by ID translator for input buffer ((b1) in Figure 3.18). Then the packet traverses as same as a normal packet to the next router. The forwarded packet to the Slave node eventually arrives at the router of the Master node, a flit of the packet is compared with a flit of the packet from the PE of the Master node in sequential order by Comparator ((c) in Figure 3.18). If any comparison mismatching of packet are found, some adequate processes for recovery from the errors are done.

Multifunction router requires additional resource for two special-purpose virtual channels and packet verification. These virtual channels use 2 input queues for each external direction, and packet verifi-

Figure 3.19 Task Allocation (Separate, Master-Slave Distance = 1)

cation uses 1 input queue for the copy buffer. DMR execution with multifunction routers uses 9 input queues and some control logics for each router.

3.8.2 Task Allocation for DMR Executions

In many-core processors, task allocation pattern affects each application performance and whole the processor throughput, as I explained above. Since DMR execution increases on-chip traffics to verify the execution status in SmartCore system, task allocation problem is ever more complex.

A simple task allocation strategy to avoid the slow down due to DMR executions is that two tasks of each DMR pair are put side by side. In contrast, in order to increase the dependability, the nodes of each DMR pair should be separately allocated so that it avoids falling in a failure mode from the same causes, such as voltage variation, GND noise and radiations.

In this case study, I tested several allocation patterns to explore the effective allocation strategy to achieve both superior performance and dependability. Now we assume a situation that multiple parallel applications are executed on an identical many-core processor. In this experiment, the number of threads of each application is 16, and each application performs in DMR mode. Totally 64 threads are executed in a single processor all together.

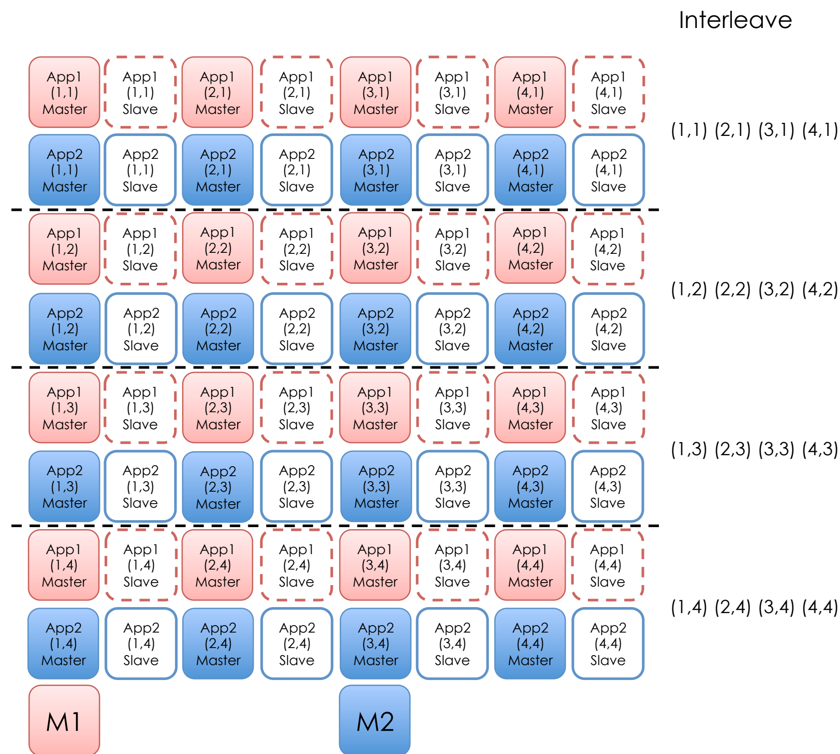


Figure 3.20 Task Allocation (Interleave, Master-Slave Distance = 1)

Side-by-Side Allocation

First, I show 3 patterns of the side-by-side allocation in Figure 3.19, Figure 3.20 and Figure 3.21, respectively. All the distances of master-slave pairs in these allocations are 1. Note that **M1** and **M2** in the figures represent off-chip memories for application 1 and application 2, respectively. On the allocation in Figure 3.19 (separate), applicatoin 1 and application 2 are separated for each other. On the allocation in Figure 3.20 (interleave), applicatoin 1 and application 2 are separated and interleaved on the Y-axis. On the allocation in Figure 3.21 (RMAP), applicatoin 1 and application 2 are still interleaved based on RMAP task allocation which I explained the previous case study. Since all the master-slave pairs are allocated side-by-side, the impact of additional communications of DMR executions to the performance might be small.

Separated Allocation

Then, I show 4 patterns that master-slave pairs are separated in Figure 3.22, Figure 3.23, Figure 3.24 and Figure 3.25, respectively. On the allocation in Figure 3.22 (block), all master nodes and all slave nodes are packed, respectively. On the allocation in Figure 3.23 (interleave ROT), the positions of all slave nodes in Figure 3.20 are exchanged in Y-axis. On the allocation in Figure 3.24 (interleave

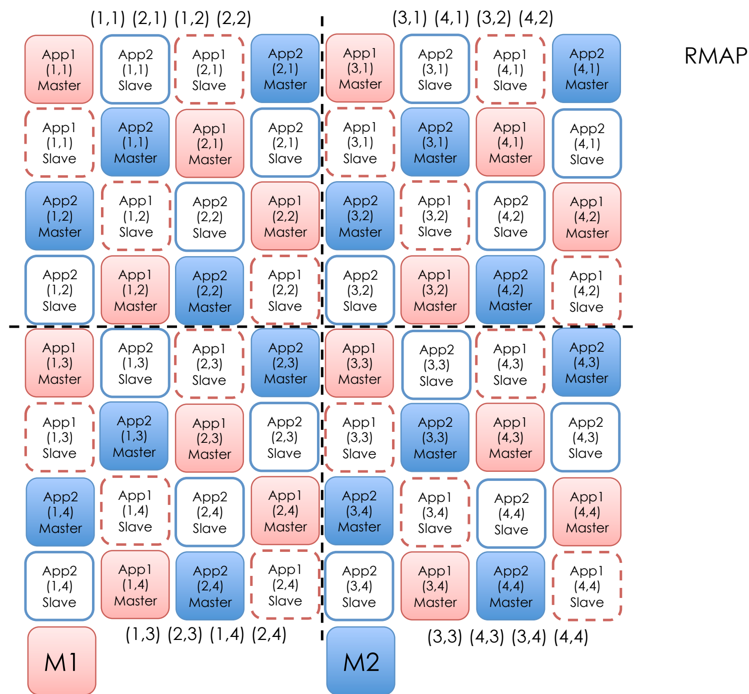


Figure 3.21 Task Allocation (RMAP, Master-Slave Distance = 1)

ROTx2), the positions of all slave nodes in Figure 3.23 are exchanged in X-axis. On the allocation in Figure 3.25 (RMAP ROT), the positions of all slave nodes in Figure 3.21 are exchanged for each block.

The average distances between a mater node and a slave node are 4 in Figure 3.22, 5 in Figure 3.23, 8 in Figure 3.24 and Figure 3.25, respectively. Since the distances are greater than the previous ones, the impact of additional communications of DMR execution to the performance will be increased

3.8.3 Evaluation

Setup

I evaluated 7 allocation patterns depicted above by using ScalableCore system. I implemented the modified M-Core architecture with multifunction on-chip routers for DMR execution on the test bed of ScalableCore system. The configuration of the simulated processor is listed in Table 3.4. 2 applications (16 threads for each application) are put together based on each task allocation I mentioned, and are executed on DMR mode of SmartCore system. I observed the impact of task allocation to the execution time of each application. The two applications are executed repeatedly so that the simulation length is 350 M cycles. I used two benchmarks for the evaluation: (1) bitonic sort for application 1 and (2) matrix multiply using Cannon's algorithm for application 2. The detailed configuration of these applications is listed as below.

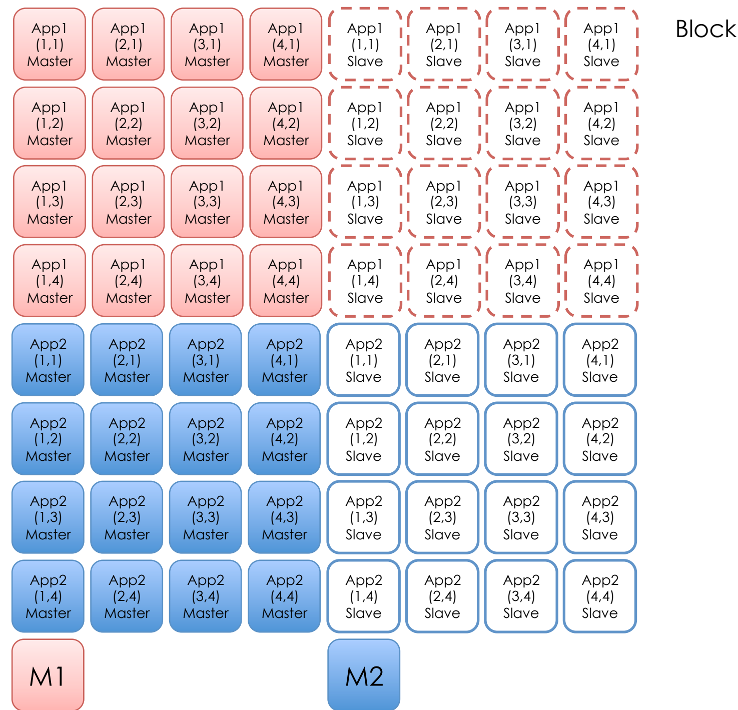


Figure 3.22 Task Allocation (Block, Master-Slave Distance = 4)

Table 3.4 Evaluation Setup

Core	MIPS32 ISA, 5-stage, Single-issue,2-memory-port (Fetch, Load/Store)
DMA controller	2-memory-port (32-bit DMA Read and DMA Write)
Network topology	2-D mesh
On-chip router	5-input/output (North, East, West, South, Core), 4-stage (NRC and VA: Next Route Computation and Virtual Channel Allocation, SA: Switch Allocation, ST: Switch Traversal, LT: Link Traversal), 2-virtual-channel, FIFO depth: 4, Credit-base flow control, X-Y Dimension Order Routing
Node memory	512KB (per Node), 32-bit width, access latency: 1 4-port (Fetch, Load/Store, DMA Read, DMA Write)
# Nodes	64 (8×8)
# DRAM Controllers	2 (Location (X,Y): App1 (1,9), App2 (5,9))

Bitonic Sort: Parallel sort[104]. The data size is 1 M entries (4MB). Executed 3 times repeatedly in the period of 350 M cycles.

Matrix Multiply: Parallel matrix multiplication using Cannon's algorithm[105]. The data size is 262144 entries (512×512, 1MB). Executed 4 times repeatedly in the period of 350 M cycles.

Figure 3.23 Task Allocation (Interleave ROT, Master-Slave Distance = 5)

Note that the simulation time of 350 M cycles for each configuration using ScalableCore system for this evaluation is about 6 minutes. Compared to the corresponding software simulator, the simulation speed is 80 times faster.

Baseline Performance without DMR Execution

First, I evaluated the baseline performance impact of the task allocations for each application without DMR execution. Instead of DMR execution, an idle thread executing nothing is allocated for every position of a slave node for each allocation pattern. Figure 3.26 shows the performance improvement from the performance on the separate allocation.

In bitonic sort, the performance on the interleave allocation is 0.1% worse than the separate allocation, due to the increase of communication latency. The performance on the block allocation is same as the separate allocation. By using RMAP allocation, the performance is improved 1.02% from the separate allocation, due to the communication contention reduction.

In contrast, performances for all allocations are worse than the separate allocation in matrix multiply. Particularly, the performance of RMAP is 0.96% worse than the separate allocation, because the amount of communication contention on matrix multiplication is small. Additionally the increase of the number of hops between a source node to a destination node decreases the performance.

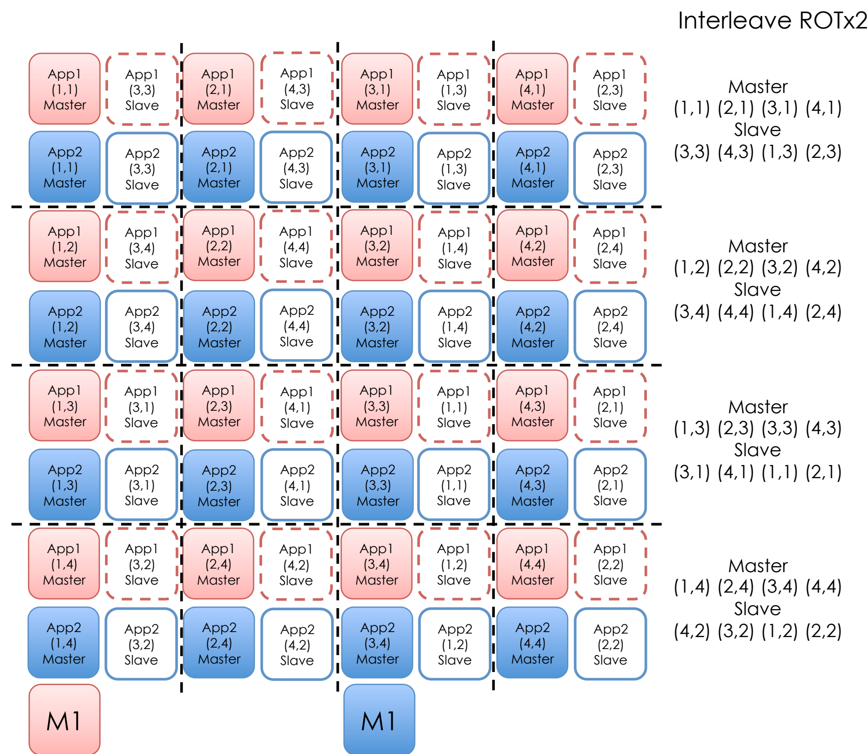


Figure 3.24 Task Allocation (Interleave ROTx2, Master-Slave Distance = 8)

Performance with DMR Execution

Figure 3.27 and Figure 3.28 show the performance improvements from baseline performances of the separate allocation without DMR executions in each task allocation.

In the figures, the x-axes represent the average distance between a master node and a slave node of a DMR pair. The y-axes represent the performance improvement from the separate allocation without DMR execution.

In bitonic sort, performances in any cases excepting the RMAP allocation (Figure 3.21) are worse, 0.83% in the separate allocation and 0.79% in the interleave allocation, respectively, than the baseline performance, due to the additional communications for DMR executions. In the RMAP allocation, since the amount of communication contentions within each application is reduced by the task allocation effect, the performance is improved 0.41% in spite of DMR execution overheads.

In the allocations with a long distance between the master and the slave, there is a huge variability among the allocations. In the block allocation (Figure 3.22), the interleave ROT allocation (Figure 3.23) and the interleave ROTx2 allocation (Figure 3.24), the performance degradations are 4.28%, 3.88% and 4.83%, respectively. The reason why the degradation of the interleave ROT is less than the block allocation is that the bi-section on-chip bandwidth of the interleave ROT allocation is higher than

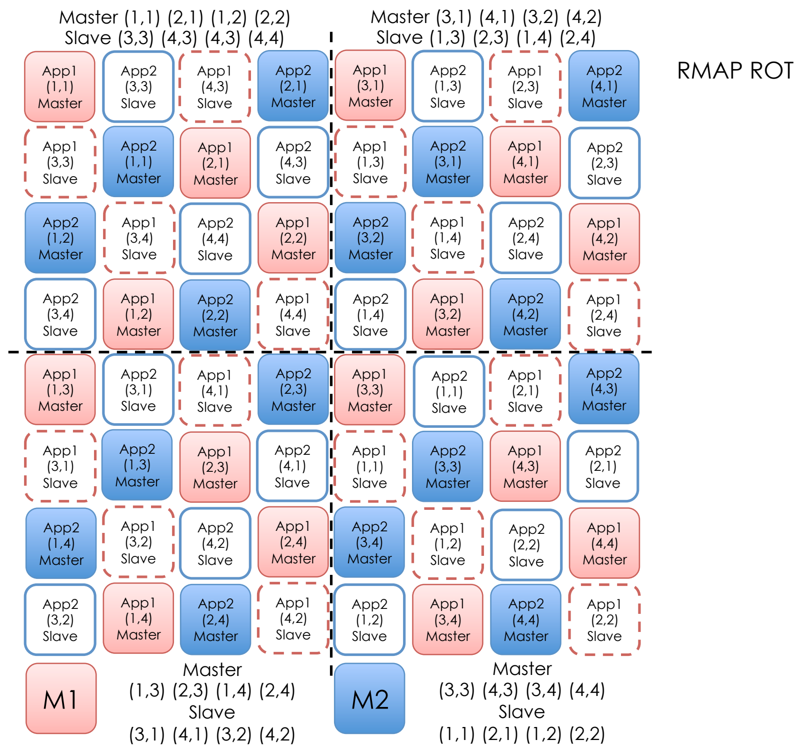
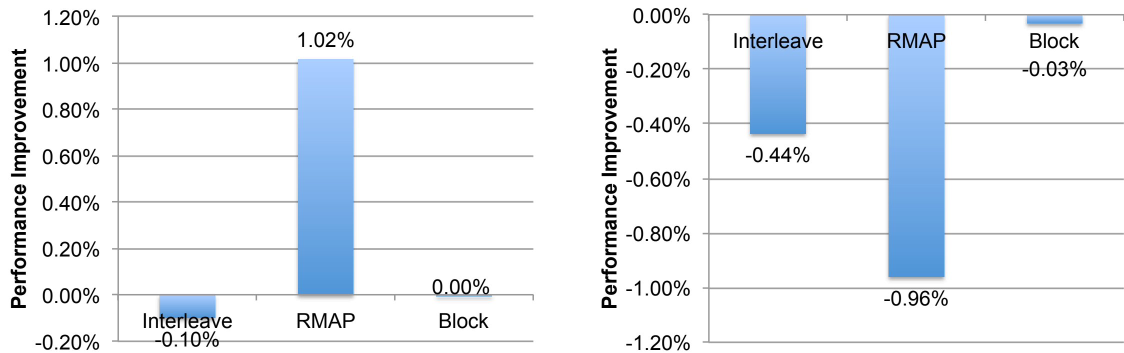


Figure 3.25 Task Allocation (RMAP ROT, Master-Slave Distance = 8)



(a) Bitonic Sort (b) Matrix Multiply

Figure 3.26 Performance (without DMR mode)

the block allocation.

In contrast, the RMAP ROT allocation (Figure 3.25) achieves good performance that the performance degradation is 1.75% in spite of the same master-slave distance as the interleave ROTx2 allocation. While the RMAP ROT allocation is an alternative allocation of the original RMAP allocation so as to increase the master-slave distance, RMAP ROT still has a few traffic contentions. It improves the

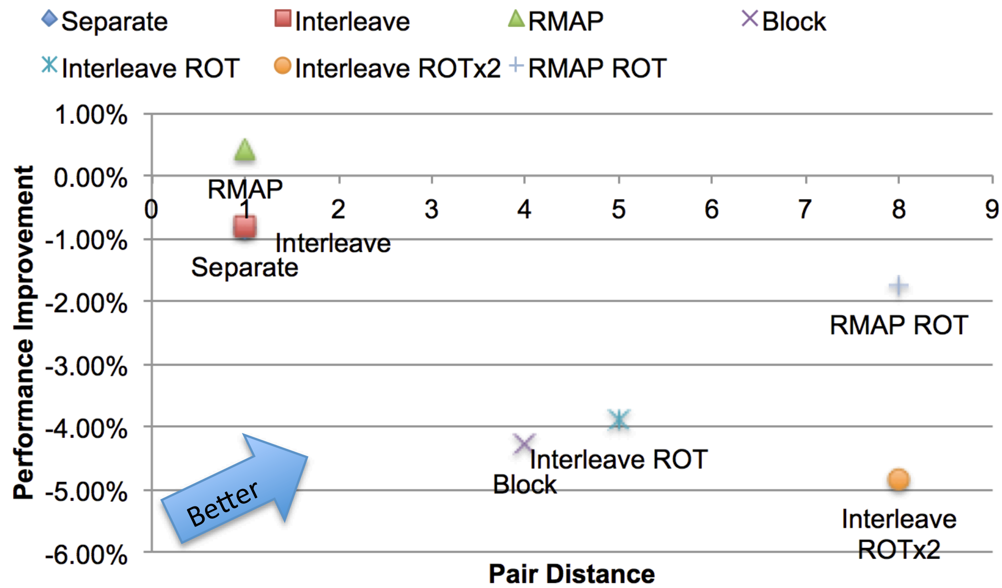


Figure 3.27 Relationship between Master-Slave Distance and Task Allocation Pattern (Bitonic Sort)

performance.

In matrix multiplication, performances in all cases are worse than the baseline. In the allocations with small master-slave distances, the performance degradations are 0.19% in the separate allocation, 0.69% in the interleave allocation and 0.44% in the RMAP allocation, respectively.

In the allocations with long master-slave distance, the performance degradations are large, 3.33% in the interleave ROT allocation, 4.09% in the interleave ROTx2 allocation and 4.17% in the RMAP ROT allocation, respectively. However, the performance degradation of the block allocation is not high, 0.76%.

According to the previous evaluation results of non-DMR executions, the RMAP allocation is not good for matrix multiplication. Otherwise the block allocation works very well. Even if DMR execution is enabled, it does not increase serious traffic contentions in matrix multiplication with the block allocation. In contrast, the performance of bitonic sort is improved by the RMAP allocation and the RMAP ROT allocation, because the amount of traffic contentions on bitonic sort is reduced by the RMAP allocation also on DMR executions.

In conclusion, I found that task allocation strategy should be selected founded on the amount of traffic contentions on the original application without DMR mode. For an application with many traffic contentions, an allocation with lesser traffic contentions is effective. In contrast, for an application with less traffic contentions, an allocation with short master-slave distances of DMR pairs is effective.

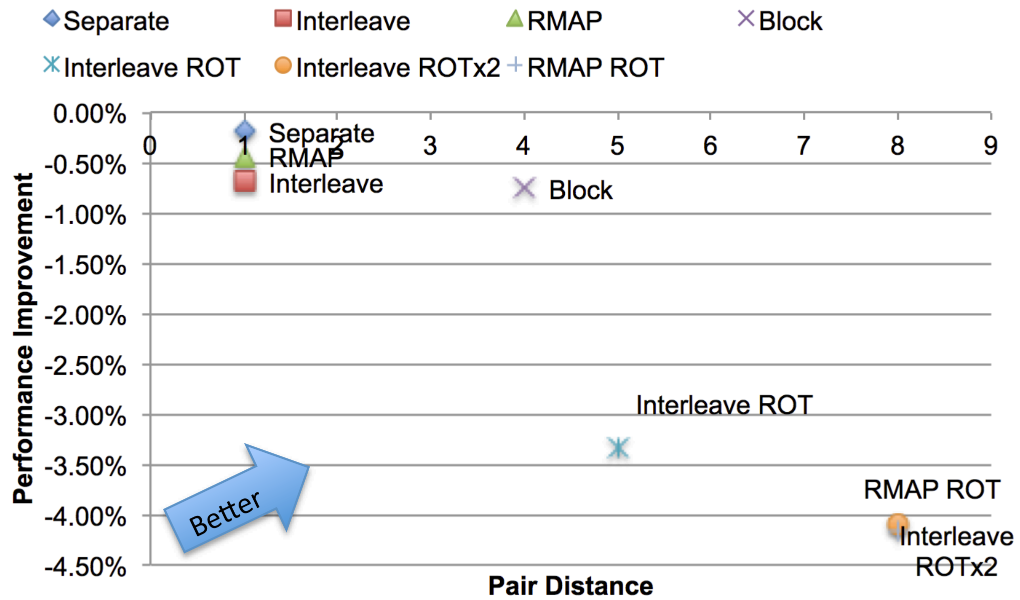


Figure 3.28 Relationship between Master-Slave Distance and Task Allocation Pattern (Matrix Multiply)

3.8.4 Summary of Case Study 2

In this case study, I evaluated various task allocation patterns to explore an effective allocation strategy for DMR execution mode on many-core processors. From the evaluation results, the effective allocation strategy depends on the amount of traffic contentions on the original application without DMR mode. By using ScalableCore system, this evaluation time is reduced to one-eightieth of that of the software simulator.

3.9 Summary

In this chapter, I introduced ScalableCore system, the acceleration method of cycle-accurate processor simulations on multi-FPGA based platforms. ScalableCore system aims to counter scalability issues of the simulation speed and the synthesis time.

I described the overall concept and architecture of ScalableCore system, which realizes the scalable simulation speed with keeping the cycle-accuracy. Then I described the test bed implementation features for viability evaluations of the proposed method.

I presented the evaluation results using the test bed system. I discussed the achieved performance of ScalableCore system, compared with a corresponding software-based simulator and prior FPGA-based prototyping platforms. The evaluation result shows that the system achieves good scalability in

simulation speed. The simulation speed of a many-core processor with 100 nodes on the developed test bed is 129 times faster than the corresponding software simulator.

Finally I introduced two case studies of many-core platform evaluations using the test bed system. As the first case study using ScalableCore system, I showed the evaluation results of task allocation method on ScalableCore system. As the second case study, I explained research on task allocation strategy on dependable many-core processor with multifunction NoC support. In both cases, the simulation times are dramatically reduced by using ScalableCore system.

Chapter 4

flipSyrup: An FPGA-based Flexible Prototyping Methodology

In this chapter, I introduce the design methodology under the resource abstraction of FPGA platforms, in order to reduce the development complexity of FPGA-based processor simulators. To counter the absence of appropriate abstractions for portable FPGA-based rapid prototyping, I propose *flipSyrup* (flexible interface for prototyping system development on reconfigurable platforms), an RTL modeling methodology and tool-chain with appropriate resource abstractions on FPGA platforms.

I introduce the overview of the proposed design methodology that mitigates the critical gap between ideal LSI-oriented RTL designs and FPGA-prototyping-oriented RTL designs. In order to prevent FPGA-specific implementations of simulated processors, the design methodology offers two abstract objects corresponding to memory resources of FPGA and inter-FPGA communications on multi-FPGA platforms.

To this end, first I developed Pyverilog, an open-source toolkit for design analysis and code generation of RTL designs written in Verilog HDL. Pyverilog offers (1) code parser, (2) dataflow analyzer, (3) control-flow analyzer, (4) visualizer and (5) code generator for Verilog HDL.

Then I developed the Python-based design tool-chain that automatically synthesizes ready-to-implement RTL designs for actual FPGA platforms from target RTL descriptions under the abstraction. This methodology enables designers to model a prototyping target processor without concern for actual platform resources.

I evaluated simulation speed under the abstraction using a standard FPGA platform with large capacity of logic and memory. The evaluation result shows that the simulation speed degradation under the abstraction is not critical so that the abstraction tool-chain offers the helpful support to develop a high-speed processor simulator rapidly.

Finally, I present the evaluation result of the proposed methodology on ScalableCore system. The evaluation result shows that the proposed design methodology works well for multi-FPGA based platforms. The simulation system synthesized automatically by the flipSyrup tool-chain archives almost

equivalent performance to manual-tuned ScalableCore system. The integrated framework aggressively improves the prototyping efficiency for emerging many-core processors by providing the sufficient simulation speed and the effective abstraction reducing the development complexity.

The main contributions of this chapter are as follows:

- to describe the overall concept and tool-chain architecture that provides appropriate abstractions for rapid FPGA-based prototyping;
- to describe the detailed implementation of the Python-based open-source toolkit for Verilog HDL designs;
- to describe the evaluation results of the proposed methodology on a single FPGA platform; and
- to describe the evaluation results of the proposed methodology on ScalableCore system as a multi-FPGA based platform.

4.1 Motivation

In order to map a target processor design to an FPGA, emulator developers should consider resource limitations on FPGA. It requires detailed implementation suited for each FPGA's characteristics. In memory perspective, in order to assemble fast evaluation environment, on-chip memory fabrics on FPGA (such as block RAM of Xilinx FPGAs) should be efficiently utilized. However, the amount of on-chip memory fabrics is limited. Memory systems in a processor design might be modified to fit the memory capacity. If the original design requires the larger capacity than the amount of on-chip memory fabrics on the FPGA, the prototyping system can use a huge off-chip memory, such as DRAM. It is time-consuming to develop such an FPGA system with complicated hierarchical memory systems.

Another problem of employing a hierarchical memory system is how to satisfy the cycle-accuracy of logic emulation. In every clock cycle, if the required data for logic emulation are not ready on the top of the memory hierarchy, emulated hardware must be stalled to wait for the correct data for cycle-accurate emulation. In order to correctly emulate the behavior of hardware, an additional mechanism to throttle the hardware is required. It is equally time-consuming to implement complicated logics for cycle-accuracy without any mistakes.

In case of multi-FPGA based prototyping, the emulator designer should also consider arbitration logics among FPGAs. In order to simulate the target processor with keeping the cycle-accuracy, the status of boundary signals crossing multiple FPGAs is carefully managed. Actually, in ScalableCore system, the control system for a memory system and synchronization is manually designed and implemented. If the target architecture is changed, these complex and important parts should be carefully redesigned. Therefore appropriate abstractions of inter-FPGA communications and memory system are required for rapid development of useful FPGA-based prototyping systems.

4.2 Methodology Overview

I propose flipSyrup, a novel design methodology to automatically synthesize a coordinated system of memory, communication and arbitration logics for FPGA-based rapid prototyping. The tool-chain of flipSyrup generates an IP-core (Intellectual Property core) package of specialized RTL for FPGA-based prototyping from RTL design of the simulated processor. The tool-chain supports both single FPGA platform and multi-FPGA platform as implementation environments of prototyping systems.

The tool-chain of flipSyrup provides two key abstractions of FPGA resources for simulated processors based on the RTL. The first is just an abstraction of a memory system that is like a fast large memory space via a simple interface by employing on-chip memory blocks and off-chip large-capacity memory components. The on-chip memory fabrics are invoked as cache memories to improve the total memory performance. Emulated logics can use a fast and large memory space without the on-chip memory capacity limitation of FPGA.

The second abstraction is for communication channel between FPGAs. The tool-chain provides also an abstraction of communication channels to the neighbor FPGAs through FPGA I/Os. Emulated hardware RTL can use them as registers. Read/write operations to abstract registers are automatically translated into FPGA-FPGA communications with keeping the cycle level accuracy of the total simulation result.

The key point of flipSyrup is to keep the cycle-accuracy of emulated hardware in spite of two resource abstractions. In order to sustain the cycle-accuracy, flipSyrup synthesizes a management hardware that throttles the emulated hardware when any miss events happen, such like cache misses on abstract memories and transmission rendezvous of abstract communications. To control the emulated hardware from a manager of the tool-chain, RTL design is automatically converted into a controllable design by using static code analysis.

4.3 Design Flow

4.3.1 Flow Overview

Figure 4.1 shows the development flow of a processor prototype using flipSyrup. FlipSyrup tool-chain generates an IP-core package from the simulated processor RTL designs. In developments with flipSyrup, all the processor design is defined in HDL. In the current implementation of flipSyrup, Verilog HDL is only supported for target processor implementation.

In order to synthesize the ready-to-implement RTL design as an IP-core package from the pure RTL design under the flipSyrup abstractions, the original RTL designs are converted into AST (Abstract Syntax Tree). Then the corresponding parts of the AST (input port declarations, instance declarations

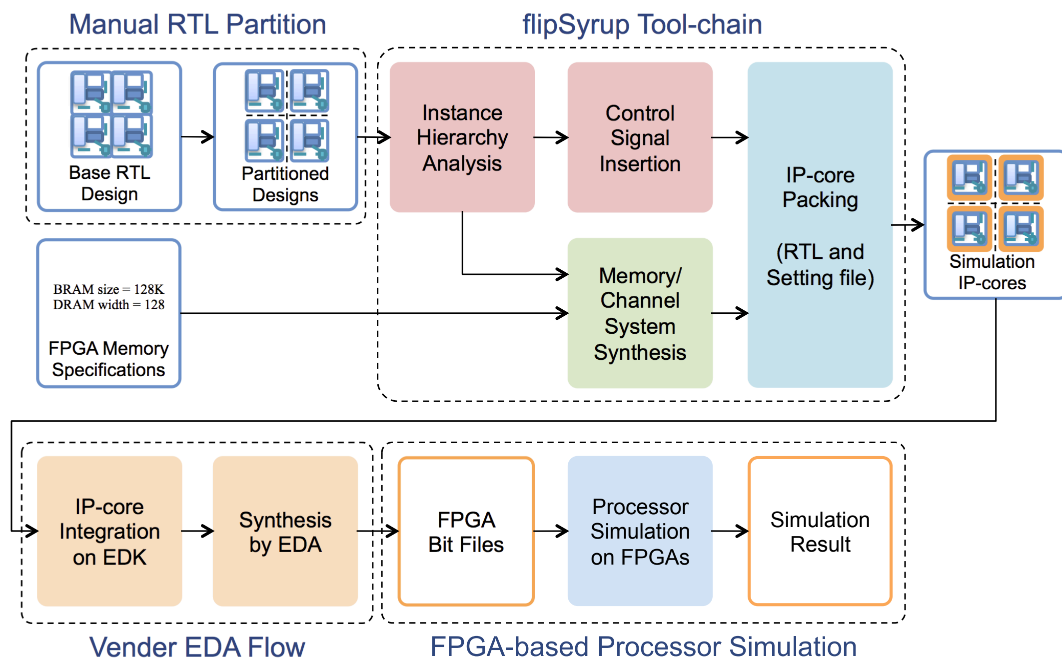


Figure 4.1 Development Flow of Prototyping System with flipSyrup

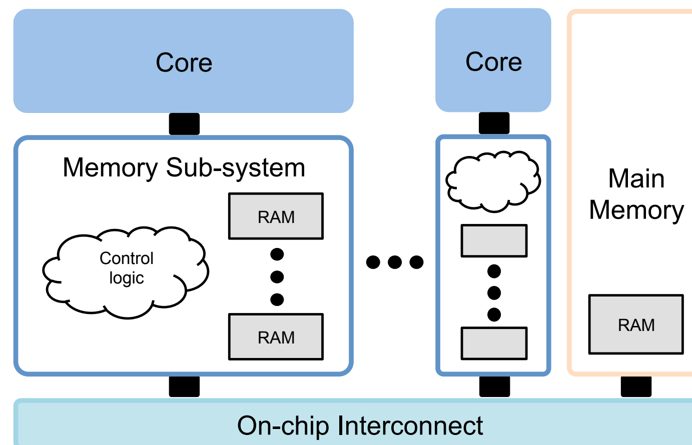


Figure 4.2 Base Processor Design

and always-statements) are modified. Finally, the modified AST is converted into source code of Verilog HDL. To this end, I developed Pyverilog, a Python-based hardware design processing toolkit for Verilog HDL, as I describe later.

Now, we assume a standard multicore processor as a prototyping target, as shown in Figure 4.2. The processor includes several processor cores and memory sub-systems, such as caches and scratchpads. The memory sub-systems include some RAM objects, such as a data array of a cache memory. These

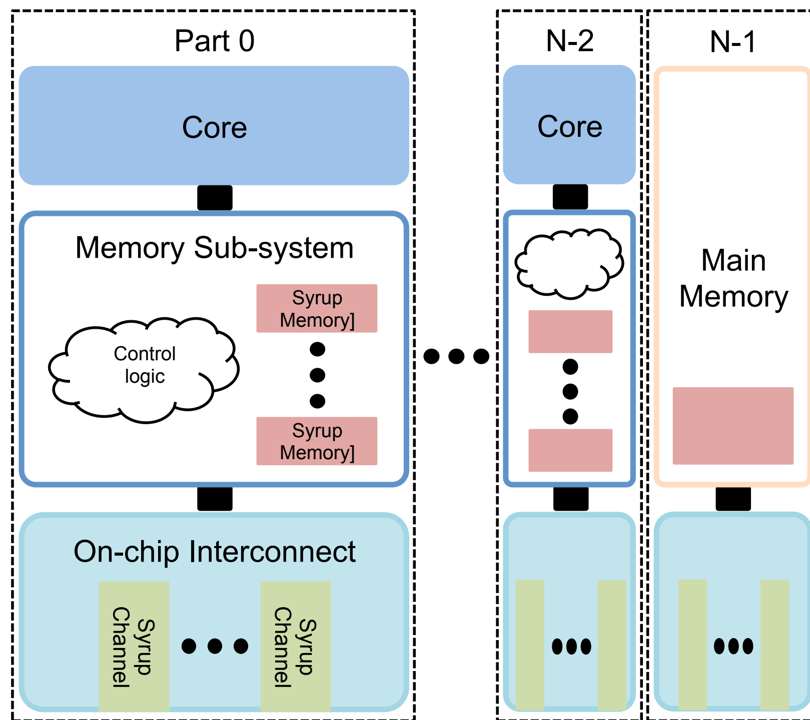


Figure 4.3 Partitioned Processor Design for flipSyrup

hardware components are connected by any on-chip interconnects, such as NoC.

I explain each design step of processor prototyping with the flipSyrup tool-chain, as follows.

4.3.2 Step 1: Design Partitioning

To generate a simulation system design in RTL from the pure processor designs, it takes several steps. Unfortunately, the current implementation of flipSyrup does not provide fully-automatically synthesis functionality for the final IP-core design of a simulated processor. In the preprocess step, RTL designs of the simulated processor should be manually partitioned into multiple regions, as well as ScalableCore system, if multi-FPGA platform is utilized. Otherwise any partitions are not required.

An example of a partitioned design is illustrated in Figure 4.3. Each partition will be implemented on each sub-FPGA system, as well as ScalableCore system. In order to partition the on-chip interconnect, flipSyrup provides an *abstraction of communications* to neighbor partitions on the different sub-FPGA systems. Additionally, flipSyrup offers an *abstraction of memory systems* on each FPGA to eliminate capacity limitations of FPGA on-chip memory systems.

4.3.3 Step 2: Object Replacement

Simulator developers should modify some of the baseline RTL designs in order to flipSyrup methodology. The developers should replace (1) a RAM component represented in standard inferable descriptions (such as block RAM) with an abstract memory (Syrup Memory in the figure) and insert an abstract channel (Syrup Channel) into the boundary between the two partitioned designs.

4.3.4 Step 3: flipSyrup Synthesis

After the partitioning, the flipSyrup tool-chain automatically generates an IP-core design for actual FPGA platforms from each partitioned RTL design. The current tool-chain supports both the AMBA AXI4[106, 107] interconnect architecture and the general handshake protocol for IP-core interfaces. The tool-chain uses a specification of FPGA memory systems, in order to synthesis appropriate memory sub-systems to increase the simulation speed. The specification includes the capacity of on-chip memory systems (block RAM), the data width of the off-chip memory controller to an external memory.

4.3.5 Step 4: Bitstream Synthesis on Standard EDA

Figure 4.4 shows the final overview of the generated simulation system designs by flipSyrup tool-chain. Each sub-FPGA system includes one IP-core corresponding to each sub-region of the simulated processor. FlipSyrup synthesizes a coordinated system of (1) a memory system abstracting memory components, (2) a communication channel for each abstract channel to the neighbor FPGA, and (3) cycle-accurate manager to control the simulated hardware throttling.

The generated IP-core package for processor simulations can be integrated with other essential components to construct a complete FPGA system, such as a memory controller, on the standard FPGA system development flow. Then the FPGA bitstream for processor simulations on the actual FPGA platform is obtained through the standard EDA flow. Finally the simulated processor is realized on the FPGA platform, by using the generated circuit images.

4.4 RTL Modeling under flipSyrup Abstractions

In order to automatically synthesizes ready-to-implement RTL designs for actual FPGA platforms from target RTL descriptions under the abstraction, flipSyrup provides two kinds of abstractions in user HDL level: (1) a *abstract memory system* and (2) a *abstract channel* between the neighbor FPGA. They enable designers to shape a prototyping target processor without concern for actual platform resources. Figure 4.5 shows Verilog HDL templates for these abstract objects.

I describe the purpose and use of each flipSyrup abstraction as follows.

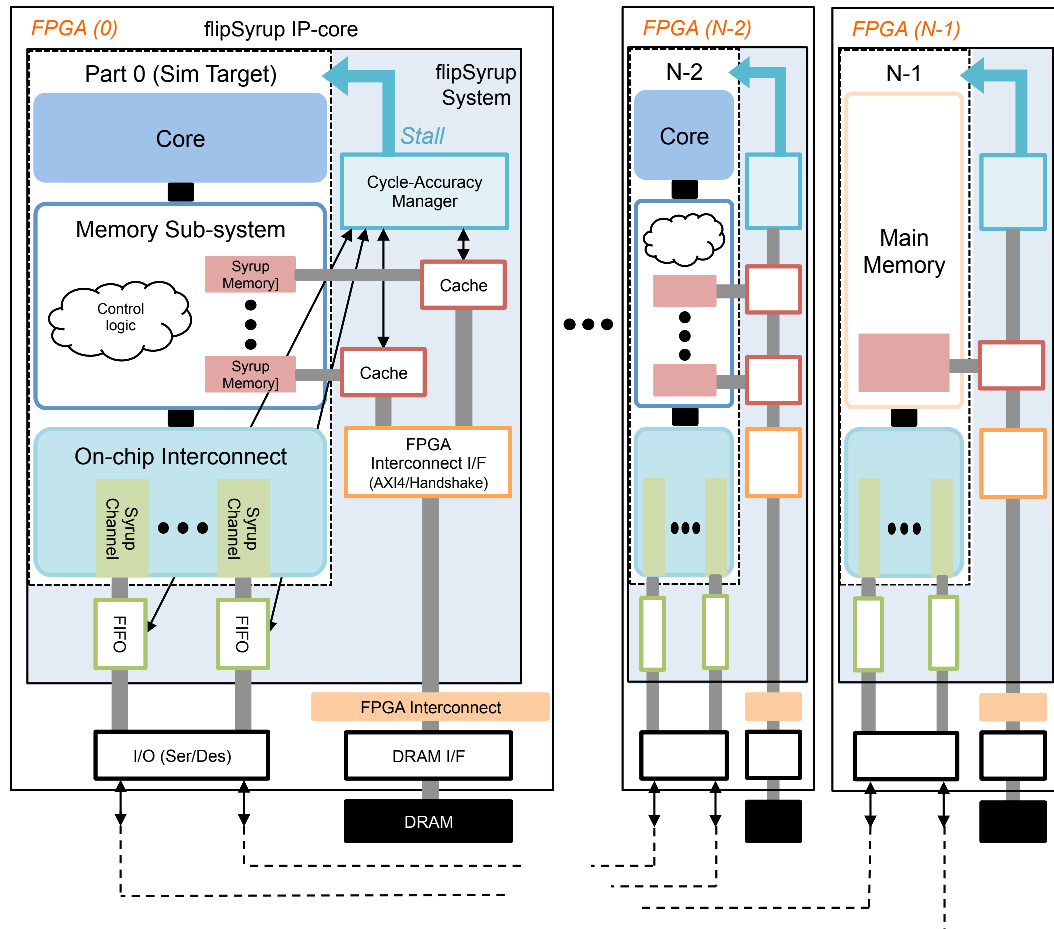


Figure 4.4 Generated Simulation System with flipSyrup

4.4.1 Syrup Memory

The first abstraction of flipSyrup is for on-chip memory systems that each FPGA platform has. Every FPGA has some on-chip memory blocks, such as block RAM and LUT RAM. Unfortunately, the amount of them is restricted and very small. In order to expand the available memory capacity, off-chip DRAM also can be utilized together. In general, off-chip DRAM has a much greater capacity than the on-chip memory blocks. However, just using such off-chip memory system cannot easily satisfy the cycle-accuracy of simulation results, because these external memory system and their access interface have very different characteristics opposed to on-chip memory blocks. On-chip memory blocks usually can be accessed within the constant latency (in most cases, the latency is 1 cycle). In contrast, the access latency of such external memory systems is not constant and varied. Therefore, in order to satisfy the cycle-accuracy of simulation results by using the external memory systems, decoupling clock cycles on the FPGA platform and clock cycles on the simulated processor is essential. Then a mechanism

Syrup Memory (1-port)

```
SyrupMemory1P #
  (
   .DOMAIN("domain"),
   .ID(0),
   .ADDR_WIDTH(W_A),
   .DATA_WIDTH(W_D),
   .WAY(1),
   .LINEWIDTH(128),
   .BYTE_ENABLE(0)
   )
 inst_mem0
  (
   .CLK(CLK),
   .ADDR(addr),
   .D(data_in),
   .WE(wen),
   .Q(data_out),
   .RE(ren),
   .BE()
   );
```

Syrup Out Channel

```
SyrupOutChannel #
  (
   .DOMAIN("domain"),
   .ID(0),
   .DATA_WIDTH(W_D)
   )
 inst_outchannel
  (
   .CLK(CLK),
   .D(data_in),
   .WE(wen)
   );
```

Syrup In Channel

```
SyrupInChannel #
  (
   .DOMAIN("domain"),
   .ID(0),
   .DATA_WIDTH(W_D)
   )
 inst_inchannel
  (
   .CLK(CLK),
   .Q(data_out),
   .RE(ren)
   );
```

Figure 4.5 Abastract Objects of flipSyrup

throttling the simulated hardware is also required, as described in the chapter of ScalableCore system.

For user-level RTL designs of simulated processors, flipSyrup provides a *Syrup memory* that behaves an ideal on-chip memory block in the simulated processor world. Syrup memory is just an abstract memory using a cache memory with on-chip memory block of FPGAs and an external memory system (DRAM).

Syrup memory (1-port) in Figure 4.5 represents an example instance template of Syrup memory with a single access port in Verilog HDL. The module of *SyrupMemory1P* corresponds to an ideal single-cycle memory block with 1 memory access port in the simulated processor world. To easily use a Syrup memory in the user-level RTL design, Syrup Memory has a generic interface with standard block RAM. The simulator designers should change the original instance declarations of on-chip memory blocks with instance declarations of the Syrup memory. For more complex memory blocks with multiple access ports, flipSyrup provides not only a single-port memory module but also multi-ported memory modules. For instance, *SyrupMemory2P* corresponds to dual-ported (2-port) memory block.

In user-level RTL designs, the simulated hardware can read and write data on an abstract memory by using the standard read/write protocol. As the user-level RTL view, the abstract memory can be accessed within just 1 cycle in latency. Using this abstraction of memory systems, system designers can express the simulated hardware without any concerns of on-chip memory capacity limitations.

Syrup memory in the simulated processor design can be treated as an ideal on-chip memory block of FPGA, without the actual capacity limitations of on-chip memory blocks. Actually, these Syrup memory instances are translated into system level cache memories to be attached to an external memory, in the flipSyrup tool-chain. Therefore the maximum size is constrained by the capacity of the external memory, not by the capacity of on-chip memory blocks. Syrup memory behaves as a very large on-chip memory block on FPGAs, while the actual implementation uses a cache memory for abstractions. If a miss event happens, the cycle-accuracy manager generated by flipSyrup stalls the entire simulated processor. Then the cycle-accuracy of simulation result is naturally kept.

Since the logical size of whole Syrup memory objects on each FPGA is not limited by the on-chip memory capacity for each FPGA, the designer can choose the appropriate size for each emulated hardware component. It improves the programmability to develop a target processor on FPGAs, because the designers do not have to consider the actual limitation of on-chip memory capacity. The logical size and the data width of Syrup memory are defined by the parameters of passing arguments of instantiation, as shown in the figure. By using given memory specification of FPGA system, flipSyrup automatically determine the cache size and the bus width to the external memory. In contrast, by passing the parameter of cache line size and the number of ways, the designer can optimize the cache performance.

4.4.2 Syrup Channel

The second abstraction of flipSyrup is for inter-FPGA communications. In manual developments of multi-FPGA based simulation systems, designers should think about how to partition the simulated processor into multiple parts. Signal states on the boundary of each partition should be shared with the neighbor FPGAs for correct simulation results. Since communications across FPGAs take several latency, they should implement a control mechanism keeping the cycle-accuracy that depends inter-FPGA communications, as well as memory system abstractions mentioned above.

In order to represent a boundary of a logical partition in user-level RTL designs, flipSyrup provides a *Syrup channel* that behaves an ideal single-cycle communication interface in the simulated processor world. An instance of Syrup channel corresponds to an abstract I/O interface via the inter-FPGA communication link between the neighbor FPGA. Actual communications across FPGAs elapse several periods to be completed. The cycle-accuracy issue of the simulated processor behavior is well managed by the cycle-accuracy manager, as well as memory system abstractions.

As shown as Figure 4.5, there are separate abstract templates for two-way communication directions: *SyrupOutChannel* for output interface to the neighbor FPGA, and *SyrupInChannel* for input interface from the neighbor FPGA. Every SyrupOutChannel has a corresponding SyrupInChannel in the other logical partition. Beforehand, system designers should define every connectivity between the SyrupOutChannel and the corresponding SyrupInChannel.

In user-level RTL designs, sending a value to the neighbor partition is represented by a write operation to the corresponding SyrupOutChannel. The data port (D) of SyrupOutChannel is set with the sent value, and the enable port (WE) is asserted as well. In the user-level RTL view, the written value on the data port is transferred to the corresponding SyrupInChannel within 1 cycle. As well as write operations, read operations are also abstracted. In the user-level RTL designs, receiving a value from the neighbor partition is represented by a read operation to the corresponding SyrupInChannel. The enable port (RE) is asserted. Then, at the next clock cycle in the simulated processor world, the value can be read from the data port (Q). Using this abstraction of inter-FPGA communications, system designers can express the simulated hardware without any concerns of the elapsed latency for each communication and the cycle-accuracy matter.

In order to keep the cycle-accuracy of the simulated processor behavior, the cycle-accuracy manager stalls the entire simulated processor, when a corresponding read value is not prepared. In contrast to memory abstractions, the actual connectivity of these abstractions depends on the architecture of the FPGA platform. As the actual low-level hardware of these abstract interfaces, the tool-chain of flipSyrup produces a FIFO-based standard interface accessing data in the abstract interface. Then system designers should connect the corresponding generated interfaces for each other on the utilized FPGA platform.

4.5 flipSyrup in Detail

This section presents the detailed features of flipSyrup. The flipSyrup tool-chain automatically synthesizes low-level hardware designs that are able to be implemented on an actual FPGA platform. To this end, the tool-chain generates various additional hardware components for accurate simulations, and then translates the input RTL design of the simulated processor into controllable designs by the generated additional hardware.

4.5.1 Abstract Object Conversion

As I described above, Syrup memory can be used just an single-cycle on-chip memory block of FPGAs without capacity limitation in the user-level RTL designs. The actual implementation of the Syrup memory is a cache memory to be attached to external DRAM through an on-chip interconnection network. The access latency of a cache primitive is 1 clock cycle for both read and write operations. Since the cache primitive has just one shared port for all read and write requests, a request arbiter is also used together, in order to handle simultaneous multiple requests in the simulated processor world.

A cache memory consists of two raw memory instances: Tag RAM and data RAM. Both memory instances are implemented using true dual-port memory blocks on FPGAs. An entry of tag RAM consists of 4 kinds of information: an original tag (represented as a part of address), a valid-bit, a dirty-

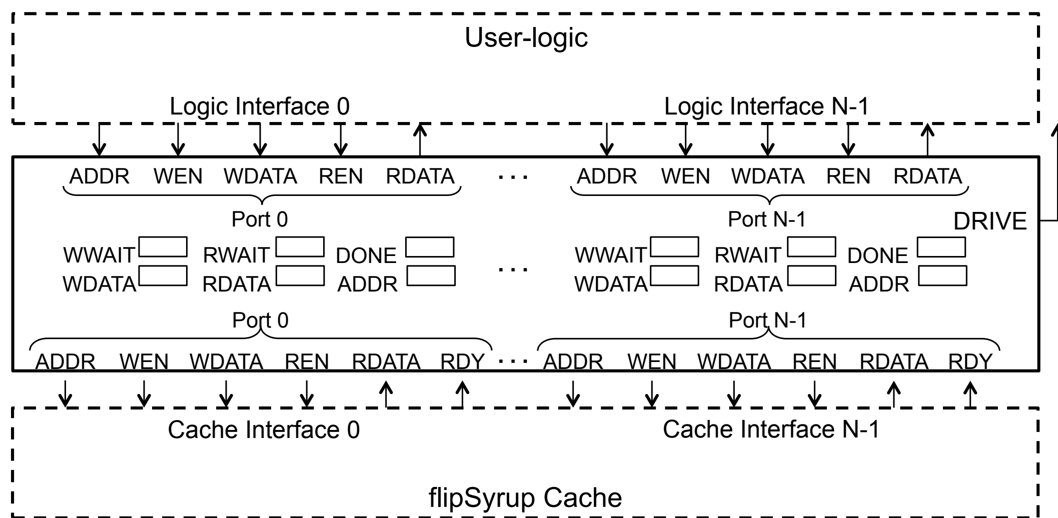


Figure 4.6 Cycle-Accuracy Manager

bit and an access-bit (indicating whether the line has been ever accessed or not). Data RAM consists of multiple independent banks of byte-granularity in order to support byte-access.

As well as Syrup memory, a Syrup channel can be utilized just an single-cycle communication interface that is connected to the neighbor region in the user-level RTL designs. The actual implementation of the Syrup channel is a FIFO to be attached to external I/O ports connected to the neighbor FPGAs. In the user-level RTL view, interfaces of the SyrupChannel can be viewed as a single-cycle shared register interface without any stalls.

A generated interface of SyrupChannel should be connected with that of the neighbor FPGA. Since the connectivity definition of SyrupChannel objects is not managed by the flipSyrup tool-chain, designers beforehand determine how to partition the simulated processor and how to connect them.

4.5.2 Architecture of Cycle-Accuracy Manager

In order to keep cycle-level accuracy of the simulation results under the flipSyrup abstractions, a hardware mechanism to throttle the entire simulated hardware when any miss events occur, such as a cache misses on the cache memory of a Syrup memory. In the previous chapter about ScalableCore system, the simulated hardware is controlled by the manually pre-defined state machine controller for local barrier synchornizations by decoupling the FPGA clock cycle and the simulated clock cycle.

In flipSyrup, the tool-chain automatically synthesizes such a hardware controller to throttle the entire simulated hardware, named as *cycle-accuracy manager*. I present here the architecture of the cycle-accuracy manager as follows. For the simplicity of descriptions, we assume a single FPGA platform without any Syrup channels to be connected with other FPGAs.

Figure 4.6 shows the structure of the cycle-accuracy manager with N ports for independent memory accesses on Syrup memory objects. *DRIVE* is a throttle signal port supplied to the simulated hardware. To stall the simulated hardware, the DRIVE port is de-aaserted. Otherwise, the DRIVE port is asserted when there are not any uncompleted operations related to the flipSyrup abstractions.

At the side of simulated hardware, the controller provides simple interfaces similar to the standard memory interface of on-chip memory fabric of FPGAs. Each interface consists of 5 signals for memory operations: (1) address, (2) write enable, (3) write data, (4) read enable and (5) read data. For raw cache memories of Syrup memory objects, the controller provides the same interfaces as the simulate hardware. It also provides ready signal ports to receive notifications of memory access completions in low-level memory systems.

In order to throttle the simulated hardware, the cycle-accuracy manager internally stores some information from the low-level cache memories: (1) address, (2) write data, (3) whether read enable port or write enable port is asserted or not from the simulated hardware,(4) and whether the previous requests are completed or not. The controller controls the throttle signal for the simulated hardware using these information for cycle-level accuracy of simulation results.

For each clock cycle on an FPGA, the controller checks whether there are any uncompleted requests. Then the controller asserts DRIVE signal if all the requests are completed. If there are any requests in processing, the controller stalls the simulated hardware by de-asserting the DRIVE signal. Because the simulated hardware assumes an ideal memory block with 1-cycle in latency, the simulated hardware might issue new requests at the timing that the simulated hardware is already stalled. To correctly process all requests, the controller keeps the request information issues in the previous clock cycle.

4.5.3 Throttling by Cycle-Accuracy Manager

Figure 4.7 illustrates a timing chart of two read requests that arbitrated by the cycle-accuracy manager for cycle-accurate simulations. The main contribution of cycle-accurate manager is to throttle the simulated hardware components, when requested operations are not yet completed. By this, simulated hardware can use low-level memory objects and inter-FPGA communications as ideal single-cycle interfaces. I explain the behavior of cycle-accuracy manager by using the simple example depicted in the figure.

At cycle 0, the DRIVE signal (enable signal to process the hardware state, which is an inverted Boolean of stall signal) is asserted because LogicREN0 and LogicREN1 (read enable signals from the emulated hardware) are not asserted.

At cycle 1, the state of emulated hardware proceeds for 1-cycle by the asserted DRIVE signal at cycle 0. Then the emulated hardware generates two new memory requests. The controller receives these requests and directly passes to the lower memory system. So CacheREN0 and CacheREN1 (read

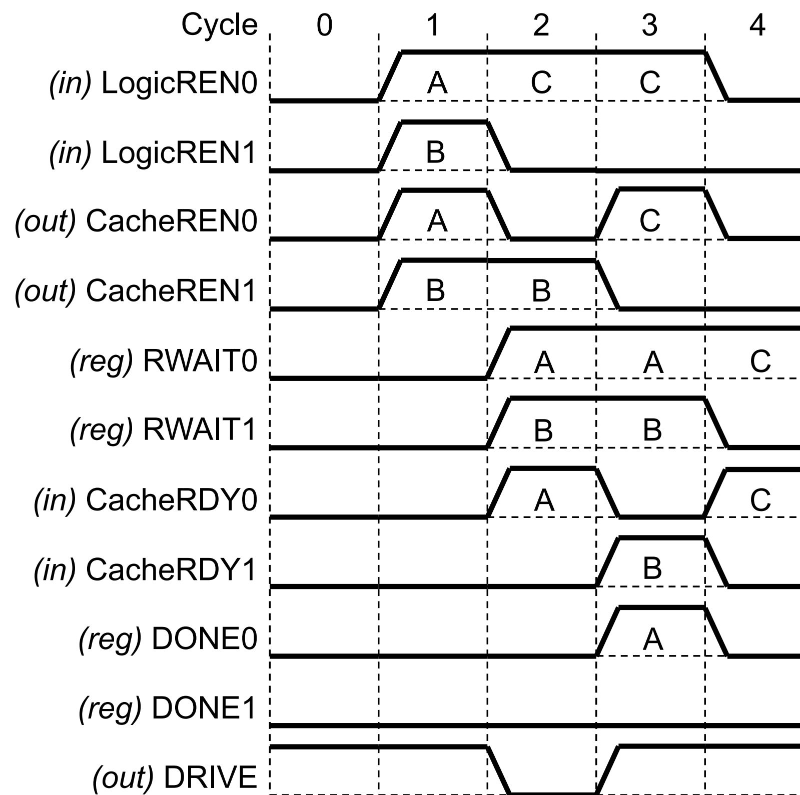


Figure 4.7 Timing Chart Example of Cycle-Accuracy Manager

enables signals to the cache memories) are asserted.

At cycle 2, the controller de-asserts DRIVE signals because the all requests are not completed at cycle 0. Because of the asserted DRIVE signal at cycle 0, status of LogicREN0 and LogicREN1 is updated by 1-cycle progress. At a time, two internal registers (RWAIT0 and RWAIT1) record that those requests at cycle 1 are still in processing, so that the controller correctly asserts CacheREN0 and CacheREN1 until all the requests are completed. In this example, at this cycle, CacheRDY0 is asserted by the lower memory. Then the controller de-asserts the CacheREN0 because all the requests are not accomplished at cycle 0.

At cycle 3, the completion of CacheREN0 request at cycle 2 is registered by the internal DONE0 register. In this example, CacheRDY1 is asserted, so that the controller recognizes its request completion at a time. Because all requests at cycle 1 are completed at this time, the controller asserts DRIVE signal to go forward the emulated hardware for 1 cycle progress.

The number of memory access ports and inter-FPGA communication ports depends on the simulated processor structure. The flipSyrup tool-chain automatically synthesizes a suitable cycle-accuracy controller for each configuration.

```
1  generate for(i=0; i<2; i=i+1) begin: loop
2   SyrupMemory1P
3    #(
4      .DOMAIN("domain"),
5      .ID(i),
6      .ADDR_WIDTH(W_A),
7      .DATA_WIDTH(W_D)
8      )
9   inst_memory_name
10     (
11     .CLK(CLK),
12     .ADDR(mem_addr),
13     .D(mem_d),
14     .WE(mem_we),
15     .Q(mem_q)
16     );
17 end endgenerate
18
```

(a) Instantiation Declaration in Input RTL

```
1  generate for(i=0; i<2; i=i+1) begin: loop
2   if((i == 0)) begin
3    SyrupMemory1P
4     #(
5       .DOMAIN("domain"),
6       .ID(i),
7       .ADDR_WIDTH(W_A),
8       .DATA_WIDTH(W_D)
9       )
10   inst_memory_name
11     (
12     .CLK(CLK),
13     .ADDR(mem_addr),
14     .D(mem_d),
15     .WE(mem_we),
16     .Q(mem_q),
17     .p0_addr(domain_syrupmemory_0_addr),
18     .p0_d(domain_syrupmemory_0_d),
19     .p0_we(domain_syrupmemory_0_we),
20     .p0_q(domain_syrupmemory_0_q)
21     );
22  end
23
24 else if((i == 1)) begin
25   SyrupMemory1P
26    #(
27      .DOMAIN("domain"),
28      .ID(i),
29      .ADDR_WIDTH(W_A),
30      .DATA_WIDTH(W_D)
31      )
32   inst_memory_name
33     (
34     .CLK(CLK),
35     .ADDR(mem_addr),
36     .D(mem_d),
37     .WE(mem_we),
38     .Q(mem_q),
39     .p0_addr(domain_syrupmemory_1_addr),
40     .p0_d(domain_syrupmemory_1_d),
41     .p0_we(domain_syrupmemory_1_we),
42     .p0_q(domain_syrupmemory_1_q)
43     );
44  end
45 end endgenerate
46
```

(b) Instantiation Declaration in the Converted RTL

Figure 4.8 Inserting Control Ports of Abstract Objects

4.5.4 RTL Design Conversion

The flipSyrup tool-chain generates a controller hardware to satisfy the cycle-level accuracy of simulation results, as I described above. Then the tool-chain automatically translates the pure RTL designs of the simulated processor into controllable RTL designs by the generated controller. The RTL design conversion takes two independent steps: (1) inserting control ports for abstract objects, and (2) inserting throttling ports for cycle-accuracy manager.

Inserting Control Ports for Abstract Objects

To define an ideal memory or communication interface, system designers can use stubs of Syrup objects in the baseline RTL code. Figure 4.8(a) shows an example instance template of a Syrup memory object. To identify the personality of the abstract object, several parameters are essential for instance creation: (1) domain name, (2) object id, (3) data width of the user-side interface and (4) address length to

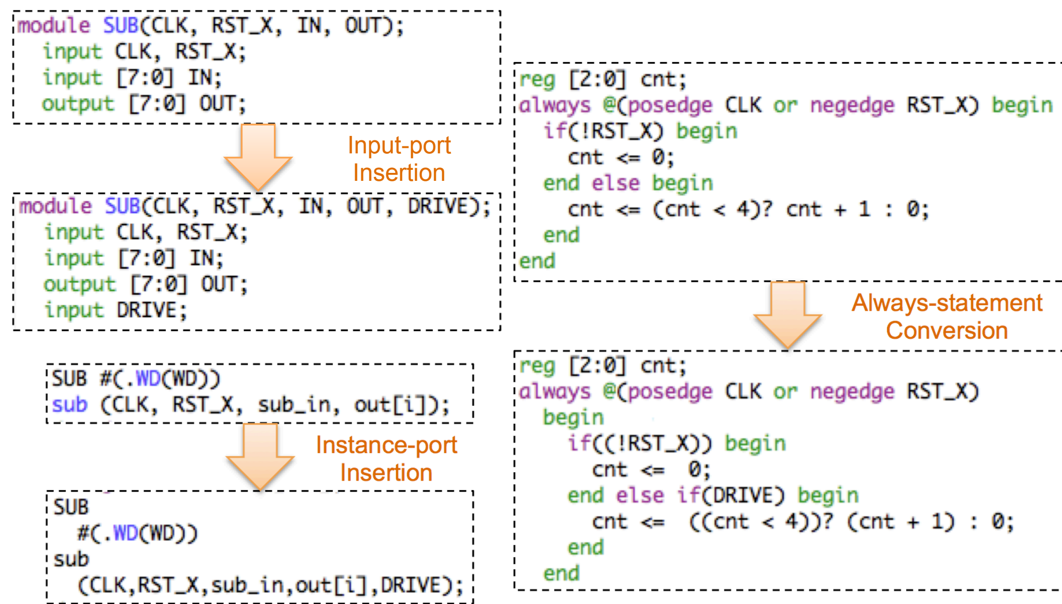


Figure 4.9 RTL Conversion Rules for Inserting Throttling Ports

identify the memory space capacity.

In the flipSyrup compilation step, the user RTL translator (Instance Hierarchy Analysis in Figure 4.1) analyzes the input RTL code, in order to replace stubs in the code with actual hardware components for abstractions. Then the translator automatically inserts additional signals for abstract objects into the RTL design. Therefore the designers do not have to prepare any signal ports that control abstract objects from outside of the user-level RTL.

As illustrated Figure 4.8(a), a *generate* statement can be used to represent a parameterized module/instance hierarchy. As shown in Figure 4.8(b), several signals for abstract objects are inserted to the instance port of an abstract instance. When a generate statement is used, some *if* statements are appended to switch the inserted signals. If instances of the module including abstract objects are created twice or more, the module definition in the RTL design is duplicated to avoid any name conflicts caused by the additional signals.

Inserting Throttle Ports for Cycle-Accuracy Manager

If any miss events, such as cache misses or port conflicts occur, the corresponding cycle-accuracy manager stalls the simulated hardware by using public throttle signal. The tool-chain of flipSyrup automatically inserts a throttle signal into RTL designs of the simulated processor.

Figure 4.9 shows the conversion rules for the throttle signal insertion. The signal of *DRIVE* is a throttle signal port that is controlled by the cycle-accuracy manager. The RTL conversion tool adopts three conversions for the RTL designs: (1) adding an input port of the throttle signal for each module

definition, (2) adding a signal port for each instance definition, and (3) transforming always-statements as the all registers are synchronously updated with the external throttle signal.

By adding an input port by the rule (1) and the rule (2), the throttle signal on the top level of the RTL design is propagated to the lowest module in the RTL. By adding an update condition into all always-statements by the rule (3), the cycle-accuracy controller can stall the entire simulated processor[*1].

Since most processors have mechanisms to stall the computation pipelines by the internal controller, the degradation of the maximum operation frequency due to these code translations is quite small. Additionally, modern FPGAs have gated-clock mechanisms to reduce power consumptions, such throttling signals can be naturally implemented using the functionality of FPGA raw primitives.

4.6 Pyverilog: A Python-based Hardware Design Processing Toolkit for Verilog HDL

As I described, the tool-chain of flipSyrup automatically synthesizes an IP-core from the input pure RTL designs under the resource abstractions. For such RTL design handling, I developed *Pyverilog*, a hardware design processing toolkit for Verilog HDL. Pyverilog is a Python-based open-source toolkit of various independent functions for analysis, code translation and code generation of Verilog HDL RTL designs. Pyverilog is entirely implemented in Python for its portability.

There are some popular open-source Verilog simulators that encourage the hardware development community[38, 39, 40]. These simulators are originally designed for practical Verilog simulations, not for reuses of the design analysis result by other software tools. To the best of our knowledge, however, there are no open-source products for both analysis and code generation of Verilog hardware design.

In this section, I describe the design and implementation of Pyverilog, especially for flipSyrup.

4.6.1 Overview

As the fundamental knowledge, Verilog HDL is the most-used design language to express the fabric of a hardware structure in the register transfer level (RTL) for both ASIC implementation and FPGA-based implementation. Since Verilog HDL has been used for many years, a lot of useful information and common design patterns are available in the literature and the web. Additionally, constructing an effective hardware structure for high performance and low power requires cycle-by-cycle scheduling definitions for computations and data movements. In cases that the high-level design languages do not fit with the application characteristics, the designers would prefer to use a general register transfer level HDL. Verilog HDL is certainly useful for such fine-grain hardware definitions in any cases, because Verilog HDL has enough capability to express cycle-level circuit behaviors. Thus Verilog HDL will

[*1] The current tool does not support level-sensitive latches and multiple clocks.

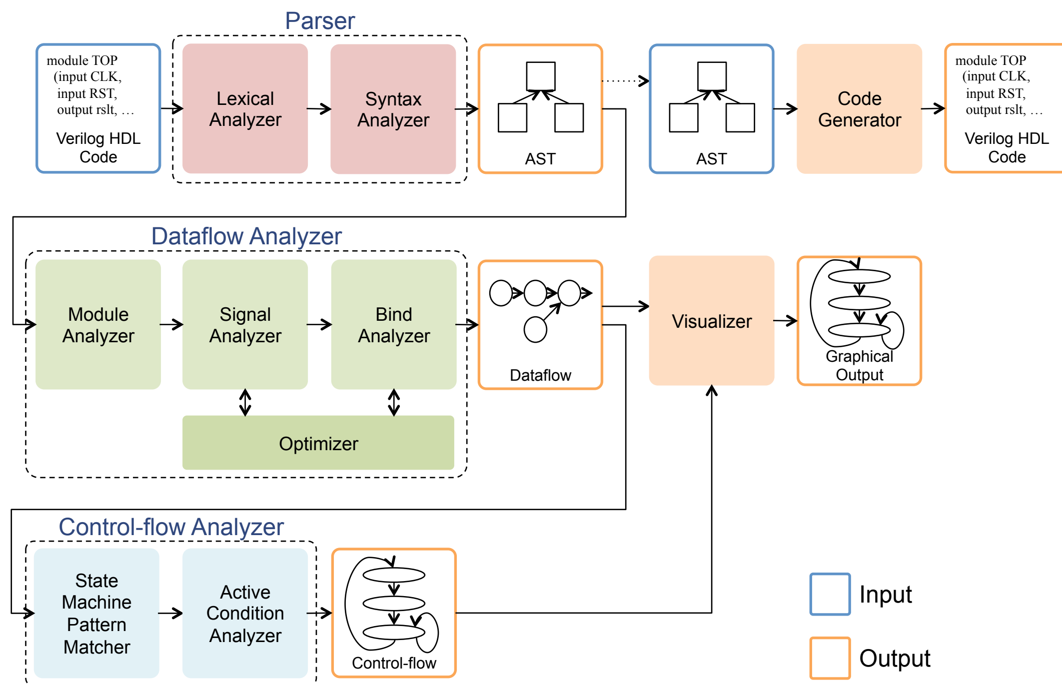


Figure 4.10 Pyverilog Toolkit Overview

certainly be important as both a cycle-by-cycle modeling language for highly tuned hardware structures, and a low-level format of hardware, like assembly languages of conventional software environments.

Figure 4.10 presents the overview of Pyverilog toolkit. Pyverilog offers (1) code parser, (2) dataflow analyzer, (3) control-flow analyzer, (4) visualizer and (5) code generator for Verilog HDL. The objective of Pyverilog is to make it easy for researchers and engineers to implement a novel CAD tool for Verilog HDL design. I describe the function and design alternatives for each useful tool of Pyverilog, as follows.

4.6.2 Design and Implementation

Code Parser

The code parser of Pyverilog is a fundamental tool to analyze the definition in Verilog HDL source codes. The code parser generates an abstract syntax tree (AST) from the code of Verilog HDL.

We chose PLY (Python Lex-Yacc)[108] as the compiler-compiler for all Pyverilog tools, which is a lightweight implementation of a lexical analyzer and an LR-parser. Since PLY is entirely implemented Python, the portability of the code parser is satisfied enough.

Dataflow Analyzer

The dataflow analyzer of Pyverilog is to create a dataflow graph that represents the relationship among the defined signals by using the AST. The generated dataflow graph is also used as the intermediate representation in the next step for control-flow analysis. The dataflow analyzer has 3 sub-analyzers to analyze the entire structure of codes: (1) *module analyzer*, (2) *signal analyzer* and (3) *bind analyzer*. All tools of dataflow analyzer are implemented using the standard visitor pattern that calls an appropriate function of the visited AST node recursively.

The module analyzer first traverses the input RTL code to get the list of modules. The signal analyzer then traverses the same RTL code again to gather signal definitions. At the same time, the optimizer resolves the constant value definitions, such as parameter and localparam, in order to determine the precise instance hierarchy. The bind analyzer finally generates the assignment tree each signal definition.

In the tool-chain of flipSyrup, it uses the module analyzer and the signal analyzer to identify the instance hierarchy and to insert control signals and throttle signals for abstract objects.

Control-flow Analyzer

The control-flow analyzer of Pyverilog is to generate an intermediate representation of finite state machines (FSMs) in the Verilog code. The control-flow analyzer uses the analysis result of the signal definitions and their relationship on the dataflow analyzer. The control-flow analyzer has 2 sub-analyzers: (1) state machine pattern matcher and (2) active condition analyzer.

The state machine pattern matcher explores representable signals of FSMs by using pattern matching schemes of typical signal names, by using the signal lists provided by the dataflow analyzer.

The active condition analyzer then analyzes the conditions that the value of the FSM candidate signal is modified. The dataflow analyzer provides the assignment definition of each signal. They contain the assigned value and its assignment condition for each signal. The control-flow analyzer infers the values of candidate conditions from these assignment conditions. Results of active condition analyses are represented as a number of pairs of cause signal name and its actual values for each state transition.

The control-flow analyzer can independently identify the conditions that a signal is asserted or deasserted, so that it can identify the conditions of state transitions. This feature can be also used to identify the asserted conditions of non-FSM signals. It helps to automatically synthesize an additional circuit to accelerate computational components in part.

Note that the tool-chain of flipSyrup does not use the control-flow analyzer, while the control-flow analyzer provides the advanced functionality for RTL design analysis.

Code Generator

The code generator of Pyverilog is to generate a source code written in Verilog HDL from the internal representation of ASTs. Common source code generators in any languages have a number of template texts of output codes in their internal source codes. Therefore, the source codes tend to be larger than the other parts. In order to separate the implementation of the entire code generator into code write parts using template texts and the control parts, I used a template engine. I used Jinja2[109], a major template engine in Python community. By using the template engine, all template texts of Verilog HDL codes are removed from the Python source code. It simplifies the software structure of the tool. As well as the dataflow analyzer, the code generator is also implemented using the standard visitor pattern that calls an appropriate function of the visited AST node recursively.

The tool-chain of flipSyrup uses the code generator to convert the AST of the modified RTL designs by the signal insertion tools of flipSyrup into actual source codes written in Verilog HDL.

Visualizer

The visualizer of Pyverilog is used to obtain a graphical output of the analysis results, such as dataflow graphs and finite state machines. The implementation of the visualizer of Pyverilog is very simple. The visualizer uses Graphviz[110] with its Python binding. The main operations of the visualizer are to convert a graph of a dataflow or a finite state machine into a graph representation in the Graphviz format.

Figure 4.11 is an example code of a vector-add circuit in Verilog HDL. This circuit calculates the summation value of an array stored in the external memory. The circuit has memory access ports that the name prefixes are *MEM_*. The circuit also includes a finite state machine of the variable *state* to control these memory access ports.

Figure 4.12 shows a dataflow graph picture of the variable *MEM_RE*. The value definition of the variable MEM_RE is represented as a tree of several (1) arithmetical or logical operators and (2) assignment conditions represented as *Branch*. The graphical representation of dataflow graphs makes us easy to find out the definition of a value by intuition.

Figure 4.13 shows the control-flow graph picture of the finite state machine of the variable *state*. The picture represents the FSM of the variable state consists of a cyclic transition pattern. The graphical representation of control-flow graphs also makes us easy to figure out the control structure of the circuit.

4.6.3 Summary of Pyverilog

Pyverilog is an open-source hardware design processing toolkit purely implemented in Python. Pyverilog is already released for public. I believe that understanding the detailed implementation of Pyverilog by reading this paper makes it easy to create another innovative software using Pyverilog for advanced

```
1  module TOP(CLK, RST_X,
2             MEM_A, MEM_RE, MEM_WE, MEM_D, MEM_Q, MEM_BUSY, MEM_DONE);
3    input CLK;
4    input RST_X;
5
6    parameter WA = 32;
7    parameter WD = 32;
8    parameter SIZE = 1024 * 32;
9
10   localparam OFFSET0 = 0;
11   localparam OFFSET1 = SIZE * 1;
12   localparam OFFSET2 = SIZE * 2;
13
14   output reg [WA-1:0] MEM_A;
15   output reg MEM_RE;
16   output reg MEM_WE;
17   output reg [WD-1:0] MEM_D;
18   input [WD-1:0] MEM_Q;
19   input MEM_DONE;
20   input MEM_BUSY;
21
22   reg [WA-1:0] cnt0;
23   reg [WA-1:0] cnt1;
24   reg [WA-1:0] cnt2;
25   reg [WD-1:0] readdata0;
26   reg [WD-1:0] readdata1;
27   reg [WD-1:0] writedata;
28   reg [3:0] state;
29
30   localparam ST_INIT = 0;
31   localparam ST_READ0 = 1;
32   localparam ST_READWAIT0 = 2;
33   localparam ST_INTERVAL = 3;
34   localparam ST_READ1 = 4;
35   localparam ST_READWAIT1 = 5;
36   localparam ST_CALC = 6;
37   localparam ST_WRITE = 7;
38   localparam ST_WRITEWAIT = 8;
39   localparam ST_DONE = 9;
40
41   always @(posedge CLK or negedge RST_X) begin
42     if(!RST_X) begin
43       state <= ST_INIT;
44       cnt0 <= 0;
45       cnt1 <= 0;
46       cnt2 <= 0;
47     end else begin
48       if(state == ST_INIT) begin
49         MEM_WE <= 0;
50         MEM_RE <= 0;
51         if(!MEM_BUSY) state <= ST_READ0;
52       end else if(state == ST_READ0) begin
53         MEM_RE <= 1;
54         MEM_A <= cnt0 + OFFSET0;
55         if(MEM_BUSY) state <= ST_READWAIT0;
56       end else if(state == ST_READWAIT0) begin
57         MEM_RE <= 0;
58         readdata0 <= MEM_Q;
59         if(MEM_DONE) state <= ST_INTERVAL;
60       end else if(state == ST_INTERVAL) begin
61         if(!MEM_BUSY) state <= ST_READ1;
62       end else if(state == ST_READ1) begin
63         MEM_RE <= 1;
64         MEM_A <= cnt1 + OFFSET1;
65         if(MEM_BUSY) state <= ST_READWAIT1;
66       end else if(state == ST_READWAIT1) begin
67         MEM_RE <= 0;
68         readdata1 <= MEM_Q;
69         if(MEM_DONE) state <= ST_CALC;
70       end else if(state == ST_CALC) begin
71         writedata <= readdata0 + readdata1;
72         if(!MEM_BUSY) state <= ST_WRITE;
73       end else if(state == ST_WRITE) begin
74         MEM_WE <= 1;
75         MEM_D <= writedata;
76         MEM_A <= cnt2 + OFFSET2;
77         if(MEM_BUSY) state <= ST_WRITEWAIT;
78       end else if(state == ST_WRITEWAIT) begin
79         MEM_WE <= 0;
80         if(MEM_DONE) begin
81           cnt0 <= cnt0 + 32;
82           cnt1 <= cnt1 + 32;
83           cnt2 <= cnt2 + 32;
84           if(cnt0 < SIZE) state <= ST_INIT;
85           else state <= ST_DONE;
86         end
87       end else if(state == ST_DONE) begin
88         $display("Done");
89         //do nothing
90       end
91     end
92   end
93 endmodule
```

Figure 4.11 Example Verilog HDL Code (Vector-Add)

FPGA/ASIC development.

Note that Pyverilog is used for the fundamental back-end tool of the other CAD tool I developed and published. I developed PyCoRAM[111], an efficient CAD tool for FPGA-based accelerator development with memory system abstraction and the Python-to-Verilog high-level synthesis. Pyverilog is

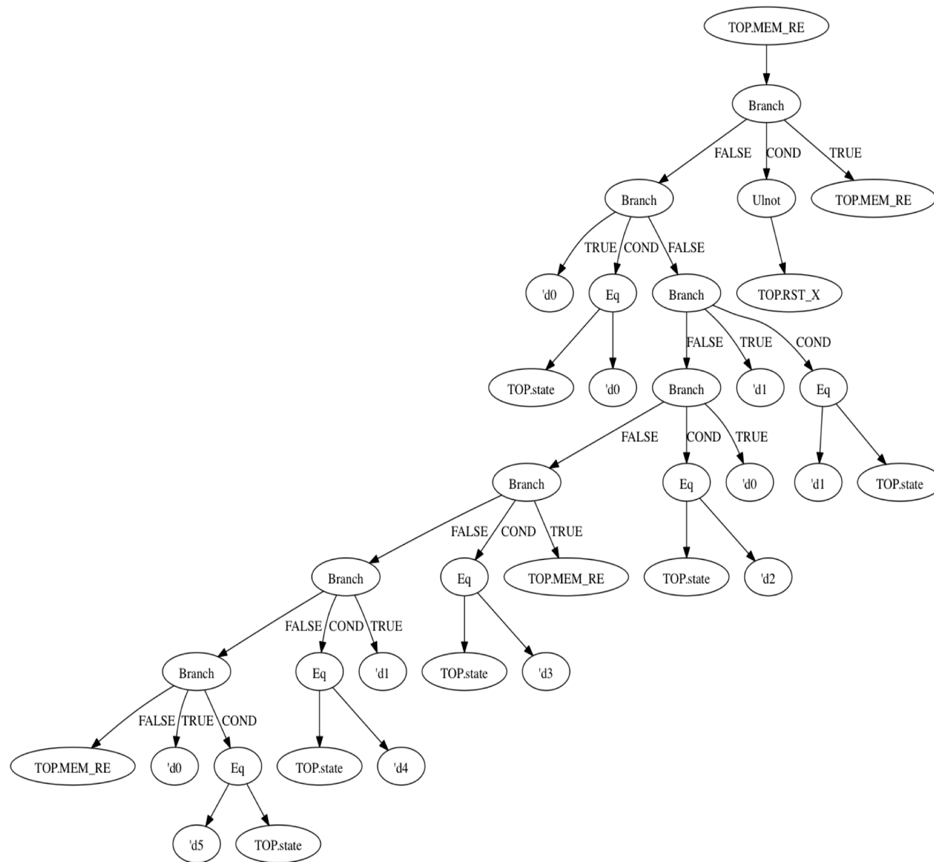


Figure 4.12 Graphical Output of Dataflow Graph

used for code analysis and generation of Verilog HDL, as well as flipSyrup.

4.7 System Generation by flipSyrup

4.7.1 Memory Specification Reader

As I depicted in Figure 4.1, the tool-chain of flipSyrup uses a memory specification file to determine the used capacity of on-chip memory blocks on FPGAs. In order to read the specification, I developed a specification reader implemented in Python with ANTLR[112] (a major LL grammar parser generator) and its Python binding.

4.7.2 Control Circuit Synthesis

The tool-chain of flipSyrup then synthesizes a coordinated simulation system with a cycle-accuracy manager. The code generator of the entire Verilog HDL codes is implemented in Python with the Jinja2[109] template engine.

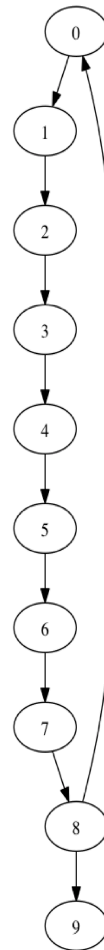


Figure 4.13 Graphical Output of Finite State Machine

4.8 Multi-core Processor Simulation on a Single FPGA Platform

As the first evaluation, I evaluate flipSyrup on a standard evaluation board with a single and large FPGA, so that the simulated processor design is not partitioned in this situation. I used M-Core architecture, NoC-based many-core architecture used for the evaluation of ScalableCore system, as a prototyping target processor.

4.8.1 Evaluation Methodology

Table 4.1 shows the evaluation setup. I evaluated the tool-chain of flipSyrup in performance (simulation speed), resource utilization and maximum operation frequency.

I used Xilinx VC707[115] evaluation board. The prototype is deployed as an AXI4 master IP and connected to the DRAM controller via an AXI4 interconnect. The operation frequency of the system

Table 4.1 Evaluation Setup

Implementation Environment	
Development tool	Xilinx ISE 14.5, Xilinx PlanAhead 14.5, Xilinx Platform Studio 14.5
FPGA	Xilinx Virtex-7 XC7VX485T (VC707 Evaluation Board)
On-chip memory (Block RAM)	18Kbit × 2,060 blocks and 36Kbit × 1,030 blocks, Maximum 37,080Kbit
Off-chip memory (DRAM)	SODIMM DDR3-800 1GB
Operation frequency	50MHz (Target processor and flipSyryp system)
flipSyrup cache	64byte per line, 1-way (Direct map), 1-outstanding miss, no-prefetching
On-chip interconnection	AXI4 256bit width, Cross-bar mode (Performance optimized), 100MHz op.
Target Processor Architecture	
Architecture	M-Core architecture[28] (DMA-based many-core processor)
Core	MIPS32 ISA, 6-stage pipeline, single-issue, 2-memory-port (Fetch, Load/Store)
DMA Controller	2-memory-port (32-bit DMA Read and DMA Write)
Network topology	2-D mesh
On-chip router	4-stage pipeline, 3-virtual-channel, FIFO depth: 4 Credit-base flow control, X-Y Dimension Order Routing
Node memory	512KB (per Node), 32-bit width, access latency: 1 4-port (Fetch, Load/Store, DMA Read, DMA Write)
Number of Nodes	8 (4 × 2, Size of each flipSyrup Cache : 256KB) 16 (4 × 4, Size of each flipSyrup Cache : 128KB) 24 (6 × 4, Size of each flipSyrup Cache : 64KB)
Benchmark	dhrystone[113] (dh), n-queens solver (nq) matrix multiply (mm), 5-point stencil[114](st)

(excepting the AXI4 interconnect and peripheral controllers) is 50MHz. As data RAM of the synthesized cache memories, 2MB of whole block RAM is assigned. The associativity of the cache memories is 1 (direct-map). The line size of the caches is 64 bytes (512 bits) as same as the DRAM burst length.

The target architecture is M-Core architecture as well as the evaluation of ScalableCore system. In this evaluation, I do not assume the external off-chip memory as prototyping target. The tool-chain of flipSyrup generates a cache memory for each scratchpad memory on the target. In order to measure the performance variations according to the core count, I tested three core counts: 8-core 16-core and 24-core. I used 4 benchmarks running on the target processor. To execute these applications on the prototype of FPGA, it takes about several ten seconds.

The hardware components (core, network interface and on-chip router) are described in about 6,300 lines Verilog HDL codes. Although whole the memory system of flipSyrup includes multiple definitions of many similar components, it is described in about 38,000 lines of Verilog HDL, in the case of 24-core.

4.8.2 Simulation Speed

I evaluated the performance of a simulation environment with flipSyrup by using some performance counters implemented for each cache memory. Figure 4.14 shows the normalized cycle count on an

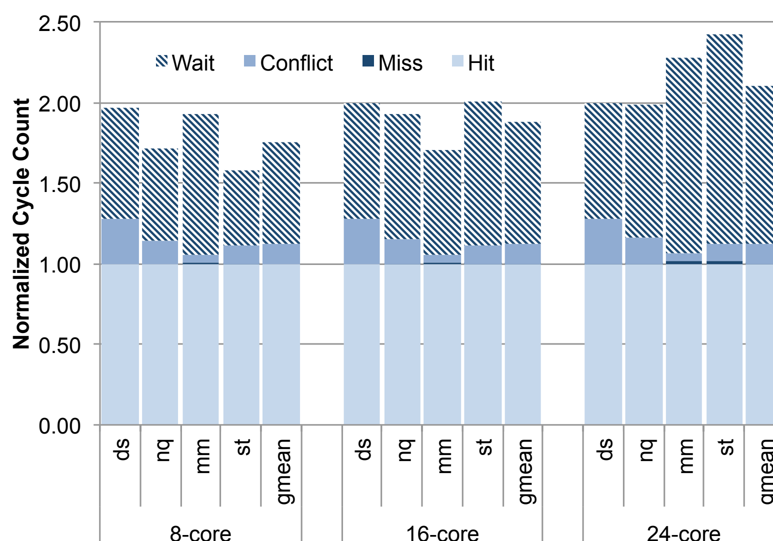


Figure 4.14 Simulation Speed of Single FPGA Simulator with flipSyrup

actual FPGA that elapsed for the execution for each configuration. The each vertical value is normalized by the elapsed FPGA-side clock cycle count on the same clock frequency operation on an ideal FPGA with infinite on-chip memory.

If the FPGA with infinite on-chip memory is used, any memory parts with any ports and capacity can be directly implemented on the FPGA. Hence there are no stalls by any cache misses or port-conflicts. In contrast, using virtual memory blocks of flipSyrup memory system on an actual FPGA, it increases the simulation cycle count due to stalls caused by their cache misses or port-conflicts.

In order to analyze the reasons for simulation slowdown, I classified the elapsed cycle count into 4 parts as follows by using the performance counters. Note that the performance counter value of each cache memory is different because each cache memory is corresponding to each scratchpad memory on the simulated hardware, and each core executes a program on a different path for each other. As representing value for each cause of slowdown, I used a geometrical mean for all cache memories.

- Hit: The cycle count where the simulated hardware is running without any stalls. This value is same as the number of total elapsed cycles on an ideal FPGA with infinite on-chip memory in the same clock frequency.
- Miss: The cycle count where the simulated hardware is being stalled due to misses of the own cache memory.
- Conflict: The cycle count where the simulated hardware is being stalled due to port-conflicts on the own cache memory for multiple memory requests.
- Wait: The cycle count where the simulated hardware is being stalled to wait for the other cache

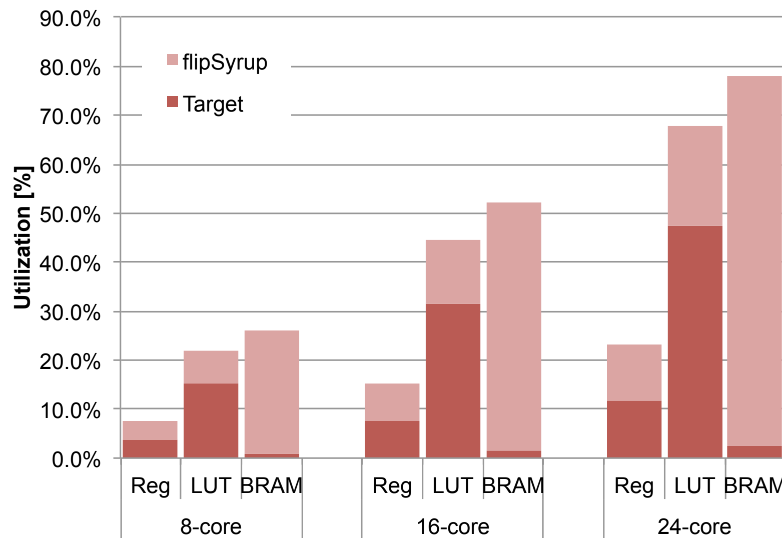


Figure 4.15 Resource Utilization Breakdown of Single FPGA Simulator with flipSyrup

memory that their requests are in processing.

In 8-core, flipSyrup-based system simulates the target hardware in 1.79x longer clock cycles in average with cycle-accurate manner than the ideal FPGA with infinite on-chip memory. In other words, the average FPGA-side clock cycle rate to simulate 1 clock cycle on the simulated hardware with its cycle-accuracy is 1.79. In 16-core, the rate is 1.91, and in 24-core, the rate is 2.16, respectively. These evaluation results show that employing a flipSyrup memory system can simulate an ideal hardware assuming infinite on-chip memory in about 2x clock cycles.

However, it increases the clock cycles by the increase of the target core count. I discuss the reasons based on the performance counter values. In any cases, compared to the cycle count without stalls, the rate of stalls by cache misses (Miss) is very low. Rates of stalls by port-conflicts (Conflict) are not very increased by increasing the core count: 12.0% in 8-core, 12.5% in 16-core and 12.6% in 24-core, respectively. In contrast, rates of stalls for waiting for memory access completion of the other cache memories are increased according to the core count: 63.5% in 8-core, 74.5 in 16-core and 98.3% in 24-core, respectively. In flipSyrup memory system, any cache misses or port-conflicts incur some stalls for entire interfaces in the same domain. Therefore the larger design with more interfaces on a single domain is more sensitive to stalls from the other cache memories. In order to decrease the stall rate, it is better to replace the current single-port caches with multi-port caches that can accept multiple requests at a time, described in [116].

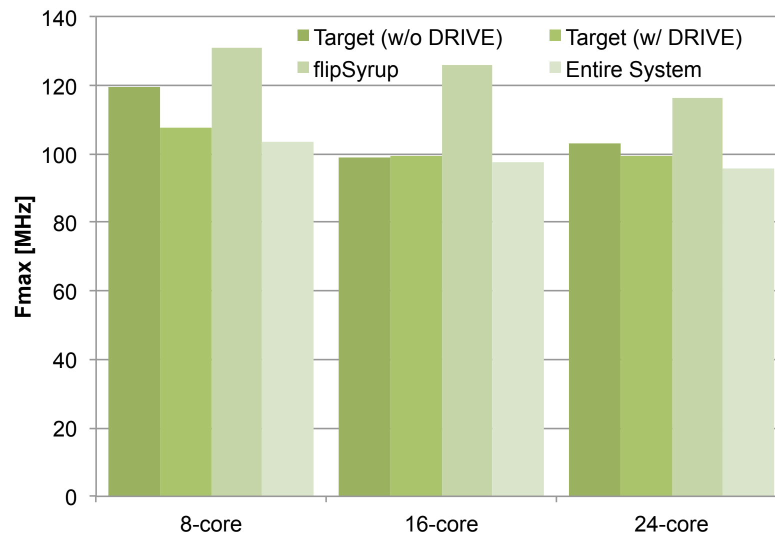


Figure 4.16 Maximum Frequency of Each Module

4.8.3 Resource Utilization

Figure 4.15 shows the resource utilization of register, LUT and on-chip memory, for each configuration. By increasing the number of cores, the amount of occupied resources for both the simulated hardware and the flipSyrup memory system is increased linearly. Compared to the resource amount of the simulated hardware, the flipSyrup memory system occupied almost same amount of registers and about 43% of LUTs in any cases.

We assigned 2MB capacity of block RAMs for the flipSyrup cache memories in all cases, so that the utilized amount is constant in any cases. However, the amount of utilized block RAMs is also increased by increasing of the number of cores. It is thought to be due to the data RAM structure of cache that is bank-partitioned for each byte in a cache line. In order to reduce memory fabric consumption, it requires a more sophisticated data RAM structure that FPGA resource characteristics are considered.

4.8.4 Clock Frequency

Figure 4.16 shows the maximum operation frequency for each configuration. We evaluated the frequencies of independent modules: a flipSyrup memory system, a simulated hardware and the overall system that we implemented on an FPGA. Each frequency value is calculated in logic synthesis phase in the Xilinx ISE.

First of all, in order to evaluate the frequency degradation caused by the addition of a throttling signal, we compare the two configurations, the raw simulated hardware (Target w/o DRIVE) and the modified

simulated hardware with throttling signal (Target w/ DRIVE). With the addition of a throttling signal, the maximum frequency has declined 10.1% in 8-core, and 3.6% in 24-core, respectively. However, the frequency is increased 0.3% in 16-core. These values are in estimation of the synthesis tool. There are some margins from the actual maximum operation frequency that determined after the place and routes. In average, the maximum frequency has declined 4.5%. These results show that the frequency degradation by the addition of a throttling signal is compact and tolerable.

Maximum frequency of independent flipSyrup memory system is decreased by increase of the number of cores. It is likely to be caused by the increase of the number of ports in the off-chip memory arbiter and the number of signals that observed by the cycle-Accuracy manager to generate a throttling signal.

For simplicity of development, all memory ports to the off-chip memory are integrated in to a single interface with an internal arbiter in this implementation. In order to reduce the degradation due to the off-chip memory arbiter, it is better that the flipSyrup cache entities are directly connected to a device-level interconnect (such as AXI). It is also better that the arbitration of requests to the off-chip memory is delegated to the interconnect arbiter.

4.8.5 Summary of Evaluation on a Signle FPGA Platform

I evaluated the framework on a standard FPGA platform with a single and large FPGA. The evaluation result shows that behavior of the multicore can be simulated at about half speed of ideal FPGA with infinite on-chip memory, by using an existing FPGA with the synthesized memory system and controllers of the tool-chain.

4.9 Many-core Processor Simulation on a Multi-FPGA Platform

The second evaluation uses ScalableCore system as a multi-FPGA based prototyping environment. In this situation, the simulated processor is partitioned into each core, as well the original ScalableCore system. By using flipSyrup, the manual scheduling of hardware simulation and local barrier synchronization is not required. Instead, the tool-chain of flipSyrup automatically synthesizes such synchronization logics and simulation logics.

This evaluation is very important, because the configuration of this evaluation includes both main contributions of this thesis: the acceleration method of cycle-accurate processor simulations on multi-FPGA based platforms; and the design methodology under the resource abstraction of FPGA platforms, in order to reduce the development complexity of FPGA-based processor simulators.

Table 4.2 Evaluation Setup

Implementation Environment	
Development tool	Xilinx ISE 14.6
FPGA	Xilinx Spartan-6 XC6SLX16 for each ScalableCore Unit
On-chip memory (Block RAM)	18 Kbit × 32 blocks, Maximum 576 Kbit
Off-chip memory (SRAM)	8-bit × 512K entry
Operation frequency	40MHz (Target processor and flipSyryp system)
flipSyrup cache	16byte per line, 1-way (Direct map), 1-outstanding miss, no-prefetching
Target Processor Architecture	
Architecture	M-Core architecture[28] (DMA-based many-core processor)
Core	MIPS32 ISA, 6-stage pipeline, single-issue, 2-memory-port (Fetch, Load/Store)
DMA Controller	2-memory-port (32-bit DMA Read and DMA Write)
Network topology	2-D mesh
On-chip router	4-stage pipeline, 1-virtual-channel, FIFO depth: 4 Credit-base flow control, X-Y Dimension Order Routing
Node memory	512KB (per Node), 32-bit width, access latency: 1 4-port (Fetch, Load/Store, DMA Read, DMA Write)
Number of Nodes	8, 16, 32, 64, 128
Benchmark	n-queens solver (nq) and matrix multiply (mm)

4.9.1 Evaluation Methodology

I evaluated flipSyrup with ScalableCore system, a multi-FPGA platform for processor prototyping. I used M-Core architecture again as a prototyping target. The configuration of this evaluation is listed in Table 4.2. The processor configuration is nearly same as the evaluation of the original ScalableCore system. However, due to the hardware resource consumption, the number of virtual channels on the on-chip network is lowered to 1.

Base on the development flow of flipSyrup, I manually partitioned the entire M-Core processor design for each FPGA unit. Since the configuration of every Node is identical, the identical IP-core is used for all the FPGA units. In order to verify the simulation performance differences by the number of FPGAs, I test 4 configurations of the core count of M-Core: 16, 32, 64 and 128. As well as the original evaluation of ScalableCore system, I used 2 benchmarks: N-Queen (NQ) and Matrix Multiplication (MM).

4.9.2 Simulation Speed

Figure 4.17 shows the obtained simulation speed for each configuration. The evaluation result shows that the number of FPGAs and the behavior of benchmarks do not affect the achievable performance. In all cases, simulation speeds are 1111 [KHz], which are almost same as the original ScalableCore system.

The reason for this stability and fast simulation speed is that inter-FPGA communications are the

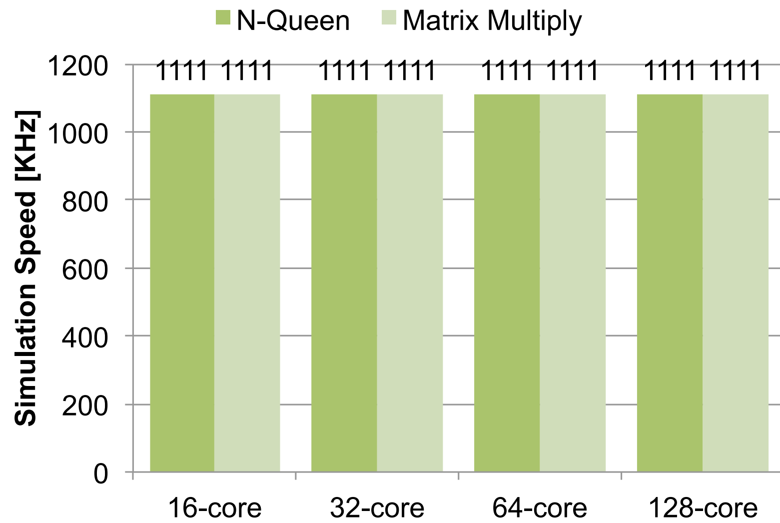


Figure 4.17 Simulation Speed of ScalableCore System with flipSyrup

bottleneck of simulation performance, in spite of flipSyrup generates on-chip cache systems to improve the simulation performance. If FPGAs have faster interconnections, the bottleneck of simulation speeds will be changed to the memory system on simulation platforms.

The multi-FPGA based with flipSyrup achieves the equivalent performance compared to the manual-scheduled ScalableCore system. This result shows that the integrated framework of ScalableCore system and flipSyrup improves the prototyping efficiency for emerging many-core processors by providing the sufficient simulation speed and the effective abstraction reducing the development complexity.

4.9.3 Resource Utilization

Figure 4.18 shows the breakdown of the resource utilizations of ScalableCore Unit with flipSyrup abstractions. Part of *target* corresponds to the simulated processor. Part of *system* corresponds to the hardware units of ScalableCore system. Part of *flipSyrup* corresponds to the simulation functions generated by the tool-chain.

The flipSyrup system consumes 12.0% LUTs and 7.2% registers of the entire FPGA resource. Compared to the target processor that consumes 59.7% LUTs, these resource impacts are not serious. Actually, the generated system by flipSyrup consumes a lot of BRAMs, due to the system level cache memory. Since designers can control the amount of used BRAMs via the parameters to identify the Syrup memory personality, this is not also a serious problem.

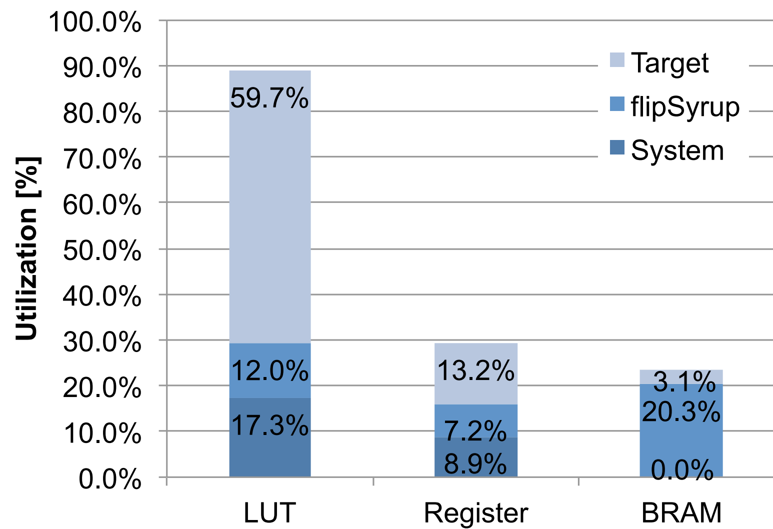


Figure 4.18 Resource Utilization Breakdown of ScalableCore System with flipSyrup

4.9.4 Summary of Evaluation on a Multi-FPGA Platform

I evaluated flipSyrup on *ScalableCore system* as a multi-FPGA based platform. In this situation, the simulated processor is partitioned for each core, as well as the original implementation of ScalableCore system. By using flipSyrup, the manual scheduling of hardware simulation and local barrier synchronization is not needed. Instead, flipSyrup automatically synthesizes such a synchronization hardware system with adequate RTL translations.

The evaluation result shows that ScalableCore system with flipSyrup system achieves equivalent performance compared to the manual-scheduled ScalableCore system. The tool-chain of flipSyrup improves the programmability of FPGA-based processor prototyping with keeping the coordinative performance.

This evaluation is very important for this thesis. The result shows that the integrated framework of two main contributions aggressively improves the prototyping efficiency for emerging many-core processors by providing the sufficient simulation speed and the effective abstractions to reduce development complexity.

4.10 Comparison with Other Methodology

Some researchers have proposed attractive frameworks for effortless implementation of a coordinated memory system on reconfigurable logics.

Yiannacouras et al.[117] proposed an automatic cache generation tool for FPGA-based computing.

The tool supports various configurations of caches, and can effortlessly create different caches to meet the target system.

Closely related research to our work is LEAP scratchpads[118, 119]. LEAP is just an integrated framework to manage data placement on hierarchical caches of on-chip memory, off-chip DRAM and host-side memory. Similar to our work, LEAP provides a simple interface to access the abstracted memory system of hierarchical caches for user logics. Memory space limitation caused by small on-chip memory capacity is eliminated so that the user logic can use the large space of memory.

CoRAM[120] and PyCoRAM[111] are memory abstraction frameworks for reconfigurable computing. The feature of CoRAM is that data movements between on-chip memory and off-chip memory are explicitly managed by control threads, user-defined state machines represented in the software model.

In processor prototyping, these memory abstraction frameworks can expand the available memory capacity used for emulations. However, such caching and replacement mechanisms in the memory system are not sufficient to sustain the cycle-accuracy of logic emulation.

The latency of on-chip memory in most FPGAs is very short, which is usually 1 cycle. So it is the same for implement optimized computing logics with assumption of the fixed single-cycle latency. However, the typical latency to complete a memory request to the off-chip memory is variable and long, due to the behavior of DRAM controllers and intermediate interconnects. In order to exactly emulate the hardware with fixed single-cycle memories on an abstracted memory environment, a kind of synchronization mechanism to throttle the emulated hardware behavior for each clock cycle is required.

The tool-chian of flipSyrup provides a fast and huge memory space for user-level RTL design, by employing on-chip memory fabrics on FPGA as a fast cache memory via simple accessing interface. The major difference from previous works is that FlipSyrup framework also provides a synchronization controller managing all memory requests from the user-level design and throttling its behavior when any cache misses or port-conflictions occur.

4.11 Summary

In this chapter, I propose a novel design methodology under the abstraction of various resources on FPGAs. Hardware components within FPGAs, such as memory block and communication interfaces, are abstracted and given to simulator designers. This methodology enables designers to model a prototyping target processor without concern for actual platform resources.

To this end, first I developed Pyverilog, an open-source toolkit for design analysis and code generation of RTL designs written in Verilog HDL. Pyverilog offers (1) code parser, (2) dataflow analyzer, (3) control-flow analyzer, (4) visualizer and (5) code generator for Verilog HDL. Then I developed a Python-based software tool-chain that realizes the proposed methodology. I described the implementation features of the tool-chain.

I evaluated simulation speed under the resource abstraction on the single large FPGA platform. The evaluation result shows that the simulation speed degradation under the abstraction is not critical so that the abstraction tool-chain offers the helpful support to develop a high-speed processor simulator rapidly.

Finally I evaluated the integrated framework of the two contributions in this thesis: scalable simulation accelerator and the abstraction methodology. The evaluation result shows that the simulation system automatically synthesized by the abstraction tool-chain archives almost equivalent performance to manual-tuned multi-FPGA based simulator. The integrated framework aggressively improves the prototyping efficiency for emerging many-core processors by providing the sufficient simulation speed and the effective abstraction reducing the development complexity.

Chapter 5

Conclusion

5.1 Concluding Remarks

In this thesis, I proposed a sophisticated prototyping framework for future many-core processor evaluations. The framework comprises of the acceleration method of many-core processor simulations and the design methodology to decrease the development complexity.

The contributions in this thesis are as follows:

- to propose an FPGA-based simulation method which achieves the scalable simulation speed against the increasing core count of simulated processors with the cycle-level accuracy of simulation results;
- to show that the proposed simulation method is feasible by designing and developing an actual multi-FPGA based acceleration system;
- to propose a novel design methodology under the physical resource abstraction of FPGAs, with its software tool-chain to improve the efficiency of prototyping system development; and
- to present and evaluate the integrated framework that wraps up the multi-FPGA based simulation method and the prototype design methodology.

I proposed a system architecture of fast and cycle-accurate processor simulator employing multiple FPGAs, focused on the structure of the on-chip-network and the memory systems on a many-core processor for future many-core processors with over 100 cores. I developed a test bed platform of multiple FPGAs to evaluate the viability of the proposed method. I evaluated the proposed method by using the test bed system in point of simulation speed. The evaluation result shows that the proposed method achieves effective scalability of the simulation speed to simulate a large scale many-core processor with keeping the cycle-accuracy of the simulation consequences.

I present a novel design methodology under the abstraction of various resources FPGA platforms have, such as memory systems and inter-FPGA interconnections. I developed the Python-based design tool-chain that automatically synthesizes ready-to-implement RTL designs for actual FPGA platforms

from target RTL descriptions under the abstraction. This methodology enables designers to model a prototyping target processor without concern for actual platform resources. I evaluated simulation speed under the abstraction using a standard FPGA platform with large capacity of logic and memory. The evaluation result shows that the simulation speed degradation under the abstraction is not critical so that the abstraction tool-chain offers the helpful support to develop a high-speed processor simulator rapidly.

Finally I evaluated the integrated framework of the two contributions, scalable simulation accelerator and the abstraction methodology. The evaluation result shows that the simulation system automatically synthesized by the abstraction tool-chain archives almost equivalent performance to manual-tuned multi-FPGA based simulator. The integrated framework aggressively improves the prototyping efficiency for emerging many-core processors by providing the sufficient simulation speed and the effective abstraction reducing the development complexity.

5.2 Open Research Topics

There are several remaining topics from this research. I describe some of them as follows:

- to develop a method for automatic design partitioning of simulated RTL design under the flipSyrup abstractions;
- to adopt flipSyrup abstractions for general FPGA-based computing; and
- to integrate flipSyrup abstractions to the multithreaded simulation technique.

The current implementation of the flipSyrup tool-chain requires manual partitionings of the simulated processors. The first topic is that to explore an automatic partitioning way of the entire simulated processor. The prior researches for multi-FPGA based logic emulations had proposed various clustering algorithms the input logic designs into multiple parts. By using these approaches, the automatic partitioning is possible.

In this thesis, I evaluated the tool-chain of flipSyrup for the processor prototyping purposes. However, architecture and methodology of flipSyrup are not actually limited to the processor prototyping purpose. Cache-based computing on FPGAs will improve the development efficiency as well as processor prototyping. Additionally, employing a more sophisticated cache architecture will improve the performance of general applications, if flipSyrup is adopted to the general computing on FPGAs.

The last open research topic is to integrate the methodology of flipSyrup with simulation-oriented approaches. As I described in the background chapter, simuation-oriented approaches require additional implementations of different RTL designs just for the simulation. I believe that we can realize a stylish way to automatically synthesize the simulation-oriented RTL designs from the pure processor RTL designs, by using my flipSyrup tool-chain and Pyverilog toolkit.

Acknowledgement

This work has been supported in part by JST CREST and the fellowship program of JSPS.

My delightful research life for 6 years, including 1 year of my bachelor course, 2 years of my master course and 3 years of my doctor course, has received the support of a lot of people. First of all, I would like to say my great gratitude to Associate Professor Kenji Kise. He has been my supervisor and has supported me for 6 years. I feed his remarkable breadth of mind from his continuous guidance for a self-willed and insolent student like me. Thanks to precious chances he gave me and his personal magnetism, I have got a lot of friends of distinguished researchers in both in domestic and international, and really have enjoyed my research life. I would like to appreciate him again.

I also would like to thank all the members at Kise Laboratory, particularly Mr. Shimpei Sato and Mr. Shimpei Watanabe, Dr. Naoki Fujieda, Mr. Jiang Xuan, Mr. Yoshito Sakaguchi, Mr. Shintaro Sano and Mr. Ryosuke Sasakawa for their help of research and development of ScalableCore system.

I would like to sincerely thank Professor Haruo Yokota, Professor Jun Miyazaki, Associate Professor Takuo Watanabe and Associate Professor Haruhiko Kaneko for many exact indications and suggestions as members of the thesis committee.

I would like to thank Professor James C. Hoe at Carnegie Mellon University for his sound guidance about FPGA systems when I stayed CMU as a visiting scholar.

I would like to thank Mr. Takefumi Miyoshi in e-trees.Japan, Inc for his cheerful advises about my research and my life since I was a master course student. I will defeat you someday.

I would like to acknowledge the doctor and everyone of Jiseikai Ukima Funado hospital who rescued me when I dropped in spite of the deadline of this thesis.

I would like to thank all those who I met at IPSJ SIG-ARC, IEICE SIG-RECONF and IEICE SIG CPSY for their aggressive discussion with me. I would like to thank particularly Mr. Takaaki Miyajiama, Mr. Keisuke Dohi and Mr. Toshiya Komoda for innovative and crazy discussions without any constraints. Let's continue on our fantastic works.

Mr. Masahito Ohue is a friend of mine in the same affiliation, since we were 3rd grade students of bachelor. He is the closest rival researcher while we are working on different research areas. I would like to thank him for good competitions. Take our best as a competent researcher.

Both my families of Takamaedas and Yamazakis have supported my research life from my faraway

hometown. I would wish to express my profound gratitude to them.

Finally, I would like to express my deepest gratitude to Kaori, my dearest wife. Thank you always for everything. I'll always be with you.

本研究の一部は，科学技術振興機構・戦略的創造研究推進事業 (CREST) の「アーキテクチャと形式的検証の協調による超ディペンダブル VLSI」および「ディペンダブルネットワークオンチッププラットフォームの構築」の支援によります．また，本研究の一部は，日本学術振興会の特別研究員制度の支援によります．

学部 4 年生の 1 年間，修士課程の 2 年間，そして博士課程の 3 年間，合計 6 年間，充実した研究生活を送ることができたのは，非常に沢山の方のご支援によるものです．まず，終始熱心なご指導を賜りました，指導教員の吉瀬謙二准教授に深く感謝に意を表します．私のような，我が強く，言うことを聞かない生意気な学生の面倒を 6 年間も見ていただいたことに，吉瀬先生の懐の広さを感じております．吉瀬先生が与えてくれた機会と，吉瀬先生の人望のおかげで，研究会や国際会議など国内外問わず，様々な場で沢山の研究者と知り合うことができ，非常に充実した楽しい研究生活を送ることができました．ここに改めて感謝申し上げます．

また，吉瀬研究室の皆様には，活発な議論と多くの貴重な意見を頂き，本当にお世話になりました．特に，ScalableCore システムに関する研究を遂行する上で，佐藤真平氏，渡邉伸平氏，藤枝直輝氏，姜軒氏，坂口嘉一氏，佐野伸太郎氏，笹河良介氏には多大なるご協力を頂きました．ここに深く感謝いたします．

本博士論文の審査員として，横田治夫教授，宮崎純教授，渡部卓雄准教授，金子晴彦准教授には，多くの的確な指摘と提案をしていただきました．ここに深く感謝いたします．

カーネギーメロン大学への留学の際に，James C. Hoe 教授には FPGA システム研究について，非常に熱心で刺激的なご指導をして頂きました．ここに深く感謝いたします．

株式会社イーツリーズ・ジャパンの三好健文氏には，修士課程 1 年生のときより，良き先輩として，研究から生き様まで，本当に沢山のことを教わりました．ここに深く感謝いたします．そして，いつか倒します．

健康面では，年末の博士論文の提出間際に，扁桃炎で倒れかけた私を助けてくれた，慈誠会浮間舟渡病院の皆様に感謝いたします．

研究室以外の研究の場として，情報処理学会 計算機アーキテクチャ研究会 (ARC)，電子情報通信学会 リコンフィギャラブルシステム研究会 (RECONF) および電子情報通信学会コンピュータシステム研究会 (CPSY) で出会い，活発な議論をした皆様に感謝いたします．特に，博士課程の同期である，慶應義塾大学の宮島敬明氏，長崎大学の土肥慶亮氏，東京大学の鳶田登志矢氏とは，研究室の垣根を越えた，しがらみのない議論をすることができました．ありがとうございました．これからも良き研究者仲間として，共に頑張りましょう．

学部 3 年次編入学より，修士課程，そして博士課程まで同じ学科・専攻の同期であった大上雅史氏とは，分野は異なるものの，最も身近なライバルとして切磋琢磨することができました．ここに感謝いたします．これからも研究者仲間として頑張りましょう．

私の自由気ままな研究生活を，遠くから支えてくれた，高前田家および山崎家の両家家族に深く感謝いたします．

最後に，良き伴侶として，私の研究生活を最も傍で支えてくれた，最愛なる妻のかおりに心より感謝の意を表します．いつも本当にありがとう．そして，これからもよろしく．

Bibliography

[1] A. Patel, F. Afram, Shunfei Chen, and K. Ghose. Marss: A full system simulator for multicore x86 cpus. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pp. 1050–1055, 2011.

[2] Standard performance evaluation corporation. http://www.spec.org/.

[3] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pp. 72–81, New York, NY, USA, 2008. ACM.

[4] Vivado design suite. http://www.xilinx.com/products/design-tools/vivado/index.htm.

[5] Xilinx Platform Studio. http://www.xilinx.com/tools/xps.htm.

[6] A. Khan, M. Vijayaraghavan, S. Boyd-Wickizer, and Arvind. Fast and cycle-accurate modeling of a multicore processor. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pp. 178–187, 2012.

[7] Kenji Kise, Takahiro Katagiri, Hiroki Honda, and Toshitsugu Yuba. The simcore/alpha functional simulator. In *Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture*, WCAE '04, New York, NY, USA, 2004. ACM.

[8] James R Larus. *Spim s20: A mips r2000 simulator*. Center for Parallel Optimization, Computer Sciences Department, University of Wisconsin, 1990.

[9] Irina Branovic, Roberto Giorgi, and Enrico Martinelli. Webmips: A new web-based mips simulation environment for computer architecture education. In *Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture*, WCAE '04, New York, NY, USA, 2004. ACM.

[10] Naoki Fujieda, Takefumi Miyoshi, and Kenji Kise. SimMips: A MIPS System Simulator. pp. 32–39, 2009.

[11] Naoki Fujieda, Shimpei Watanabe, and Kenji Kise. A MIPS System Simulator SimMips for Education and Research of Computer Science (in Japanese). *IPSJ Journal*, Vol. 50, No. 11, pp. 2665–2676, nov 2009.

[12] Fabrice Bellard. Qemu open source processor emulator. *URL: http://www. qemu. org*, 2007.

[13] S. Raghav, M. Ruggiero, D. Atienza, C. Pinto, A. Marongiu, and L. Benini. Scalable instruction set simulator for thousand-core architectures running on gpgpus. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pp. 459 –466, 28 2010-july 2 2010.

[14] T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *Computer*, Vol. 35, No. 2, pp. 59–67, 2002.

[15] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, Vol. 26, No. 4, pp. 52–60, 2006.

[16] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, Vol. 35, No. 2, pp. 50–58, 2002.

[17] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, Vol. 33, No. 4, pp. 92–99, 2005.

[18] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, Vol. 39, pp. 1–7, August 2011.

[19] Ehsan K. Ardestani and Jose Renau. Esesc: A fast multicore simulator using time-based sampling. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 448–459, 2013.

[20] Manifold. http://manifold.gatech.edu/.

[21] Aamer Jaleel, Robert S. Cohn, Chi keung Luk, and Bruce Jacob. Cmp$im: A pin-based on-the-fly multi-core cache simulator.

[22] Zhenman Fang, Qinghao Min, Keyong Zhou, Yi Lu, Yibin Hu, Weihua Zhang, Haibo Chen, Jian Li, and Binyu Zang. Transformer: A functional-driven cycle-accurate multicore simulator. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pp. 106–114, New York, NY, USA, 2012. ACM.

[23] Pengju Ren, M. Lis, Myong Hyon Cho, Keun Sup Shim, C.W. Fletcher, O. Khan, Nanning Zheng, and S Devadas. Hornet: A cycle-level multicore simulator. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, Vol. 31, No. 6, pp. 890–903, June 2012.

[24] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simula-

tion of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pp. 475–486, New York, NY, USA, 2013. ACM.

[25] Shimpei Sato, Naoki Fujieda, Akira Moriya, and Kenji Kise. SimCell: A Processor Simulator for Multi-Core Architecture Research. *IPSJ Transactions on Advanced Computing Systems*, Vol. 2, No. 1, pp. 146–157, mar 2009.

[26] Gpgpu-sim. http://www.gpgpu-sim.org/.

[27] Koh Uehara, Shimpei Sato, Takefumi Miyoshi, and Kenji Kise. A Study of an Infrastructure for Research and Development of Many-Core Processors. In *Workshop on Ultra Performance and Dependable Acceleration Systems held in conjunction with PDCAT'09*, pp. 414–419, 2009.

[28] Koh Uehara, Shimpei Sato, and Kenji Kise. A Practical Infrastructure for Research and Education of Many-Core Processors (in Japanese). *IEICE Journal D*, Vol. 93, No. 10, pp. 2042–2057, oct 2010.

[29] Nan Jiang, D.U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D.E. Shaw, J. Kim, and W.J. Dally. A detailed and flexible cycle-accurate network-on-chip simulator. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pp. 86–96, 2013.

[30] Hang-Sheng Wang, Xinping Zhu, Li-Shiuan Peh, and S. Malik. Orion: a power-performance simulator for interconnection networks. In *Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, pp. 294–305, 2002.

[31] Joel Hestness, Boris Grot, and Stephen W. Keckler. Netrace: Dependency-driven trace-based network-on-chip simulation. In *Proceedings of the Third International Workshop on Network on Chip Architectures*, NoCArc '10, pp. 31–36, New York, NY, USA, 2010. ACM.

[32] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pp. 83–94, 2000.

[33] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pp. 469–480, New York, NY, USA, 2009. ACM.

[34] Gpuwattch. http://www.gpgpu-sim.org/gpuwattch/.

[35] Synopsys vcs. http://www.synopsys.com/products/simulation/simulation.html.

[36] Caence. http://www.cadence.com/us/pages/default.aspx.

[37] Mentor modelsim. http://www.mentor.com/products/fpga/model.

[38] Icarus verilog. http://iverilog.icarus.com/.

[39] Gplcver. http://gplcver.sourceforge.net/.

[40] Verilator. http://www.veripool.org/wiki/verilator.

[41] Ghdl. http://ghdl.free.fr/.

[42] Rishiyur Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pp. 69–70, 2004.

[43] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pp. 1216–1225, New York, NY, USA, 2012. ACM.

[44] P. Bellows and B. Hutchings. Jhdl-an hdl for reconfigurable systems. In *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pp. 175–184, 1998.

[45] J. I. Villar, J. Juan, M.J. Bellido, J. Viejo, D. Guerrero, and J. Decaluwe. Python as a hardware description language: A case study. In *Programmable Logic (SPL), 2011 VII Southern Conference on*, pp. 117–122, 2011.

[46] S. Sato and K. Kise. Archhdl: A new hardware description language for high-speed architectural evaluation. In *Embedded Multicore Socs (MCSoC), 2013 IEEE 7th International Symposium on*, pp. 107–112, 2013.

[47] Laurence William Nagel and Donald O Pederson. *SPICE: Simulation program with integrated circuit emphasis*. Electronics Research Laboratory, College of Engineering, University of California, 1973.

[48] Hspice. http://www.synopsys.com/tools/Verification/AMSVerification/CircuitSimulation/HSPICE/Pages/defa

[49] Zhangxi Tan, Andrew Waterman, Henry Cook, Sarah Bird, Krste Asanović, and David Patterson. A case for fame: Fpga architecture model execution. *SIGARCH Comput. Archit. News*, Vol. 38, No. 3, pp. 290–301, June 2010.

[50] M.A. Kinsy, M. Pellauer, and S Devadas. Heracles: Fully synthesizable parameterized mips-based multicore system. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pp. 356–362, 2011.

[51] Michel A. Kinsy, Michael Pellauer, and Srinivas Devadas. Heracles: A tool for fast rtl-based design space exploration of multicore processors. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pp. 125–134, New York, NY, USA, 2013. ACM.

[52] Nehir Sonmez, Oriol Arcas, Gokhan Sayilar, Osman Unsal, Adrian Cristal, Ibrahim Hur, Satnam Singh, and Mateo Valero. From plasma to beefarm: Design experience of an fpga-based multicore prototype. In Andreas Koch, Ram Krishnamurthy, John McAllister, Roger Woods, and Tarek El-Ghazawi, editors, *Reconfigurable Computing: Architectures, Tools and Applications*,

Vol. 6578 of *Lecture Notes in Computer Science*, pp. 350–362. Springer Berlin / Heidelberg, 2011.

[53] N. Njoroge, J. Casper, S. Wee, Y. Teslyar, D. Ge, C. Kozyrakis, and K. Olukotun. Atlas: A chip-multiprocessor with transactional memory support. In *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pp. 1–6, 2007.

[54] Huandong Wang, Xiang Gao, Yunji Chen, Dan Tang, and Weiwu Hu. A multi-fpga based platform for emulating a 100m-transistor-scale processor with high-speed peripherals (abstract only). In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10, pp. 283–283, New York, NY, USA, 2010. ACM.

[55] S. Lyberis, G. Kalokerinos, M. Lygerakis, V. Papaefstathiou, D. Tsaliagkos, M. Katevenis, D. Pnevmatikatos, and D. Nikolopoulos. Formic: Cost-efficient and scalable prototyping of manycore architectures. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pp. 61–64, 2012.

[56] Xinyu Li and O. Hammami. Multi-fpga emulation of a 48-cores multiprocessor with noc. In *Design and Test Workshop, 2008. IDT 2008. 3rd International*, pp. 205–208, 2008.

[57] Sameh Asaad, Ralph Bellofatto, Bernard Brezzo, Chuck Haymes, Mohit Kapur, Benjamin Parker, Thomas Roewer, Proshanta Saha, Todd Takken, and José Tierno. A cycle-accurate, cycle-reproducible multi-fpga system for accelerating multi-core processor simulation. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pp. 153–162, New York, NY, USA, 2012. ACM.

[58] K. Banovic, M. A S Khalid, and E. Abdel-Raheem. Fpga-based rapid prototyping of digital signal processing systems. In *Circuits and Systems, 2005. 48th Midwest Symposium on*, pp. 647–650 Vol. 1, 2005.

[59] Y. Nakamura, K. Hosokawa, I. Kuroda, K. Yoshikawa, and T. Yoshimura. A fast hardware/software co-verification method for systern-on-a-chip by using a c/c++ simulator and fpga emulator with shared register communication. In *Design Automation Conference, 2004. Proceedings. 41st*, pp. 299–304, 2004.

[60] Yuichi Nakamura and Kouhei Hosokawa. Fast fpga-emulation-based simulation environment for custom processors. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, Vol. E89-A, No. 12, pp. 3464–3470, December 2006.

[61] Ari Kulmala, Erno Salminen, and Timo D. Hämäläinen. Evaluating large system-on-chip on multi-fpga platform. In *Proceedings of the 7th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS'07, pp. 179–189, Berlin, Heidelberg, 2007. Springer-Verlag.

[62] P. Meloni, S. Secchi, and L. Raffo. An fpga-based framework for technology-aware prototyping

of multicore embedded architectures. *Embedded Systems Letters, IEEE*, Vol. 2, No. 1, pp. 5–9, 2010.

[63] S. Secchi, P. Meloni, and L. Raffo. Exploiting fpgas for technology-aware system-level evaluation of multi-core architectures. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pp. 194–202, 2010.

[64] S. Hauck and G. Borriello. Pin assignment for multi-fpga systems. In *FPGAs for Custom Computing Machines, 1994. Proceedings. IEEE Workshop on*, pp. 11–13, 1994.

[65] R. Burra and D. Bhatia. Timing driven multi-fpga board partitioning. In *VLSI Design, 1998. Proceedings., 1998 Eleventh International Conference on*, pp. 234–237, 1998.

[66] Scott Hauck and Gaetano Borriello. Logic partition orderings for multi-fpga systems. In *Proceedings of the 1995 ACM Third International Symposium on Field-programmable Gate Arrays*, FPGA '95, pp. 32–38, New York, NY, USA, 1995. ACM.

[67] C. Bolchini and C. Sandionigi. A reliability-aware partitioner for multi-fpga platforms. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2011 IEEE International Symposium on*, pp. 34–40, 2011.

[68] S. Hauck, G. Borriello, and C. Ebeling. Mesh routing topologies for multi-fpga systems. In *Computer Design: VLSI in Computers and Processors, 1994. ICCD '94. Proceedings., IEEE International Conference on*, pp. 170–177, 1994.

[69] Wen-Jong Fang and A.C.-H. Wu. Integrating hdl synthesis and partitioning for multi-fpga designs. *Design Test of Computers, IEEE*, Vol. 15, No. 2, pp. 65–72, 1998.

[70] M. Vootukuru, R. Vemuri, and N. Kumar. Resource constrained rtl partitioning for synthesis of multi-fpga designs. In *Proceedings of the Tenth International Conference on VLSI Design: VLSI in Multimedia Applications*, VLSID '97, pp. 140–, Washington, DC, USA, 1997. IEEE Computer Society.

[71] J. Babb, R. Tessier, M. Dahl, S.Z. Hanono, D.M. Hoki, and A. Agarwal. Logic emulation with virtual wires. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, Vol. 16, No. 6, pp. 609–626, 1997.

[72] J. Babb, R. Tessier, and A. Agarwal. Virtual wires: overcoming pin limitations in fpga-based logic emulators. In *FPGAs for Custom Computing Machines, 1993. Proceedings. IEEE Workshop on*, pp. 142–151, 1993.

[73] D. Baviskar and S. Patkar. A pipelined simulation approach for logic emulation using multi-fpga platforms. In *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, pp. 1141–1144, 2009.

[74] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A. Patil, William Reinhart, Darrel Eric Johnson, Jebediah Keefe, and Hari Angepat. Fpga-accelerated simulation technologies (fast): Fast,

full-system, cycle-accurate simulators. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pp. 249–261, Washington, DC, USA, 2007. IEEE Computer Society.

[75] Eric S. Chung, Michael K. Papamichael, Eriko Nurvitadhi, James C. Hoe, Ken Mai, and Babak Falsafi. Protoflex: Towards scalable, full-system multiprocessor simulations using fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, Vol. 2, No. 2, pp. 15:1–15:32, June 2009.

[76] Zhangxi Tan, Andrew Waterman, Rimas Avizienis, Yunsup Lee, Henry Cook, David Patterson, and Krste Asanović. Ramp gold: An fpga-based architecture simulator for multiprocessors. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pp. 463–468, New York, NY, USA, 2010. ACM.

[77] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer. Hasim: Fpga-based high-detail multicore simulation using time-division multiplexing. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 406 –417, feb. 2011.

[78] Michael Pellauer, Muralidaran Vijayaraghavan, Michael Adler, Arvind, and Joel Emer. A-ports: an efficient abstraction for cycle-accurate performance models on fpgas. In *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, FPGA '08, pp. 87–96, New York, NY, USA, 2008. ACM.

[79] Muralidaran Vijayaraghavan and Arvind Arvind. Bounded dataflow networks and latency-insensitive circuits. In *Proceedings of the 7th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, MEMOCODE'09, pp. 171–180, Piscataway, NJ, USA, 2009. IEEE Press.

[80] Danyao Wang, Natalie Enright Jerger, and J. Gregory Steffan. Dart: A programmable architecture for noc simulation on fpgas. In *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '11, pp. 145–152, New York, NY, USA, 2011. ACM.

[81] Michael K. Papamichael, James C. Hoe, and Onur Mutlu. Fist: A fast, lightweight, fpga-friendly packet latency estimator for noc modeling in full-system simulations. In *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '11, pp. 137–144, New York, NY, USA, 2011. ACM.

[82] Kouadri-Mostefaoui, Abdellah-Medjadji, B. Senouci, and F. Petrot. Large scale on-chip networks : An accurate multi-fpga emulation platform. In *Digital System Design Architectures, Methods and Tools, 2008. DSD '08. 11th EUROMICRO Conference on*, pp. 3–9, 2008.

[83] A.-M. Kouadri-Mostefaoui, B. Senouci, and F. Petrot. Scalable multi-fpga platform for networks-on-chip emulation. In *Application -specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, pp. 54–60, 2007.

[84] Kuan Fang, Yufei Ni, Jiayuan He, Zonghui Li, Shuai Mu, and Yangdong Deng. Fastlanes: An

fpga accelerated gpu microarchitecture simulator. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pp. 241–248, 2013.

[85] Shinya Takamaeda-Yamazaki, Shintaro Sano, Yoshito Sakaguchi, Naoki Fujieda, and Kenji Kise. Scalablecore system: A scalable many-core simulator by employing over 100 fpgas. In *Proceedings of the 8th international conference on Reconfigurable Computing: architectures, tools and applications*, ARC'12, pp. 138–150, Berlin, Heidelberg, 2012. Springer-Verlag.

[86] Shinya Takamaeda-Yamazaki, Ryosuke Sasakawa, Yoshito Sakaguchi, and Kenji Kise. An FPGA-based Scalable Simulation Accelerator for Tile architectures. *SIGARCH Comput. Archit. News*, Vol. 39, pp. 38–43, December 2011.

[87] Kermin Elliott Fleming, Michael Adler, Michael Pellauer, Angshuman Parashar, Arvind Mithal, and Joel Emer. Leveraging latency-insensitivity to ease multiple fpga design. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pp. 175–184, New York, NY, USA, 2012. ACM.

[88] O. Mencer, Kuen Hung Tsoi, S. Craimer, T. Todman, W. Luk, Ming Yee Wong, and Philip Leong. Cube: A 512-fpga cluster. In *Programmable Logic, 2009. SPL. 5th Southern Conference on*, pp. 51–57, 2009.

[89] Chen Chang, Kimmo Kuusilinna, Brian Richards, and Robert W. Brodersen. Implementation of bee: A real-time large-scale hardware emulation engine. In *Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*, FPGA '03, pp. 91–99, New York, NY, USA, 2003. ACM.

[90] K. Sano, T. Iizuka, and S. Yamamoto. Systolic architecture for computational fluid dynamics on fpgas. In *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pp. 107–116, 2007.

[91] K. Sano, Y. Hatsuda, and S. Yamamoto. Scalable streaming-array of simple soft-processors for stencil computations with constant memory-bandwidth. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pp. 234–241, 2011.

[92] Wang Luzhou, K. Sano, and S. Yamamoto. Local-and-global stall mechanism for systolic computational-memory array on extensible multi-fpga system. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pp. 102–109, 2010.

[93] Hirokazu Morisita, Kenta Inakagata, Yasunori Osana, Naoyuki Fujita, and Hideharu Amano. Implementation and evaluation of an arithmetic pipeline on flops-2d: Multi-fpga system. *SIGARCH Comput. Archit. News*, Vol. 38, No. 4, pp. 8–13, January 2011.

[94] Takashi Yokota, Masamichi Nagafuchi, Yoshito Mekada, Tsutomu Yoshinaga, Kanemitsu Ootsu, and Takanobu Baba. Real-time medical diagnosis on a multiple fpga-based system. In Manfred

Glesner, Peter Zipf, and Michel Renovell, editors, *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, Vol. 2438 of *Lecture Notes in Computer Science*, pp. 1088–1091. Springer Berlin Heidelberg, 2002.

[95] Edward C. Lin and Rob A. Rutenbar. A multi-fpga 10x-real-time high-speed search engine for a 5000-word vocabulary speech recognizer. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '09, pp. 83–92, New York, NY, USA, 2009. ACM.

[96] N. Kapre and A. DeHon. Accelerating spice model-evaluation using fpgas. In *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on*, pp. 37–44, 2009.

[97] R. Kobayashi, S. Takamaeda-Yamazaki, and K. Kise. Towards a low-power accelerator of many fpgas for stencil computations. In *Networking and Computing (ICNC), 2012 Third International Conference on*, pp. 343–349, 2012.

[98] Hiroshi Sasaki, Teruo Tanimoto, Koji Inoue, and Hiroshi Nakamura. Scalability-based manycore partitioning. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pp. 107–116, New York, NY, USA, 2012. ACM.

[99] Hiroshi Sasaki, Satoshi Imamura, and Koji Inoue. Coordinated power-performance optimization in manycores. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pp. 51–62, Piscataway, NJ, USA, 2013. IEEE Press.

[100] S. Sano, M. Sano, S. Sato, T. Miyoshi, and K. Kise. Pattern-based systematic task mapping for many-core processors. In *Networking and Computing (ICNC), 2010 First International Conference on*, pp. 173 –178, nov. 2010.

[101] Shintaro Sano and Kenji Kise. A Task Mapping Method to Mitigate Network Contention for Many-core Processors (in Japanese). *IPSJ Transactions on Advanced Computing Systems*, Vol. 4, No. 4, pp. 96–109, Oct. 2011.

[102] S. Takamaeda, S. Sato, T. Miyoshi, and K. Kise. SmartCore System for Dependable Many-Core Processor with Multifunction Routers. In *International Conference on Networking and Computing (ICNC2010)*, pp. 133 –139, nov. 2010.

[103] Shinya Takamaeda-Yamazaki, Naoki Fujieda, and Kenji Kise. Network Performance of Multi-function On-chip Router Architectures. *IEICE SIG-CPSY 2012-11*, 2012.

[104] D. Nassimi and S. Sahni. Bitonic sort on a mesh-connected parallel computer. *IEEE Trans. Comput.*, Vol. 28, pp. 2–7, January 1979.

[105] Lynn Elliot Cannon. *A cellular computer to implement the kalman filter algorithm*. PhD thesis, Bozeman, MT, USA, 1969. AAI7010025.

[106] Amba open specifications. http://www.arm.com/products/system-ip/amba/amba-open-

specifications.php.

[107] Ug761 axi reference guide - xilinx. http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf.

[108] Ply (python lex-yacc). http://www.dabeaz.com/ply/.

[109] Jinja. http://jinja.pocoo.org/.

[110] John Ellson, Emden Gansner, Lefteris Koutsofios, StephenC. North, and Gordon Woodhull. Graphviz: Open source graph drawing tools. In *Graph Drawing*, Vol. 2265 of *Lecture Notes in Computer Science*, pp. 483–484. Springer Berlin Heidelberg, 2002.

[111] Shinya Takamaeda-Yamazaki, Kenji Kise, and James C. Hoe. Pycoram: Yet another implementation of coram memory architecture for modern fpga-based computing. In *Intersections of Computer Architecture and Reconfigurable Logic (CARL 2013)*, 2013.

[112] Antlr. http://www.antlr.org/.

[113] Reinhold P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM*, Vol. 27, No. 10, pp. 1013–1030, October 1984.

[114] Jaswinder Pal Sing, David E. Culler, and Anoop Gupta. *Parallel Computer Architecture A Hardware/Software Approach*. Morgan Kaufmann, 1998.

[115] Virtex-7 fpga vc707 evaluation kit. http://www.xilinx.com/products/boards-and-kits/EK-V7-VC707-G.htm.

[116] Jongsok Choi, K. Nam, A. Canis, J. Anderson, S. Brown, and T. Czajkowski. Impact of cache architecture and interface on performance and area of fpga-based processor/parallel-accelerator systems. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pp. 17–24, 2012.

[117] P. Yiannacouras and J. Rose. A parameterized automatic cache generator for fpgas. In *Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on*, pp. 324–327, 2003.

[118] Michael Adler, Kermin E. Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. Leap scratchpads: automatic memory and cache management for reconfigurable logic. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, pp. 25–28, New York, NY, USA, 2011. ACM.

[119] Hsin-Jung Yang, K. Fleming, M. Adler, and J. Emer. Optimizing under abstraction: Using prefetching to improve fpga performance. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pp. 1–8, 2013.

[120] Eric S. Chung, James C. Hoe, and Ken Mai. Coram: an in-fabric memory architecture for fpga-based computing. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, pp. 97–106, New York, NY, USA, 2011. ACM.

Publication

5.3 Journal Paper

1. **Shinya Takamaeda**, Shimpei Sato, Naoki Fujieda, Takefumi Miyoshi, and Kenji Kise: ScalableCore System: Hardware Environment for Many-core Architectures Evaluation, IPSJ Transaction on Advanced Computing Systems, Vol.4, No.1, pp. 24-42 (2011) (In Japanese).

5.4 International Conference Paper

2. **Shinya Takamaeda-Yamazaki**, Kenji Kise and James C. Hoe: PyCoRAM: Yet Another Implementation of CoRAM Memory Architecture for Modern FPGA-based Computing, Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL 2013) (December 2013).
3. Ryohei Kobayashi, **Shinya Takamaeda-Yamazaki**, and Kenji Kise: Towards a Low-Power Accelerator of Many FPGAs for Stencil Computations, Workshop on Challenges on Massively Parallel Processors (CMPP 2012) held in conjunction with ICNC'12, pp.343-349 (December 2012).
4. Takakazu Ikeda, **Shinya Takamaeda-Yamazaki**, Naoki Fujieda, Shimpei Sato,and Kenji Kise: Read Density Aware Fair Memory Scheduling (Performance Track Award), 3rd JILP Workshop on Computer Architecture Competitions (JWAC-3): Memory Scheduling Championship (MSC) in conjunction with ISCA-39th (June 2012).
5. **Shinya Takamaeda-Yamazaki**, Shintaro Sano, Yoshito Sakaguchi, Naoki Fujieda, and Kenji Kise: ScalableCore System: A Scalable Many-core Simulator by Employing Over 100 FPGAs, International Symposium on Applied Reconfigurable Computing (ARC 2012) (March 2012).
6. **Shinya Takamaeda-Yamazaki**, Ryosuke Sasakawa, Yoshito Sakaguchi, and Kenji Kise: An FPGA-based Scalable Simulation Accelerator for Tile Architectures, ACM COMPUTER ARCHITECTURE NEWS, Vol.39, No.4, pp.38-43 (International Workshop on Highly-Efficient Accelerators and Reconfigurable Technologies HEART2011, pp. 35-40, June 2011), (September 2011).
7. **Shinya Takamaeda**, Shimpei Sato, Takefumi Miyoshi, and Kenji Kise: SmartCore System

for Dependable Many-core Processor with Multifunction Routers, International Conference on Networking and Computing (ICNC'10), pp.133-139 (November 2010).

8. Yuhta Wakasugi, Naoki Fujieda, **Shinya Takamaeda**, and Kenji Kise: MipsCoreDuo: A Multifunction Dual-core Processor, International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS) , pp. 587-590 (December 2009).
9. **Shinya Takamaeda**, Shimpei Watanabe, Takefumi Miyoshi, and Kenji Kise: ScalableCore:The Concept of Practical and Low-Cost Prototyping System for Many-Core Processor Research and Education, The 4th Workshop on Architectural Research Prototyping (WARP 2009) held in conjunction with the ISCA-2009, Austin, USA (June 2009).

5.5 Domestic Conference Paper (with Review)

10. 笹河 良介, 藤枝 直輝, **高前田（山崎） 伸也**, 吉瀬 謙二：ネットワークオンチップにおける仮想チャネル利用法の再考と評価, 先進的計算基盤システムシンポジウム SACSIS2013 論文集, 於 仙台国際センター (May 2013) (to appear).
11. 小林 諒平, **高前田（山崎） 伸也**, 吉瀬 謙二：多数の小容量 FPGA を用いたスケーラブルなステンシル計算機の開発, 先進的計算基盤システムシンポジウム SACSIS2013 論文集, 於 仙台国際センター (May 2013) (to appear).
12. 小林 諒平, 佐野 伸太郎, **高前田（山崎） 伸也**, 吉瀬 謙二：メッシュ接続 FPGA アレーにおける高性能ステンシル計算, 先進的計算基盤システムシンポジウム SACSIS2012 論文集, 於 神戸国際会議場 (2012 年 5 月 17 日発表), pp.142-149 (May 2012).
13. **高前田 伸也**, 渡邉伸平, 姜軒, 植原昂, 藤枝直輝, 三好健文, 吉瀬謙二：メニーコアアーキテクチャの HW 評価環境 ScalableCore システムの開発, 先進的計算基盤システムシンポジウム SACSIS2010 論文集, 於 奈良県新公会堂, 2010 年 5 月 28 日発表, pp. 287-294 (May 2010).
14. 植原 昂, 佐藤 真平, **高前田 伸也**, 渡邉 伸平, 吉瀬 謙二：メニーコアプロセッサの HW/SW 研究開発を加速する実用的な基盤環境, 先進的計算基盤システムシンポジウム SACSIS2009 論文集, 於 広島国際会議場, 2009 年 5 月 29 日発表, pp. 199-207 (May 2009).

5.6 Technical Report

15. **高前田（山崎） 伸也**, 吉瀬 謙二：メモリ抽象化フレームワーク PyCoRAM を用いたソフトプロセッサ混載 FPGA アクセラレータの開発 **(The 1st IPSJ SIG-ARC High-Performance Processor Design Contest 学生部門 準優勝)**, 情報処理学会研究報告 2014-ARC-208, 於 東京工業大学 (2014 年 1 月 23 日発表) (January 2014).
16. **高前田（山崎） 伸也**, 吉瀬 謙二：FPGA プロトタイピング向けメモリ管理フレームワークの

開発, 電子情報通信学会研究報告 RECONF2013-35，於 北陸先端科学技術大学院大学 (2013 年 9 月 19 日発表), pp. 91-96 (September 2013).

17. **高前田（山崎） 伸也**, 吉瀬 謙二：RTL 静的解析による FPGA アクセラレータ向けアプリケーション特化メモリプリフェッチャー, 情報処理学会研究報告 2013-ARC-204, 於 情報交流センタービッグ・ユー (2013 年 3 月 26 日発表) (March 2013).
18. 小林 諒平, **高前田（山崎） 伸也**, 吉瀬 謙二：メッシュ接続 FPGA アレーを用いた高性能ステンシル計算機の設計と実装, 電子情報通信学会研究報告 RECONF2013，於 慶応義塾大学 日吉キャンパス (2013 年 1 月 17 日発表), pp.159-164 (January 2013).
19. **Shinya Takamaeda-Yamazaki**, Naoki Fujieda, and Kenji Kise: Network Performance of Multifunction On-chip Router Architectures, IEICE-CPSY 2012-11, at Kyushu University, (November 2012).
20. 笹河 良介, 藤枝 直輝, **高前田（山崎） 伸也**, 吉瀬 謙二：ネットワークオンチップにおける仮想チャネル利用法の再考, 電子情報通信学会研究報告 CPSY2012-51，於 九州大学百年講堂 (2012 年 11 月 27 日発表) (November 2012).
21. 池田 貴一, **高前田（山崎） 伸也**, 吉瀬 謙二：ロード命令のプログラムカウンタを用いたメモリスケジューリング手法, 情報処理学会研究報告 2012-ARC-201, 於 とりぎん文化会館 (2012 年 08 月 01 日発表), pp. 1-8 (August 2012).
22. **高前田（山崎） 伸也**, 佐藤 真平, 吉瀬 謙二：高機能ルータを利用した DMR 実行メニーコアにおける効率的なタスク配置手法の検討, 情報処理学会研究報告 2011-ARC-199, 於 長崎大学 2012 年 3 月 27 日発表 (March 2012).
23. **高前田（山崎） 伸也**, 吉瀬 謙二：DMA ベースメニーコアにおける通信オーバーヘッド削減手法, 情報処理学会研究報告 2011-ARC-196, 於 かごしま県民交流センター 2011 年 7 月 27 日発表, pp. 1-6 (August 2011).
24. **高前田 伸也**, 笹河 良介, 吉瀬 謙二：FPGA によるメニーコアシミュレータ ScalableCore システムの正当性検証, 電子情報通信学会研究報告 RECONF, 於 慶應義塾大学 2011 年 1 月 18 日発表, pp. 187-192 (January 2011).
25. 坂口 嘉一, **高前田 伸也**, 吉瀬 謙二：ScalableCore システム 2.0 の実装と評価, 電子情報通信学会研究報告 RECONF, 於 静岡大学 2010 年 9 月 17 日発表, pp. 121-126 (September 2010).
26. 坂口 嘉一, モッハマドアスリ, **高前田 伸也**, 金子 晴彦, 吉瀬 謙二：誤り訂正符号を用いた軽量な高速シリアル通信機構の実装と評価, 電子情報通信学会研究報告 CPSY2010-19, 於 金沢市文化ホール 2010 年 8 月 4 日発表, pp. 67-72 (August 2010).
27. 佐野 正浩, **高前田 伸也**, 芝 哲史, 曹 哲, 伊藤 宗平, 川合 秀実, 笹田 耕一, 吉瀬 謙二：Mieru システムソフトウェア, 情報処理学会研究報告 2010-ARC-189, 於 ラフォーレ伊東 2010 年 4 月 22 日発表, pp. 1-9 (April 2010).
28. **高前田 伸也**, 渡邉 伸平, 姜 軒, 藤枝 直輝, 植原 昴, 三好 健文, 吉瀬 謙二：メニーコアアーキテ

クチャ研究のためのスケーラブルな HW 評価環境 ScalableCore システム, 情報処理学会研究報告 2009-ARC-185, 於 東京工業大学, 2009 年 10 月 26 日発表, pp. 1-10 (November 2009).

29. 渡邉 伸平, **高前田 伸也**, 姜 軒, 三好 健文, 吉瀬 謙二：小容量 FPGA によるスケーラブルなシステム評価環境の構築手法, 電子情報通信学会研究報告 RECONF2009-31, 於 宇都宮大学, 2009 年 9 月 18 日発表, pp. 73-78 (September 2009).
30. 若杉 祐太, 佐藤 真平, 植原 昂, 藤枝 直輝, 渡邉 伸平, **高前田 伸也**, 森 洋介, 吉瀬 謙二：極めて低コストで効率的な VDEC チップ試作・検証システムの開発と応用, 情報処理学会研究報告 2009-ARC-183, 於 沖縄県青年会館, 2009 年 4 月 22 日発表, pp. 1-8 (April 2009)
31. 渡邉 伸平, 藤枝 直輝, 若杉 祐太, **高前田 伸也**, 森 洋介, 吉瀬 謙二： MIPS システムシミュレータ SimMips を活用した組込みシステム開発の検討, 情報処理学会研究報告 2008-EMB-10, 於 キャンパスプラザ京都, pp. 23-28 (November 2008).
32. 植原 昂, 佐藤 真平, 森谷 章, 藤枝 直輝, **高前田 伸也**, 渡邉 伸平, 三好 健文, 小林 良太郎, 吉瀬 謙二：シンプルで効率的なメニーコアアーキテクチャの開発, 情報処理学会研究報告 2008-ARC-180, 於 二日市温泉, pp. 39-44 (October 2008).

5.7 Other Presentation and Poster

33. **高前田（山崎）伸也**, 吉瀬 謙二：FPGA ベースアクセラレータ向けメモリプリフェッチ機構の検討, 情報処理学会第 75 回全国大会, 東北大学 川内キャンパス (2013 年 3 月 6 日発表) (March 2013).
34. Ryohei Kobayashi, **Shinya Takamaeda-Yamazaki**, and Kenji Kise: Design of Synchronization Mechanism to Conquer the Clock Oscillator Variation for High Performance Stencil Computation Accelerator, 情報処理学会第 75 回全国大会, 東北大学 川内キャンパス (March 2013).
35. **高前田（山崎）伸也**, 吉瀬 謙二, 佐藤 充：集約光インターコネクトにおける高性能トラフィックスケジューラ, 情報処理学会第 74 回全国大会, 名古屋工業大学 御器所キャンパス (2012 年 3 月 7 日発表) (March 2012).
36. 笹河 良介, **高前田 伸也**, 藤枝 直輝, 吉瀬 謙二：ScalableCore システムの挙動検証 〜ソフトウェアシミュレータと比較して〜 , 先進的計算基盤システムシンポジウム SACSIS2011 論文集, 於 秋葉原コンベンションホール (2011 年 5 月 26 日発表), pp. 262-263 (May 2011).
37. **高前田 伸也**, 吉瀬 謙二：HW メニーコアシミュレータ ScalableCore システムの高速化 **(情報処理学会 第 73 回全国大会 学会推奨修士論文)** , 情報処理学会第 73 回全国大会, 東京工業大学 大岡山ャンパス (2011 年 3 月 2 日発表) (March 2010).
38. Shimpei Sato, **Shinya Takamaeda**, and Kenji Kise: DMR mode of SmartCore system, IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'10), National Institute of Informatics Japan (2010-12-15 Presentation) (December 2010)

39. **高前田 伸也**, 佐藤 真平, 三好 健文, 吉瀬 謙二 : メニーコアアーキテクチャの HW 評価環境 ScalableCore システムの活用 〜ディペンダブルプロセッサの実装〜 **(優秀ポスター賞)**, 先進的計算基盤システムシンポジウム SACSIS2010 論文集, 於 奈良県新公会堂, 2009 年 5 月 27 日発表, pp. 115-116 (May 2010).
40. **高前田 伸也**, 吉瀬 謙二：メニーコアプロセッサにおけるコア間通信レイテンシ隠蔽手法の検討, 情報処理学会第 72 回全国大会, 東京大学 本郷キャンパス (2010 年 3 月 9 日発表), Vol.1, No. 2M-6, pp. 173-174 (March 2010).
41. 渡邉 伸平, **高前田 伸也**, 姜 軒, 三好 健文, 吉瀬 謙二： メニーコアプロセッサ向けプロトタイピングシステムの高速化, 情報処理学会第 72 回全国大会, 東京大学 本郷キャンパス (2010 年 3 月 10 日発表), Vol.1, No. 4M-7, pp. 205-206 (March 2010).
42. 姜 軒, **高前田 伸也**, 渡邊 伸平, 三好 健文, 吉瀬 謙二：マルチプロセッサシステムにおけるルータの実装と評価, 情報処理学会第 72 回全国大会, 東京大学 本郷キャンパス (2010 年 3 月 9 日発表), Vol.1, No. 2M-5, pp. 171-172 (March 2010).
43. 佐野 正浩, **高前田 伸也**, 藤枝 直輝, 吉瀬 謙二 : 計算機システムとソフトウェアシミュレータと組込みソフトウェアの三位一体開発のすすめ, 組込みシステムシンポジウム 2009, 於 国立オリンピック記念青少年総合センター (October 2009).
44. **高前田 伸也**, 渡邉 伸平, 吉瀬 謙二：メニーコアプロセッサの高速プロトタイピングシステム ScalableCore, 先進的計算基盤システムシンポジウム SACSIS2009 論文集 (ポスター発表), 於 広島国際会議場, 2009 年 5 月 28 日発表, pp. 145-146 (May 2009).
45. **Shinya Takamaeda**, Shimpei Watanabe, Shimpei Sato, Koh Uehara, Yuhta Wakasugi, Naoki Fujieda, Yosuke Mori, and Kenji Kise: ScalableCore : High-Speed Prototyping System for Many-Core Processors, in International Symposium on Low-Power and High-Speed Chips (COOL Chips), Yokohama Japan(2009-04-16 Poster Short Speech), p. 161 (April 2009).
46. **高前田 伸也**, 渡邉 伸平, 吉瀬 謙二： メニーコアプロセッサの高速プロトタイピングシステム ScalableCore の提案 **(学生奨励賞)**, 情報処理学会第 71 回全国大会, 立命館大学びわこ・くさつキャンパス No. 3K-1, pp. 91-92 (March 2009).
47. **高前田 伸也**：これが中身が見える計算機システム MieruPC-2008 だ！ **(三菱電機 Changes for the better 賞)**, The 5th IEEE Tokyo Young Researchers Workshop (December 2008).
48. 吉瀬 謙二, 佐藤 真平, 森谷 章, 藤枝 直輝, 若杉 祐太, 渡邉 伸平, 植原 昂, 森 洋介, **高前田 伸也**, 高橋 朝英, 棟岡 朋也, 山田 裕介, 権藤 克彦, 小林 良太郎, 三好 健文, 中條 拓伯：MieruPC プロジェクト: 中身が見える計算機システムを構築する研究・教育プロジェクト **(最優秀ポスター賞)**, コンピュータシステム・シンポジウム (ComSys2008), 於 キャンパス・イノベーションセンター東京 (November 2008).