UC Irvine
UC Irvine Electronic Theses and Dissertations

Title
Computation Model Based Automatic Design Space Exploration

Permalink
https://escholarship.org/uc/item/0h1876f9

Author
Kim, Kyoungwon

Publication Date
2014

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

Computation Model Based Automatic Design Space Exploration

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Electrical Engineering and Computer Science

by

Kyoungwon Kim

Dissertation Committee:
Professor Daniel D. Gajski, Chair
Professor Rainer Dömer
Professor Tony Givargis

2014

Portion of Chapter 3 © 2010 Center for Embedded Systems
Portion of Chapter 3 © 2012 Center for Embedded Systems
Chapter 4 © 2014 IEEE
Portion of Chapter 5 © 2014 Center for Embedded Systems
Portion of Chapter 6 © 2014 Center for Embedded Systems
Chapter 7 © 2014 IEEE
All the other materials © 2014 Kyoungwon Kim

TABLE OF CONTENTS

Page

LIST OF FIGURES **v**

LIST OF TABLES **vii**

ACKNOWLEDGMENTS **viii**

CURRICULUM VITAE **x**

ABSTRACT OF THE DISSERTATION **xii**

1 Introduction **1**
- 1.1 Productivity Gap in Embedded System Design . . . 1
- 1.2 System-Level Design . . . 5
- 1.3 Transaction Level Model-Based Design . . . 6
- 1.4 Transaction Level Model Estimation . . . 12
- 1.5 Computation Model Based Automatic Design Space Exploration . . . 14
- 1.6 Scope . . . 16
- 1.7 Tool . . . 17
- 1.8 Contribution . . . 17
- 1.9 Dissertation Overview . . . 19

2 Related Work **21**
- 2.1 System-Level Design Methodologies . . . 21
- 2.2 Computation Model Based Initial Design Decision-Makings . . . 23
- 2.3 Transaction Level Model Estimation . . . 26

3 Embedded System Environment (ESE) **28**
- 3.1 ESE Idea . . . 28
- 3.2 EDS(ESE Data Structure) . . . 30
  - 3.2.1 The Structure of EDS . . . 31
  - 3.2.2 Update of EDS on Changes in System Specification . . . 35
- 3.3 Transaction Level Model . . . 42
  - 3.3.1 TLM Overview . . . 42
  - 3.3.2 Computational Part of TLM : PE . . . 45
  - 3.3.3 Communicational Part of TLM . . . 51

3.4 Transaction Level Model Generation 71
3.4.1 Transaction Level Model Generation : PE 76
3.4.2 Transaction Level Model Generation : UBC 79
3.4.3 Transaction Level Model Generation : Tx 81

4 Hierarchy-Aware Mapping of Pipelined Applications 83
4.1 Application Model and Platform 84
4.2 Problem Definition 87
4.3 Hierarchy-Aware Mapping 87
4.3.1 Algorithms 87
4.3.2 Heuristics for Repartitioning and Remapping 90
4.4 Case Study 91

5 N-Way Clustering and Mapping 94
5.1 Problem Definition 96
5.2 Motivation of N-Way Clustering and Mapping 98
5.3 Closeness Function of NWCM 100
5.4 Algorithm 101
5.5 Case Study 106
5.5.1 MP3 Decoder And JPEG Encoder 106

6 Cycle-Approximate Estimation Based Mapping 110
6.1 Application Model 111
6.2 Cycle-Approximate Estimation Based Mapping 113
6.2.1 Overview 113
6.2.2 Algorithm 114
6.3 Case Study 117
6.3.1 MP3 Decoder And JPEG Encoder 117

7 Trace-Driven Performance Estimation of Multi-core Platforms 120
7.1 Trace-Driven Performance Estimation 121
7.1.1 Assumptions 121
7.1.2 Definitions: Execution, Communication Events and Traces . . 121
7.1.3 Trace-Driven Performance Estimation 123
7.1.4 Abstract RTOS, Memory Hierarchy, Processing Element and Bus Protocol Models 125
7.2 Case Study 127

8 Conclusion 129
8.1 Summary 129
8.2 Initial Mapping 131
8.3 Iterative Improvement 133
8.4 Transaction Level Model Estimation 133

9 Future Work 135

Bibliography **136**

LIST OF FIGURES

Page

1.1 Moore's Law (source : [1]) . . . 2
1.2 Automatic Design Space Exploration for TLM-Based System Design . 7

3.1 ESE Design Flow and Front-End Tool . . . 29
3.2 SW Architecture for System Specification . . . 31
3.3 The Structure of EDS . . . 32
3.4 Synchronization Tables for Buses for The Example in Figure 3.3 . . . 33
3.5 The Implementation of EDS . . . 34
3.6 Change in Channel-Route Mapping . . . 37
3.7 Change in PE-Process Mapping . . . 39
3.8 Change in Platform : Move A PE . . . 40
3.9 Change in PE-Process Mapping . . . 41
3.10 Transaction Level Model . . . 43
3.11 Structure of PE in TLM . . . 46
3.11 Structure of PE in TLM (Cont'd) . . . 47
3.12 Sequential Execution of Processes on RTOS Model . . . 49
3.13 Same Communication API with Different Implementation . . . 50
3.14 UBC . . . 53
3.15 The Structure of UBC . . . 54
3.16 Communication Delay Table in UBC . . . 58
3.17 Data Transaction Scenario in Tx . . . 59
3.17 Data Transaction Scenario in Tx (Cont'd) . . . 60
3.18 SW Architecture of Tx . . . 62
3.19 Old Tx Model with Deadlock . . . 63
3.20 Number of Request Buffers in A Tx . . . 66
3.21 Active to Active Buffer : Redundant Cycle in A Route . . . 67
3.22 Tx Routing Table . . . 69
3.23 Two Types of Information for TLM Generation . . . 71
3.24 PE Generation . . . 76
3.25 RTOS Model Generation . . . 78
3.26 Communication API Selection . . . 78
3.27 UBC Generation . . . 80
3.28 UBC Template . . . 81
3.29 Tx Generation . . . 82

4.1 Application Model, Platform, and Mapping 85
4.2 Modeling Example: SpecC 86
4.3 Hierarchy-Aware Mapping: $\|\boldsymbol{V}\| = \|\boldsymbol{S}\|$ 88
4.4 Hierarchy-Aware Mapping: $\|\boldsymbol{V}\| \neq \|\boldsymbol{S}\|$ 89
4.5 ExecTime(G) Function 91

5.1 Design Flow in Model-Based Design 97
5.2 Simple Mapping Example with and without Considering Process Scheduling 99
5.3 N-Way Clustering and Mapping Example 102
5.4 N-Way Clustering and Mapping (NWCM) Algorithm 103
5.5 Application and Platform for Experiment 106

6.1 Application Model 111
6.2 CAEBM 112
6.3 Flow Chart of CAEBM 114
6.4 Changes in The Ratio of Execution Time over The Initial Mapping During CAEBM's Process on MP3 + JPEG Application 119

7.1 Alignment Example w/ RTOS Model and Bus Protocol Model 122
7.2 Trace-Driven Performance Estimation 123
7.3 Design Flow with Trace-Driven Performance Estimation 124

LIST OF TABLES

Page

3.1 Example of RTOS Overhead Table 48

4.1 Canny Edge Detector: Average Execution Time per Frame 92
4.2 JPEG: Average Execution Time per Frame 92

5.1 Total Execution Delays on Each PE 108
5.2 Overall Latency 108

6.1 Errors in Estimation: SPEA and LB 118

7.1 Comparison in Estimation Time 126
7.2 Accuracy Results: Error % against TLM Estimation (SW, SW+1) . . 126
7.3 Accuracy Results: Error % against TLM Estimation (SW+2, SW+4) 126

ACKNOWLEDGMENTS

Above all, I would like to express my best gratitude to my advisor, professor Daniel Gajski. I have been fortunate to have an advisor who gave me the freedom to explore on my own, and at the same time the guidance to recover when my steps faltered. His technical advice was essential to finish this dissertation.

My thanks also go to the Committee members for providing their priceless comments that improved the presentation and this dissertation.

I would like to thank my friend and ex-colleague, Yonghyun Hwang. Without his guide, this dissertation would not be able to be completed. The friendship of Yonghun Eom, Changik Lee, and Grace Wu is much appreciated. It is their various forms of supports during my graduate study that always encouraged me to keep going on. I am also thanksful to CECS staffs. Their helps and care helped me overcome setbacks and stay focused on my study.

Last but not least, I would like to thank my fiancee, Cory Bayrante. Her support and encouragement was in the end what made this dissertation possible.

I would like to inform the readers that the following CECS technical reports written solely by the author of this dissertation are permitted to be reused in this dissertation:

- Kyounwon Kim. Embedded System Environment: Overview. CECS Technical Report TR11-11, University of California Irvine, Dec., 2011.
- Kyounwon Kim. TLM Generation in ESE. CECS Technical Report TR12-02, University of California Irvine, Feb., 2012.

The text of this dissertation is a reprint of the material as it appears in the following publications:

- Kyounwon Kim and Daniel Gajski. Automatic Partitioning and Process Mapping for Model-Based Design. CECS Technical Report TR14-09, University of California Irvine, May, 2014.
- Kyounwon Kim and Daniel Gajski. Cycle-Approximate Estimation-Based Mapping. CECS Technical Report TR14-10, University of California Irvine, May, 2014.
- Kyoungwon Kim and Daniel Gajski, Hierarchy-Aware Mapping of a Pipelined Application. In Proceedings of the IEEE 57th International Midwest Symposium on Circuits and Systems, College Station, TX, USA, August, 2014

- Kyoungwon Kim and Daniel Gajski, Trace-Driven Performance Estimation of Multi-Core Platforms. In Proceedings of the IEEE 57th International Midwest Symposium on Circuits and Systems, College Station, TX, USA, August, 2014

The co-author listed in the four publications above directed and supervised researches which form the basis of this dissertation.

My thank must also go to IEEE, who allows the publications that are listed below, have been accepted, but have not been published yet:

- Kyoungwon Kim and Daniel Gajski, Hierarchy-Aware Mapping of a Pipelined Application. In Proceedings of the IEEE 57th International Midwest Symposium on Circuits and Systems, College Station, TX, USA, August, 2014
- Kyoungwon Kim and Daniel Gajski, Trace-Driven Performance Estimation of Multi-Core Platforms. In Proceedings of the IEEE 57th International Midwest Symposium on Circuits and Systems, College Station, TX, USA, August, 2014

CURRICULUM VITAE

Kyoungwon Kim

EDUCATION

Ph.D., Electrical Engineering and Computer Science **2014**
University of California, Irvine *Irvine, California*

M.S., Electrical Engineering and Computer Science **2009**
Seoul National University, Seoul *Seoul, South Korea*

B.S., Electrical Engineering **2007**
Seoul National University *Seoul, South Korea*

RESEARCH EXPERIENCE

Graduate Research Assistant **2010–2014**
University of California, Irvine *Irvine, California*

Visiting Scholar **2009–2010**
University of California, Irvine *Irvine, California*
Graduate Research Assistant **2007–2009**
Seoul National University *Seoul, South Korea*

SELECTED HONORS AND AWARDS

Engineering Fellowship **2010**
University of California, Irvine

REFEREED CONFERENCE PUBLICATIONS

Hierarchy-Aware Mapping of a Pipelined Application **Aug. 2014**
The IEEE 57th International Midwest Symposium on Circuits and Systems (accepted)

Trace-Driven Performance Estimation of Multi-Core Platforms **Aug. 2014**
The IEEE 57th International Midwest Symposium on Circuits and Systems (accepted)

SELECTED TECHNICAL PUBLICATIONS

TLM Generation with ESE **Jul. 2010**
CECS Technical Report TR10-07

Embedded System Environment: Overview **Dec. 2011**
CECS Technical Report TR11-11

TLM Generation in ESE **Feb. 2012**
CECS Technical Report TR12-02

Automatic Partitioning and Process Mapping for Model-Based Design **May. 2014**
CECS Technical Report TR14-09

Cycle-Approximate Estimation-Based Mapping **May. 2014**
CECS Technical Report TR14-10

SOFTWARE

ESE(Embedded System Environment) Front-End `www.cecs.uci.edu/~ese`
The front-end of a toolset for modeling, validation and synthesis of embedded system designs

ABSTRACT OF THE DISSERTATION

Computation Model Based Automatic Design Space Exploration

By

Kyoungwon Kim

Doctor of Philosophy in Electrical Engineering and Computer Science

University of California, Irvine, 2014

Professor Daniel D. Gajski, Chair

Embedded system designers continuously face a twofold challenge handling the ever-increasing complexity of design and meeting the ever-shrinking time-to-market timeline. To meet such a challenge, the system design paradigm has shifted to platform-based design characterized by intensive use of software and aggressive reuse of verified components. More and more designs are turning to heterogeneous platforms to meet design constraints in multiple criteria. However, the use of such and heterogeneity generates more challenges for platform-based design. To address the challenges, system designers must utilize system-level methodologies for specification and design. Here the designers begin the design process by coming up with a computation model that captures the behavior of the system. The computation model is successively refined down to the structural model in the system level. In each refinement, the designers explore the design space. The design space comprises a set of alternatives concerning hardware/software partitioning, platform selection, and mapping. The design space is huge, however, necessitating the automation of its exploration. The platform is very often fixed regarding either availability or legacy reasons, making mapping crucial. This dissertation focuses on the automatic mapping of the computation model for the given heterogeneous platform.

This dissertation presents a new automatic mapping technique. The proposed technique consists of two separate phases: initial mapping and improvement driven by cycle-approximate estimation. Existing mapping techniques depend on early estimation so a dilemma arises from the fact that cycle-approximate estimation cannot precede mapping. The dilemma can be gotten around by our performing initial mapping based on rough estimation and then making iterative improvements based on cycle-approximate estimation. While earlier work has been domain specific, the mapping techniques in the proposed work are driven by a general computation model that includes hierarchy, state transitions, dynamic data-oriented behavior, and imperative languages. The proposed work also addresses mapping with an awareness of general hierarchy in pipelined applications.

In such a mapping technique, the size of the design space explored is limited by the speed of cycle-approximate estimation. Earlier work has realized such a fast cycle-approximate estimation by generating and simulating Transaction Level Models. However, simulation has to be performed whenever there is any change in the platform or mapping. That is not necessary and therefore there is still room for improvement regarding speed. This dissertation presents a new trace-driven estimation that is orders of magnitude faster than simulation-based cycle-approximate estimation while losing neither accuracy nor generality.

We have applied the proposed mapping techniques to multiple multimedia applications and the techniques outperformed in terms of execution time the competitors by proportions ranging from 23.3% through 36.3%.

Chapter 1

Introduction

1.1 Productivity Gap in Embedded System Design

An embedded system is a computing system that is designed for mission-specific computing devices rather than general purpose computing systems such as personal computers. Therefore, embedded systems are intended to serve a dedicated set of specific functions. Typical examples include:

- consumer electronic appliances such as smartphones and smartpads
- office automation such as wired/wireless printers
- automotive systems
- home appliances such as freezers, ovens and laundry machines

According to Vahid and Givargis [2], embedded systems are differentiated from other computing devices by the following characteristics:

- The system is dedicated to a set of specific functions rather than designed for general purpose.
- Design constraints such as power consumption, size, cost, and requirements for real-time processing are much tighter than those of general computing systems.
- The systems must be fault tolerent.

Embedded system designers are genenerally operating under time-to-market pressure.

Microprocessor Transistor Counts 1971-2011 & Moore's Law

Figure 1.1: Moore's Law (source : [1])

On top of going with these factors, embedded system designers must contend with the rapidly increasing hardware density. In accordance with Moore's Law, advances in semiconductor technology have increased at an exponential rate the transistor-carrying capacity of a single chip. This technological advancement has generated a positive feedback loop with the ever-increasing user demands for performance.

In an effort to come to terms with the design complexity and time-to-market pressures and tight design constraints inherent in embedded systems, the design paradigm has shifted to software-centric, platform-based design [3]. In a traditional embedded system design, a processor mainly served as a commodity part while the system was close to being an amalgamation of nonprogrammable componnents such as Application-Specific Integrated Circuits (ASICs), peripherals, buses, and glue logic held together by. However, such a hardware implementation was ill-equipped to take on the increasing design complexity of contemporary and future embedded systems. Generally, software implementations can better handle higher complexity than can hardware implementations. Fortunately, embedded processors have improved in terms of capabilities and power. Moreover, with rising non-recurring engineering costs such as the cost of mask, it has become very crucial to maximize post-fabrication reuse of the verified components. Such progress has encouraged designers to implement large portions of the given system to software running on general purpose and domain-specific processors such as Digital Signal Processors (DSP) and network processors, all of which are available in the platform components library.

In such software-centric platform-based designs, the system may have performance issues: in general, hardware implementations outperform software implementations in terms of performance. Indeed, software implementations have become prevalent not for their performance but for the design complexity of embedded systems and productivity concerns. The common engineering practice in software-centric designs to ensure the required performance is to rely on frequency ramping followed by CMOS scaling. CMOS scaling, however, is hobbled by physical and economic limitations giving it a short future life [4]. Moreover, in embedded system designs, performance is not at all the single optimization goal. Especially, in many embedded systems, power consumption is no more a secondary issue. Frequency is proportional to operation voltage, and dynamic power dissipation is proportional to the square of operation

voltage. Thus, the increase in power consumption has also been making infeasible frequency ramping.

As an alternative to frequency ramping, engineers have proposed the integration of multiple processors and/or cores into a single chip. In this design style, engineers put onto a single chip an entire system that include multiple emdedded processors and/or cores, specialized digital hardware, and, often, mixed-signal circuits. The advances in semiconductor technology enabled this design style. Rather than increasing frequency, the design replicates the functionality to exploit parallelism.

The disadvantage of this approach is the increased design complexity. One of the most difficult design challenges lies in the programming models, which are required to map application software into efficient implementations [5]. Several decades of computing history have taught designers to think sequentially and most programming languages encourage sequential thinking. Nevertheless, embedded systems are concurrently executed with process synchronization depending on dynamic data-oriented behaviors. Another design challenge in multi processor/core designs is that analysis of such a system executed in parallel is far harder than that of single processor/core systems.

Heterogeneity puts additional challenges to software-centric platform-based design. Although homogeneous platforms can be manufactured in enormous volumes and programming for such homogeneous platforms is easier [6], many embedded applications still call for several reasons heterogeneous platforms. The tasks in an embedded application vary greatly, which encourages the use of different types of processors. In addition, many embedded applications are memory-oriented, and the required access times of the tasks in such applications are not uniform. Memory subsystems are prime determinants of power consumption, and a platform may need different memory subsystems, with each subsystem providing different memory access time. As configurable and extensible processors are replacing ASICs [5], platforms tend to be

more heterogeneous. Heterogeneity, however, even harder programming and thorough design space exploration for mapping the given application into implementations even harder.

Inevitably, these design challenges result in a longer design cycle. Nonetheless, time-to-market windows tend, contrarily, to decrease. Therefore, all major technology roadmaps to address productivity gap in embedded system designs [7]. The productivity gap, for sure, augments the need for well-defined system design methodologies accompanied by Computer-Aided Design tool(s).

1.2 System-Level Design

A well-known solution intended to gain the required productivity is to raise the level of abstraction of the design process to the system-level [7]. System-level design uses models at the system level, where the unnecessary implementation details are left out. The advantage is that the designers can worry less about implementation specifics and concentrate more on the behavior of the system and on high-level design decisions such as hardware/software partitioning, platform selection, and mapping. In addition, as the granularity of the design components are much coarser at the system level than at any lower level, the design complexity can be quickly relaxed. The coarse granularity of the system-level design allows components, connectivity, and parameters to be easily changed. Therefore, the designers can explore a broad range of the design space with an eye on high-level design decisions.

Even at the system level, however, the design complexity is still high. The mapping of a given application processes to the given, fixed homogeneous platform is only a small subset of all the problems at system-level design but already an NP-complete

problem [8]. Therefore, the design complexity at the system level highlights the needs for well-defined, system-level methodologies for specification and design.

A common system-level design methodology is called platform methodology [9]. Designers begin the design process by defining the platform. The defined platform could be informal block diagrams for the structure of the system. The block diagrams are created by chief architects who rely on their experience and intuition. Or, the platform exists as a legacy. The design process often serves as an improvement of the given platform. However, when these block diagrams are created, the functionality of the system is not fully understood. This results in inconsistencies discovered only late in the design process, neccessitating unnecessary iterations that result in a long design time. Therefore, many studies on system-level design methodologies have suggested that the design process should start with, rather than platform, an executable, purely behavioral models for the system and non-functional design constraints.

In such design methodologies, however, designers must address the large gap between system-level behavioral models and implementations. In general, the transformation cannot be done in a single step. Therefore, the design process is broken into manageable pieces at the expense of losing global optimality. In this regard, it is a must to have well-defined levels of abstractions, models at each level of abstraction, and transformation rules.

1.3 Transaction Level Model-Based Design

One such system-level design methodology is the Transaction Level Model-based design [9] shown in Figure 1.2. Transaction Level Model-based design starts with not the platform but a set of non-functional design constraints and Behavioral Models of

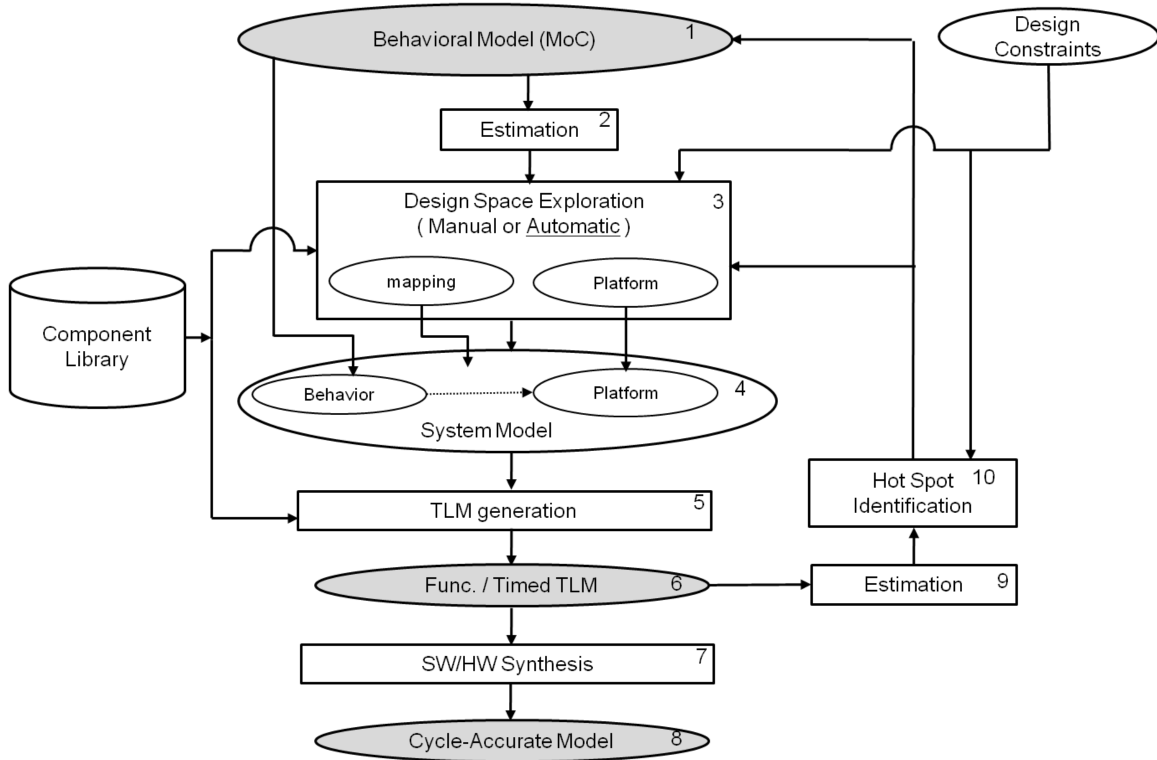


Figure 1.2: Automatic Design Space Exploration for TLM-Based System Design

the system, captured with computation models [10, 11, 12, 13]. Computation models are generalized ways to describe the system-level behavior of the design.

To close the gap between computation models and implementations, Transaction Level Model-based design offers several orthogonal levels of abstraction and models to utilize the 'divide and conquer' strategy. The design flow in Transaction Level Model-based design is a series of successive refinements from one model to another. At each refinement step, the designers are encouraged to explore the design alternatives at the given level of abstraction but discouraged to worry about the details below that level. In fact, it could be argued that the ideal is the minimum number of required levels of abstraction. One way to determine what this number is to consider the number of different types of designers in contemporary embedded system design practices. Generally, designers are of three types: application programmers, system

designers, and implementation. Accordingly, it is reasonable to define three orthogonal levels of abstractions for the three types of designers: the Behavioral Level (shape 1) for application programmers, the Transaction Level (shape 6) for system designers, and the Cycle-Accurate Level (shape 8) for implementation designers. In this regard, Transaction Level Model-based design offers the three levels of abstractions and defines the design process as two refinements: one from Behavioral Model to Transaction Level Model [14] and the other from Transaction Level Model to Cycle-Accurate Models.

Behavioral Models are computation models that are purely functional and preferably executable; all implementation details are abstracted out; execution of the Behavioral Models allows, above all, validation of the functionality. Moreover, executable computation models allow rough estimation on the design decisions. Based on the estimation, designers can explore the design space concerning early design decisions such as hardware/software partitioning, platform selection, and mapping. Once the design decisions are made, the Transaction Level Model (Shape 6) is automatically generated (Shape 5).

In this design stage, however, design space is too broad, so automation is a must, although Transaction Level Model-based design is missing such automation. This dissertation is intended to define automatic design space exploration for Transaction Level Model-based design. There have been a large number of papers addressing automatic design space exploration. The papers have presented many different algorithms to automate design space exploration in this design stage [15, 16, 17, 18, 19]. These algorithms, however, have limitations regarding the estimation they depend on. The limitations highlight the need of new approaches to automatic design space exploration and refinement.

In the algorithms, estimation is assumed to be given ahead of design decisions. The

estimation is also a prime determinant of the design quality. For the algorithms to ensure the required design quality, the estimation has to meet the following three requirements: fast, accurate, and general. If the speed of estimation is too low, the size of the design space explored will be narrowed. Accuracy greatly impacts the design quality. For an estimation to be general, it should be applicable to both hardware and software. Otherwise, many platforms that include custom hardware to accelerate critical functions with reasonable power consumption will not be available for evaluation. The estimation should not require Cycle-Accurate Models, either. Otherwise, it is not general enough because Cycle-Accurate Models may not, in the early design stages, be available for one or more platform components.

The estimation techniques [20, 21, 22, 23, 24] by which the automatic design space algorithms can be facilitated do not, however, meet altogether all three requirements. The estimation techniques include Cycle-Accurate Model estimations [20, 21], static analysis-based estimations [22], trace-driven estimations [23, 24]. Cycle-Accurate Model estimations tended to be too slow for early design space exploration. Moreover, it may not be available for the entire platform. Static analysis-based estimations are not at the cycle level. Trace-driven estimations either need Cycle-Accurate Models or cannot be applied to hardware. Thus, trace-driven estimations are not sufficiently general. Therefore, the exisiting techniques may not be sufficient regarding the limitation in estimation rather than the quality of those algorithms themselves.

Transaction Level Model estimation (shape 9, 10) [25, 26, 27] can satisfy all three requirements at the same time. Transaction Level Models are abstract, executable models for the system, where communication and computation are separated and one of them or both are approximate-timed instead of cycle-timed. Generally, Transaction Level Model estimation has advantages over Cycle-Accurate Model estimations. It is faster by orders of magnitude, available for the entire platform even in the early

design stages, and yet cycle-approximate. Therefore, design decisions can be evaluated very fast and accurately so it dramatically reduces the design cycle. However, in the viewpoint of design space exploration concerning hardware/software partitioning, mapping, and platform selection, there exists a conflict that gives rise to a "chicken-or-the-egg" type of dilemma: the design decisions need estimation, yet Transaction Level Model estimation cannot be fed to the input of the design space exploration process. This conundrum underscores the need to change design space exploration techniques. As Transaction Level Model estimation is applicable to a very small number of platforms at a time, one reasonable way to utilize Transaction Level Model estimation is to improve the given design by local searches.

Therefore, this dissertation proposes new approaches to automatic design space exploration. The approaches consist of two phases: initial design decision and Transaction Level Model estimation-based iterative improvements. In the first phase, computation models are roughly estimated and any existing automatic design space exploration algorithm could be applied to make the initial design decisions. In the second phase, the given platform and mapping are iteratively improved based on the Transaction Level Model estimation until all the design constraints are met.

Once the design constraints are met, the Transaction Level Models are refined to Cycle-Accurate Models. Generated in this step are the interface hardware and system software such as device drivers. Cycle-Accurate Models describe the state of the system-level components on each clock cycle. These are synthesized by existing logic synthesis tools to be transformed into the final implementation. This dissertation, however, focuses on the refinement from the computation model to the Transaction Level Model so that the design constraints are met.

In Transaction Level Model-design, two crucial issues should be addressed: efficient Transaction Level Model estimation and definition of computation models and algo-

rithms driven by the computation models.

Transaction Level Model estimtion plays a crucial role in the second phase of transformation from computation model to Transaction Level Models. The size of the design space that can be explored is limited by the speed of the Transaction Level Model estimation. Existing Transaction Level Model estimations require simulation of the entire platform model. Depending on the application, this may not be necessary. Therefore, there is still room for improvement regarding speed. In this dissertation, a Trace-Driven estimation in Transaction Level is proposed to complement the existing, simulation-based Transaction Level Model estimations to enhance the speed of estimation without losing generality and accuracy.

In the Transaction Level Model-based design, many different computation models could be used for various applications and optimization goals. Generally, there exist requirements that must met by computation models. Computation models, first of all, should be complete so as to capture the behavior of the entire system. Thus, concurrency, state-transition, structural/behavioral hierarchy, and synchronization depending on data and control should be supported. Moreover, as verification between models at different levels of abstraction is a bottleneck in embedded system design flow, automation in refinement processes are a must. Computation models must be defined with an eye on synthesis. In this dissertation, computation models are defined to answer the requirements, and algorithms for automation in design space exploration, for both phases are proposed based on the defined computation models.

1.4 Transaction Level Model Estimation

Estimation on the design decisions can be performed at several different levels of abstraction. One method to provide estimates on the system in the early design stages is to use the computation models. As computation models are very often executable, estimates can be given by simulating the model on the host machine or an Instruction Set Simulator. While these approaches are fast and available in the early design stages, they fall short of ensuring the required accuracy. The design metrics are impacted by many different factors, such as the datapath of the processing elements, memory hierarchy, RTOS scheduling policy and overhead, and delays regarding bus protocol. In estimation based on computation models, however, these factors are ignored. Another method is to depend on Cycle-Accurate Models. Cycle-Accurate Models provide highly accurate estimation regarding any design metric. Where the design space is vast, however, the speed of estimation is too slow. Moreover, Cycle-Accurate Models for a portion of hardware components in the platform may not be available, yet.

Transaction Level Model-based design requires a fast and cycle-approximate estimation, on which can depend the process of iterative improvements of the design. That is where the Transaction Level Model comes. Transaction Level Model estimations provide cycle-approximate estimates on design decisions that are orders of magnitude faster than Cycle-Accurate Model esitmations. It is applicable to both hardware and software from the early design stages on.

The existing Transaction Level Model estimations [25, 26, 27] depend on generation and simulation of Transaction Level Models. Such simulation-based Transaction Level Model Estimations offer speed, accuracy, and generality at the same time. The speed is close to that of native simulation as Transaction Level Models are typically written

in a single System-Level Design Language such as SystemC [28], natively compiled to the binary for the simulation host, and simulated. Compared to Cycle-Accurate Model estimations, simulation-based Transaction Level Model estimations are order of magnitude faster. Researchers have claimed that accuracy is close to Cycle-Accurate Models. The error ratio is less than 10% that of the board measurement [26], and in this regard, estimations based on Transaction Level Models are claimed to be cycle-approximate. Simulation-based Transaction Level Model estimations are retargetable in the sense that they are applicable to both hardware and software. Moreover, as simulation-based Transaction Level Model estimations require no Cycle-Accurate Models, they are applicable to most of the platforms in early design stages.

However, the design space is still too broad. There is also room for improvement regarding speed. This dissertation proposes a new trace-driven estimation based on Transaction Level Model. The Trace-Driven estimation is complementary to simulation-based Transaction Level Model estimations. For applications in which the execution path of each task depends only on data, Trace-Driven estimation offers orders of magnitude faster estimation than does simulation-based Transaction Level Model estimation, and without losing accuracy or generality. Trace-Driven estimation generates, only once, the ordered list of continuous communication and computation events at the Transaction Level annotated with delay estimates before any design decision is made. Once a new platform selection and mapping has been given, Trace-Driven estimation places the traces at the right location in the global timeline. Note that simulation-based Transaction Level estimation must generate the Transaction Level Model and simulate the entire platform from scratch. Thus, Trace-Driven estimation is faster than simulation-based Transaction Level Model estimation. The delay estimates are calculated in exactly the same way as was performed in Hwang et al [26]. Placing the events along the global time line takes into consideration the abstract RTOS model [25], memory hierarchy models, Processing Element configura-

tions [26], and bus protocol models [29]. All of these abstract models are Transaction Level Models. Moreover, the delay estimate calculation is performed at the Transaction Level. Thus, Trace-Driven estimation lose accuracy and generality.

1.5 Computation Model Based Automatic Design Space Exploration

In any system-level design methodology including Transaction Level Model-based design, computation models are necessary for thorough exploration of all architectural design alternatives. A computation model is a generalized way to describe the functionality of the system, although it can be defined in several different ways [10, 11, 12, 13].

To guide the system-level design flow, computation models must meet several requirements. First of all, a computation model should provide completeness [30]. It must be able to capture the complex behavior of the entire given system. Completeness refers the concept of explicit state transitions, concurrency, timing, leaf-behaviors described in imperative language, and synchronization. Moreover, to ensure the required productivity, the computation model should be able to be combined with well-established, automated design flow.

Existing computation models can be categorized into two groups: process-based and state-based models [9]. Not one of them is, however, complete. A process-based model [31, 32, 33, 34] is a set of concurrently executed, communicating processes. Thus, process-based ones take advantage of explicitly exposed concurrency by focusing only on data dependencies throughout the model. However, process-based models are mainly limited to modeling dynamic behaviors in data oriented applications such as

modern multimedia applications. There is no concept of explicit state transitions. On the contrary, state-based models [35, 36, 37] generally target control-oriented applications such as automotive systems. Although such state-based models provide hierarchy and concurrency, they tend to be more on lock-step synchronization or, at least, require the local clock for the leaf behaviors. Compared to process-based models where processes are executed in a completely asynchronous manner, state-based models have the limitation of describing the dynamic behavior depending on data. Thus, state-based models are not complete in terms of timing.

A general computation model such as Program State Machine [9] combines both process-based and state-based models to achieve the completeness. A Program State Machine is close to hierarchical concurrent state machines. However, each program-state is a function written in an imperative language. Program states can runs in a completely asynchronous manner, and can have synchronization depending on data.

On the other hand, a general model of computation such as Program State Machines already have well-established automated refinement flow provided by Computer-Aided Design tools such as System-on-Chip Environment [38].

Therefore, it is valuable to define the Transaction Level Model-based design flow that starts with a general computation model. Existing automatic design space algorithms may not be efficient for such a general model. The algorithms tend to have assumptions that are held by a small set of domain-specific computation models. For example, Erbas et al [18] implicitly assume that computation model is decomposed into completely parallel, non-hierarchical processes communicating with non-hierarchical channels. Therefore, it is not efficiently applicable to Program State Machines, where explicit sequential executions are defined in forms of state transitions.

This dissertation proposes, for general model of computation, several algorithms for

initial design decisions. First of all, this dissertation addresses optimization in mapping of pipelined applications. Previous works [39, 40, 41, 42] have assumed that a period of executions can be flattened into an acyclic directed graph. However, our general model can have many program-states inside a pipeline stage and complex state transitions can be defined over the program-states. This general model can hardly be flattened. Thus, to balance the pipeline stages, design space exploration algorithms must be aware of hierarchy. Besides pipelined structures, automatic design space algorithms for a general computation model should address process scheduling. Process scheduling can greatly impact the performance of the design. In general computation models, processes can run in a completely asynchronous manner and complex synchronization based on data dependencies can be defined. Thus, process scheduling is not highly predictable, so a general models does not accomodate existing algorithms. Nonetheless, some information is available regarding process scheduling, as any two processes can be either parallel or sequential. This dissertation presents heuristic mapping algorithms that put together into the objective functions all of process scheduling, communication, and computation.

For iterative improvements of the given design decisions, this dissertation proposes heuristics that performs local searches near the given design point.

1.6 Scope

The input of this dissertation is a general computation model, design constraints such as cost, and a platform model. This dissertation focuses on mapping only. Mapping is a crucial problem as the platform is very often, for legacy reasons or availability, given and fixed [9]. The optimization goal is to reduce execution time as much as possible while meeting all the other design constraints such as cost. Since the performance

demands of the marketing departments are ever-increasing, a primary issue is still performance. The output is a Transaction Level Model that represents the platform given by the design decisions.

1.7 Tool

The most recently released version of Embedded System Environment is the Embedded System Environment 2.0 [43]. It is intended to realize the concept of Transaction Level Model-based design. Embedded System Environment is a Computer-Aided Design tool set that helps users specify the system and explore the design space concerning hardware/software partitioning, platform selection, and mapping. Embedded System Environment also provides automatic refinements, so as to ensure the required productivity, from the computation model to the Transaction Level Model and from the Transaction Level Model to the Cycle-Accurate Model. Still missing in Embedded System Environment, however, is automatic design space exploration.

By adding automatic design space exploration, this dissertation takes Embedded System Environment to the next level of a system-level design tool set. The automatic design space exploration process that consists of two phases: initial design decision-making and iterative improvements based on Transaction Level Model estimations.

1.8 Contribution

The contribution of this dissertation can be enumerated as follows:

- Data structure for Transaction Level Model-based Design: Any system-level

methodology is accompanied with Computer-Aided Design tools. Transaction Level Model-based design was realized in Embedded System Environment 2.0. The implementation of Embedded System Environment 2.0 failed to provide efficient data structure and API to describe general computation models and broad design space explorations. In this dissertation, data structures and APIs for specification and design space exploration have been redesigned. Moreover, Transaction Level Model generation requires well-established data structure and API, which were missing from Embedded System Environment 2.0.

- A Transaction Level Modeling style that has synthesis semantics: The modeling style of Embedded System Environment Transaction Level Models were proposed with synthesis semantics. However, the style in Embedded System Environment 2.0 allows only very limited number of platforms to be transformed into Transaction Level Models. For Transaction Level Model generation to be allowed for most feasible platforms, Transaction Level Model components are redefined and re-implemented.

- New Transaction Level Model-based design flow: This dissertation proposes the new design flow of Transaction Level Model-based design. Design space exploration is decomposed into two phases depending on the estimation that each phase uses. The first phase is an initial design decision-making based on rough estimation. The second phase is iterative improvements based on Transaction Level Model estimation.

- The algorithms for initial mapping of a general computation model to the given platform: For initial mapping, any existing algorithm can be applied. However, most existing algorithms depend on a domain-specific computation model, and, for a general compuation model, are not efficient. This dissertation, first of all, addresses the pipelined application with complex hierarchy captured with

a general compuation model. Unlike previous works, the proposed algorithms balance the pipeline stages by considering complex hierarchy inside each stage.

- The algorithms for improvements in mapping based on Transaction Level Model estimation: Transaction Level Models are available only for a limited number of design alternatives at a time. Therefore, it requires a set of algorithms different from the ones for initial mapping. Those algorithmes tend to be local search from the given design point. This dissertation suggests heuristic algorithms based on local search.

- Trace-Driven Performance Estimation: Transaction Level Model-based design requires fast, general, and cycle-approximate estimation using Transaction Level Models. Existing works use Transaction Level Model generation and simulation. For a set of applications, Trace-Driven Performance Estimation speeds up, without losing generality and accuracy, simulation-based Transaction Level Model estimations. Trace-Driven Performance Estimation records, at the Transaction Level, communication API calls and executions between communication API calls, thus generating an ordered list of execution and communication events for each process. The events are placed while abstract RTOS, memory hierarchy, and bus models at the Transaction Level are emulated. As simulation of the entire platform is unnecessary, Trace-Driven Performance Estimation is orders of magnitude faster than simulation-based Transaction Level Model estimation.

1.9 Dissertation Overview

The rest of this dissertation is organized as follows. Chapter 2 offers a review of previous works. Chapter 3 explains the latest generation of Embedded System Environment, which is extended from Embedded System Environment 2.0. Chap-

ters 4 and 5 detail the proposed initial design decision-making algorithms: Hierarchy-Aware Mapping of a Pipelined Application and N-Way Clustering and Mapping algorithms. Hierarchy-Aware Mapping of a Pipelined Application addresses mapping of a pipelined application captured with a general computation model to the given platform while execution time is minimized. N-Way Clustering and Mapping algorithms reduces the execution time of the system by putting altogether into the objective functions process scheduling, communication, and computation. Chapter 6 discusses Cycle-Approximate Estimation Based Mapping, which is a set of algorithms for iterative improvements of the given design decisions based on cycle-approximate estimation such as Transaction Level Model estimation. Mapping algorithms based on Transaction Level Model estimation calls for even faster Transaction Level Model estimation. Chapter 7 shows one such faster Transaction Level Model estimation technique: Trace-Driven Performance Estimation. Trace-Driven Performance Estimation is orders of magnitude faster than simulation-based Transaction Level Model estimations while losing neither accuracy nor generality. Chapter 8 offers the conclusion. In Chapter 9, the future work is stated.

Chapter 2

Related Work

2.1 System-Level Design Methodologies

To tackle the high design complexity of contemporary embedded systems, one must address productivity in any design methodology. In order to achieve the required productivity gain in embedded system design, the level of abstraction of the design process should be raised to so called Electronic System-Level (ESL). System-level design typically starts with specification of the system: behavioral models capturing the system's functionality and non-functional design constraints. As there is a huge gap between specification and implementation, the transformation from specification to implementation cannot be completed in a single step. The transformation is divided into pieces at the expense of losing global optimality. Several orthogonal models are defined in different levels of abstractions, design spaces are explored to optimized the required design metrics, and successive refinements from a model to

another follows design space exploration based on the "correct-by-construction" principle. Moreover, automation in such design space exploration and refinements is a must to achieve the required productivity gain. Thus, system-level design methodologies have to be accompanied by Electronic Design Automation (EDA) tools that also target synthesis of the specification models. Densmore et al. [44] reviewed more than 90 different EDA tools, many of which are, however, only for simulation. There have been several EDA tools that intended automated system-level synthesis [9] of the given behavioral model of the system to the platform. Daedalus [15, 16] provided a highly-automated framework for system-level architecture exploration, modeling and platform selection as well as system synthesis. SystemCoDesigner was intended to implement an EDA tool that offers automatic mapping from a given behavioral model to the selected, heterogeneous MPSoC platform. The behavioral model is synthesized by automatic refinements. Peace [17] is an EDA tool that started a Ptolemy II [45]. Peace offers a seamless hardware/software co-design flow including from behavioral simulation through system synthesis. HOPES [46] [47] was, more recently, proposed as an enhancement to Peace. HOPES mainly focuses on generation of MPSoC software to overcome the limitations of OpenMP and MPI. All of these tools, however, do not state approaches to automatic design space exploration facilitated by cycle-approximate estimations such as Transaction Level Estimation. Moreover, these are more on data-oriented applications.

On the contrary, System-on-Chip Environment (SCE) [38] focuses on the general computation models. The design flow starts by capturing the system's functionality with a Program State Machine. Following that, SCE offers well-established definition of orthogonal models at different levels of abstraction and transformation from a model to another. However, in System-on-Chip Environment, design space exploration is not automated.

2.2 Computation Model Based Initial Design Decision-Makings

Many applications such as multimedia streaming applications can run in a pipelined manner. In these applications, balancing pipeline stages plays a crucial role in minimizing the execution time. Ideally, the execution time of the system is proportional to the delay of the stage with the critical path.

As it is a challenging problem, a great deal of papers have appeared, over the past few years, in the literature try to solve it. Lin et al [39] suggested pipeline-aware mapping of an application that is based on heuristics using Strength Pareto Evolutionary Algorithm (SPEA) II [48] and Integer Linear Programming to find Pareto optimal solutions in terms of throughput, latency and cost. Gordon et al [49] devised heuristics to optimize throughput in mapping of a streaming application to the target platform. However, these works require the application to be captured in Synchronous Data Flow (SDF) models. Javaid and Parameswaran [42] conducted multiobjective optimization based on ILP and the heuristics the authors suggested. Optimal configurations for the given, reconfigurable processor-based platform are explored followed by mapping. Benoit et al [40, 41] established the complexity of pipeline-aware mapping problems and suggested heuristics if the time complexity is NP-hard.

All of these studies, however, assume that a period of execution of the application can become an acyclic directed graph without hierarchy. The assumption may not hold in general computation models. In general models, a stage could be decomposed into many subprogram-states, over which very complex state transitions depending on data are defined.

In this dissertation, Hierarchy-Aware Mapping of Pipelined Applications are pre-

sented for such general computation models. It performs mapping of a given pipelined application captured with general computation models to the given multi-processor/core platform. The optimization goal is to reduce the execution time as much as possible under the cost constraints. The application model is decomposed into one or more pipeline stages or, interchangeably, stages. Each stage consists of parallel and/or sequential hierarchical tasks. Each task can be decomposed into sequential and/or parallel tasks, recursively. Since a period of the application can be hardly flattened to an acyclic directed graph, hierarchical tasks will tend to be large in previous approaches. In this dissertation, a large hierarchical stage or task can be divided into small pieces, and each piece can be mapped separately. By partioning a large hierarchical stage, the proposed approach makes the delay of each stage approximately the same.

In computation model based initial design decision-making, process scheduling also greatly impacts the quality of the design. There have, in the past few years, been a large amount of library mapping computation models to platforms. Among them, several studies takes into consideration process scheduling. There have been several early studies focusing on Synchronous Data Flow models [31]. Bonfietti et al [50] used graph-based solutions to optimize throughput in mapping Synchronous Data Flow models to the platform. Oh and Ha [51] presented mapping Synchronous Data Flow models to the platform to optimzes cost with real-time constraints. Lin et al [39] showed mapping Synchronous Data Flow models to heterogeneous MPSoC. Multi-objective optimization based on Strength Pareto Evolutionary Algorithm (SPEA) II [48] and Integer Linear Programming (ILP) is performed to achieve Pareto front in terms of latency, throughput, and cost. They take process scheduling into consideration as well as computation and/or communication. However, in Synchronous Data Flow models, there is trade-off between analyzability and generality. In the models, a task must consume all input data at the beginning of execution and pro-

duce all output data at the end of execution. The number of tokens that the task consumes or produces are statically given and fixed. There is no control dependency among tasks. Thus, Synchronous Data Flow model-based mapping techniques are mainly applicable to data-oriented applications only. The assumptions do not hold in general computation models.

Thompson et al [15], Nikolov et al [16], and Erbas et al [18] perform mapping of process network models to the given platforms. Although both communication and computation are taken into account, process scheduling is not considered.

Ferrandi et al [19] took as an input Hierarchical Task Graph (HTG) [52] and mapped the input to a heterogeneous MPSoC platform. The optimization goal is to minimize overall execution time [19], which is the time difference between the start time of the start task and the end time of the end node. Most computation models can be refined to this HTG and thus the tecniques are generally applicable. However, HTG in its nature does not represent communication overhead. In addition, Ferrandi et al did not address the impact of process scheduling caused by data dependency between parallel processes.

We propose heuristics that take into account process scheduling as well as both computation and communication. The proposed heuristics can take as an input a general MoC to map it to heterogeneous MPSoC while latency is minimized as much as possible and all the other constraints are met. To the best of our knowledge, there is no previous work that maps general MoCs, while communication, computation and process scheduling are involved in the optimization processes.

2.3 Transaction Level Model Estimation

Transaction Level Model estimation plays a crucial role in Transaction Level Model-based design. During iterative improvements of the given design, the design process must depend on Transaction Level Model estimation. For Transaction Level Model estimation to meet the need of the iterative improvement phase of Transaction Level Model-based design, Transaction Level Model estimations should be fast, accurate, and general.

Existing estimation tecniques but Transaction Level Model estimation could use include; Cycle-Accurate Model estimation [20, 21], static analysis-based estimations [22], and trace-driven estimations [23, 24]. Cycle-Accurate Model based estimation is generally very accurate. However, they may not be available for the entire platform especially in early design stages where early design decisions are made. In addition, Cycle-Accurate Model based estimation is too slow so the size of the design space that can be explored is limited. Static analysis-based estimations are not in cycle level. The design will still take the risk of over-design while depending on static analysis-based estimations. Trace-driven estimations are not applicable to custom hardware. In addition, existing trace-driven estimation requires Cycle-Accurate Models or Instruction Set Simulators, which limits the estimation in terms of generality; Generality refers that the estimation should be applicable to both hardware and software and not require Cycle-Accurate Models so that it can be applied to any platforms in early design stages.

Recently, novel, fast, and general cycle-approximate estimation techniques based on Transaction Level Model generation and/or simulation have been proposed [25, 26, 27]. Such simulation-based Transaction Level Model Estimations offer speed, accuracy, and generality at the same time. The speed is close to that of native simulation

as Transaction Level Models are typically written in a single System-Level Design Language such as SystemC [28], natively compiled to the binary for the simulation host, and simulated. Compared to Cycle-Accurate Model estimations, simulation-based Transaction Level Model estimations are order of magnitude faster. The accuracy has been claimed to be close to Cycle-Accurate Models. The error ratio is less than 10% compared to the board measurement [26], and, in this regards, estimations based on Transaction Level Models are claimed to be cycle-approximate. Simulation-based Transaction Level Model estimations are retargetable in the sense that they are applicable to both hardware and software. Moreover, as simulation-based Transaction Level Model estimations do not require any Cycle-Accurate Models, they are applicable to most of the platforms in early design stages.

This dissertation proposes a new trace-driven estimation performed in Transaction Level. Our Trace-Driven Performance Estimation is complementary to simulation-based Transaction Level Model estimations. For applications in which the execution path of each task depends only on data, Trace-Driven Performance Estimation offers order of magnitude faster estimation compared to simulation-based Transaction Level Model estimation without losing accuracy and generality. Since Trace-Driven Performance Estimation generates traces once and places the events in the global timeline whenever new design decisions are made, simulation of the entire platform model is not necessary.

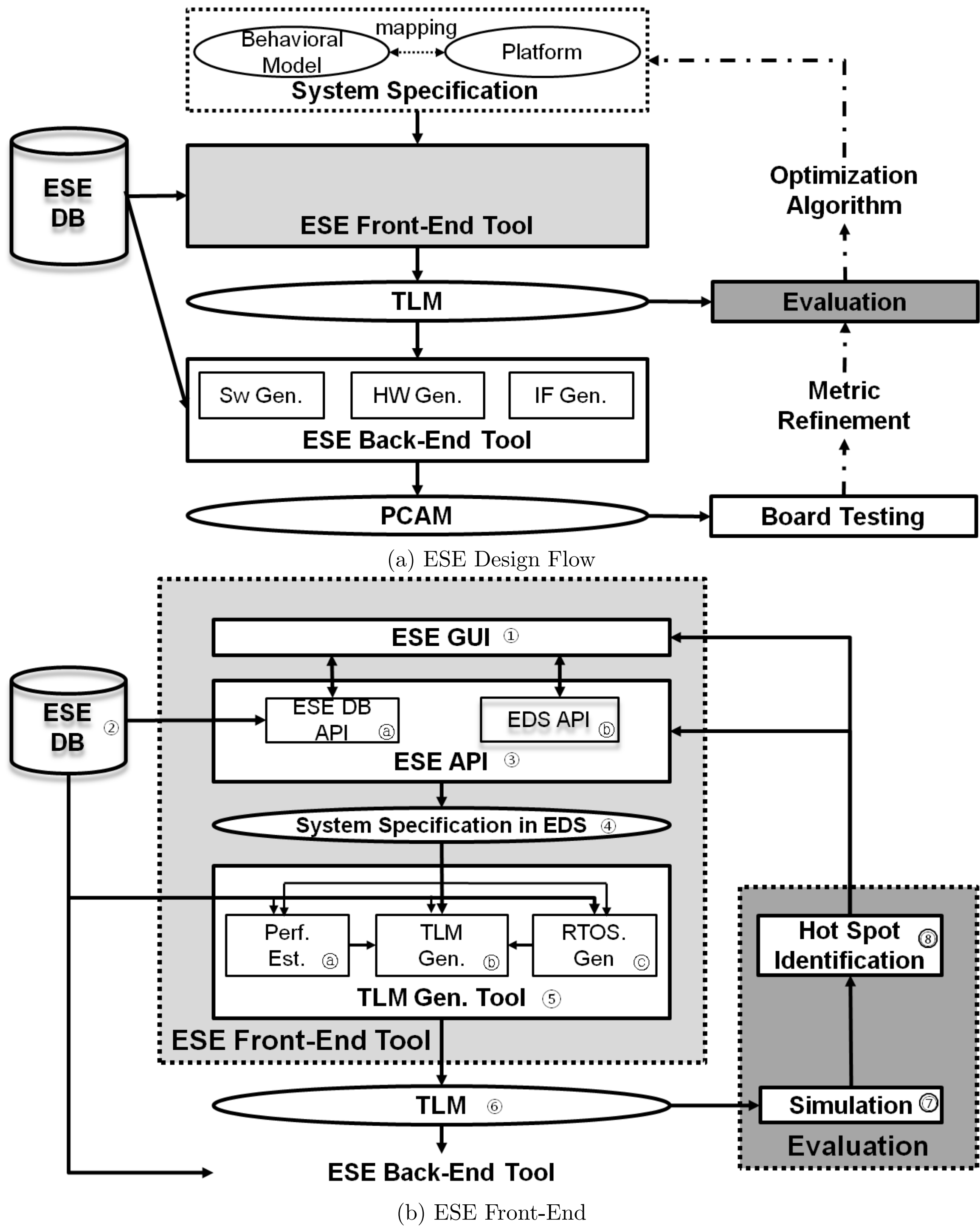
Chapter 3

Embedded System Environment (ESE)

3.1 ESE Idea

ESE is intended to realize the concept of Transaction Level Model-based design. Figure 3.1a shows the design flow with ESE. With the given combination of behavioral model, platform and mapping, ESE front-end tools automatically produces TLM. TLM allows fast and accurate early estimation so designers can explore broader design space. ESE back-end takes TLM as an input and generates PCAM, which can be measured on the board for further metric refinements.

This dissertation is focused on the front-end tool. The main task of ESE front-end tools is the following:



(a) ESE Design Flow

(b) ESE Front-End

Figure 3.1: ESE Design Flow and Front-End Tool

- system specification
- initial design space exploration concerning hardware-software partioning, platform selection, and mapping

- TLM generation
- TLM estimation
- iterative improvements of the given design based on TLM estimation

Figure 3.1b explains ESE front-end in greater details. First of all, designers will write the behavioral model with GUI and source code editors. Platform will be selected with the help of GUI. Following that, design space exploration will be performed either automatically or manually. In the view point of the ESE front-end tools, behavioral model, platform, and mapping is stored in ESE Data Structure (EDS). The platform components are retrieved from ESE Database (ESE DB). Design space exploration is to manipulate EDS through EDS API and ESE DB API. Thus, EDS and ESE API, which consists of ESE DB and EDS API, should be sufficiently efficient for broad design space exploration. On the other hand, EDS includes all required information for TLM generation. This dissertation offers EDS and ESE to meet those needs.

3.2 EDS(ESE Data Structure)

System specification should include all necessary information for TLM and PCAM generation and be well-organized. In Figure 3.2, SW architecture for system specification is depicted. That is the part of ESE for system specification. ESE DB stores platform components and is accessed via ESE DB API. They will be discussed in Section 3.3 and 3.4. Except what a platform component is, everything is kept in EDS so that what EDS has and how EDS is organized are the core.

In our implementation, EDS keeps system specification by separating platform, behavioral model and mapping(④). API is also separated as depicted in Figure 3.2. In

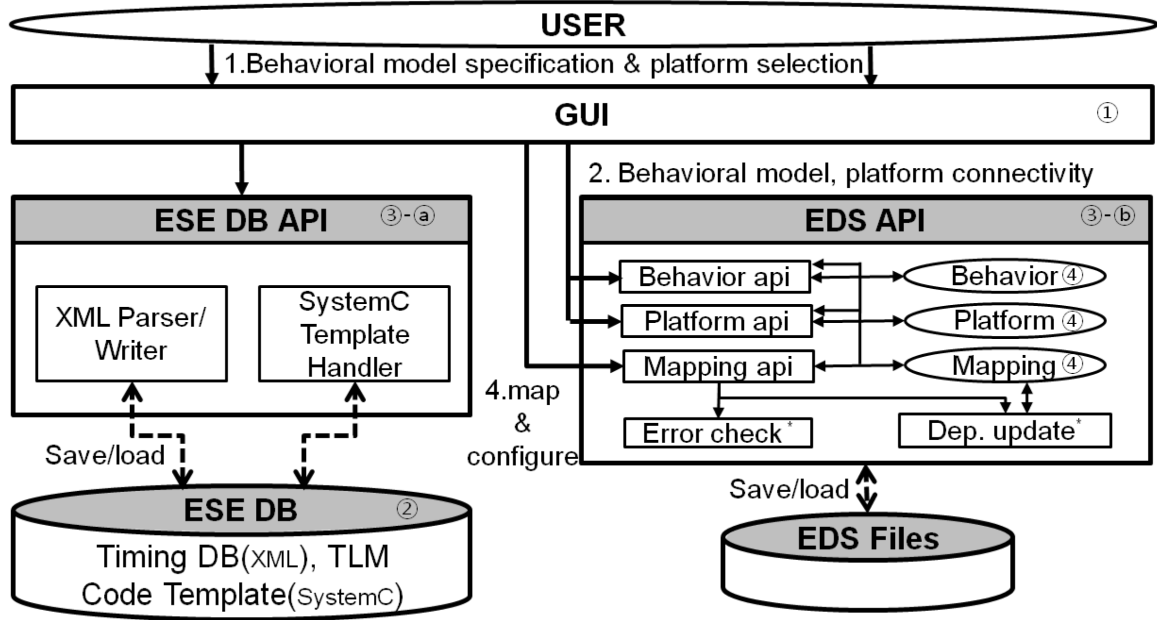


Figure 3.2: SW Architecture for System Specification

addition to error checking modules, modules to update dependencies may be needed. For example, when a PE(Processing Element) is removed, the processes on the PE should become unmapped. This kind of dependencies are handled by the modules. In the rest part of this section, the structure of EDS is explained in section 3.2.1 and what the required changes in system specification and how EDS should be updated on each change is presented in section 3.2.2.

3.2.1 The Structure of EDS

This section is written to show the structure of EDS to keep system specification. Conceptually, data structure for system specification looks like Figure 3.3. For now, we assume that a behavioral model in ESE is a flattened program state machine. It has processes communicating via channels. A process is a C function. Platform is a net-list of platform components such as PEs, Tx(transducer)s and buses. Mapping is

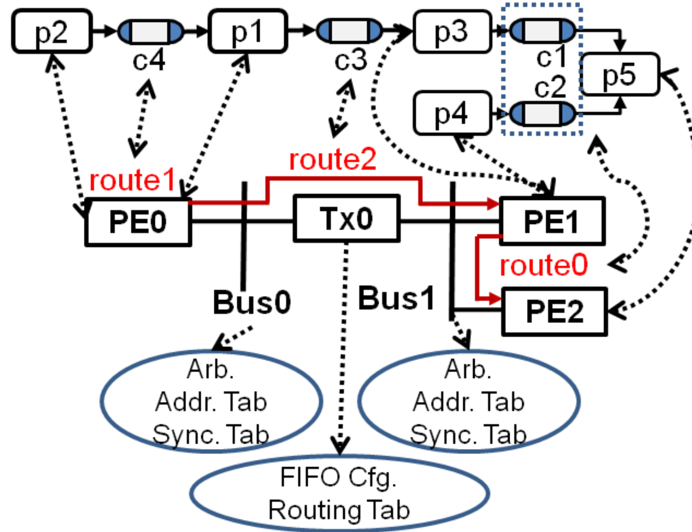


Figure 3.3: The Structure of EDS

basically channel to route mapping and process to PE mapping.

The eclipses in Figure 3.3 with a blue line are not explained, yet. A transducer has an internal FIFO memory. A channel passing the transducer may share the FIFO with other channels or exclusively occupy a designated partition of the FIFO [53]. Therefore, the configuration of the Tx FIFO is given for each transducer. Aside from that, since a Tx may be connected to many buses, the Tx should be able to know from which bus address to which bus, to whom on the bus the data must be transferred. The information is kept in a routing table. A bus may keep a bus address space and have arbitration.

All that have been explained so far in this Section, Section 3.2.1 are intuitive. The only thing that is not explained is bus synchronization table. Our bus model implements point-to-point transactions while communication API for channels are in the network layer according to OSI 7 layers [53] [54]. C3 mapped to route2 is implemented by two point-to-point transactions; from p1 on PE0 to Tx0 and from Tx0 to p3 on PE1. Each transaction is synchronized [53] [54]. One of two communicating partners is the resetter while the other the initiator. A resetter does different from an initiator.

Therefore, bus functions in TLM needs to be different depending on whether the caller is the resetter or not. That is why a bus should keep the list of resetters using the bus. Synchronization table is the list.

Bus0 Sync. Tab

Channel	Send	Recv	Resetter
c3	p1	**Tx0**	p1

Bus1 Sync. Tab

Channel	Send	Recv	Resetter
c1	p3	p5	p3
c2	p4	p5	p4
c3	**Tx0**	p3	p3

Figure 3.4: Synchronization Tables for Buses for The Example in Figure 3.3

Figure 3.4 is an example. For example, only route2 mapped to c3 passes bus0. What bus0 should do for c3 is to transfer data from p1 to Tx0. However, for that, bus0 should know which one is the resetter. In this example, p1 is the resetter on bus0. Likewise, route2 and route0 shared by c1 and c2 passes bus1. Thus, the synchronization table of bus1 has three entries, each of which is for c1, c2 and c3 respectively. In each entry, the resetter on bus1 is specified as well as the sender and the receiver.

Figure 3.5 shows our implementation for the EDS concept. Behavioral model, platform and mapping are separated since unseparated EDS was a major source of bugs in the previous ESE [55]. EDS for a behavioral model includes processes, channels and their connectivity only. There is no platform-related information unlike the previous versions of ESE. In EDS for platform, there are nothing but PEs, buses, transducers and their connectivity.

Mapping in system specification is kept in EDS for mapping. Process-PE mapping and channel-route mapping are intuitive. The only thing needed to be mentioned is the list of routes. A route is one directional list of pointers to platform components,

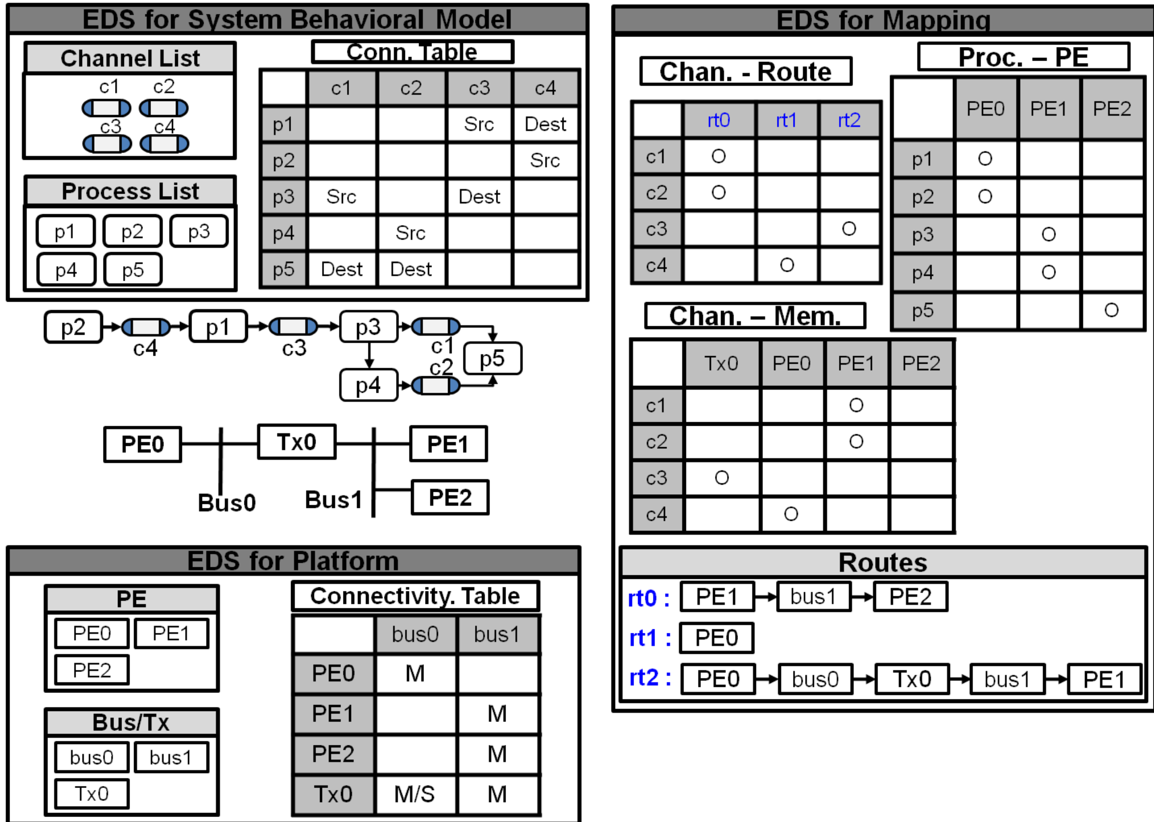


Figure 3.5: The Implementation of EDS

every of which is in EDS for platform. EDS keeps a list of the used routes. A channel may need a memory. The location of the memory may differ even if a single route is shared by two different channels. That is why we need a separate channel-memory mapping, which is also new.

Note that every connection or mapping is written in C++ template, generic 2D mapping table.

The information depicted in Figure 3.5 is not enough. The followings are the omitted information.

A. RTOS type on each PE if applicable

B. Bus address table and synchronization table on each bus

C. Tx FIFO configuration and routing table on each Tx

RTOS type is stored in each PE if needed. Bus address tables and synchronization tables are kept in each bus. Each Tx must keep its own routing table and FIFO configuration so that the data structures are in the Tx.

Following subsections explains how the EDS structure should be updated on each required change in system specification.

3.2.2 Update of EDS on Changes in System Specification

System specification may change during iterative design cycles either manually by designers or automatically by synthesis tools. For example, the designers feels a bus has too heavy traffic so that they make up their mind to duplicate the bus. Or, a PE is identified as a hot spot and the synthesis tools may move some processes on the PE to a different PE.

EDS should have a structure that can be efficiently updated on each change. For that, we need to enumerate the required changes first and then present how EDS can be updated on each change. The followings are the list of the required changes.

A. Change in Mapping

    i. Channel-Route Mapping

    ii. PE-Process Mapping

B. Change in Platform

    i. Addition of PE/Tx/Bus

    ii. Removal of PE/Tx/Bus

C. Change in Behavioral Model

Firstly, mapping may change while platform and behavioral model does not. In that case, a channel may be mapped to a different route while PE-Process mapping does not change. However, note that, when a process is mapped to a different PE, channel-route mappings may need to be changed due to the changed PE-Process mapping.

Secondly, platform may change while behavioral model is not changed. A platform component can be either added or removed. In the former case, mapping does not change. In the latter, everything mapped to the platform component becomes unmapped and should be re-mapped at some point.

Thirdly, behavioral model itself may change. For example, designers may find out that the design constraints cannot meet unless they split a heavy process and map them to multiple PEs. In this case, platform will not change. However, mapping have to be sometimes changed and automatic update of EDS is complicated while manual update with GUI is not a big deal.

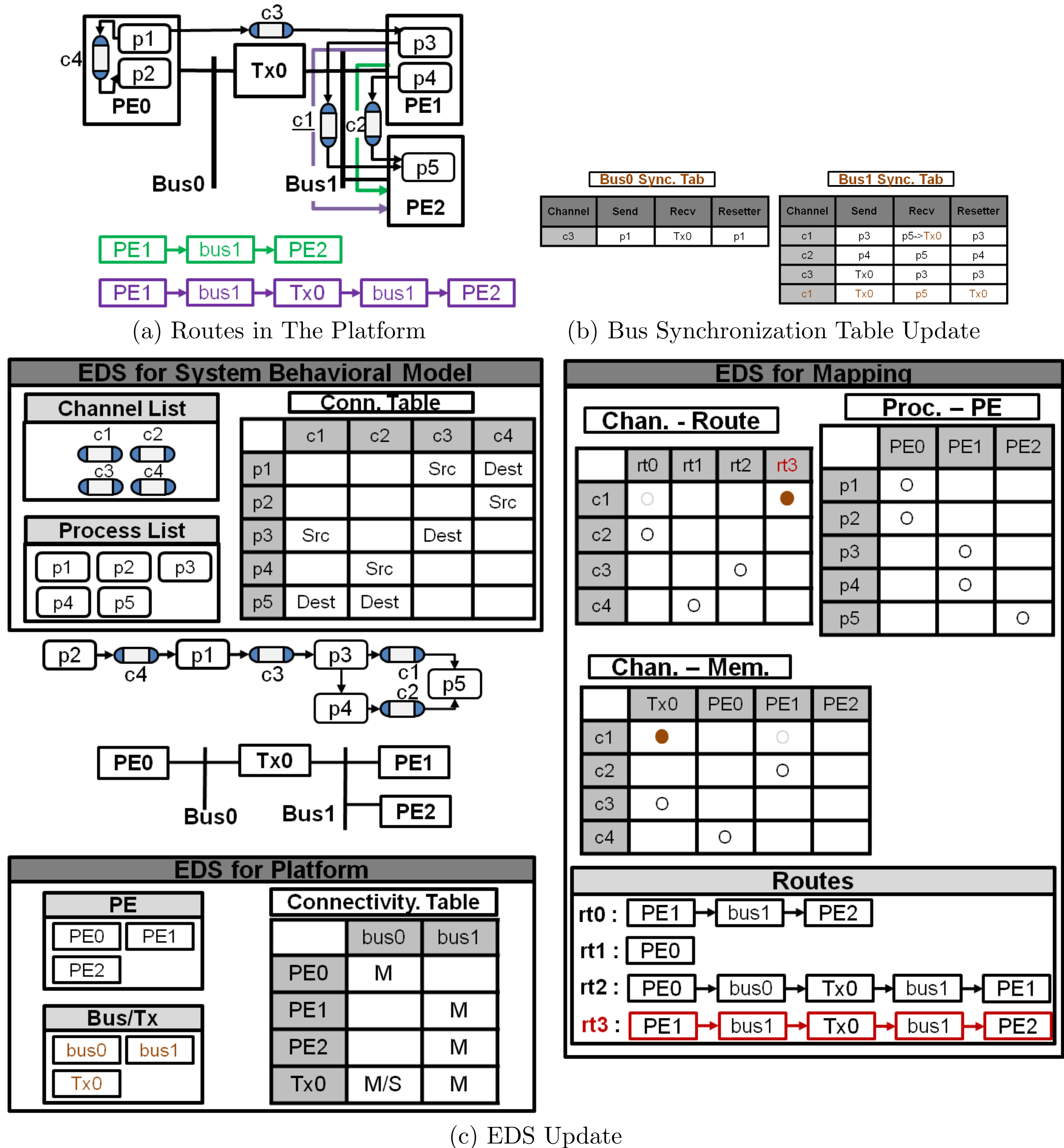
In the following three subsections, these three cases are reviewed in great details.

Change in Mapping

As depicted in Figure 3.6, a channel can be mapped to a different route without changing anything. In the Figure, channel c1 was mapped to route rt0, which does not pass Tx0 and is mapped to route rt3 passing Tx0. This change does not affect PE-Process mapping so that the simplest case.

The list of operations that should be done is as follows.

A. Add a route object to EDS if the new route does not exist



(a) Routes in The Platform

(b) Bus Synchronization Table Update

(c) EDS Update

Figure 3.6: Change in Channel-Route Mapping

B. Update channel-route mapping and channel-memory location mapping in EDS for mapping.

C. Update synchronization tables, bus address tables, Tx FIFO configurations and routing tables

The updates are depicted in Figure 3.6c. Route rt3 is added in EDS for mapping.

Also, channel-route mapping table needs a new column for rt3. C1 is now mapped to not rt0 but rt3. Therefore, channel-route mapping table is updated. Tx0 has an internal memory to save spaces for c1. Designers may decide to the memory location for c1 to Tx0. That is why the channel-memory mapping table is updated in Figure 3.6c. In addition to that, the bus address table and synchronization table of bus1 has been changed as well as routing table and FIFO configuration of Tx0. Figure 3.6b shows the bus synchronization table as an example. Via bus1, p3 was supposed to directly send data to p5. However, since the route for c1 is changed to rt3, c1 is implemented with two point-to-point communication. p3 now has to talk to Tx0 and Tx0 to p5.

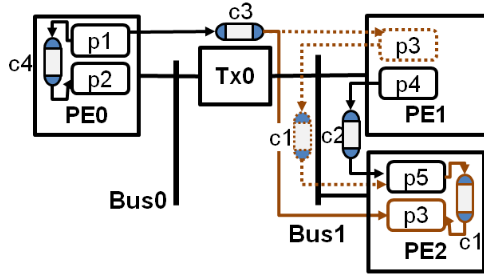
A process can be mapped to a different PE. For example, designers find out that a PE is utilized 95% based on TLM estimation. Since TLM estimation is not 100% accurate, the designers may move some processes on the PE to a different, less utilized PE. Figure 3.7a shows an example. p3 was mapped to PE1 and has been moved to PE2. Note that the change implies extra changes in channel-route mapping. In the example, the route for c1 is changes to PE2 and the route for c3 has been also changed.

The list of necessary operations are as follows.

A. Update PE-Process mapping table in EDS for mapping

B. Change channel-route mapping following i. in Section 3.2.2

Figure 3.7c shows an example. PE-Process mapping table has been updated. p3 is now mapped to PE2 rather than PE1. The channels connected to p3, c1 and c3 needed to find different routes. The routes are rt3 and rt4, which were not in the route list. Thus, rt3 and rt4 has been added to the route list as well as to the channel-route mapping table. c1 and c3 are now mapped to rt4 and rt3 respectively. Memory



(a) Move A Process to A Different PE

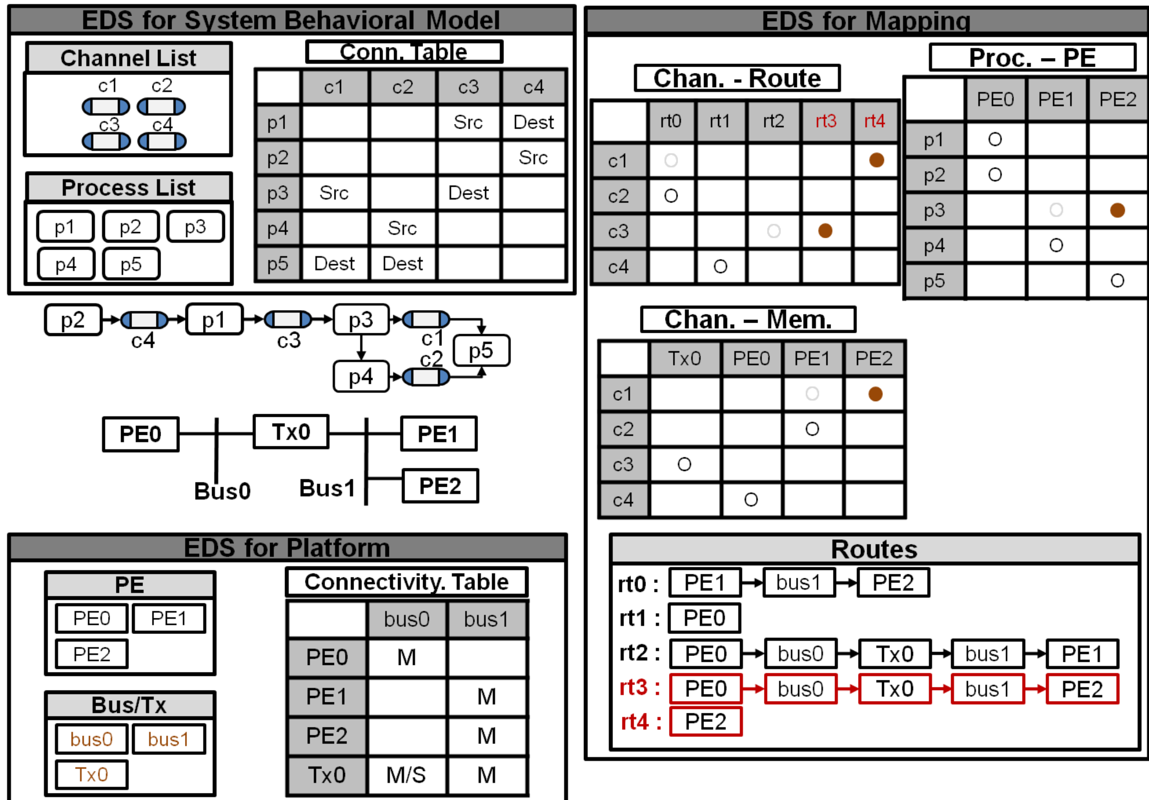
Bus0 Sync. Tab

Channel	Send	Recv	Resetter
c3	p1	Tx0	p1

Bus1 Sync. Tab

Channel	Send	Recv	Resetter
c1	p3	Tx0	p3
c2	p4	p5	p4
c3	Tx0	p3	p3->Tx0
c1	Tx0	p5	Tx0

(b) Bus Synchronization Table Update



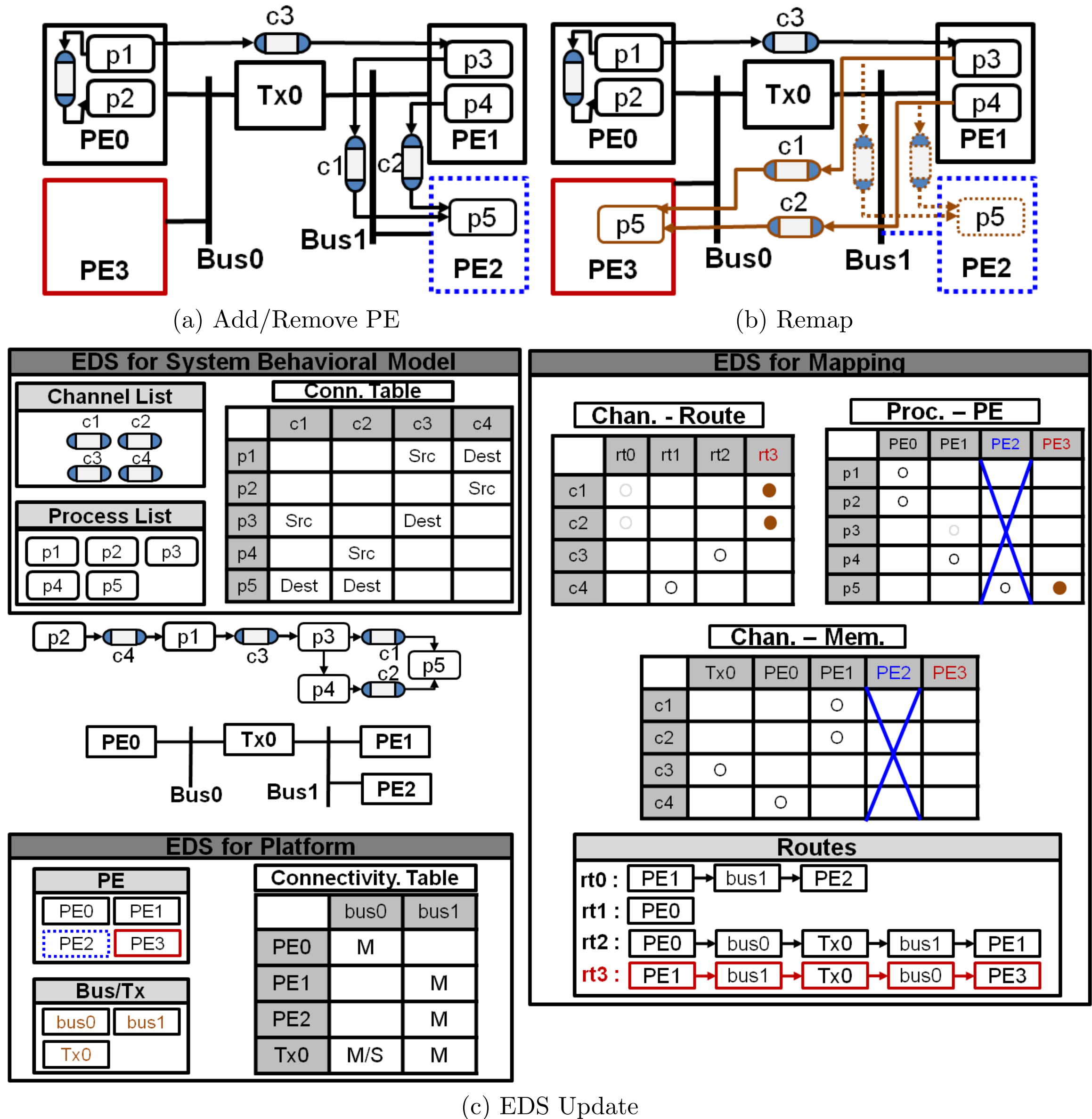
(c) EDS Update

Figure 3.7: Change in PE-Process Mapping

location for c1 was in PE1. This should be changed. In this example, it has been changed to PE2 so that the channel-memory mapping table is updated. Bus address tables, synchronization tables, FIFO configurations for Tx0 and routing tables in Tx0 should be changed. Figure 3.7b shows change in the synchronization table inside bus1. c1 is not passing bus1 no more so that two entries for c1 are removed. Since p3 is moved to a different PE, the resetter for c3, which is connected to p3, may change

and changed in this example to Tx0.

Change in Platform



(a) Add/Remove PE (b) Remap

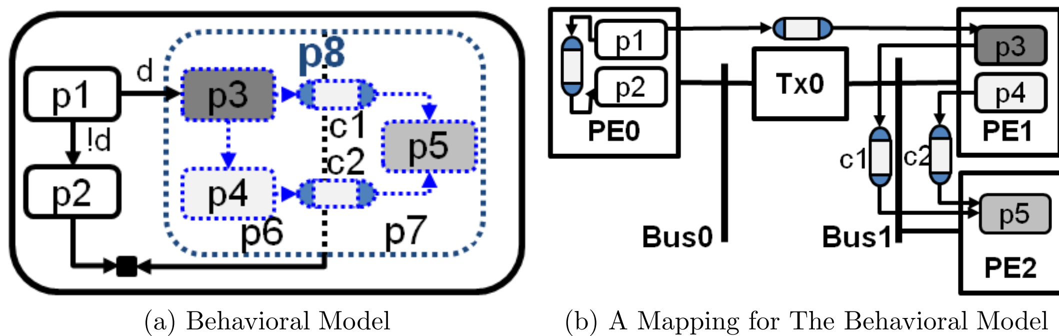
(c) EDS Update

Figure 3.8: Change in Platform : Move A PE

Platform components may be added or removed. Moving a platform component is equal to addition and then removal. Thus, the example in Figure 3.8c shows both of the cases at the same time. In the example, PE2 on bus1 is moved to bus0.

As depicted in Figure 3.8a, PE3 is added and PE2 is removed. Addition is easily done by updating EDS for platform. However, when PE2 is removed, p5, c1 and c2 become unmapped. Therefore, remove a PE implies change in PE-Process mapping explained in the Section 3.2.2. That is what Figure 3.8b is explaining. Removing a bus or Tx is similar. The channels depending on the bus or Tx become unmapped and should be remapped later. Figure 3.8c shows how EDS is updated when a PE is moved from bus1 to bus0. In EDS for platform, PE2 is removed and PE3 is added. PE-Process table is also updated. The column for PE2 are deleted and the one for PE3 is added. p5 is moved from PE2 to the new PE3. C1 and c2 share a new route, rt3. Rt3 is added and channel-route mapping table is updated so that c1 and c2 are now mapped to rt3. Since routes that the channels mapped to have been changed, FIFO configuration and routing table in Tx0 become different. Synchronization tables and address spaces in bus0 and bus1 have been also changed.

Change in Behavioral Model



(a) Behavioral Model (b) A Mapping for The Behavioral Model

Figure 3.9: Change in PE-Process Mapping

Behavioral model also changes. For example, designers may think there is no way to meet all design constraints but splitting a heavy process and map the new processes to multiple PEs.

However, Figure 3.9a shows that automatic update of EDS may not be simple in this

case. In the Figure, the behavioral model is rewritten so that all p8 had is manually moved into p1 and p2. Figure 3.9b should be updated in accordance with the change. However, since the behavioral model may have hierarchy, this is not easy.

On the contrary, redo the mapping from the scratch is not a big deal with the help of GUI. Therefore, the current ESE is asking designers to redo the mapping if they change the behavioral model.

3.3 Transaction Level Model

Purely behavioral models such as ours do not allow accurate estimation. On the contrary, PCAM is ready at late design stages and slow to simulate. TLM is in the middle. TLM is functionally equivalent to the behavioral model and has platform in it to allow accurate estimation on the system.

TLM is a working code functionally equivalent to the behavioral model and having platform inside. In this section, an overview of TLM and TLM structure are given as well as the SW architecture of each TLM components and how the architecture works. This section is organized as follows.

3.3.1 TLM Overview

Figure 3.10 is an overview of a typical example of our TLM. TLM is a net-list of TLM platform components written in SystemC. As depicted, the three core components are PE(PE0, PE1, PE2), Tx(Tx0) and UBC(Bus0, Bus1), which is a bus model. PE composes the computational part of TLM while Tx and UBC the communicational part.

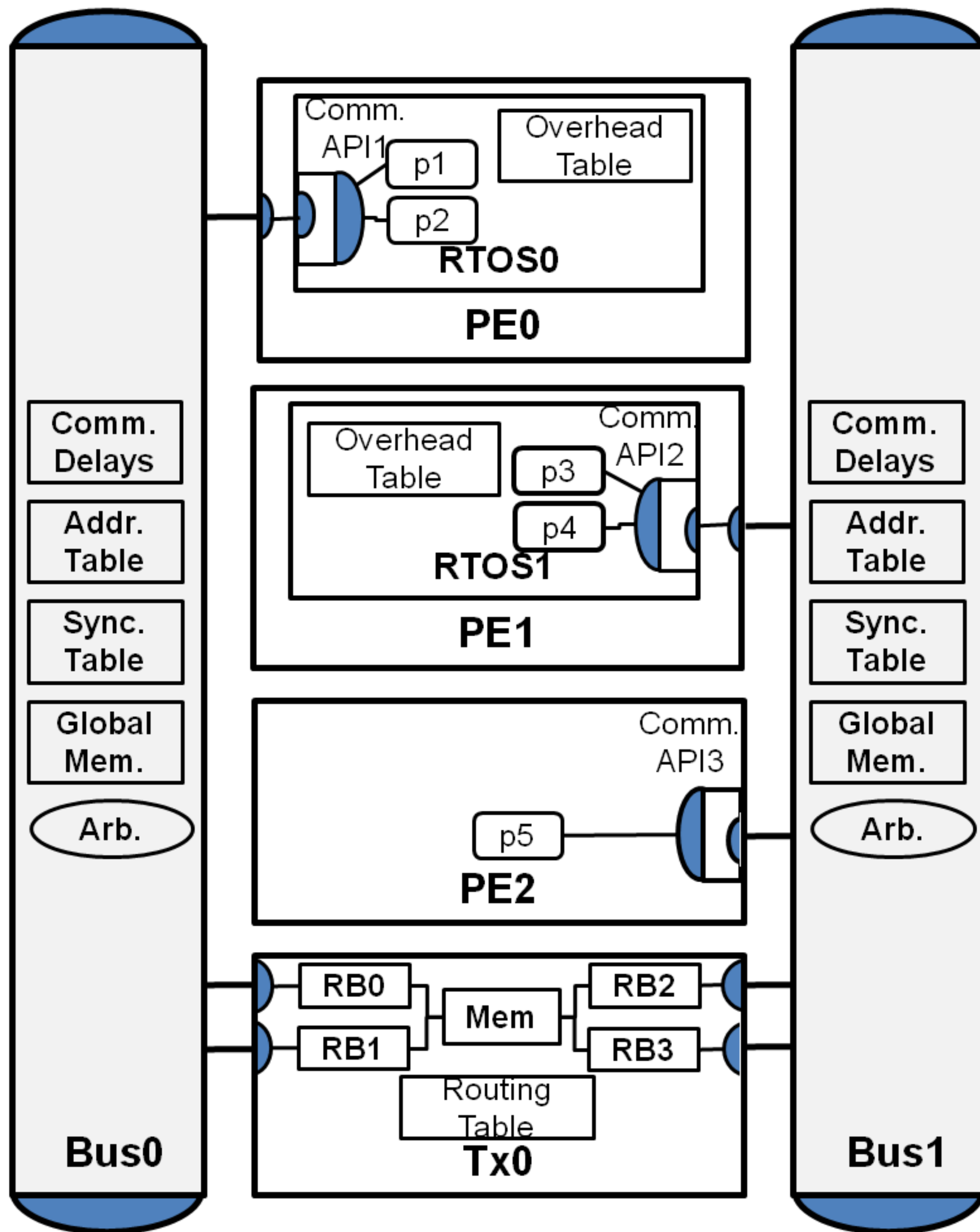


Figure 3.10: Transaction Level Model

Computation means running processes. What runs the processes may be either HW(PE2) or an RTOS on a CPU(RTOS0 on PE0, RTOS1 on PE1). HW directly runs a single, timed process while a CPU only has an RTOS and the processes are not visible to the CPU. Instead, the RTOS runs multiple timed processes. In addition

to the timing delays in the timed processes [26], RTOS also has extra delays due to its operations [25]. Since the amount of the delays may change from RTOS type to RTOS type, from CPU type to CPU type and so on, RTOS keeps its overhead table and wait statements in RTOS refer the table.

Communication means an end-to-end data transaction by calling communication API. For example, p1 sends data to p3 and each of them calls communication API for the purpose. The end-to-end transaction is implemented with a couple of point-to-point communications, each of which needs an UBC or a Tx. For example, for p1 to send data to p3 on bus1, the following steps should follow.

A P1 sends data to Tx0

B Tx0 moves data from Bus0 to Bus1 by using the memory as a buffer

C P3 receives data from Tx0

A and C are done by UBC functions of bus0 and bus1, respectively. B is done by the operations of Tx0.

A Tx moves data from a bus to another or the same bus. For the purpose, a Tx has two request buffers per each connected bus. In Figure 3.10, Tx0 has RB0 and RB1 for bus0 as well as RB2 and RB3 on the side of bus1. Each request buffer takes data from the bus and pushes the data into the memory inside the Tx, or vice versa. RB0 may take data from p1 and pushes it into the memory so that RB2 picks up the data. Or, RB3 may pop data from the memory and sends the data to p5. The routing table keeps necessary information for a request buffer to take its actions. The necessary information will be explained later on in the following subsections.

Before an UBC is briefly reviewed, we introduce definition of a terminology as follows.

Entity or Entity on A Bus: A Communicating Process or Request Buffer of Tx on A Bus

The main task of an UBC is to transfer data from an entity to another entity. For example, bus0 should transfer data from p1 to RB0 by implementing UBC functions and bus1 has to transfer data from p4 to p5, from p3 to RB2 and so on. An UBC has only a set of functions and data structures needed by the functions.

In the following subsections, we will discuss the TLM components one by one in great details.

3.3.2 Computational Part of TLM : PE

What should be discussed in this section is the followings;

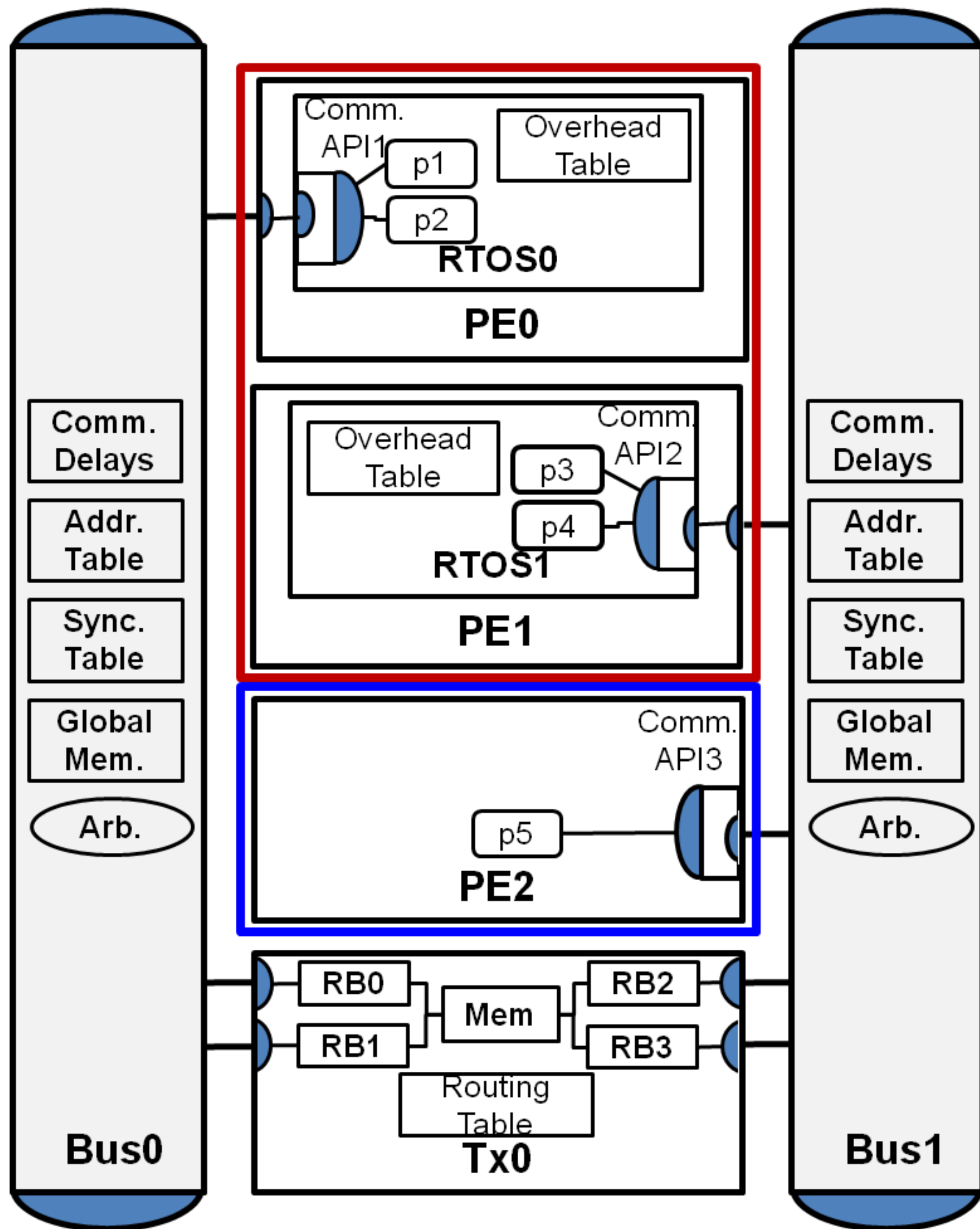
- How a PE runs processes
- How communication is viewed in the view of a process

In this subsection, the former is explained first and then the latter will discussed.

Figure 3.11 shows the structure of two types of PE, CPU and HW. PE0 and PE1 in the red-lined box in Figure 3.11a are CPU types and PE2 in the blue-lined box is a custom HW type. Both of the two types are SystemC SC_MODULEs. In other words, they are C++ classes inheriting SC_MODULE defined in the SystemC library.

HW and CPU, however, are different in how they run processes and, as a result, in SW architecture.

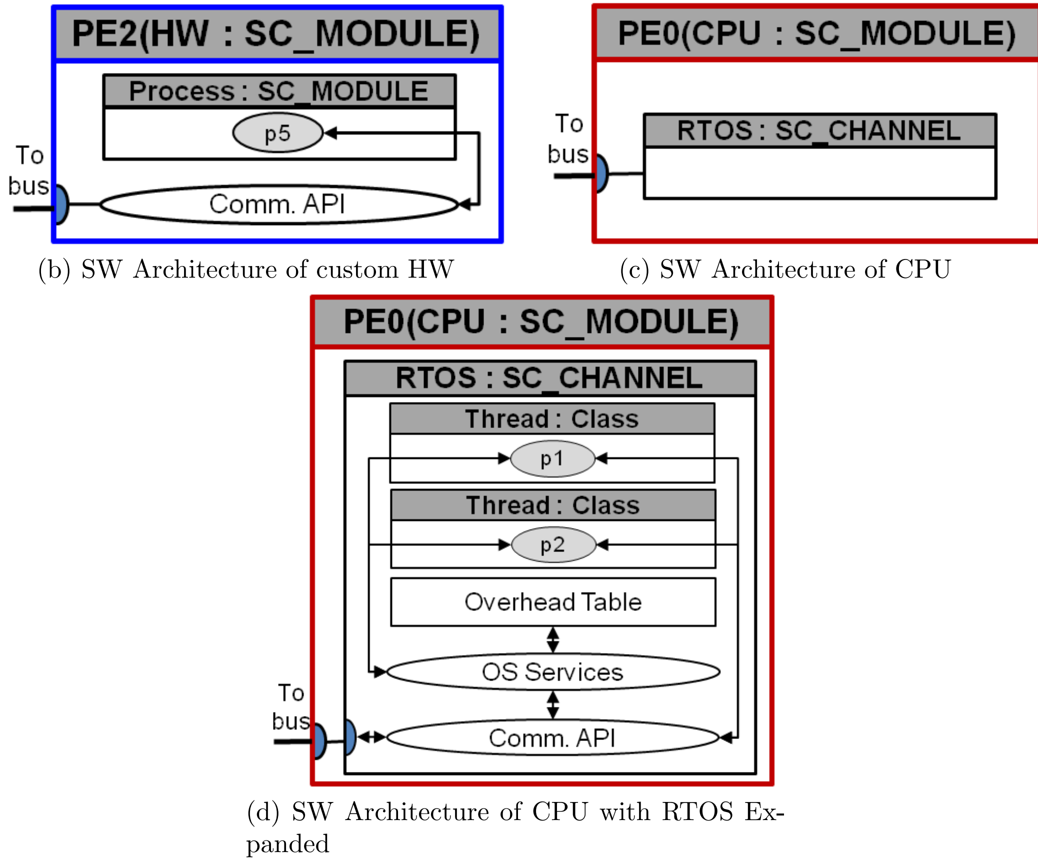
HW directly runs a single user process. Note that the user process is written in C while the entire TLM in SystemC. Therefore, a SystemC wrapper is needed for the



(a) PE in TLM

Figure 3.11: Structure of PE in TLM

user process. In addition to that, the HW is functionally equal to PEs in functional TLM, which is not discussed here, except that a PE in functional TLM runs multiple processes in parallel. To be reused in functional TLM as well, the SystemC wrapper



(b) SW Architecture of custom HW

(c) SW Architecture of CPU

(d) SW Architecture of CPU with RTOS Expanded

Figure 3.11: Structure of PE in TLM (Cont'd)

is written in SystemC SC_Module. The SystemC wrapper is called 'Process' and only runs a single user process as a SC_THREAD. An example is depicted in Figure 3.11b. PE2 is a custom HW running p5. Therefore, it has a sub module, which is also SC_MODULE called 'Process' and the sub module runs p5 as a SC_THREAD.

On the contrary, as depicted in Figure 3.11c, a CPU such as PE0 and PE1 can not even directly see the processes. What runs the processes is the RTOS on a CPU. Therefore, the SW architecture of a CPU is simple; it has an RTOS, which is an SC_CHANNEL. A CPU has almost no code but instantiating and initiating its RTOS.

Primitive Operations	Estimated Delays
Context Switch	45 cycles
IRQ Return	17 cycles
Task Creation	33 cycles
...	...

Table 3.1: Example of RTOS Overhead Table

All complexities are implemented in the RTOS models, an example of which is depicted in Figure 3.11d. RTOS in the Figure keeps multiple processes, p1 and p2. As explained, each user process require a SystemC wrapper, which is a C++ class called 'Thread'. RTOS operations such as scheduling, interrupt handling and so on are implemented as the member functions of the RTOS class. Each of the functions needs an overhead value since our RTOS models support RTOS overhead estimation. The delay value for each operation differs from RTOS type to RTOS type, from CPU type to CPU type and so on. One easy way for all RTOS instances to share a common RTOS class is that every RTOS instance keeps its overhead table, a small data structure, and the RTOS services read the table. Figure 3.11d shows that as well. A very simplified example of an RTOS table is shown in Table 3.1. The table is obtained with the help of the techniques introduced in [25]. We will not discuss that in great details.

How an RTOS model sequentializes the execution of multiple processes is complicated. Basically, an user process is bound with an event, which is an sc_event. The RTOS model has the event object with the corresponding 'Thread' object of the user process. When the user process is suspended, the process called either a wait statement or other system call to explicitly yield the CPU. Inside any system call including a wait statement, there is the event waiting. Therefore, when the RTOS scheduler resumes the process, it notifies the event bound with the process at last.

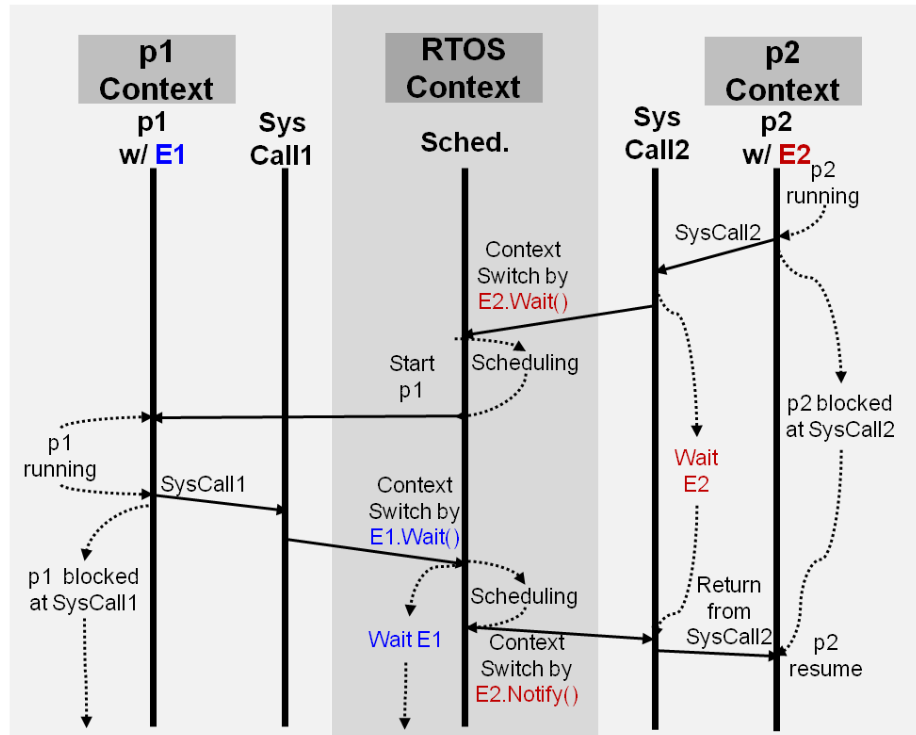


Figure 3.12: Sequential Execution of Processes on RTOS Model

Figure 3.12 shows an example, where p1 and p2 are running on the same CPU. P2 was already running and p1 had not started yet. The RTOS is non-preemptive so that the context can be switched only by explicit yielding through system calls. P2 called a system call, SysCall2. Since, inside the system call, there is a wait statement for the event, E2, bound with p2, p2 began waiting for notification of E2 and became suspended. On the other hand, RTOS took control, did scheduling and made a decision to initiate p1. RTOS switched the context so that p1 began execution. P1 also called a system call, SysCall1. RTOS took the control again. Since p2 was ready to resume at that time, RTOS awoke p2 by notifying the event, E2. Then, the wait statement for E2 can return. In addition to that, as a result, p2 has returned from SysCall2. In the viewpoint of p2, it called SysCall2, was suspended, waited for the return from SysCall2 and has just resumed execution. Note that SystemC does not have representation for pre-emption. Pre-emptive kernels can be modeled by adding the routines polling the interrupt flags to wait statements at the end of basic blocks.

A basic block cannot be pre-emptied in the middle of execution.

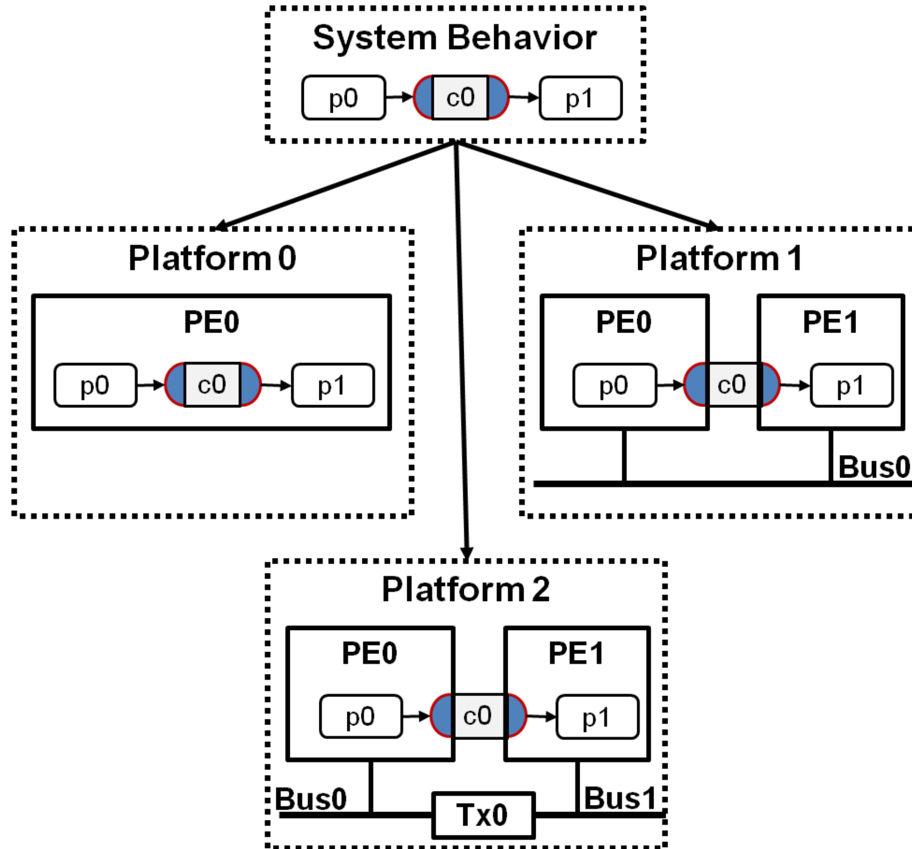


Figure 3.13: Same Communication API with Different Implementation

From now one, how communication can be viewed from the prospective of an user process will be explained. Communication API is a set of functions called by user processes. Each function in communication API is a C function. In the viewpoint of the user processes, communication API are not different. If p0 in Figure 3.13 calls the send function in communication API in the specification level, the process will call the same function in TLM. However, the implementation is different from platform to platform and depends on the location of the communicating processes.

The right communication API is implemented as a global function. As depicted in Figure 3.11, an RTOS model and HW have pointers to the proper functions in communication API, which are used by the user processes on the RTOS or HW.

Note that communication API may be implemented on the top of the UBC functions. The UBC functions can be called through bus ports to the UBC. The bus ports are depicted as the blue semi-circles in Figure 3.11. Thus, there is still one more problem; how a function in communication API identifies the bus port that it should access.

The solution is as follows. If p1 and p3 uses the same send function, we duplicate the send function and give different names. Each copy of the send function is bound with the right bus port. For example, we duplicate the send function and give 'send_p1' and 'send_p3' as the names of the copies. Then, the bus port to bus0 is bound with 'send_p1' while port to bus1 with 'send_p3'. Following that, RTOS0 in PE0 will have a pointer to 'send_p1' while RTOS1 in PE1 a pointer to 'send_p3'.

The actual implementation of communication API is not covered in this Section. It will be covered in Section 3.3.3.

Estimation in computational part is categorized into two groups; user process estimation and RTOS overhead modeling. RTOS overhead modeling is realized by RTOS member functions' referring its overhead table and calling wait statements. That is implemented during TLM generation. However, user process estimation solely depends on the timed processes.

3.3.3 Communicational Part of TLM

In a behavioral model, communication is done via channels. Since processes can be mapped to arbitrary PEs, user-level functions in communication API should provide end-to-end communications.

This end-to-end communication is actually implemented by one or more point-to-point communications. UBC and transducers are the components that implement

point-to-point communications.

An UBC model moves data for an entity on the bus to another over the bus. On the other hand, a Tx moves data between two buses connected to the Tx. Note that the two buses are not necessary to be different. For example, a Tx, Tx0, can move data from p0 on bus1 to p1 on the same bus, bus1. This case is not meaningless since the memory that Tx0 has inside serves as a smart, shared memory to both p0 and p1.

This section will explain the structure of communication components and show how communication in TLM works. We will see the structure of the components one by one. Following that, how the structures make communication work in TLM is reviewed.

Universal Bus Channel(UBC)

Figure 3.14 shows examples of UBC. Bus0 and Bus1 are the examples. The role of an UBC is to move data from an entity to another entity on the UBC. For example, bus0 moves data from p1 to RB0 or RB1 and bus1 from p4 to p5 and so on.

Each entity does data transaction by calling UBC user functions. The list of user functions are enumerated in [54].

A. Send

B. Receive

C. Write

D. Read

E. Memory Service

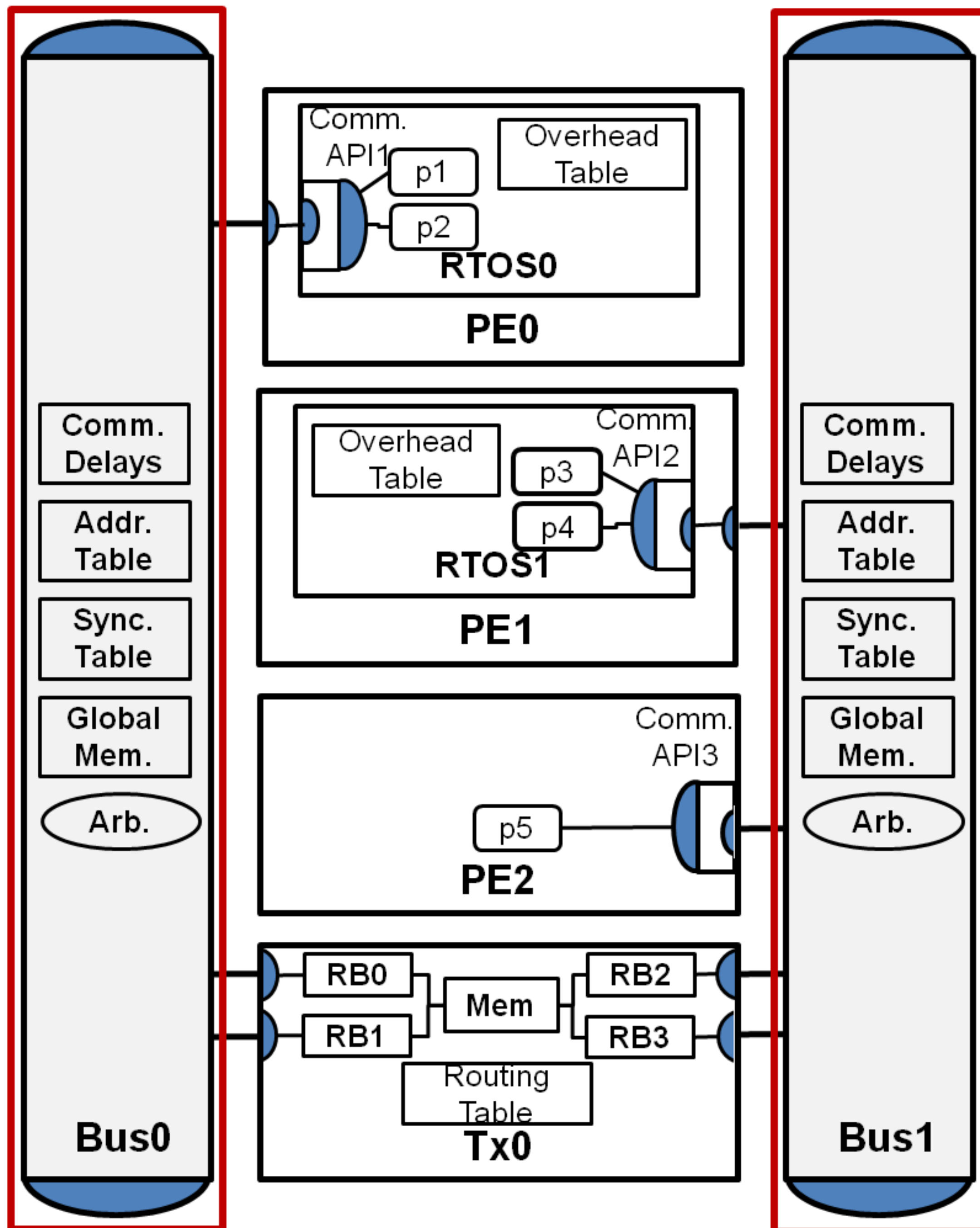
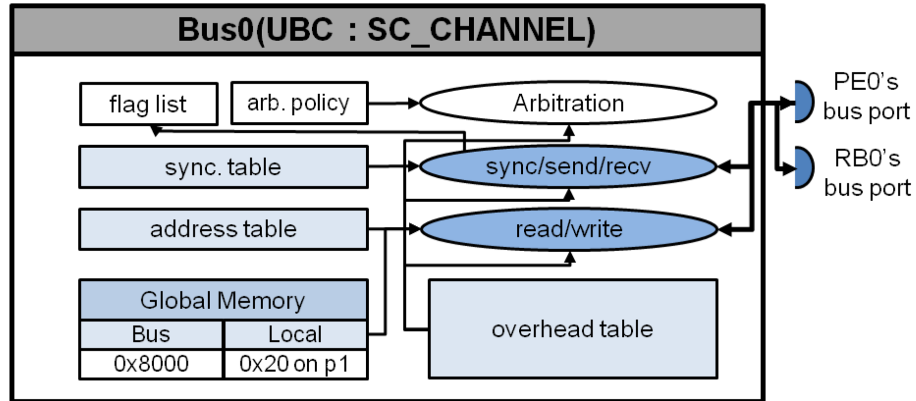
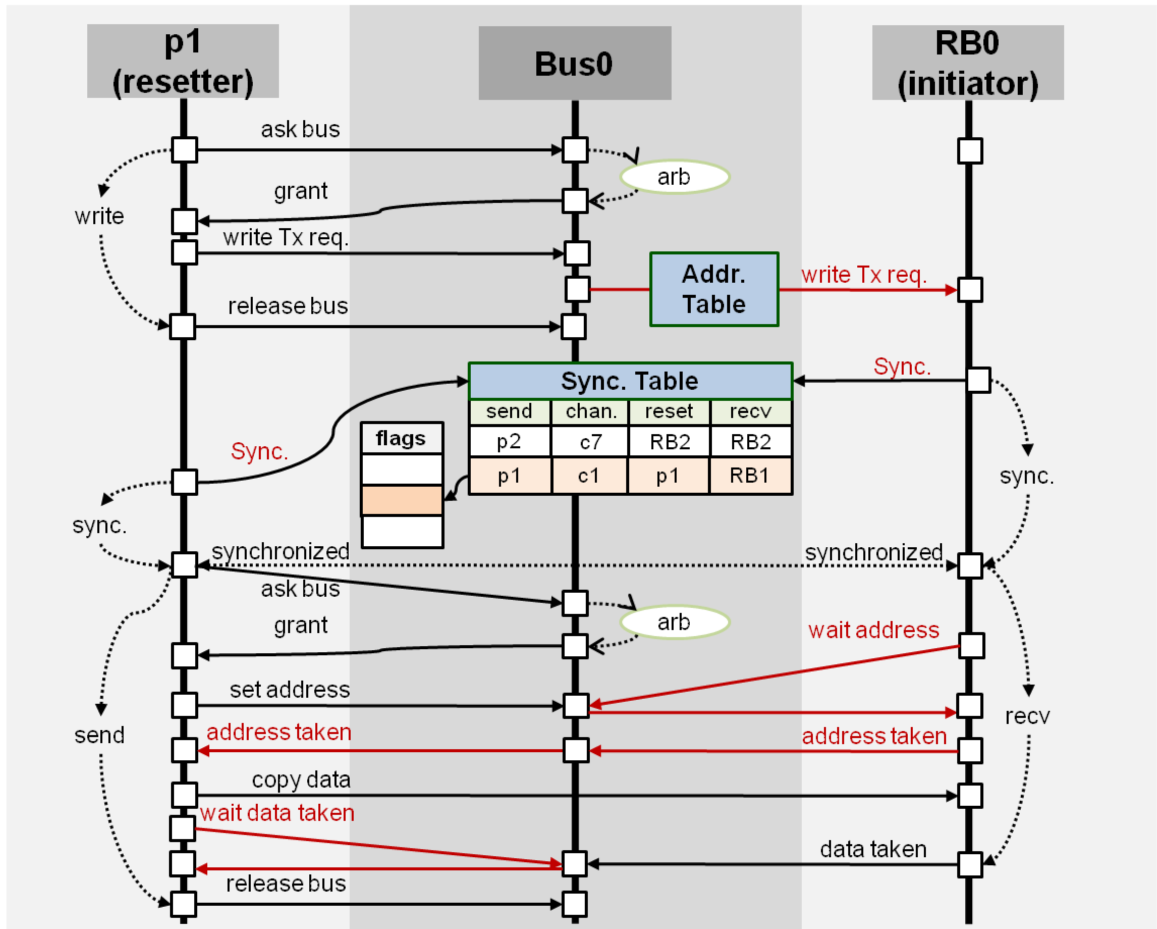


Figure 3.14: UBC

In addition to that, if two entities on the bus do a synchronized transaction, synchronization by setting/resetting a flag is implemented as UBC functions. For memory channels, test-and-set functions are needed as well. [54] provides more details.



(a) SW Architecture of UBC



(b) Functions and Data Structures in Figure

Figure 3.15: The Structure of UBC

Figure 3.15a shows SW architecture of UBC. Basically, our UBC model is SC_CHANNEL in SystemC. Therefore, through the bus ports bound with a given UBC, entities can execute UBC functions. In the Figure, PE0, RB0 and more may call UBC user func-

tions in the blue eclipses through bus ports depicted as the small, blue semi-circles.

Inside UBC, in addition to the user functions, arbitration is also implemented. Arbitration policy may differ from bus to bus. We do not want to implement a new UBC code whenever arbitration policy changes. Therefore, all arbitration functions are implemented in a single SC_CHANNEL, UBC, and selected at runtime depending on the arbitration policy. The function looks like Algorithm 1. UBC has a variable to represent the type of arbitration policy.

Algorithm 1 Runtime Selection of Arbitration

1: $ArbId = read\ arbitration\ policy\ variable$
2: **if** $ArbId = FCFS$ **then**
3:     Do First Come First Served
4: **else if** $ArbId = PRIORITY$ **then**
5:     Do Priority Based Arbitration
6: **else if** $ArbId = Something\ Else$ **then**
7:     Do Something Else
8: **end if**
9: **return** $Who\ Gets\ Bus$

The UBC functions as depicted in Figure 3.15a read the overhead table for communication estimation. The delays for each UBC primitive operations are determined at TLM generation time and are put together on the overhead table.

The user functions are categorized into two groups depending whether the function is for synchronized communication or not. The synchronization table and flags are used by functions for synchronized communication. The address table is used by read and write, which are not for synchronized communication. All of the tables and the

list of flags are simple C++ data structures inside UBC, the SC_CHANNEL.

Figure 3.15b shows a typical transaction scenario and how it is implemented. Actually, in a big picture, the resetter and the initiator are synchronized either by interrupt or by polling, the resetter takes the bus(again in case of polling-based synchronization) and begins the transaction. The Figure is thus a expansion of the big picture.

First, the resetter p1 should write a request on RB0 so that RB0 can be ready to do the transaction. That is done by p1 calling the UBC function, write. Inside the write function, the followings come after the function call. P1 asks the bus, waits for the bus and is granted. Bus0 may execute the arbitration function to choose the entity to be granted. Therefore, p1 being granted may take time. Depending on implementation, p1 can yield the CPU. Once bus0 is granted, p1 writes a request on RB0. UBC needs to have a mapping between the given bus address of RB0 and the actual address on the host machine to be written. The actual address is included in the RB0 object. The address table has this kind of mappings. That writing may fail with timed-out, success or something else. In any case, the bus is released and the write function returns. The procedures in this paragraph model writing a request on a memory-like entity, request buffer, over a bus.

So far, RB0 notices that there is a request to serve for p1. Since RB0 may have multiple requests from a couple of different sources on the bus, taking the request from p1 may take time.

Synchronization between RB0 and p1 follows. That is done in either one of the two ways; interrupt-based and polling-based. Both of them are implemented with a flag and functions inside UBC. [54] provides more details on that.

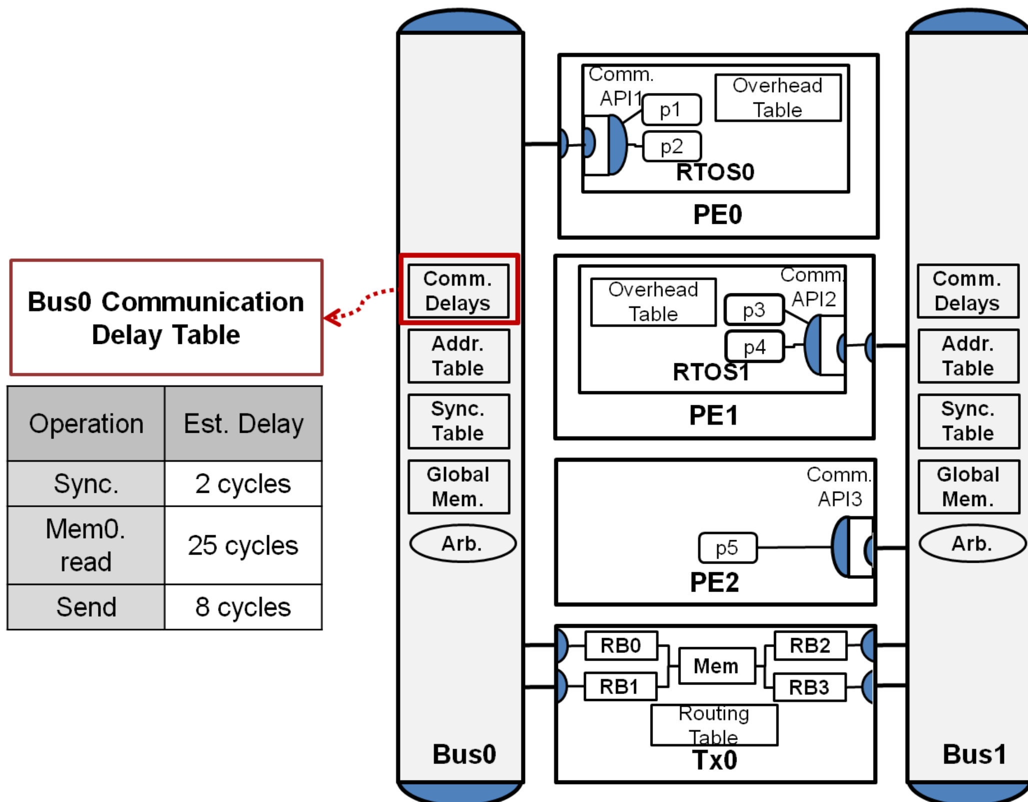
UBC keeps the list of the flags used in synchronization although the communicating processes or Tx should keep the flags in real world. The number of the flags are

scalable. We do not want to write a different UBC class whenever we change the list of synchronizing pairs. That is the reason why we keep a re-sizable array for the flags and a synchronization table to identify the right flag depending on the header of the message, which includes source/destination process and/or the channel ID. In addition to that, we have only two different UBC functions for synchronization itself. One is for interrupt-based synchronization and the other polling-based. Both of them are implemented and stored in UBC. Note that any of them needs to know which entities are trying to synchronize. Therefore, the synchronization functions has to access the synchronization table.

Once the two entities are synchronized, the resetter may try to get the bus. On the other hand, the initiator is completely ready to give or take the data. After the resetter asks the bus and is granted, the address and the data will be put on the bus and transferred. After a certain handshaking protocol, the resetter will release the bus.

The procedures are modeled as send/receive function of UBC in our TLM. Either the sender or the receiver can do the tasks of the initiator and vice versa. In Figure 3.15b, p1, the sender is the resetter and the receiver, RB0 is the initiator. Therefore, inside the send function, what happens to p1 is as follows. P1 asks the bus and is granted. Once p1 gets the bus, p1 set the address on the bus and waits until the address is taken by RB0. Following that, p1 lets the bus know the size, the source address, where the data is and so on. After that, p1 waits until RB0 notifies p1. Once p1 gets the data taken event from RB0, p1 releases the bus and returns from the send function. Note that if the resetter calls receive function to get data, the function does the same thing as the resetter version of send except the direction of the data. On the other hand, inside the receive function called by an initiator, the followings happen as depicted in Figure 3.15b. RB0, the initiator waits for the address set. Once it is

set, RB0 notifies the resetter by signaling the address taken event. Following that, RB0 can take or give the data. Finally, RB0 rises the data taken event and returns from the receive function. Likewise, the initiator version of send function is the same as the initiator version of receive function except the direction of data flow.

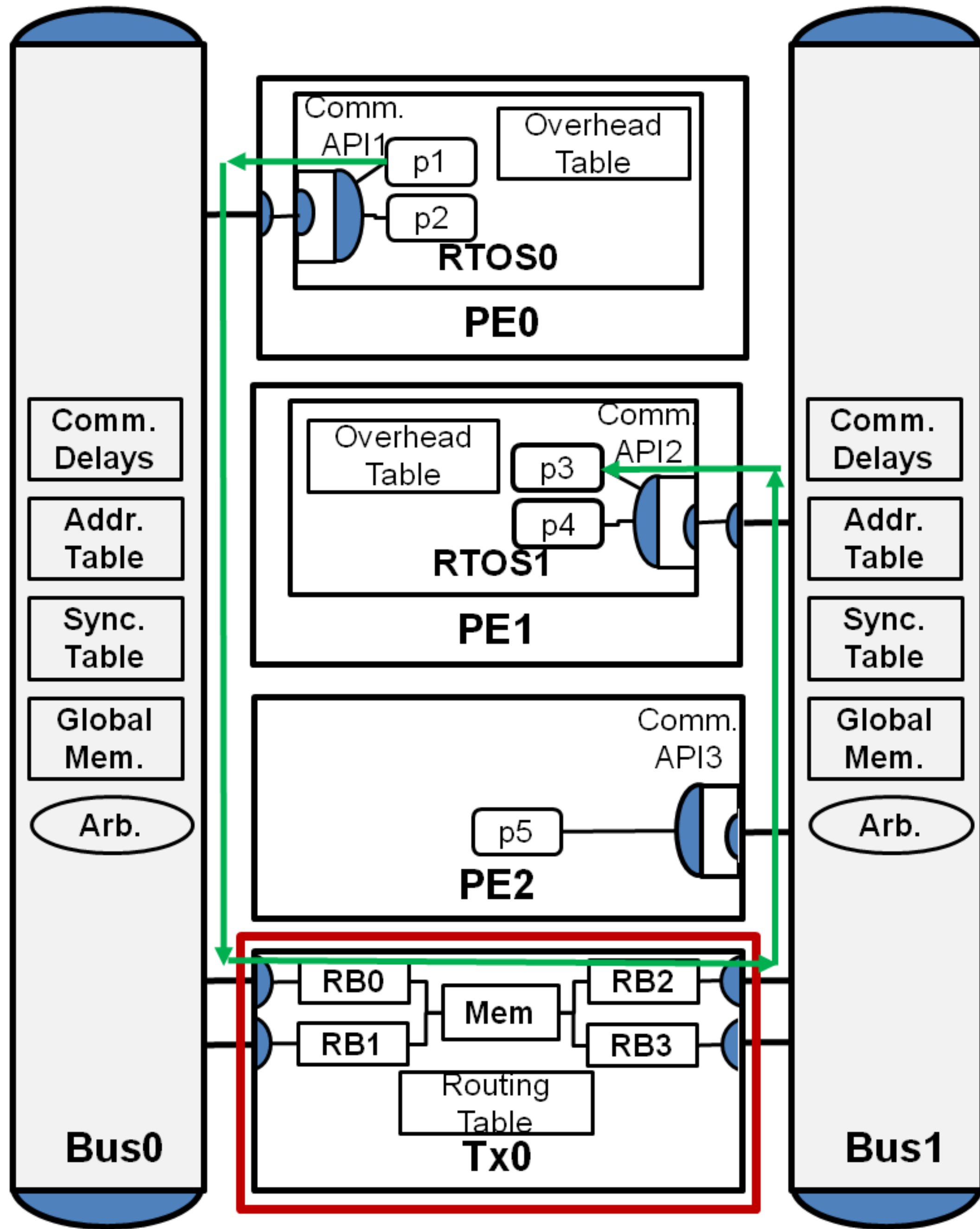


Operation	Est. Delay
Sync.	2 cycles
Mem0. read	25 cycles
Send	8 cycles

Figure 3.16: Communication Delay Table in UBC

Even though the communication delay table is not explicitly depicted in Figure 3.15b, note that all UBC functions have waits statements inside and the amount of time to wait is determined by reading the communication delay table. A simplified example of communication delay table is shown in Figure 3.16. Bus0 keeps the table. On the table, the delay of some UBC operations are enumerated. The table is read by all the UBC functions of bus0.

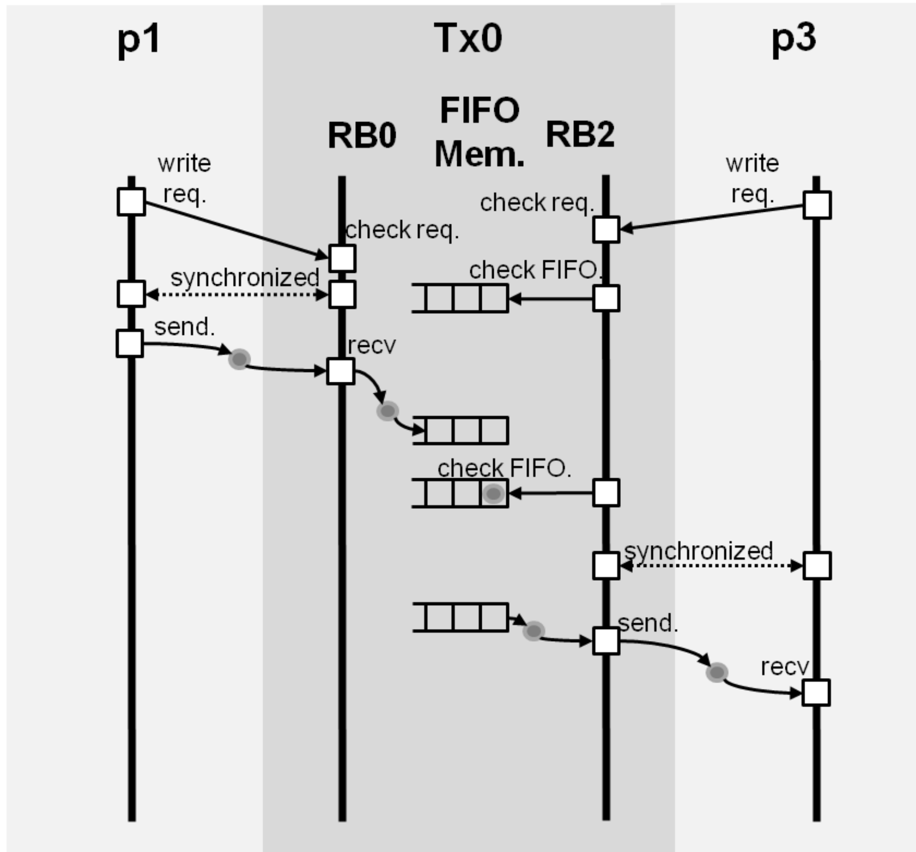
Transducers(Tx)



(a) Tx

Figure 3.17: Data Transaction Scenario in Tx

A Tx is a complex bridge serving as a router as well and connected to one ore more buses. Two communicating processes may not be connected to the same bus. In



(b) Data Transaction (source : [56])

Figure 3.17: Data Transaction Scenario in Tx (Cont'd)

Figure 3.17a where p1 sends data to p3(according to the green-lined path), p1 and p3 are connected to two different buses, bus0 and bus1 respectively. The purpose of transducer is to facilitate this kind of multi-hop transactions. That happens, for example, when IPs from different vendors are put in a system together. Each IP uses its own bus protocols so that all of them cannot be connected to a single type of bus.

The basic functionality of a Tx is to receive data from the sending process or other Tx and to send the data to the receiving process or another Tx. For example, Tx0 in Figure 3.17a receives data from p1 and sends the data to p3.

In that sense, a Tx is similar with a bridge. However, a bridge does not synchronize with its communication partner so that the communicating processes should synchro-

nize with each other. If Tx0 were a bridge, it would not synchronize with p1. On the contrary, a Tx does synchronize and has a buffer. Since Tx0 in Figure 3.17a does synchronization phase with p1, p1 can continue its execution without being synchronized with p3. In addition to synchronization and buffering, a Tx also does routing. They are differences between a Tx and a bridge.

Figure 3.17b shows more details on that. A request buffer such as RB0 or RB2 can work only when a request is written on it. That is why p1 and p3 also depicted in Figure 3.17a write a request on RB0 and RB2, respectively. The processes are expecting being synchronized with and doing data transaction by send/receive with the request buffers. What a request buffer is doing is checking the list of the written requests, doing synchronization over the connected bus if the request is selected and the FIFO has item and moving the data from/to the bus to/from the internal FIFO memory. That is how a Tx does synchronized data transaction and serves as a data buffer for outside world. In addition to that, even though RB2 is selected in Figure 3.17b to send the data to p3, Tx0 was able to choose a different request buffers according to the route assigned to the channel. A Tx0 must select a right request buffer to move the data by following the right route. That is the duty of Tx in routing.

Figure 3.18 shows the internal structure of Tx0 in Figure 3.17. SW architecture of a Tx more or less looks like Figure 3.18. The three main components are request buffers, an internal FIFO memory and a routing table. The routing table is a C++ class, the FIFO memory is implemented as a SC_CHANNEL and the request buffers as SC_MODULEs. The Tx module itself has all of them connected but almost no function.

The FIFO memory serves as a data buffer. It has a storage, which may be either shared or partitioned. If the storage is partitioned, then, each channel uses the FIFO memory has its own, separated partition. Otherwise, the entire FIFO is simply shared

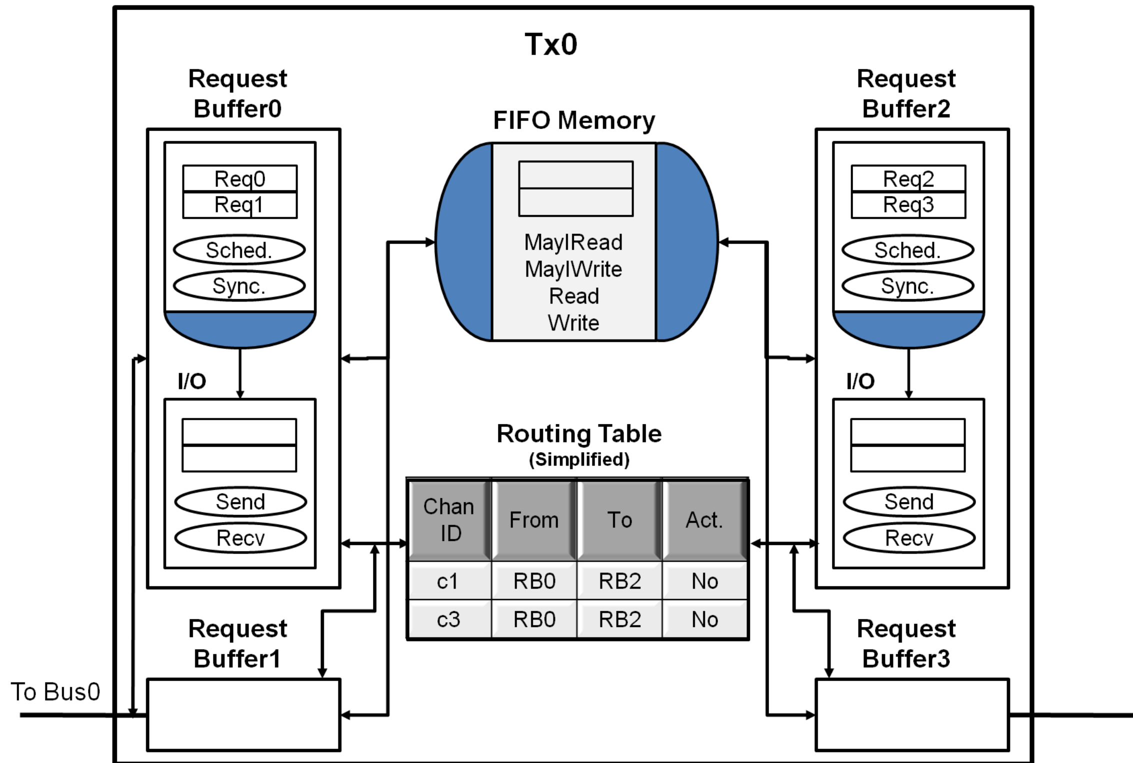
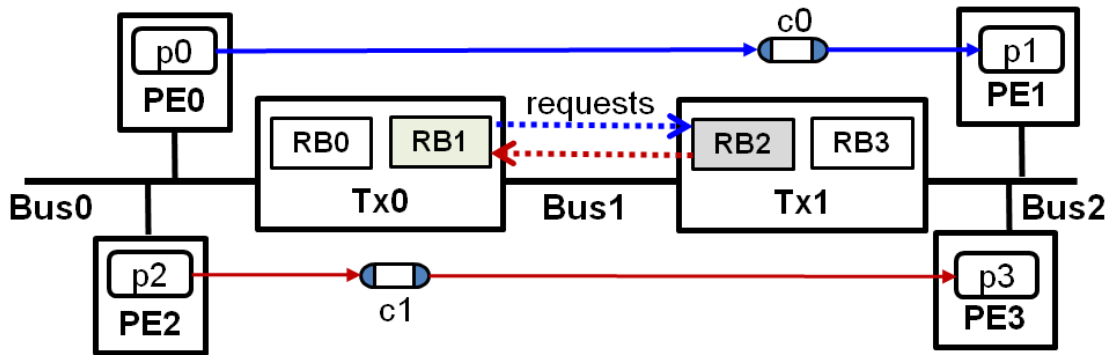


Figure 3.18: SW Architecture of Tx

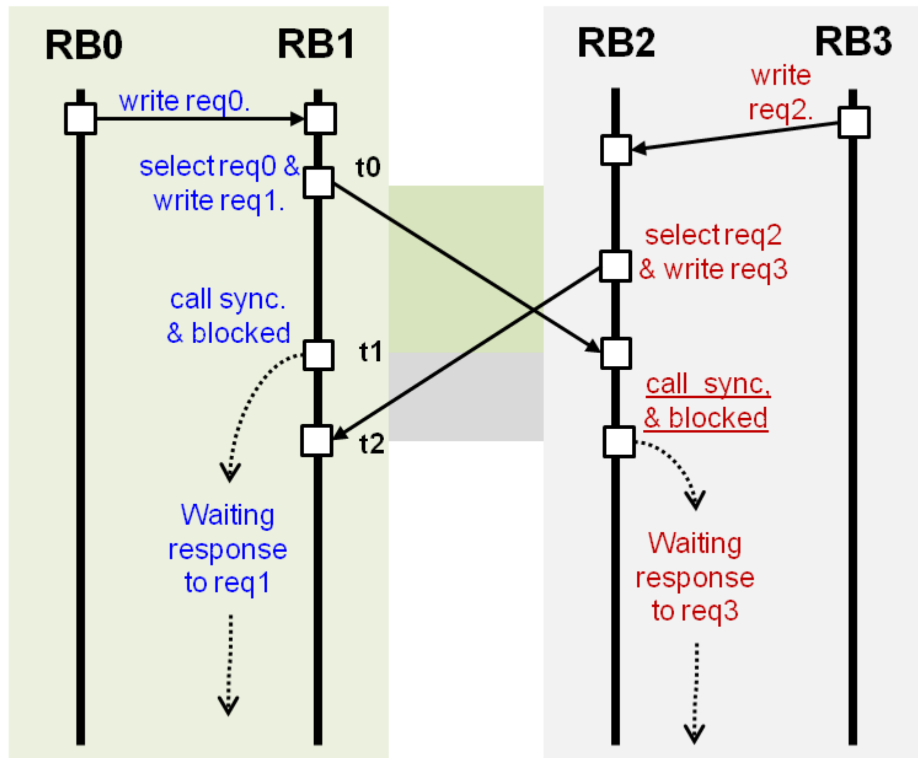
by all channels. If the FIFO memory is partitioned, the FIFO memory also includes partition identification mechanism. The internal structure of the FIFO memory need not be visible to the connected request buffers. Instead, the FIFO are accessed via channel interfaces such as checking FIFO status(MayIRead, MayIWrite, etc) or pushing/popping the data.

A request buffer hosts an active thread in our Tx model. A request buffer always checks if any request is written on it. If there are one or more requests to serve, the scheduler in the request buffer selects one at a time and serve it. Serving it means doing synchronization with the entity having made the request, making the I/O module do the actual transfer and pushing/popping the data to/from the FIFO memory. Each request buffer is given one I/O module directly connected to the bus and does the actual transfer according to the request buffer's control. For example,

request buffer0 may choose req0, which is from p1 on bus0. It synchronizes with p1 and instructs the I/O module to get the data from p1. The I/O module moves the data from p1 to the FIFO channel. Likewise, the request buffer2 may select req3, which is written by p3 and tells the request buffer to send the data to p3. According to the request, request buffer2 checks if the FIFO is available. Following that, the request buffer instructs the I/O module to pop the data and send it to p3.



(a) Old Tx Model with 1 Request Buffer per Each Connected Bus



(b)

Figure 3.19: Old Tx Model with Deadlock

Note that the number of request buffers are two per connected bus. In the previous versions of ESE, the number was one as depicted in Figure 3.19a. However, that leads a deadlock in some cases.

Figure 3.19a shows an example of the deadlock. p0 is sending data to p1. p2 is sending data to p3. A channel needs a memory location. In the example, Tx0 provides c1 with the memory. Tx1 gives the data buffer to c0. For c0, p0 writes request to RB0. For RB1 to send RB2, something should write a request on RB1 and there is nothing but RB0 to do that. Once RB1 picks up the request from RB0, RB1 is trying to write a request on RB2. Likewise, regardless of the direction of the data flow, for c1, p3 writes the request on RB3, RB3 on RB2 and RB2 on RB1. RB1 does not need to do that because the data buffer is between RB0 and RB1, which means the request from the other side of the channel c1, the process p2, will be propagated to RB0. The dotted lines between Tx0 and Tx1 represents who wrote a request and to whom the request is written. It is saying that RB1 writes a request on RB2 for c0 and RB2 writes another request on RB1 for c1.

Note that the direction of the data flow does not matter. What is important is who must write a request to whom. Depending on that, Figure 3.19a may or may not cause deadlock. The reason why is the most complicate part in our TLM and is explained in the following paragraphs.

Figure 3.19b shows a deadlock scenario. Assume that both RB1 and RB2 are masters on bus1. RB1 is written a request, req0 by RB0. Since RB0 needs to write a request, req1 on RB2, at time t0, RB1 takes bus0 and write it on RB2. RB1 successfully writes the request and then, at t1, releases the bus. At t1, RB1 does not have request3, yet. Therefore, it interrupts RB2. Interrupting RB2 means setting a flag in RB2, does nothing but waiting until something happens on bus1.

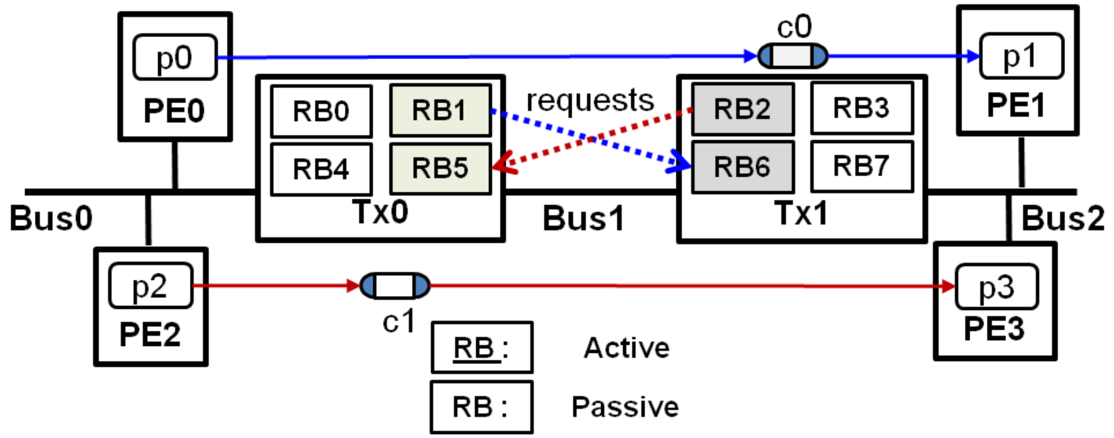
On the contrary, RB2 selects the request from RB3, req2, later than RB1 chose req0. Therefore, when RB2 is not granted for the bus by time t1. At t1, RB2 returns from the UBC write function. The previous implementation immediately called synchronization function to interrupt RB1. Then, RB2 goes a state to wait until something happens on bus1 due to RB1. RB1 will never reply.

The Figure 3.19 is the simplest case ever. It even seems that the problem is solved if RB2 checks the request buffer again before calling synchronization function. However, even that simple solution implies many change in implementation of synchronization mechanism. Moreover, the case can be even worse since more than two transducers as well as multiple processes can be connected to a single bus. Also, the fact that SystemC does not have pre-emption in its nature makes the problem harder. Even RTOS models for pre-emptive operating systems are often implemented with explicit CPU yield in wait statements at the end of every basic blocks [25] [57].

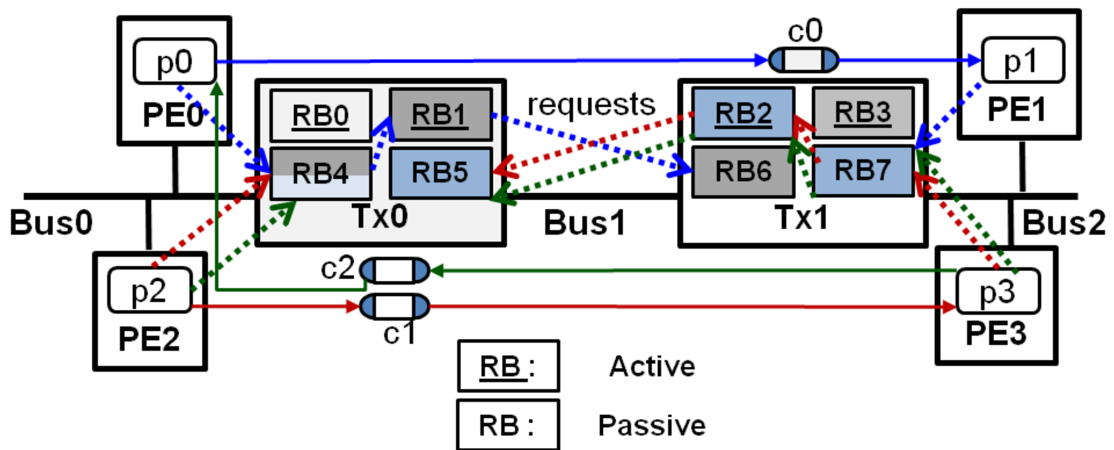
Our solution is presented in Figure 3.20. The solution is based on the following observation. Cyclic dependencies in writing requests and waiting for responses to the requests may bring about the deadlock. Therefore, disconnecting the cycle is a solution.

Our solution is neither the only way nor the best way. However, it is one of the simplest, working solution. We separate a request buffer into two, the passive request buffer and the active request buffer as depicted in Figure 3.20a. In the Figure, the active request buffers have names underline. An active request buffer takes requests that require the request buffer to remotely write a request to serve the given request. A passive request buffer takes all the other requests.

For example, the blue dotted lines in Figure 3.20b shows the direction of requests for channel c0. The memory mapped to the channel is in Tx1. Therefore, p0 must write



(a) Duplication of Request Buffers



(b) Current Tx Model with 2 Request Buffers per Each Connected Bus

Figure 3.20: Number of Request Buffers in A Tx

a request on RB4 or RB0. Any of them will not write a request over bus0. Therefore, the request should go to the passive request buffer on the bus0 side of Tx0. That is RB4. RB4 should write a request either on RB1 or on RB5. Note that this request requires the request buffer to write a request over bus1. If not, the data cannot be moved from Tx0 to Tx1. Thus, the request from RB4 for c0 should be written on the active request buffer on the bus1 side of Tx0, which is RB1. Likewise, RB1 writes a request on either RB2 or RB6 due to c0. Since any of RB2 or RB6 does not need to activate anything on bus1, the request from RB1 will go to RB6. Now, for p1 to receive the data via c0 from Tx1, p1 must write a request on Tx1. It is done either on RB3 or RB7, which are on the bus2 side. Since any of RB3 or RB7 does not activate

any on bus2 due to c0, the request from p1 should be written on RB7, the passive request buffer of Tx1 which is on the bus1 side.

Likewise, the red, dotted lines shows the flow of activation, which is writing a request and waiting for the response. P3 writes its request for c1 on RB7 not on RB3 since RB7 does not activate anything on bus2. Tx1 activates Tx0 for c1. Therefore, the request from RB7 should be written on RB2, the active request buffer. The green lines can be interpreted in the same way.

The reason why this solution is working is explained as follows. In most cases, an active request buffer writes a request on the passive request buffer. If that's the case, since the passive request buffer does not depend on any remote request buffer, there is no cycle unless the route itself is a multi-hop route with at least one cycle. We can prevent this since that is redundant.

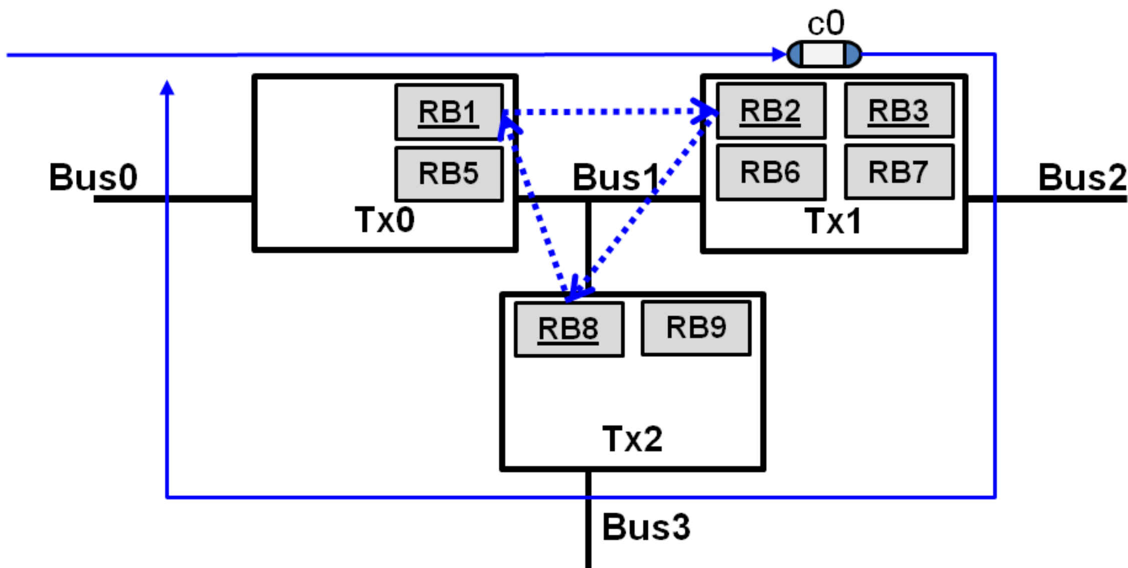


Figure 3.21: Active to Active Buffer : Redundant Cycle in A Route

If an active request buffer writes a request on another active request buffer not on a passive request buffer, a cycle can be found in the dependencies in writing requests and

waiting for the responses. However, that is not a reasonable scenario. In Figure 3.21, if Tx1 were not supposed to send the data through c0 directly to Tx2 over bus1, RB1 would have written a request on RB6 not on RB2. The thing is that RB2 is writing a request on RB8 of Tx2. That means RB2 will send the data from RB1 directly to RB8. Likewise, the Figure implies RB8 sends the data to RB1, again. That is not reasonable.

Therefore, unless any cycle is prevented from being included in a multi-hop route, our solution is working. Moreover, the constraint is very reasonable. Implementing the prevention is easy. We can detect a cycle in multi-hop routes before TLM/PCAM generation and cut the redundant cycle off.

A routing table in Figure 3.18 keeps all the information that the request buffers need at runtime. As we can see in its name, a routing table, the routing information is also kept in the routing table so that the request buffers can read the table.

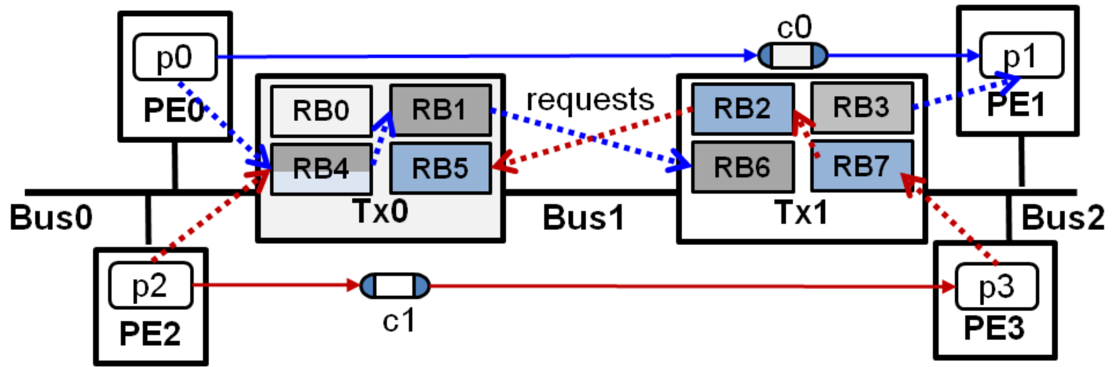
The problem is what routing means inside a Tx. Actually, the route mapped to c0 in Figure 3.20a is p0 - bus0 - Tx0(RB4 - RB1) - Bus1 - Tx1(RB6 - RB7) - p1. The reason why is already explained; to avoid deadlock. Routing in the prospective of Tx0 means the followings.

A. RB4 takes the data from p0.

B. RB4 pushes the data and triggers RB1 to get the data from the FIFO.

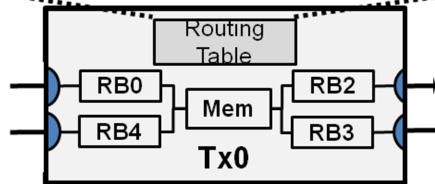
C. RB1 sends the data to RB6, neither RB2 nor any entity on bus1.

In Figure 3.22, an example is present to show how a routing table provides the Tx with the required routing information. In the example, the routing table of Tx0 is depicted.



(a) Example in Figure 3.20a

Src. Proc.	Dst. Proc.	Chan. ID	From	Via (src)	Via (dst)	To	Src. Act.	Dst. Act.
p0	p1	c0	p0	RB4	RB1	RB6	Local	Remote
p3	p2	c1	RB2	RB5	RB4	p2	N/A	N/A



(b) Routing Table of Tx0

Figure 3.22: Tx Routing Table

Since two channels, c0 and c1 pass Tx0, the routing table has two entries, in other words, two rows. The first three columns are the final source, destination processes and the channel ID. The header of the data is compared to these three columns to be identified. The 'From' column is the sender entity on the source side bus. For example, in the viewpoint of Tx0, for c0, bus0 is on the source side and p0 is the sender.

The 'From' and 'To' in Figure 3.22b let the request buffers to send/receive data over the bus to/from the right communicating partner. For example, RB1 must send data to anything else but RB6 for c0. The two columns help that. The 'From' of the first row in the table is p0. 'To' is the receiving entity on the destination side bus, bus1.

RB6 is taking the data from bus1 side so that 'To' is RB6. By looking up 'To', RB1 can send data to the right partner, RB6 and RB4 can wait for the right partner, p0.

The rest four columns, two 'Via's and two 'Activation's are needed, for example, for RB4 triggers RB1 to sned data from RB6 to implement transaction via c0. Generally speaking, since a request buffer can begin a data transaction only when a request is written on it, there are some cases that a request buffer must write a request on another request buffer either in the same Tx or in a different Tx over the bus.

In Figure 3.22a, RB1 should pop the data from the data and send to RB6. However, note that nothing but RB4 can do that. In this case, RB4 must know whether it must locally activate another request buffer in Tx0 or not or whether it must remotely activate another request buffer not in Tx0 but on bus0. In addition to that, if there is any 'yes', RB4 should also know which request buffer it must activate.

The required information is kept in the four columns in the routing table, 'Via(src)', 'Via(dst)', 'Source Activation' and 'Destination Activation'. 'Via(src)' is the source side request buffer of the Tx for the channel while 'Via(dst)' is the destination side request buffer. Therefore, depending on those two, RB4 may know that it is RB1 that RB4 must activate if activation is needed. Also, RB1 knows that, to implement c0, it should remotely activate nothing else but RB6 over bus1 as far as activation itself is necessary. 'Source Activation' and 'Destination Activation' is for the source side request buffer and the destination request buffer, respectively. For c0, the former is for RB4 while the latter is applied to RB1. For c1, the former is applied to RB5 and the latter to RB4. Each of the columns defines the following two.

- Whether the applied request buffer must locally activate the next request buffer inside the same Tx
- Whether the applied request buffer must remotely activate the next request

buffer over the bus

Based on the six columns, 'To', 'From', two 'Via's and two 'Activation's, every request buffer may figure out whether it has to activate another request buffers or, if it has to, which request buffers it should activate.

Like UBC, Tx models can have a different wait statement in each primitive operations. For now, our Tx models have the amount of time to wait hard-coded. In the future, it can be improved by having the primitive operations to refer an overhead table.

3.4 Transaction Level Model Generation

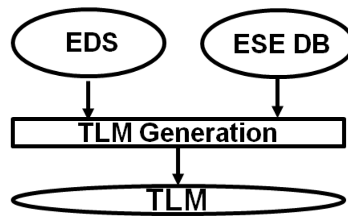


Figure 3.23: Two Types of Information for TLM Generation

As depicted in Figure 3.23, the generated TLM has two types of information; one is from EDS and the other from ESE DB. Information from EDS is dynamic information and those from ESE static.

Static information is invariant from design to design. The information mainly consists of code templates and timing databases. Timing databases includes processing unit models used for timing annotation of user processes [26], bus protocol models to estimate UBC primitive operations [29] and RTOS overhead tables [25]. Code templates are the common part of each TLM components.

Dynamic information includes the followings. Platform structure, configuration and

mapping. Platform structure is selection of platform components and their connectivity. Examples of configuration are the frequency of a selected PE, cache sizes, caching policies and so on. Mapping has been discussed in this report. It includes PE-process mapping, channel-route mapping, channel-memory location mapping, bus address space, Tx routing information and FIFO configuration and so on.

As a prerequisite, our style in instantiation of TLM components needs to be explained. That is probably helpful to understand how dynamic information and static information are added to each TLM components as well as the entire TLM.

```
1  // user functions p1, p2
2  extern void p1();
3  extern void p2();
4
5  // two HW, HW1 and HW2 running p1 and p2 respectively
6  class HW1 : public SC_MODULE {
7        // ...
8        SC_THREAD( p1 );
9        // ...
10 };
11
12 // note that HW2 type must be different from HW1
13 class HW2 : public SC_MODULE {
14       SC_THREAD( p2 );
15       // ...
16 };
17
18 class TopModule : public SC_MODULE {
19
20       TopModule()
21       {
22         // binding HW, CPU, Tx with UBC
```

```
23            // ...
24        }
25
26        HW1 hw1;
27        HW2 hw2;
28        // ...
29 };
```

Listing 3.1: Static Instantiation in The Previous Versions of ESE

Listing 3.1 shows a typical example of static instantiation used by the previous versions of ESE. In the example, two HW, HW1 and HW2 run different processes p1 and p2, respectively.

The key observations are the following two. First, as we can see at line 8 and line 14, every information is statically given. In other words, the information is known to TLM before execution of the TLM. HW1 already knows it should run p1 even before execution of the TLM. Second, HW1 and HW2 in our system specification must have different types(classes) in TLM. Each type must include a line such as line 8 and line 14. Those lines, however, cannot be determined until we have system specification including HW1, HW2, p1, p2 and their mapping. Therefore, it is hard to define a single HW class shared by all HW instances in advance. In addition to that, for example, if HW is completely instantiated at generation time, it cannot be scalable. That is, even in case that the two HW with exactly same functions running exactly same process, two different types(classes) should be introduced if they are different in the number of bus ports.

Using two different classes for two instances makes TLM generation more complicated and less maintainable. When the characteristics of C/C++ are taken into consideration, we can infer that not only the objects themselves but also the codes that use

the objects often require different codes. For example, assume that HW1 and HW2 have a run function. Also, assume that we want to write a C function to call the run function. Then, since the types of HW1 and HW2 are different, we must write two slightly different functions.

```
1  // include modules
2  #include <HW.h>
3
4  // user functions p1, p2
5  extern void p1();
6  extern void p2();
7
8  class TopModule : public SC_MODULE {
9
10         TopModule()
11         {
12                 // two HW, HW1 and HW2 running p1 and p2 respectively
13                 // Note that HW class is shared by HW1 and HW2
14                 p_HW1 = new HW( p1 );
15                 p_HW2 = new HW( p2 );
16
17                 // binding HW, CPU, Tx with UBC
18                 // ...
19
20         }
21
22         HW *p_HW1, *p_HW2;
23         // ...
24 };
```

Listing 3.2: Dynamic Instantiation in The Current Versions of ESE

Listing 3.2 shows an alternative approach, which the current version of ESE adopts.

We keep the common parts between HW1 and HW2 in a single class, HW. The parts that are not common to HW1 and HW2 are added during instantiation phase at runtime as well as generation time. The runtime instantiation is done inside the top module constructor.

First of all, line 6 - 16 and line 26 - 27 in Listing 3.1 have been changed to the code at line 2, 14 and 15. In Listing 3.2, HW.h included at line 2 defines a single, common HW class. The class has a pointer to process, which is void (*)() type. Instead of hard-coding which HW runs which process, line 14 and 15 inside the constructor of the top module, the single HW class is instantiated twice for HW1 and HW2 objects, respectively. Note that there is neither HW1 nor HW2 in TLM before the line 14 and 15 are executed. Also, note that the HW class can be instantiated with any process. Those two are the main difference between the previous static instantiation and the current dynamic instantiation.

In addition to that, dynamic instantiation also allows scalability. For example, an RTOS model may run a couple of processes and the number may vary. If we should statically generate the RTOS model, as we reviewed in Listing 3.1, a single class cannot be shared by two RTOS models different only in the number of the running processes. Also, if two PEs with exactly same functionality and running exactly the same set of processes are different in the number of bus ports, the two PEs should be most likely in different types(classes). On the contrary, dynamic instantiation allows such scalability.

Since two different objects, for example, two PEs, share the same class, the code using the objects does not need to be different. The code being common means that the code can be written in advance and stored in ESE DB. That makes TLM generation much more simpler and far more maintainable.

Why the current implementation is adopting dynamic instantiation rather than static one has been explained. Since instantiation can be done dynamically, what TLM generation does is mainly the following; It produces the top module code that instantiates the TLM components in system specification and connects them together. Simulation starts only by the function, sc_start, being executed. Before sc_start is called, instantiation of TLM components are completely done inside the constructor of the top module. Note that what we call 'TLM Generation' includes not only production of the generated TLM code but also execution of the instantiation parts of the TLM code.

In this Section, we will see how TLM generation is done based on both of static information and dynamic information component by component.

3.4.1 Transaction Level Model Generation : PE

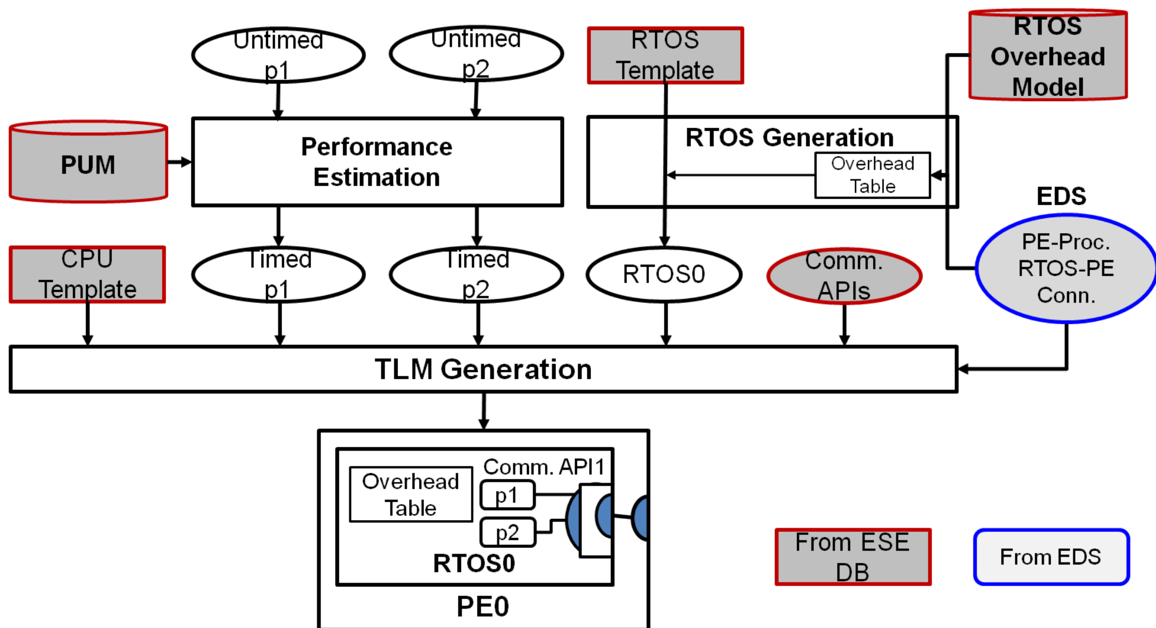


Figure 3.24: PE Generation

Figure 3.24 shows CPU generation. PE can be either a CPU or a HW type. We

starts with CPU generation. Note that all red-lined shapes with gray color come from ESE DB while blue-lined shapes in light gray come from EDS. Figure 3.24 depicts generation of PE0, to which two processes, p1 and p2 are mapped.

Figure 3.24 shows a big picture. The original, untimed user processes such as p1 and p2 are annotated with timing by performance estimation. That requires PUM in ESE DB. At the same time, RTOS overhead table is made and the pointer is added to an instance of RTOS template stored in ESE DB. The common codes among all RTOS instances are packed in the template. The template is actually a single SystemC class. Communication API will be explained later on. However, note that the required and right set of functions in communication API needs to be accessible by the processes. RTOS0 manages the resources of PE0 so that communication API is added to RTOS. The selection of communication API and the processes to be executed on RTOS0 should be done based on EDS. TLM generation process choose the right processes and communication API for RTOS0 and add them to RTOS0 during dynamic instantiation. Also, the CPU template, which is a c++ class, is instantiated with the RTOS model, RTOS0.

Note that TLM generation for a custom HW component is simpler. A HW template, which is a SystemC SC_MODULE, is instantiated with the process and the right communication API during runtime instantiation. Communication API selection will be explained later on in this section.

Figure 3.25 shows RTOS generation in great details. Above all, it shows what the RTOS template is. It is a SystemC(C++) class where all functionality of RTOS primitive operations are already implemented. It has an empty overhead table and an empty process list. RTOS primitive operations such as scheduling or services of communication API refer the overhead table and the process list. They will be filled out during RTOS generation.

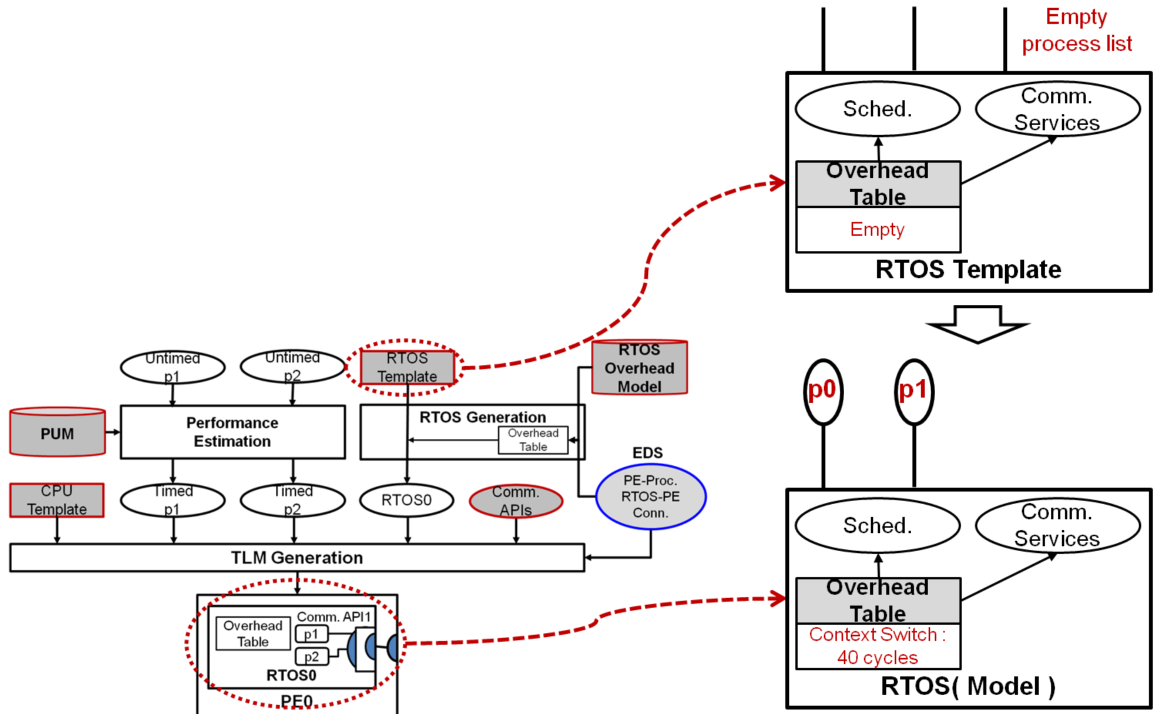


Figure 3.25: RTOS Model Generation

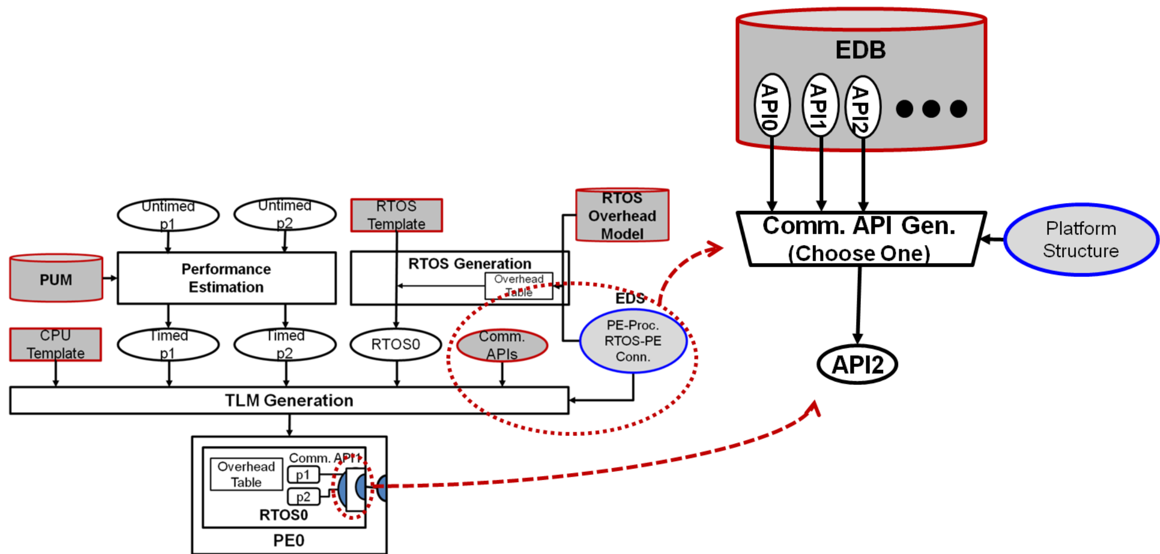


Figure 3.26: Communication API Selection

Figure 3.13 showed the same channel interface must be implemented in different ways depending on the platform and the locations of the communicating processes. The number of cases are limited and usually less than ten for a given function such as

send. In addition to that, unlike the previous versions of ESE, the current ESE has a single type for PEs, a different but single type for UBC instances and so on.

Therefore, the entire set of communication API can be implemented and stored in ESE DB. As depicted in Figure 3.26, depending on PE-Process mapping, platform connectivity and so on, all of which are given in EDS, the TLM generation process selects the right communication API. The selected communication API is added to the RTOS model if the processes using the API are mapped to a CPU. Otherwise, the communication API is directly added to the PE, which is a HW type. In the Figure, communication API2 is selected depending on EDS and added to RTOS0.

Note that, even though we adopted static instantiation as before, the bodies of communication API should be written at least once inside the TLM generation code. Also, TLM generation code must include the entire set of communication API somewhere, anyway. Therefore, implementing the entire set of API and storing them in ESE DB do not require any extra work at all.

3.4.2 Transaction Level Model Generation : UBC

UBC models a bus by defining primitive operations and providing estimation on the operations.

All UBC instances share a large portion of common code. That is stored in ESE DB and called UBC template. UBC template includes the bodies of UBC primitive functions such as send, receive, read, write and so on. Those functions need data, which varies from instance to instance. Examples of the data are bus address space, synchronization table, global memory, arbitration policy and so on. Except the communication delays, EDS has all the necessary data for bus0. The communication

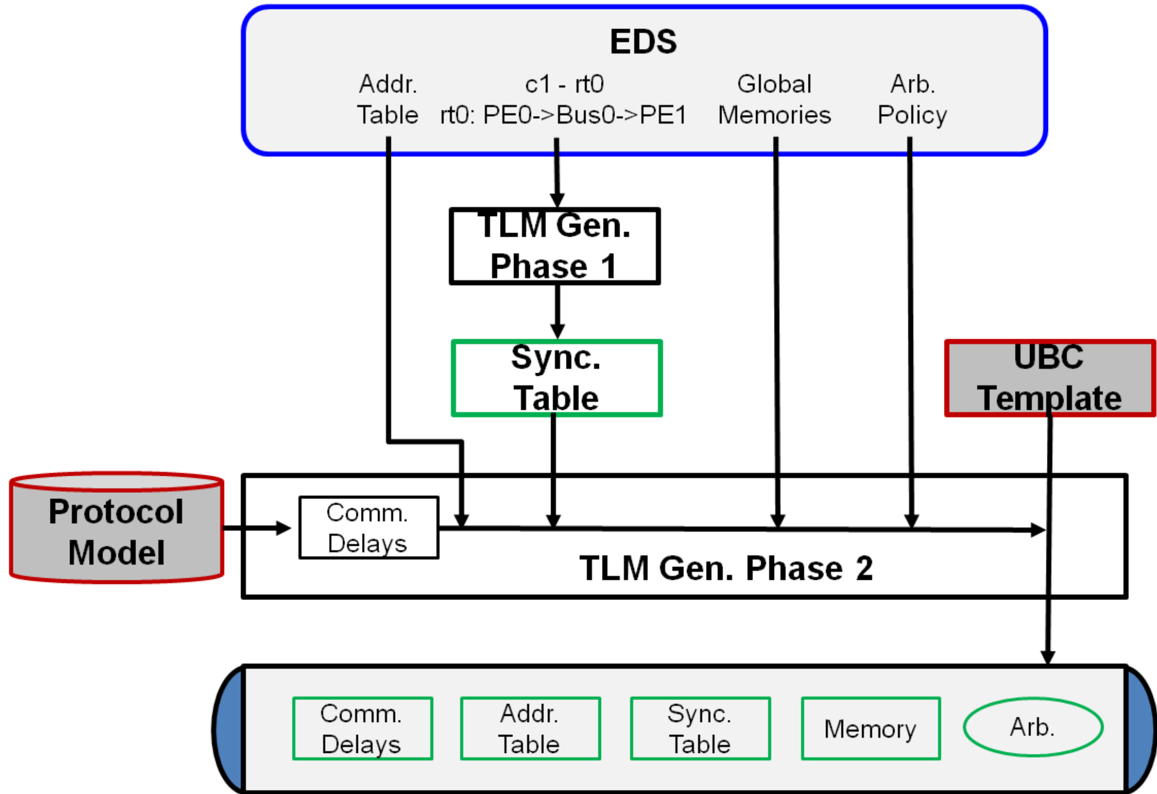


Figure 3.27: UBC Generation

delays are different from instance to instance. To estimate the delays without PCAM, we need protocol models [9] [29] in ESE DB.

In Figure 3.28, UBC template is drawn compared to UBC. That is a SystemC object, where all UBC functions are already implemented. The functions will refer synchronization table, address table, communication delay table, the list of memories and so on. Those are not common to the UBC instances. The UBC template initially has empty list and tables. A UBC model is a template with the tables and list filled out.

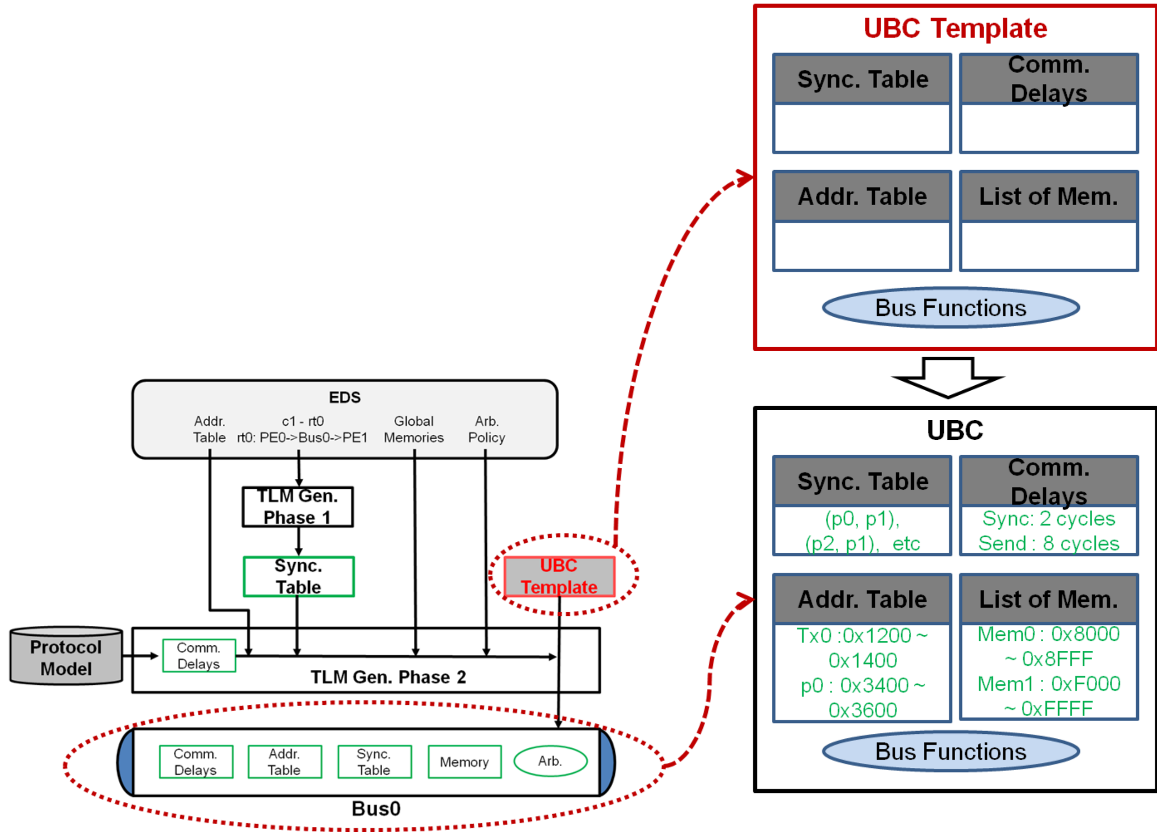


Figure 3.28: UBC Template

3.4.3 Transaction Level Model Generation : Tx

Tx generation is similar with UBC generation. Tx primitive operations are common to all Tx instances. Therefore, they can be implemented and stored in ESE DB as Tx template. As depicted in Figure 3.29, generation of Tx0 begins with retrieving Tx template from ESE DB. However, configuration for the internal FIFO, number of request buffers, the number of request buffer entries in each request buffer and routing table may vary. The information is already included in EDS. For example, since Tx0 is connected to two buses, four request buffers are needed. The connectivity is given in EDS. TLM generation produces four instances of request buffers and add them to Tx template.

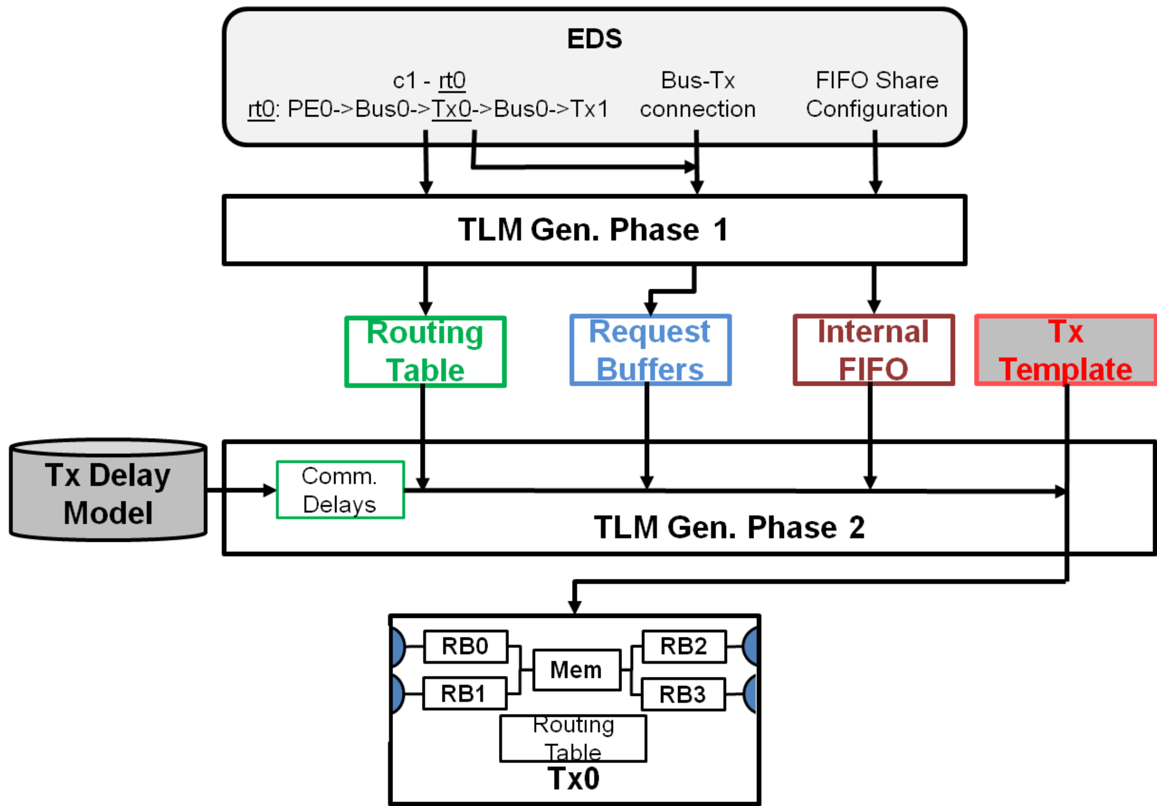


Figure 3.29: Tx Generation

Note that communication delays cannot be given in EDS. They will be computed based on Tx delay models stored in ESE DB. As a result, a small data structure containing the estimated delays for the given Tx instance, Tx0, is added to Tx template.

Chapter 4

Hierarchy-Aware Mapping of Pipelined Applications

© 2014 IEEE. Reprinted with permission from Kyoungwon Kim and Daniel Gajski, Hierarchy-Aware Mapping of A Pipelined Application, 2014

This dissertation offers mapping techniques for general computation models. As pipelined execution of the given application on a heterogeneous platform is a practical solution for several application domains such as streaming applications [58], mapping of a pipelined applications to the given platform is a problem worth solving. A general computation model may have explicit representation of pipelined execution of the application. Mapping techniques for such a model must exploit pipeline parallelism.

This chapter describes mapping of a pipelined application captured while optimizing execution time. The applications we target are executed in a pipelined manner and can be captured in a general computation model with explicit representations for pipelined execution. Such a model can be decomposed into one or more *pipeline stages*

or, interchangeably, *stages*. Each stage consists of a set of sequential and/or parallel tasks, each of which may also be decomposed, recursively, into sequential/parallel tasks. Balancing the delays of the pipeline stages, the aim of which is to make the execution time of each stage approximately the same, significantly impacts the execution time of the system. Ideally, the execution time of the system is close to that of the stage with the critical delay.

The contribution of this dissertation is, by being aware of hierarchy, to balance pipeline stages. Previous works [39, 49, 42, 40, 41] have assumed that hierarchy in the computation model is flattened so each period of execution can become an acyclic directed graph. However, in general models, a stage can have a complex hierarchy that does not easily allow such flattening. For example, in Program State Machines (PSMs) [9], a stage can be decomposed into ten program-states, over which are defined very complex state transitions depending on the data. Hierarchy-Aware mapping we present saves additional PEs by merging consecutive small stages, re-assigning the saved PEs to the large and hierarchically described stages, repartitioning the large stages and carrying out a remapping of the stages.

4.1 Application Model and Platform

The entire computation model is a pipeline, which can be decomposed into one or more *pipeline stages*, or *stages*. Any two tasks in different stages run concurrently. Any two tasks in adjacent stages may communicate through FIFO channels. Each stage is a single *program-state*. As Stage1 in Figure 4.1 exemplifies, a program-state can be either decomposed into sequential program-states, decomposed into parallel program-states or a leaf program-state we call a *process*. A process contains codes written in high level languages such as C/C++. Sequential decomposition implies

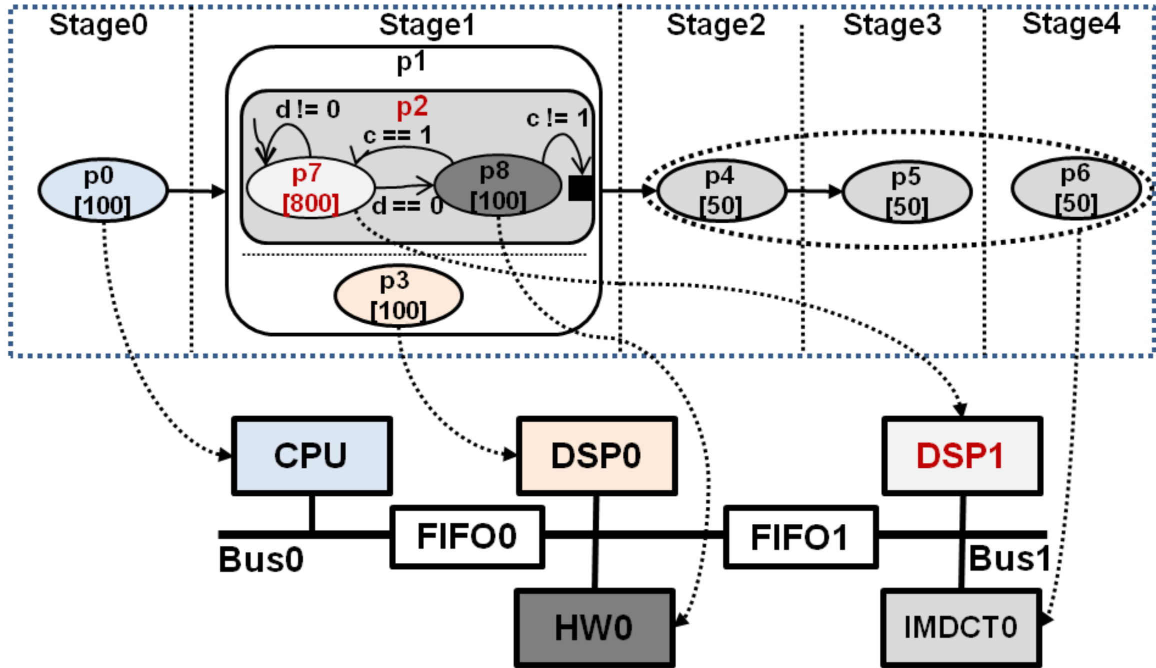


Figure 4.1: Application Model, Platform, and Mapping

there is a finite set of hierarchical sub program-states in the program-state; a set of state transitions is defined as well. A state transition may be affected by data. Parallel program states in a single stage run in parallel with their sibling program-states. Any two states may communicate through channels. For example, in Figure 4.1, p0, p1, p4, p5 and p6 are executed concurrently in a pipelined manner. p1 is decomposed into two parallel program states: p2 and p3 running in parallel with each other. p2 is sequentially decomposed into two processes: p7 and p8. However, note that, according to the data **c** and **d**, p7 and p8 can be executed any arbitrary times even in a single cycle of the pipelined execution. Figure 4.2 shows a practical example of modeling a pipelined application with a computation model with explicit representation for pipelined executions.

A practical example of the applicatio model in Figure 4.1 is shown in Figure 4.2. The code is a SpecC [30] pseudo code. SpecC is a system-level design language

```
1: behavior Application() {
2:    void main() {
3:     /* nIter is the number of iterations */
4:        pipe ( i = 0; i ¡ nIter; i++ ) {
5:          p0(); // Stage
6:          p1();
7:          p4(); // Stage2
8:          p5(); // Stage3
9:          p6(); // Stage4
10:       }
11:   }
12: }
13: behavior p1() {
14:   void main() {
15:       par { p2(); p3(); }
16:   }
17: }
18: behavior p2() {
19:   void main() {
20:       fsm { p7(); p8(); }
21:   }
22: }
```

Figure 4.2: Modeling Example: SpecC

developed by Center for Embedded Computer Systems at University of California, Irvine. SpecC provides the **pipe** keywords for pipelined execution, which can be found in line 4. From line 4 through line 10 define pipelined execution of five hierarchical program-states: p0, p1, p4, p5, and p6. Any program-state executed in a pipeline stage can be hierarchical. P1 is decomposed into two parallel program-states, p2 and p3. P2 has a sequential decomposition of p7 and p8 from line 18 through line 21.

The platforms we target are combinations of general purpose processors (GPPs), DSP, FPGA and other processing elements.

4.2 Problem Definition

The optimization goal is to minimize the execution time of the system. It is assumed that the execution time of the system depends on the execution time of the stage which has the longest delay. To minimize the execution time, small consecutive stages can be merged and each large stage can be partitioned into pieces. Each piece can be mapped separately. The application model is mapped to the given, fixed platform while the execution time is minimized under cost constraints.

This dissertation offers formalization for the problem. A partition is defined as either a subset of program-states in a single stage or the stage itself. V, S and $P(s_i)$ are the set of Processing Elements (PEs), stages and partitions in a stage s_i, respectively.

For a partition p_j, $T(p_j)$ is execution time of p_j. $T(s_i)$ is the execution time of stage s_i and defined as follows:

$$T(s_i) = max\{T(p_j)\}\ for\, p_j\ \in\ P(s_i) \tag{4.1}$$

Our goal is to find the set of stage S and mapping $M : V \rightarrow S$, where $max\{T(s_i)\} for s_i \in S$ is minimized. Note that S changes during mapping and that $T(s_i)$ may be reduced by partitioning s_i and assigning more PEs to the stage.

4.3 Hierarchy-Aware Mapping

4.3.1 Algorithms

Heuristic-based approaches can be justified as follows; If even a stage can be divided when it is hierarchically described, the complexity is not polynomial.

The number of PEs $\|V\|$ can be initially less than, equal to or greater than the number of stages $\|S\|$. Figure 4.3 covers the case in which $\|V\| = \|S\|$. The other two are very similar.

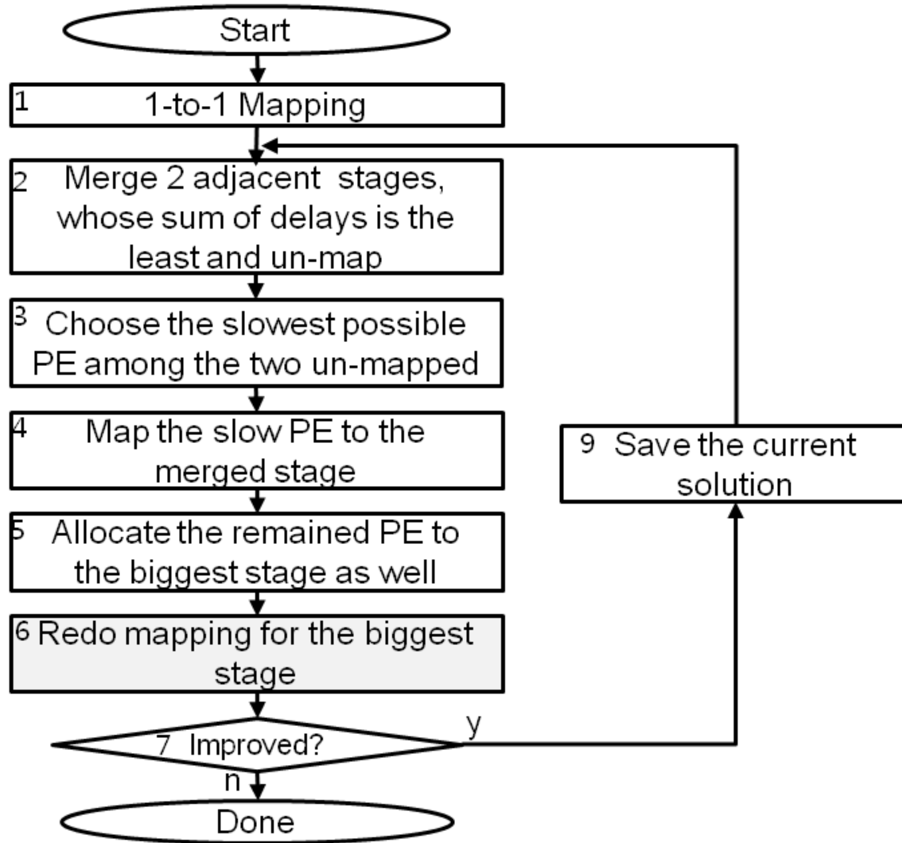
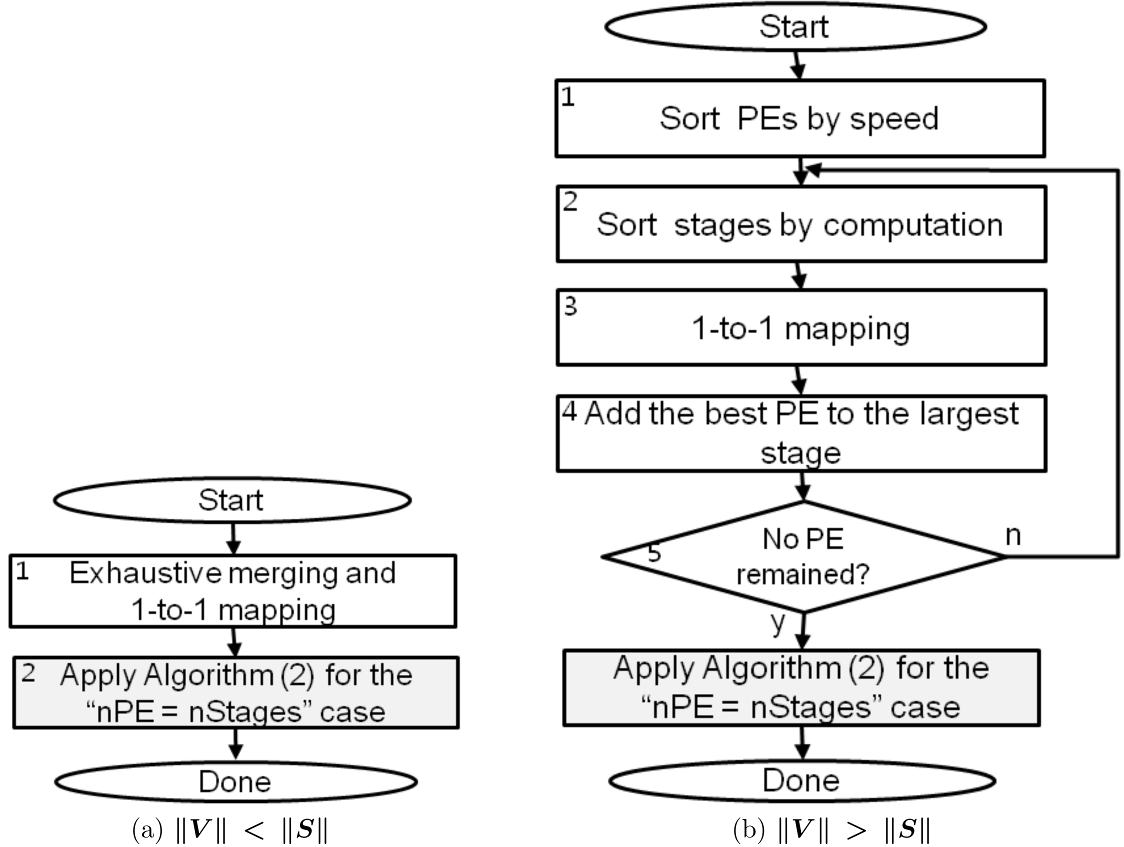


Figure 4.3: Hierarchy-Aware Mapping: $\|\boldsymbol{V}\| = \|\boldsymbol{S}\|$

In Figure 4.3, $\|S\|$ is or has become equal to $\|V\|$. S is sorted by the execution time given in the execution profile. V is sorted by speed. Following that, the one-to-one mapping in Box 1 Figure 4.3 is performed in order. One-to-one mapping has room for improvement since a set of stages are too small while another set of stages are too large. The quality of the design can improve if we save extra PEs by merging more consecutive small stages and assign the saved PEs to the stages with longer delays. In Boxes 1 through 5, Hierarchy-Aware mapping algorithms repeat merging stages, saving a PE, assigning the PE to the stage with the longest delay and testing whether

there is improvement. Once the saved PEs are added to a stage, the stage should be repartitioned and remapped. Heuristics for repartitioning and remapping will be explained in the next section.



(a) $\|\boldsymbol{V}\| < \|\boldsymbol{S}\|$ (b) $\|\boldsymbol{V}\| > \|\boldsymbol{S}\|$

Figure 4.4: Hierarchy-Aware Mapping: $\|\boldsymbol{V}\| \neq \|\boldsymbol{S}\|$

Figure 4.4a describes the algorithm that is applied when $\|V\|$ is less than $\|S\|$. An intuitive way to map starts with merging several small consecutive stages so that $\|S\|$ becomes $\|V\|$. The total number of ways to reduce S is ${}_{\|S\|}C_{\|V\|}$. In addition, for each reduced S, there are $\|V\|!$ ways of mapping. Nonetheless, $\|S\|$, and thus $\|V\|$ in this specific case, are not large in practice, so an exhaustive search could be reasonable.

In Figure 4.4b, the algorithm for the case that $\|V\|$ is greater than $\|S\|$ is explained. The basic idea is to reduce the problem to the "$\|V\| = \|S\|$" case. S is sorted by computation. V is sorted by speed. Sorting is followed by injective in-order mapping

from S to V. The same procedures in Figure 4.3 follow except that Box 2 is skipped whenever there is any remaining PE.

4.3.2 Heuristics for Repartitioning and Remapping

When a saved PE is added to a stage, the stage should be repartitioned. Repartitioning is followed by remapping. Even finding the local optimal that minimizes the execution time of the stage takes exponential time. Therefore, heuristics giving suboptimal solutions are required.

In the heuristics, a process and a PE to which the process is moved is selected. The process is moved to the PE until there is no improvement. We formalize the problem. The algorithm takes three inputs: 1) a graph G, which is the subgraph of the given computation model representing the stage under concern 2) the set of the PEs given to this stage s V_s and 3) the execution profile. The output is P_s which is a set of non-overlapping subgraphs of G and one-to-one mapping $f : P_s \to V_s$ where the $ExecTime(G)$ is minimized.

Figure 4.5 is the algorithm to compute ExecTime. If the state is a leaf process, the execution time of the leaf process on the currently mapped PE is given from the execution profile. If the decomposition is sequential, delay of each sub program-state should be accumulated. It is complicated to compute ExecTime function with a program-state decomposed into parallel program-states. If each parallel sub program-state is mapped to different PEs respectively, ExecTime returns the maximum execution time. However, several sub program-states can be mapped to the same PE. Moreover, each sub program-state is hierarchical. For example, sub program-state0 and sub program-state1 can be partially mapped to PE v0. The execution time of program-state0 is affected by PE configurations, bus arbitration, RTOS scheduling,

```
1: double ExecTime(G) {
2: if G is a leaf node then
3:    return the execution time of G on the mapped PE
4: end if
5: if G is a set of sequential sub program-states SubG_seq then
6:    return Σ ExecTime( sub_g ∈ SubG_seq )
7: end if
8: if G is a set of parallel sub program-states SubG_par then
9:    max := −∞
10:   for all v ∈ V_G where V_G is the set of PEs running part of G do
11:      delay := 0
12:      for all sub_g ∈ SubG_par do
13:         if sub_g or its part is mapped to v then
14:            delay+ = ExecTime( sub_g ) × α, 0 < α ≤ 1
15:         end if
16:      end for
17:      max = MAX( max, delay )
18:   end for
19:   return max
20: end if
21: }
```

Figure 4.5: ExecTime(G) Function

RTOS overhead and many others so is not predictable at this design stage. Therefore, approximation is required. As seen from lines 11 through 19, to compute the execution time of the sub graph of G mapped to v, we sum up ExecTime of all sub graphs, each is entirely or partially mapped to v. Before summation, we multiply α. The best value of α in Figure 4.5 could vary. We used 1, which encourages mapping parallel program-states separately.

4.4 Case Study

The case study is performed with two streaming applications: Canny Edge Detector [59] and JPEG encoder. The platforms are randomly generated with different cost constraints. In the platform library, there are MicroBlaze processors, three different

types of custom hardware and DSP models. We capture Canny Edge Detector in PSM following the modeling style of Han et al [60]. We modeled the JPEG encoder on our own. The application is decomposed into four stages. The first two stages are decomposed into several hierarchical sequential/parallel sub program-states. The rest are leaf program-states. The platforms are randomly generated. The number of PEs varies from 2 to 4. The simulation framework in [61] is used to measure the execution times. Note that the framework reflects the impact of resource sharing and communication overheads.

Table 4.1: Canny Edge Detector: Average Execution Time per Frame

Randomly Generated Platforms	Exhaustive Hierarchy-Unaware	Hierarchy-Aware	Execution Time Reduction
1	50.62 ms	38.68 ms	23.59%
2	50.32 ms	38.27 ms	23.95%
3	189.65 ms	110.45 ms	41.76%
4	39.5 ms	32.89 ms	16.73%
5	39.5 ms	23.87 ms	39.57%

Table 4.2: JPEG: Average Execution Time per Frame

Randomly Generated Platforms	Exhaustive Hierarchy-Unaware	Hierarchy-Aware	Execution Time Reduction
1	20.09 ms	14.14 ms	29.62%
2	19.03 ms	13.07 ms	31.32%
3	15.15 ms	14.14 ms	6.67%
4	19.03 ms	13.57 ms	28.69%
5	19.03 ms	17.03 ms	10.51%
6	15.15 ms	14.26 ms	5.87%
7	47.98 ms	40.99 ms	14.57%
8	41.93 ms	32.79 ms	21.80%
9	16.39 ms	16.18 ms	1.28%
10	15.15 ms	14.14 ms	6.67%

Tables 4.1 and 4.2 compare, respectively, Hierarchy-Aware mapping to an exhaustive Hierarchy-Unaware Mapping in the average execution time of Canny Edge Detector and JPEG encoder. The latter is the optimal as long as hierarchy is not taken into

account. Hierarchy-Aware mapping decreases the execution time by the average of 23.3%.

Chapter 5

N-Way Clustering and Mapping

Due to design complexity, in any design methodology, one must address productivity. In traditional design methodologies, the design process is not efficient enough, the main reason being the lack of HW/SW co-design. Virtual Platform (VP)-based design methodology allows HW/SW co-design since VP, which is a model of hardware platform, is used for software development before the prototype is ready. However, every change in the platform must be manually implemented and thus VP-based design is not flexible enough for new embedded applications.

One alternative that researchers have proposed is Transaction Level Model-based design. The design begins not with platforms but with a Model of Computation capturing the system's functionality. Designers map computation models to Model of Architecture (MoA), which is an abstract model of the selected platform. Transaction Level Model (TLM) is automatically generated for cycle-approximate evaluation so that the design quality can be evaluated without the prototype board [9].

In Transaction Level Model-based design, mapping is still intuitively done by experienced designers. Manual mapping is becoming infeasible regarding that the realistic systems are already too complex. A solution to tackle this problem is to automate mapping.

Due to trade-offs between different metrics, no single algorithm can optimize all of metrics at the same time. Designers may choose among different algorithms, each optimizing its own set of the metrics. In this dissertation, execution time is minimized under other design constraints such as cost. In optimization of execution time process scheduling plays a crucial role.

Optimization in mapping also depends on the computation model. Although Transaction Level Model-based design could starts with any computation model, this dissertation focuses on general computation models such as Program State Machines. A general computation model is complete so it can describe the entire system. Moreover, there have been well-established automated design flow for such a general computation model.

In a general computation model, processes run in an asynchronous manner and process scheduling is affected by the dynamic data-oriented behavior limited by the flow of data and data dependencies across computations. Therefore, process scheduling is not highly predictable in a general computation model. Nonetheless, the impact of process scheduling must be considered as well as communication overhead and computation. Thus, approximation for process scheduling are required.

In this chapter, N-Way Clustering and Mapping (NWCM) is proposed. NWCM is a mapping technique for a general computation model, and takes into consideration altogether communication, computation, and process scheduling. NWCM conducts clustering based on its closeness function, to which communication overhead, compu-

tation, and process scheduling are put together. Clustering is followed by one-to-one mapping. The clusters are sorted by computation and PEs by speed. Mapping is performed in order.

The case study performed with Canny Edge Detector and an application running MP3 decoder and JPEG encoder in parallel shows that NWCM outperforms any competitive algorithm by at least 24.4%.

5.1 Problem Definition

We add automatic partitioning and mapping to a Transaction Level Model-based design as depicted in Figure 5.1. In Transaction Level Model-based design, the design process begins with capturing the system's functionality in an computation model. Modeling the system's functionality and specification of the design constraints are followed by platform selection and mapping. We call the result system description. From the system description, a Transaction Level Model (TLM) is automatically generated so that fast cycle-approximate estimation can be conducted by simulating the TLM. If the design constraints are not met, mapping and/or platform selection are performed to improve the design. Or, computation model itself can be modified. Once the design constraints seem to be met according to the TLM-based cycle-approximate estimation, the back-end design procedure will refine the TLM down to the implementation.

First of all, there are trade-offs in selecting an computation model for a given application. For example, there is a trade-off between expressive power and analyzability.

In this paper, we focus on general computation models and use Program State Machine (PSM) [9].

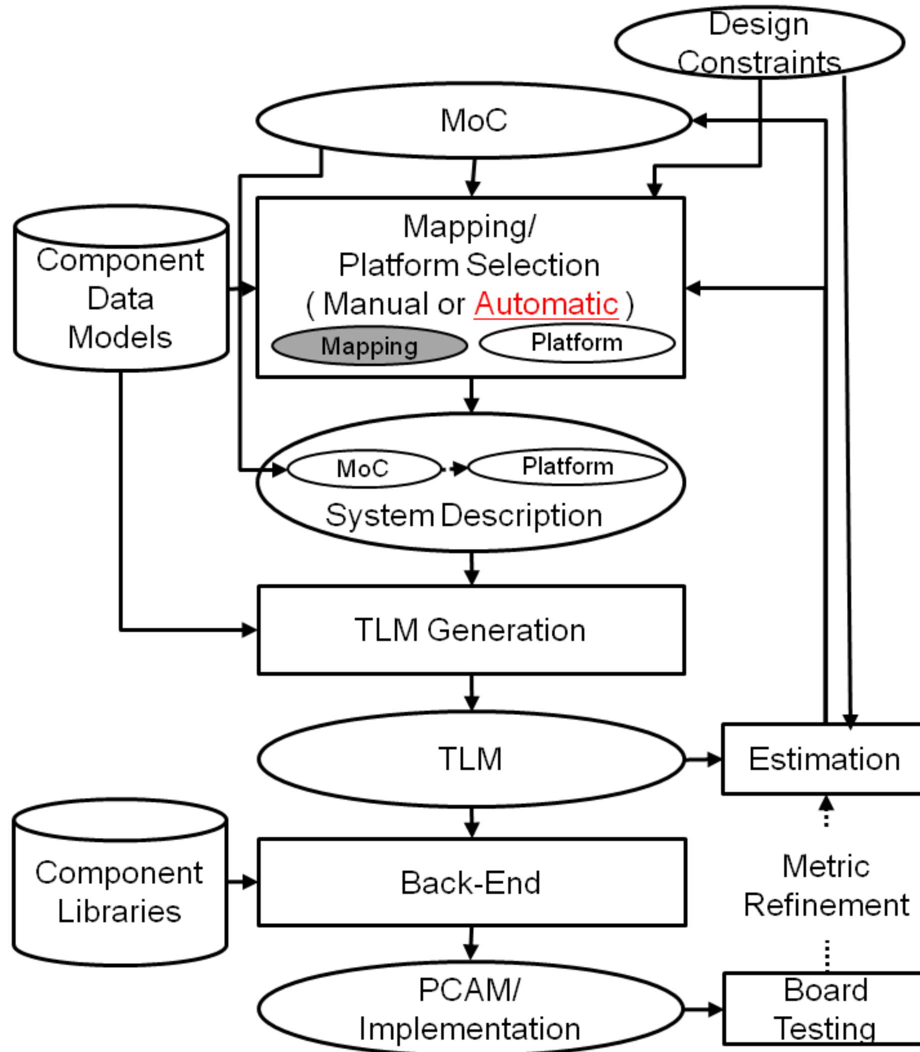


Figure 5.1: Design Flow in Model-Based Design

"The Program-State Machine (PSM) unifies the concepts of hierarchical concurrent finite-state machines, dataflow graphs and imperative programming languages in a single model of computation." (Grütter & Nebel, 2008 [62]).

A program-state can be either of the following. First, it can be further decomposed into concurrent program-states. Second, it can be decomposed into sequential states. Third, it can be a leaf program-state, which we call *a process*. Fourth, it can be a pipeline: several sub program-states are executed in a pipelined manner. We do not cover the last case: the functionality of the entire system is captured in a single

pipeline. Nonetheless, in reality, many computation-intensive applications modeled in such a way that we cover in this paper (e.g. [63] [64]).

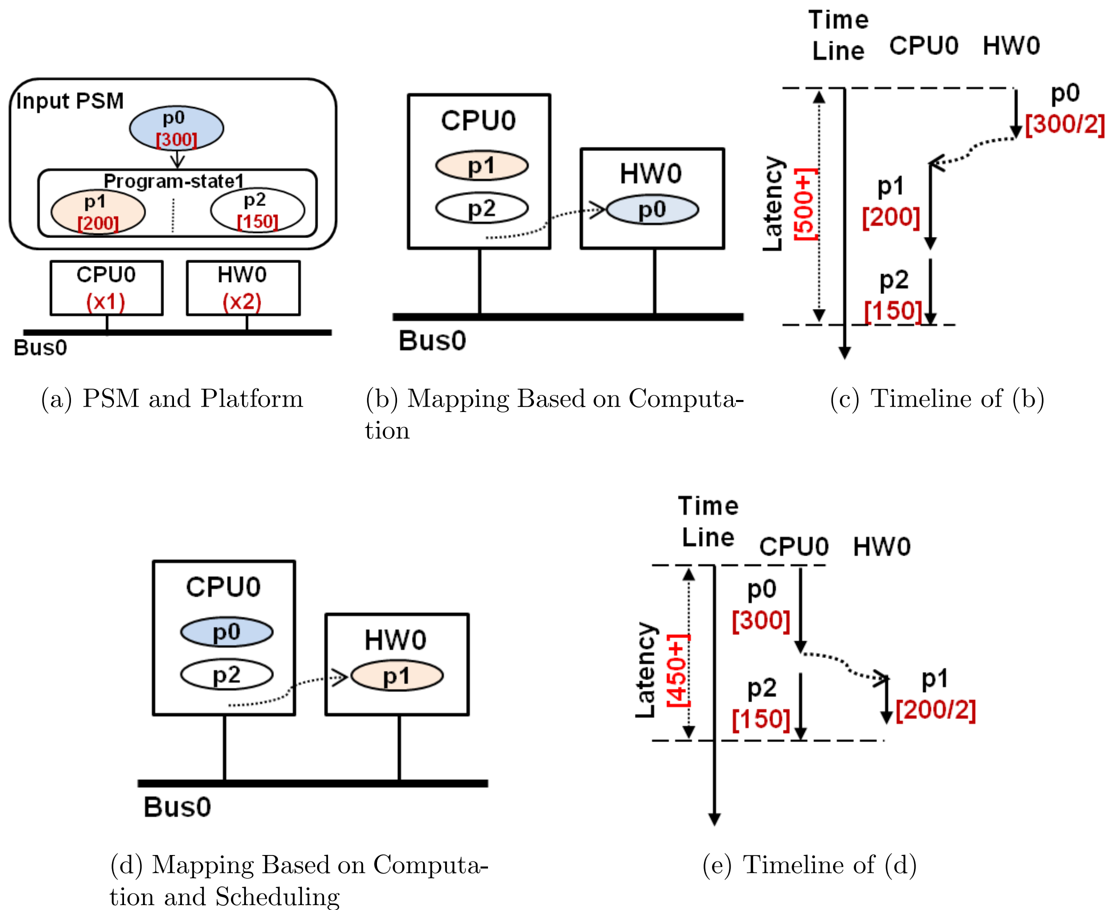
Our target platforms are a combination of processing elements (PEs) such as general processors, DSP, custom hardwares, hardware IPs, etc, as they are in [39]. We assume that a MPSoC platform is provided. In addition, the execution profile, which is the power consumption, cost and execution time of each process on each PE, is given.

Optimal partitioning and process mapping are to be constructed. Our optimization goal is low latency while meeting all the other constraints such as power consumption and cost. We use latency as defined in [65]: latency in transformative systems indicates the average time that a system takes to transform an input from the input stream to the output stream.

The assumptions made for the rest of this paper are as follows: the size of the local memory of each PE is large enough. A single type of RTOS is used for every processor and its scheduling policy is priority-based scheduling. All tasks mapped to a processor have the same priority. Channel mapping is given, once process mapping has been completed. Each process is statically mapped to a single PE. Two or more parallel processes can be mapped to a PE without any RTOS only when static scheduling of the processes is possible.

5.2 Motivation of N-Way Clustering and Mapping

Mapping processes in Transaction Level Model-based designs has been conducted manually. Designers may initially map the entire computation model to a single host processor. Then, the designers may find task-level parallelism to move some processes from the host processor to other PEs. As we see in Figure 5.2, process scheduling may



(a) PSM and Platform

(b) Mapping Based on Computation

(c) Timeline of (b)

(d) Mapping Based on Computation and Scheduling

(e) Timeline of (d)

Figure 5.2: Simple Mapping Example with and without Considering Process Scheduling

greatly impact the latency of the system. In the example, p0 and program-state1 run sequentially, while p1 and p2 run in parallel to each other. HW0 is assumed to execute only a single process in this example and to complete every operation twice as fast as CPU0 does. Without considering scheduling, Figure 5.2b is the optimal. However, in Figure 5.2b, there is no parallelism. The latency is even reduced in Figure 5.2d.

In addition to finding task-level parallelism, designers often consider the processing time of each PE defined by C. Erbas et al [18] and communication overhead. If the communication overhead between two parallel or sequential tasks is very large, designers may map the two processes to the same PE. Besides, designers are always

aware of various design constraints. Especially for real-time constraints, the processing time of each PE must be smaller than the real-time constraints. Finally, when the mapping seems to be infeasible, the designers may change some PEs to better ones and/or use additional PEs under cost constraints.

Based on these observations, we propose NWCM. NWCM performs partitioning by N-Way clustering without violating design constraints, where N is the number of PEs in the platform. For this purpose, we develop the closeness function in which communication overhead, computation, and approximatly estimated impact of process scheduling are taken into account. In addition, we formulate the constraints.

5.3 Closeness Function of NWCM

The problem is that process scheduling in PSM mostly depends on dynamic behavior. There is only limited knowledge on process scheduling available in PSM at static time; any pair of processes are either sequential or parallel to each other. However, two parallel processes may run sequentially due to complicated data/control dependency. State transitions can entirely depend on runtime behavior so that the order of execution of two sequential processes is often limitedly predictable.

Nonetheless, there are still multiple ways to take process scheduling into consideration. One is N-Way clustering based on our new closeness function, to which two boolean variables, b_p and b_s, are added. The boolean variables show, respectively, whether the pair of sets of processes are running in parallel and whether the pair can run sequentially.

In general, to exploit parallelism and to reduce the system's latency, a mapping algorithm may as well map two processes to the same PE if there is a intensive/large

data transfer between them and separate the processes if the sum of the estimated execution delays is large. In addition, there are two more basic observations. A mapping algorithm may as well:

- separate two parallel processes especially if the overall execution delay is large,
- map two sequential processes to the same PE especially if the processes run back-to-back.

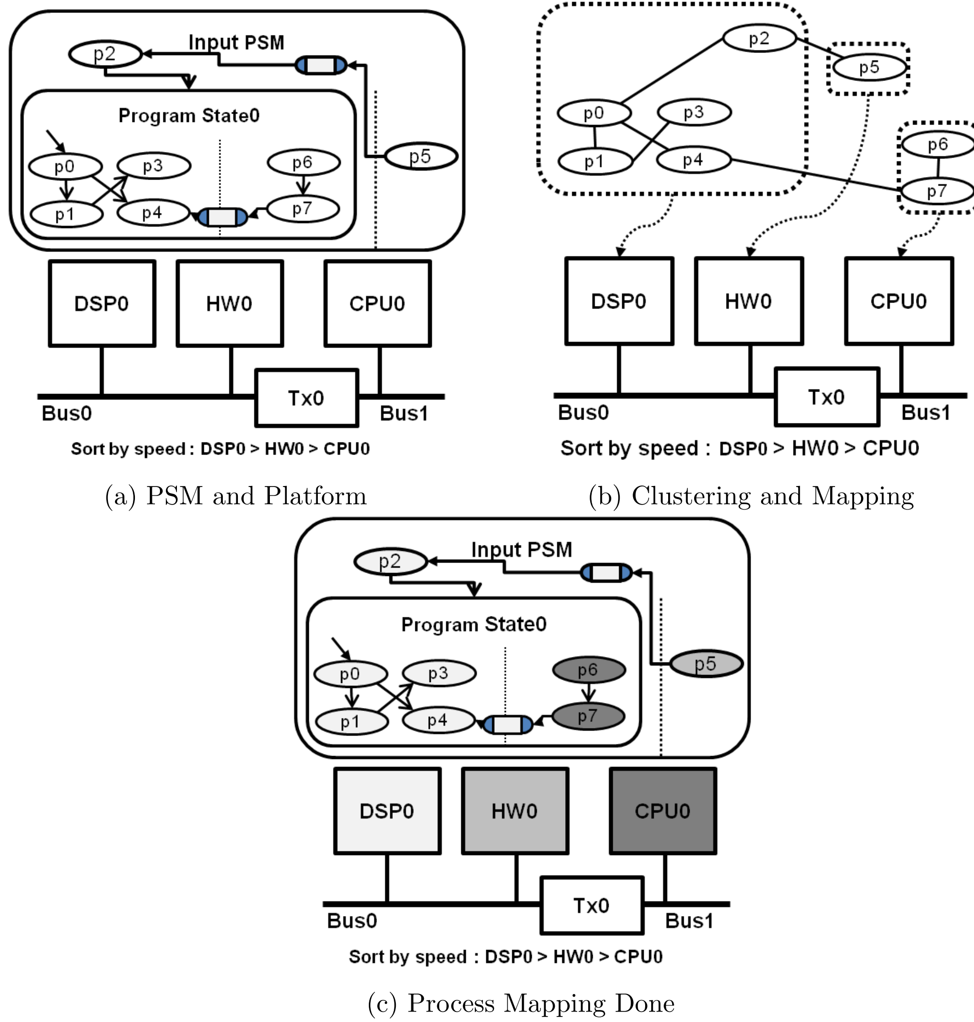
In addition, multiple sequential processes can be mapped to a single HW but not multiple parallel processes. Therefore, by considering the basic observations, we can improve HW utilization and as a result exploit parallelism even further.

Since our work is targeting computation-intensive applications, mapping is finalized as follows: the N clusters are sorted by the order of execution delay and the N PEs are sorted by speed. One-to-one mapping between PEs and clusters is performed in order.

5.4 Algorithm

We propose N-Way clustering based on our new closeness function C of process $p0$ and $p1$ in Equation 5.1. In the Equation, b_s and b_p show, respectively, whether $p0$ and $p1$ may run back-to-back and whether $p0$ and $p1$ run in parallel. D_e and D_t present, respectively, estimated execution delay of the two processes and estimated communication overhead.

$$C(p0, p1) = (c_0 + c_1 * b_s) * D_t + (c_2 - c_3 * b_p) * D_e \tag{5.1}$$



(a) PSM and Platform

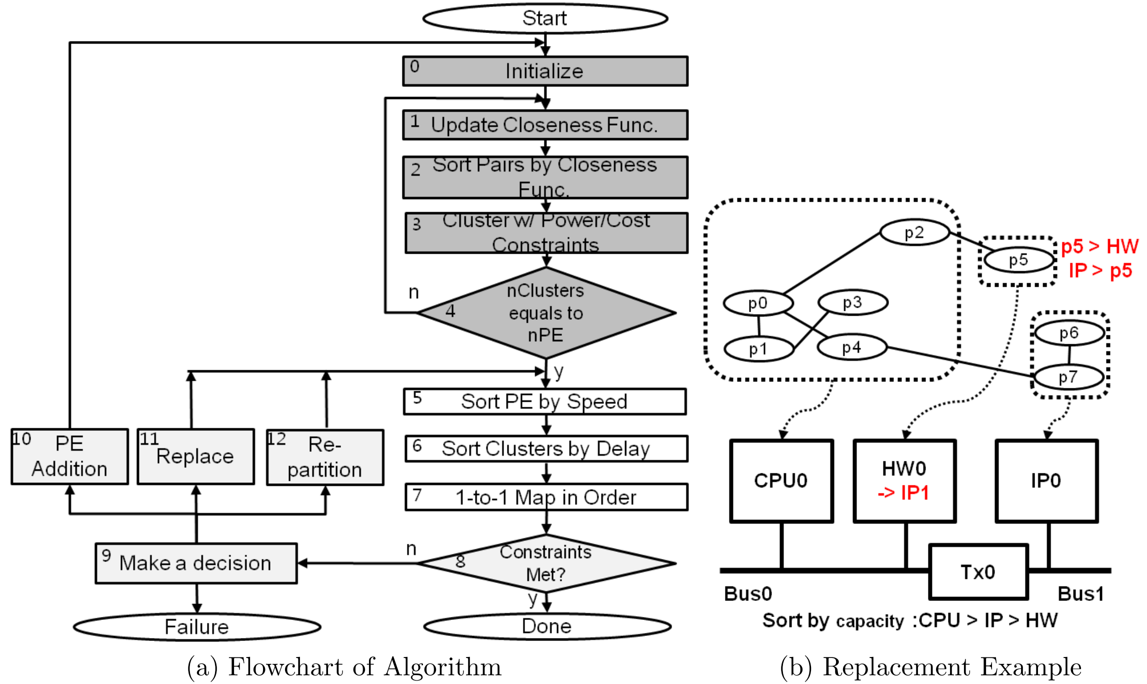
(b) Clustering and Mapping

(c) Process Mapping Done

Figure 5.3: N-Way Clustering and Mapping Example

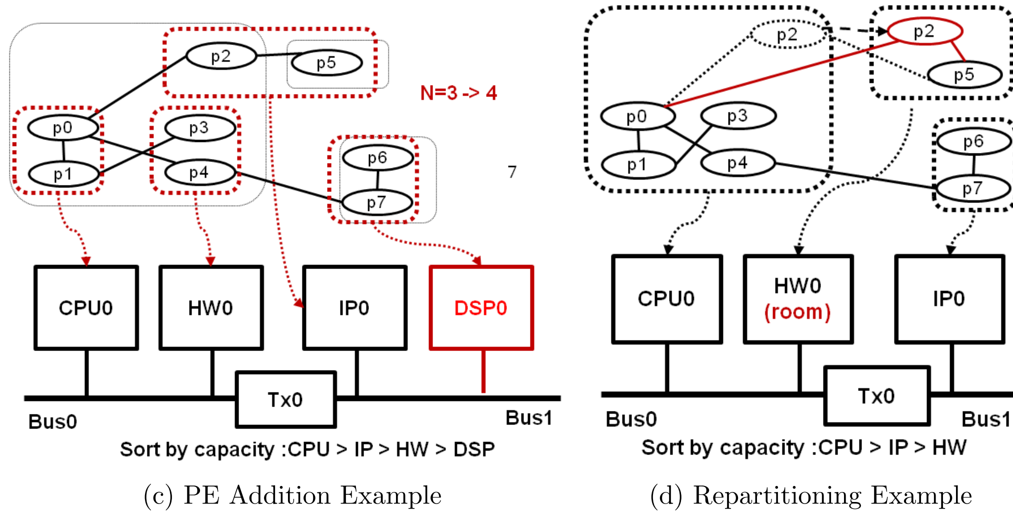
We can extends the closeness function to take two clusters, each of which is a mutually exclusive set of processes. For two clusters, $Cl0$ and $Cl1$, b_s indicates whether all processes in the two clusters can run sequentially or not and b_p shows whether all processes in the two can run in parallel. D_t and D_e are, respectively, the sum of estimated execution delays of the two clusters and overall communication overheads between the two. Equation 5.2 shows the extended closeness function of the two clusters.

$$C(Cl0, Cl1) = (c_0 + c_1 * b_s) * D_t + (c_2 - c_3 * b_p) * D_e \quad (5.2)$$



(a) Flowchart of Algorithm

(b) Replacement Example



(c) PE Addition Example

(d) Repartitioning Example

Figure 5.4: N-Way Clustering and Mapping (NWCM) Algorithm

After N-Way clustering completed, one-to-one mapping between clusters and PEs are performed. Since our approach is mainly targeting computation-intensive applications, clusters are selected one by one in order of the total estimated execution delay of the cluster and PEs are selected in order of speed.

Mapping is an NP-hard problem, resulting in several approaches based on many different algorithms. Taking into consideration process scheduling, we can apply

ILP or meta-heuristic-based approaches to our problem as well. However, ILP is not scaling well to large problem sizes and meta-heuristic based approaches offer no assurance that the required design quality is reached in a finite time. Putting this aside, we can start with a simple algorithm that works.

Figure 5.3 shows how the algorithm works. The input PSM and the initial target platform are given. From the input PSM, all leaf processes are enumerated. In this example, program state0 is not a leaf. Instead, from process p0 to process p7 are leaf processes. Between any of two processes, an edge is added and the closeness function is used as the weight of the edge. Note that edges with a zero or negative weight are omitted for simplicity. N-Way clustering is performed and what is following is one-to-one mapping depending on execution delays of the clusters and speeds of the PEs. The result is shown in Figure 5.3c.

Figure 6.3 shows the flow chart of the N-Way Clustering and Mapping Algorithm. The algorithm is decomposed into three phases, the N-Way clustering, mapping, and fixing phase, each of which is indicated by a different color.

Initially, each process constructs its own cluster. The N-Way clustering phase is a loop from box 1 to box 4 and colored dark gray. In the N-Way clustering phase, the following process is repeated;

- Update closeness function values for all pairs of clusters.
- Sort the pairs by closeness function.
- Select a cluster with the closeness function value as large as possible while keeping power/cost constraints met.

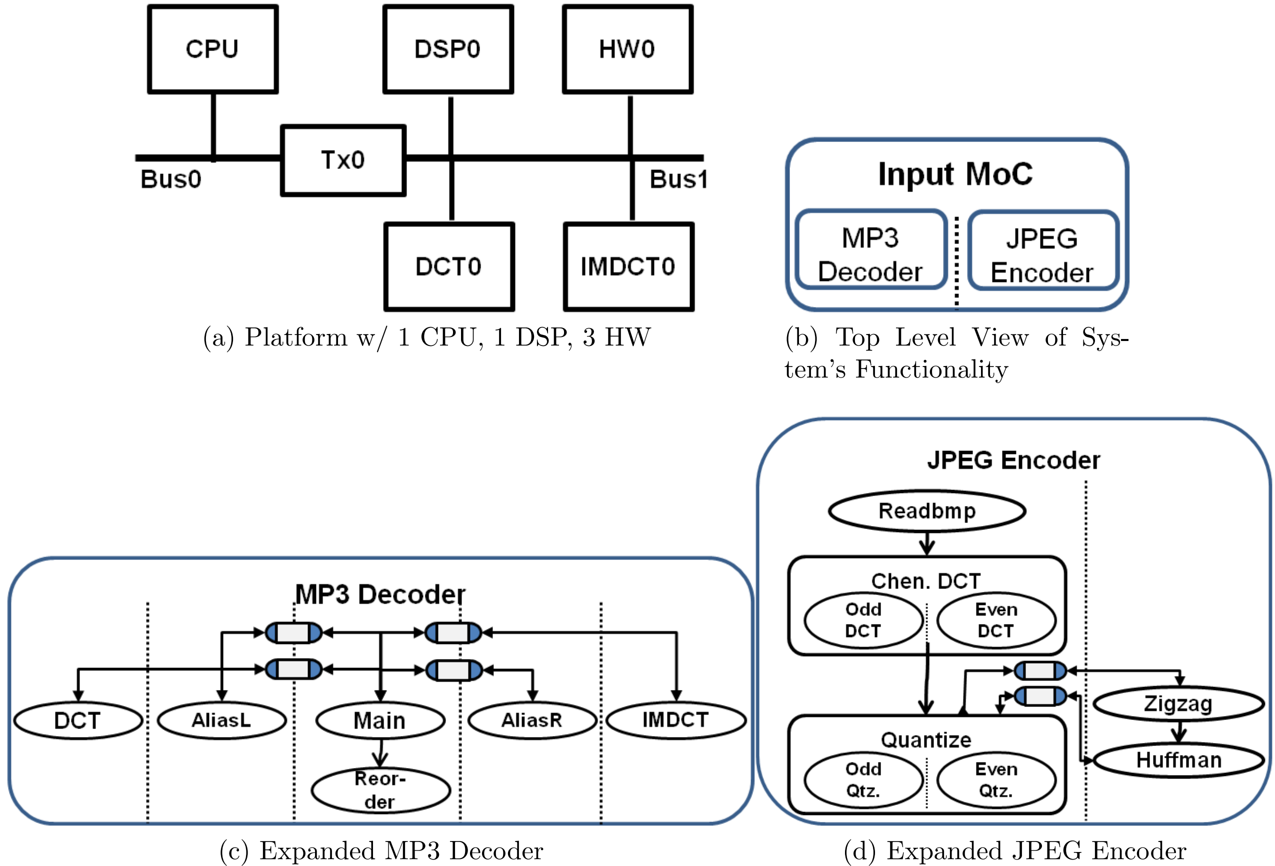
The one-to-one mapping process from box 5 to box 7 follows N-Way clustering. The

PEs are sorted by speed. The N clusters are sorted by estimated delays. One-to-one mapping is performed in decreasing order.

The last phase is to fix mapping if any of the design constraints are not met. Note that any heuristic algorithm may fail for many reasons. One reason is that the given constraints are too tight for any heuristic or are even impossible to meet. In the third phase of the proposed algorithm, the given mapping is fixed or the algorithm returns failed with the best design explored. The three ways to fix the mapping are depicted in Figure 6.3. Three options are selected by decision making schemes. In Figures 5.4b, 5.4c, and 5.4d, the real-time constraint is violated as an example. Other design constraints may be also violated. The first option is to replace PE. For example, in Figure 5.4b, HW0 is overloaded but replacing HW0 with IP1 solves the problem while meeting cost and power constraints. The algorithm replaces HW0 with IP1. As a result, the ordered list of PEs may change so that the one-to-one mapping process may need to be applied again. The second option is to add a new PE. For example, in Figure 5.4c, the real-time constraint is violated. Therefore, a new PE, DSP0 is added. In this case, 4-Way clustering is needed instead of the 3-Way clustering already completed. In general, after PE addition, with the updated N, N-Way clustering is performed again. The third option is to select a victim process and move the process to another cluster.

There are various ways to decide which option is selected to fix mapping. Also, there are multiple ways to select PE to replace or add. In addition, a number of ways exist to select both a victim process and the new cluster to locate the process. In this work, we try to replace PE first. If that fails, PE addition follows. The option selected last is repartitioning. The victim process is selected based on the closeness function at the first iteration of N-Way clustering. The cluster to which the process is moved is selected based on the constraint violated.

5.5 Case Study



(a) Platform w/ 1 CPU, 1 DSP, 3 HW

(b) Top Level View of System's Functionality

(c) Expanded MP3 Decoder

(d) Expanded JPEG Encoder

Figure 5.5: Application and Platform for Experiment

5.5.1 MP3 Decoder And JPEG Encoder

We have chosen a computation intensive multimedia application running, in parallel, an MP3 decoder and JPEG encoder. Figure 5.5a shows the given platform including 1 general purpose processor (GPP), 1 digital signal processor and 3 different pieces of custom hardware. The GPP is connected to bus0 and the others to bus1. A transducer bridges two buses. Figure 5.5b describes the top level view of the system's functionality. Two program states, MP3 decoder and JPEG encoder, are running in

parallel. Figure 5.5c and Figure 5.5d expand each program state, respectively. Shared variables accessed by sequential processes are omitted for simplicity. The frequency of each component is between 200 MHz and 400 MHz.

To prove our contribution, it is necessary to compare it to different approaches with similar algorithms and different closeness functions. For this purpose, we use CBB, LB and LPT as competitive algorithms. CBB is a modification of N-Way clustering [66]. CBB is similar to NWCM but uses a different closeness function, which includes only the number of bytes transferred between the two processes. We may compare NWCM to N-Way clustering and mapping algorithms with the closeness function taking only computation and/or communication. However, LB and LPT assure shorter latency than do the algorithms since LB and LPT considers PEs and the capacity of communication routes. Therefore, we compare NWCM to LB and LPT.

These comparisons are not sufficient to assert the quality of NWCM is acceptable. Therefore, NWCM also needs to be compared to previous, realistic approaches as well. We choose a modified version of SPEA for this kind of comparison. SPEA finds pareto optimal solutions in terms of power consumption, cost and maximum computation. For a fair comparison, we apply SPEA with a single objective function, maximum computation. The modified SPEA optimizes latency based on evolutionary algorithms. For SPEA, the number of generations, population size, crossover probability and mutation rate are kept the same as [18]. The input PSM is manually translated to a Khan Process Network and the Tx is used as a single FIFO memory. In the platform, nothing but Tx0 can be used as a shared memory.

Any of the five algorithms—NWCM, LB, LPT, CBB and modified SPEA—produces a single mapping between the given platform and the given computation model. The result of the mapping process is a system model, which is refined to a TLM. The

TLM is simulated to evaluate the design quality.

Table 5.1: Total Execution Delays on Each PE

Algorithms	CPU	HW	DCT	IMDCT	DSP	Delay
NWCM	8.5 ms	2.8 ms	2.3 ms	3.2 ms	5.9 ms	22.7 ms
SPEA	9.3 ms	1.3 ms	1.4 ms	UNUSED	8.9 ms	20.9 ms
LPT	8.5 ms	1.2 ms	2.0 ms	1.7 ms	14.0 ms	27.4 ms
LB	0.9 ms	1.2 ms	2.0 ms	1.5 ms	12.7 ms	18.3 ms
CBB	21.6 ms	0.1 ms	0.1 ms	0.1 ms	19.8 ms	41.7 ms

Table 5.2: Overall Latency

Algorithms	CPU	HW	DCT	IMDCT	DSP	Latency
NWCM	8.5 ms	2.8 ms	2.3 ms	3.2 ms	5.9 ms	16.8ms
SPEA	9.3 ms	1.3 ms	1.4 ms	UNUSED	8.9 ms	24.2ms
LPT	8.5 ms	1.2 ms	2.0 ms	1.7 ms	14.0 ms	20.9ms
LB	0.9 ms	1.2 ms	2.0 ms	1.5 ms	12.7 ms	22.6ms
CBB	21.6 ms	0.1 ms	0.1 ms	0.1 ms	19.8 ms	24.2ms

In TABLEs 5.1 and 5.2, the simulation results are enumerated. Each row shows the simulation result of the TLM obtained by each algorithm, respectively. Note that communication overhead is far smaller than latency so that it is omitted. Each row shows the distribution of the total execution delay in addition to the latency. The total execution delay is the sum of the execution times from each process, during which the process is actively running. Latency is the overall response time to process a given set of inputs.

First of all, communication overhead in this application was very small. Therefore, NWCM outperforms CBB in terms of both total execution delay and latency since CBB mainly optimizes communication.

Compared to LB, which optimizes computation only, we can point out the following:

- Execution delay is more evenly distributed in NWCM than it is in LB.
- The total execution delay of LB is, nonetheless, smaller than that of NWCM.

- Overall latency of NWCM is smaller than that of LB.

Multiple sequential processes can be mapped to an HW but multiple parallel processes cannot in general. Therefore, NWCM can map more processes to HW instead of SW and, as a result, improve HW utilization. In spite of this fact, LB usually finds better PEs to the given processes so that the total execution delay of LB is smaller than that of NWCM. However, the latency of LB is larger than that of NWCM, which implies latency does in fact depend on process scheduling. Interestingly, between LB and CBB, the difference in latency is much smaller than the difference in total computation delay. In this given application, sequential processes usually perform a large amount of communication to each other via shared variables. Therefore, CBB often binds sequential processes together. It is also proof of impact of process scheduling.

Since communication overhead is miniscule, LPT produces similar mapping to that of LB. However, LPT exploits parallelism more than LB does due to process scheduling. Thus, the latency of LPT is shorter than that of LB. In the same sense, NWCM is better than LPT.

SPEA was originally applied to Process Network Models. Therefore, it does not consider process scheduling. Therefore, as shown in TABLEs 5.1 and 5.2, even though SPEA shows lower total execution delay than NWCM does, the overall latency of SPEA is worse than that of NWCM.

In summary, the proposed algorithm, NWCM, is better than any other by at least 24.4% in terms of latency although its total execution delay is not the shortest. The reason is process scheduling and HW utilization.

Chapter 6

Cycle-Approximate Estimation Based Mapping

This chapter describes Cycle-Approximate Estimation-Based Mapping (CAEBM) of a computation model to the given multi-processor/core platform. In previous work, estimation precedes mapping, while cycle-approximate estimation follows mapping. Therefore, the estimates given by the existing estimation techniques is not vey accurate, lowering the quality of mapping. Cycle-approximate estimation improves the mapping process but requires changes to it.

CAEBM is driven by recent changes in estimation technologies that allow fast cycle-approximate estimation. With an initial mapping, CAEBM iteratively uses cycle-approximate estimation and heuristics to improve the mapping in terms of execution time. A case study is performed with a multimedia application, in parallel, an MP3 decoder and JPEG encoder. We use two different algorithms to find the initial mapping for CAEBM. CAEBM reduces the execution time by at least 36.3%.

6.1 Application Model

In this paper, we use a Program State Machine (PSM) [9] to capture the application. A general computation model should have concurrency, the concept of states, imperative programing languages, dynamic data-oriented behaviors, structural and behavioral hierarchy, etc. PSM is one such general computation model.

An application is a hierarchical program state. Each program state can be recursively defined as either a leaf program state called a process, a decomposition of parallel program states, or state transitions among sequential program states. Any pair of program states may communicate over one or more channels.

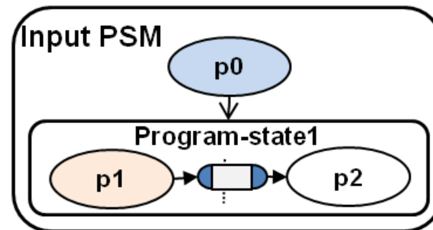
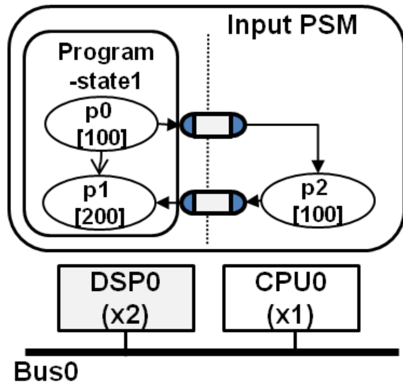
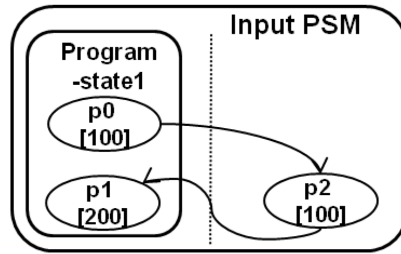


Figure 6.1: Application Model

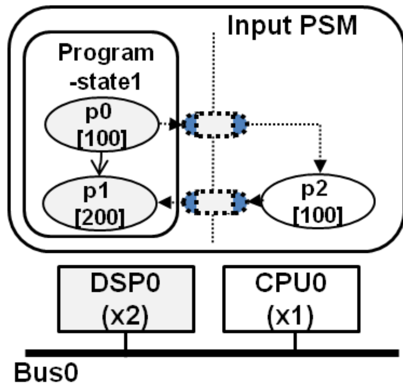
Figure 6.1 shows a simple PSM example. The PSM is sequentially decomposed into two program states: p0 and program state1. P0 is a process, or, interchangeably, a leaf program state. A process is a function written in an imperative programming language such as C. Program state1 is also decomposed into two parallel processes: p1 and p2. P1 and p2 communicate over a channel. Likewise, in general, p0 and p1 can also communicate over a channel such as a memory channel.



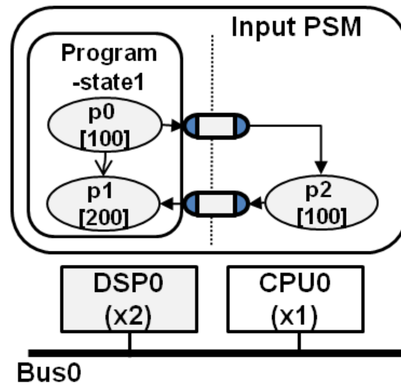
(a) Platform and computation model



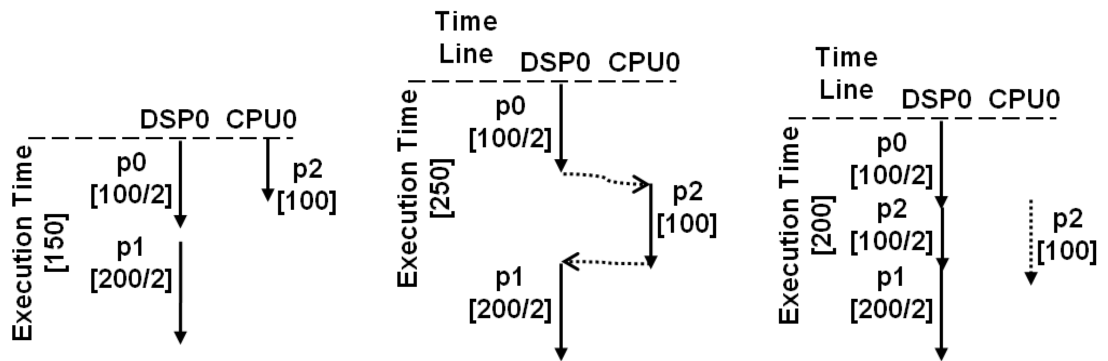
(b) Actual Execution Order Due to Data Dependencies



(c) Estimation Ignoring Data Dependencies and Mapping



(d) Improvement by CAEBM



(e) Estimated Timeline of Mapping (c)

(f) Actual Timeline of Mapping (c)

(g) Estimated/Actual Timeline of Mapping (d)

Figure 6.2: CAEBM

6.2 Cycle-Approximate Estimation Based Mapping

6.2.1 Overview

Figure 6.2 describes how CAEBM works. Figure 6.2a is the input PSM and the platform with two processing elements. DSP0 is twice as fast as CPU0. The numbers on each process is the execution time of the process on CPU0. In this example, the major problem is to select PE to locate p2.

The assumption in this example is as follows: Although p2 seems to be in parallel with the others, p0, p2, and p1 run sequentially due to the data dependency. In this simple example, the actual order of execution may be easily known ahead of time. However, in general, there are at least several tens of processes and channels. The execution order may vary depending on RTOS scheduling policies, communication routing, bus arbitration policies and many others. Thus, the actual execution order is not predictable before cycle-approximate estimation.

Figure 6.2c shows a mapping based on estimation that ignores data dependencies. The mapping compares Figure 6.2e and Figure 6.2g, although Figure 6.2e is different from the actual timeline of Figure 6.2c. Even though the mapping algorithms resulting in Figure 6.2c is the optimal, the wrong estimation misleads the design decisions. The estimated execution time of mapping in Figure 6.2c is 150, while the actual execution time is 250.

CAEBM fails to see every possible mapping. However, CAEBM compares mapping near the initial mapping depicted in Figure 6.2c based on cycle-approximate estimation. Thus, CAEBM makes a decision based on the right timelines: Figures 6.2f and 6.2g. Since these three processes are completely sequential, running all of them on the same PE is optimal in terms of execution time. Therefore, CAEBM move p2

to DSP0.

6.2.2 Algorithm

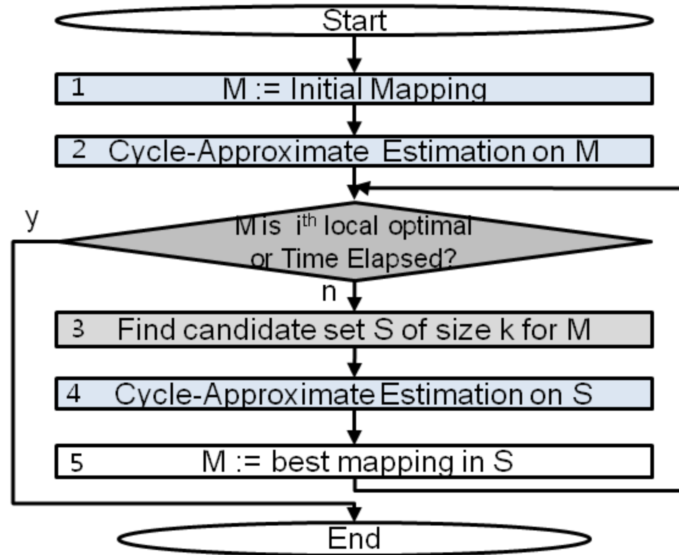


Figure 6.3: Flow Chart of CAEBM

Figure 6.3 describes the algorithm of CAEBM. An initial mapping is given. Cycle-approximate estimation is performed on the initial mapping. CAEMB uses heuristics and this cycle-approximate estimation to construct a set of candidates for the mapping that is fed to the input of the next iteration. From the constructed set, CAEBM identifies the best candidate. This decision is based on the cycle-approximate estimation on each candidate in the set.

The loop terminates if the given time is elapsed. In addition, the loop terminates if two local optimums are found. A local optimum is defined as the mapping satisfying the following two properties:

- A local optimum found at i^{th} iteration is better than the mapping constructed at $(i-1)^{th}$ iteration.

- A local optimum found at i^{th} iteration is better than any other mapping in the candidate set S constructed at i^{th} iteration.

The local optimum that comes first may not be sufficiently close to the global optimum in many applications. Thus, CAEBM finds i^{th} local optimum instead of the first local optimum. i could vary depending on the given applications and we use 2 in this paper. The size of the candidates set S is k. The greater k is, the smaller the number of iterations. k could also vary and we use 2 for it.

In construction of the candidate set S, heuristics are necessary. If P is the set of processes and V is the set of PEs, the number of different mappings can be as high as $\|P\|^{\|V\|}$. The $\|P\|^{\|V\|}-1$ mapping excluding the current mapping M could be elements of S. Performing cycle-approximate estimation on all of these mapping is not feasible. Therefore, we use heuristics using the cycle-approximate estimation on M, choose only k candidates among the $\|P\|^{\|V\|}$ mapping and perform cycle-approximate estimation on the selected candidates only.

Since CAEBM is a local search, we construct each element of S by selecting a single process and move it to an other PE. Therefore, the objective function f is defined with domain P and codomain V as follows:

$$
\begin{aligned}
f(p_i, v_j) = w_0 \sum_{p \in v_j U \{p_i\}} ExecutionTime(p) \\
+ w_1 \sum_{p \in v_j} CommOverhead(p_i, p) + w_2 Par(p_i, v_j), \\
where\, p_i \in P,\ v_j \in V,\ w_0,\ w_2 \geq 0,\ and\, w_1 \leq 0
\end{aligned}
\tag{6.1}
$$

A small f indicates p_i is recommended to be mapped to v_j. The idea is to balance computational loads, to map the processes together if there is large data transactions between the processes, and to exploit parallelism. Thus, w_0 and w_2 is a positive

coefficient, while w_1 is negative.

We compute f for all pairs of processes and PEs, and finds k pairs of p_i and v_j that have the smallest possible f and meet the following conditions as well:

- p_i is not now on the v_j.
- Moving p_i to v_j does not fall into the mapping already visited.

Execution time of each process is in a table given ahead of CAEBM. We modified [26] to print the execution time of each process on each PE by a single TLM simulation. To compute the communication-overhead function, we record the total number of bytes in the transactions for every pair of processes respectively. This procedure can be completed by the same single TLM simulation as well. We divide the number of bytes by the bandwidth of the best route between p_i and v_j.

Communication and computation must be taken into consideration and that is what $ExecutionTime$ and $CommOverhead$ functions are for. However, we include the Par function in the objective function f.

Par function represents approximately how much the process p_i can be executed in parallel with the processes on v_j. To define Par, we define $ParProcesses$ first. For any given two different processes p and q, $ParProcesses(p,q)$ is the minimum execution time of the processes if they are parallel processes in the PSM. Otherwise, it is the sum of the execution time of p and that of q. Par is defined as follows:

$$Par(p_i, v_j) = \sum_{q \in v_j} ParProcess(p_i, q) \; , \qquad where \, p_i \in P \, and \, v_j \in V \tag{6.2}$$

6.3 Case Study

6.3.1 MP3 Decoder And JPEG Encoder

We have chosen a computation-intensive multimedia application running a MP3 decoder and JPEG encoder in parallel. The PSM capturing the entire application is decomposed into two parallel program states. Two program states, MP3 decoder and JPEG encoder, are running in parallel. The number of levels of hierarchy is up to 3. There are 13 processes and 17 channels. A platform is randomly given under cost constraints. The platform consists of five different PEs and two buses.

For comparison, we choose two different algorithms to find the initial mappings for CAEBM. CAEBM starts each initial mapping, respectively. The first algorithm is Load Balancing (LB) [9]. LB is a heuristic that optimizes computation only. We also modify SPEA. SPEA [18] performs multi-criteria optimizations including approximation of execution time. We changed it to have a single optimization: execution time.

For each initial mapping, firstly, we measure the execution time by TLM simulation and compare the execution time to the estimate that each mapping uses. The objective function of SPEA is the maximum processing time, which is named by the authors. The processing time of a PE v is the sum of ${f_v}^e$ and ${f_v}^c$. ${f_v}^e$ is the sum of execution time of each process on the PE. f^c is the sum of the delays due to data transactions in which any PE on the process is involved. The objective function is the maximum of ${f_v}^e + {f_v}^c$. The objective function of LB excludes ${f_v}^c$ from the formula.

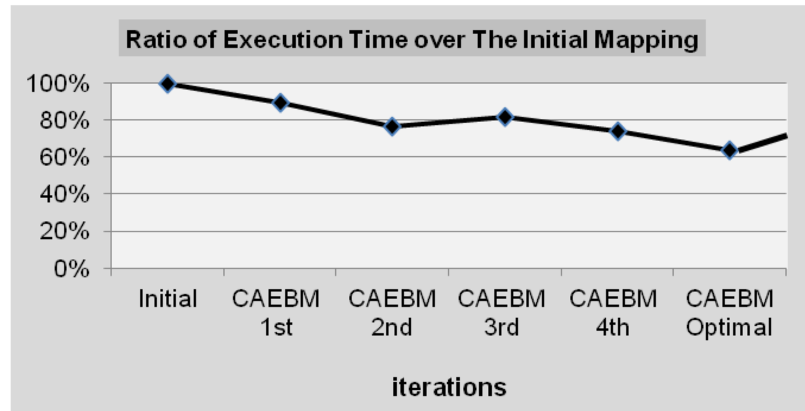
Table 6.1: Errors in Estimation: SPEA and LB

Mapping	Estimate	Execution Time	Error Percentage Time
SPEA	90.77 ms	112.17 ms	19.5%
LB	12.39 ms	65.71 ms	81.1%

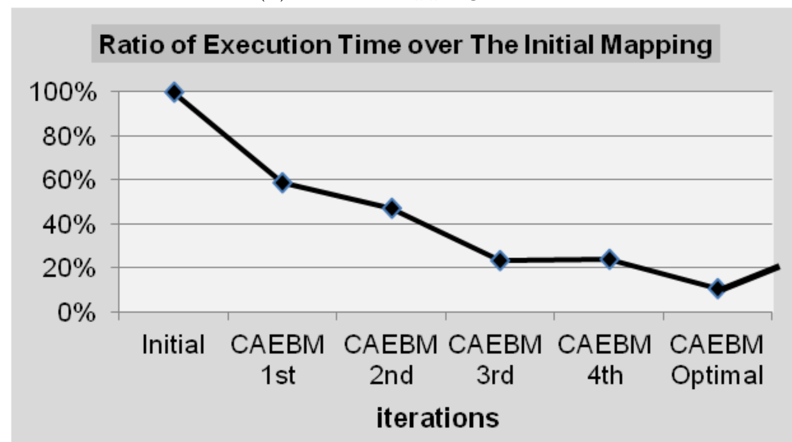
TABLE 6.1 shows the estimates and the execution times. The estimates are different from the execution time by 19.5% and 81.1%. In addition to communication and computation, process scheduling due to data dependencies and control dependencies also greatly impacts the execution time but is not included in either objective function. Therefore, the errors are not small, which indicates there is room for improvement.

Figure 6.4 shows the changes in execution time during the improvement process of CAEBM. In Figure 6.4a and Figure 6.4b, the initial mapping is LB and SPEA. CAEBM iteratively improves each of these initial mapping. The execution time is recorded at every iteration. The ratio of the execution time over the initial mapping is depicted in Figure 6.4a through Figure 6.4b. The X-axis is the number of iterations, while Y-axis represents the ratio of the execution time over the initial mapping. The CAEBM improvement process stops since the second local optimum is found. In all cases, there are improvements in execution time. The execution time is reduced at least by 36.3% compared to the initial mapping.

SPEA basically targets applications captured in a process network. In a process network, every task is assumed to run in parallel even though the tasks run sequentially due to data dependencies. In the estimation used by LB, niether communication overhead nor control flow of the application is considered. On the contrary, the heuristics used by CAEBM has a tendency to separate parallel tasks. Thus, the execution time is reduced during iterative iterations.



(a) Initial Mapping: LB



(b) Initial Mapping: SPEA

Figure 6.4: Changes in The Ratio of Execution Time over The Initial Mapping During CAEBM's Process on MP3 + JPEG Application

Chapter 7

Trace-Driven Performance Estimation of Multi-core Platforms

© 2014 IEEE. Reprinted with permission from Kyoungwon Kim and Daniel Gajski, Trace-Driven Performance Estimation of Multi-core Platforms, 2014

In Transaction Level Model-based design, a crucial role is played by Transaction Level Model estimation. One type of such estimation that is as fast as native simulation, cycle-approximate and applicable to both software and custom hardware is simulation-based Transaction Level Model (TLM) Estimation that depends on TLM simulation. For every platform selection and mapping, however, the entire platform must be simulated. The simulation overhead is reduced by Trace-driven estimation but such estimation is not applicable to custom hardware and often requires Cycle Accurate Models (CAMs), which may not available for the whole platform.

In this paper, we present Trace-Driven Performance Estimation (TDPE) of multi-

processor/core designs. TDPE is a trace-driven estimation but applicable to both software and hardware and requires no CAM. Since TDPE includes a mere single functional simulation, it is orders of magnitude faster than TLM Estimation. TLM Estimation is cycle-approximate by considering the data path of each Processing Element (PE), memory hierarchy, RTOS scheduling and overheads, bus arbitration and overhead. TDPE takes all of them into account so is as accurate as TLM Estimation.

7.1 Trace-Driven Performance Estimation

7.1.1 Assumptions

We assume that inter task communications are conducted by calling communication APIs. We assume that the order of communication API calls of each task depends on the data only. These assumptions are valid for multiple applications such as multimedia applications captured in SDF.

7.1.2 Definitions: Execution, Communication Events and Traces

A *communication event* of a task is defined as the interval between calling and returning from a communication API call. Each communication event is coupled with the number of bytes transferred. The length of a communication event is computed during alignment according to the bus protocol model. An *execution event* of a task refers to the execution of the task between any two of the following: the start of the task, the end of the task and communication events. An execution event is coupled with (1) all possible optimistic scheduling delays [26] for the event, (2) the number of branches, (3) and the total amount of local memory accesses. Optimistic scheduling

delay of an execution event is the delay estimate where there is no branch prediction failure and no cache miss. The impact by branch prediction failures and cache misses will be added in the alignment step. The number of cache misses is computed by multiplying the total amount of local memory accesses and the probability of cache miss. The penalty of a branch prediction failure and a cache miss is given as a part of the PE configuration. The *trace* of a task is an ordered list of the communication and execution events of the task. The order depends on the execution path of the task, which is affected not by the platform or mapping but by the data.

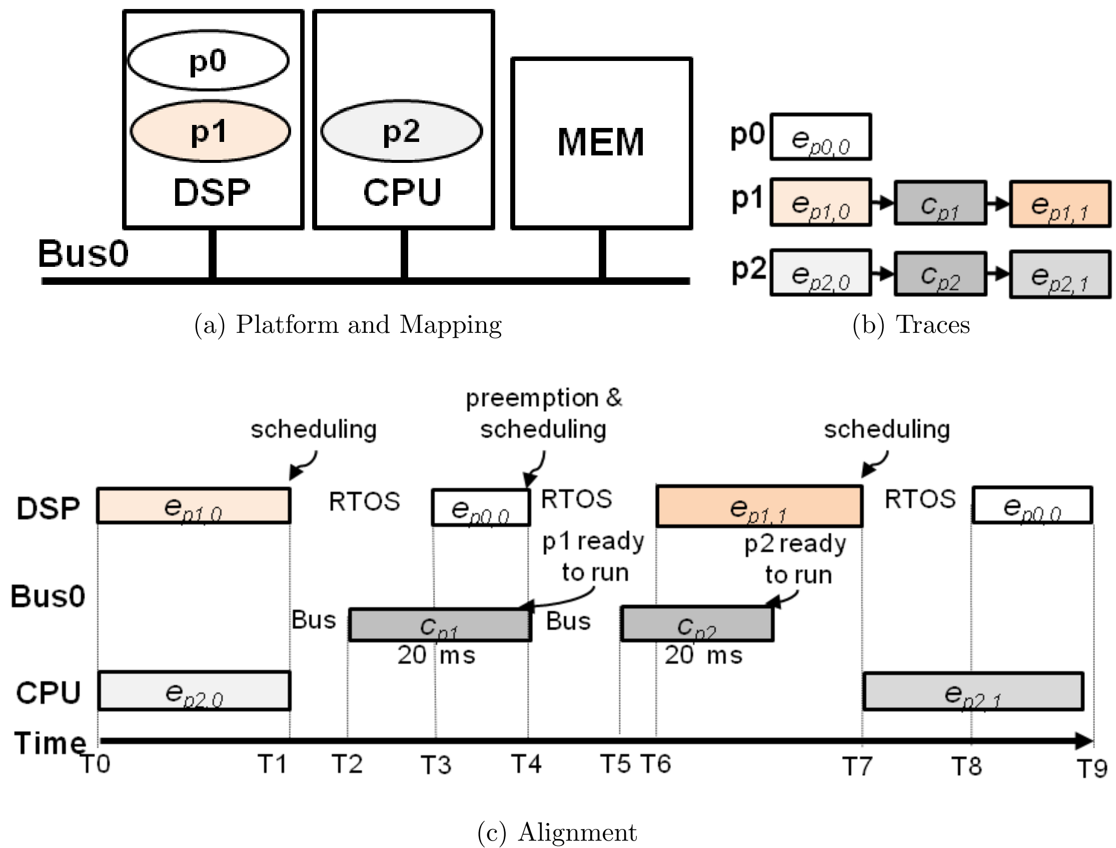
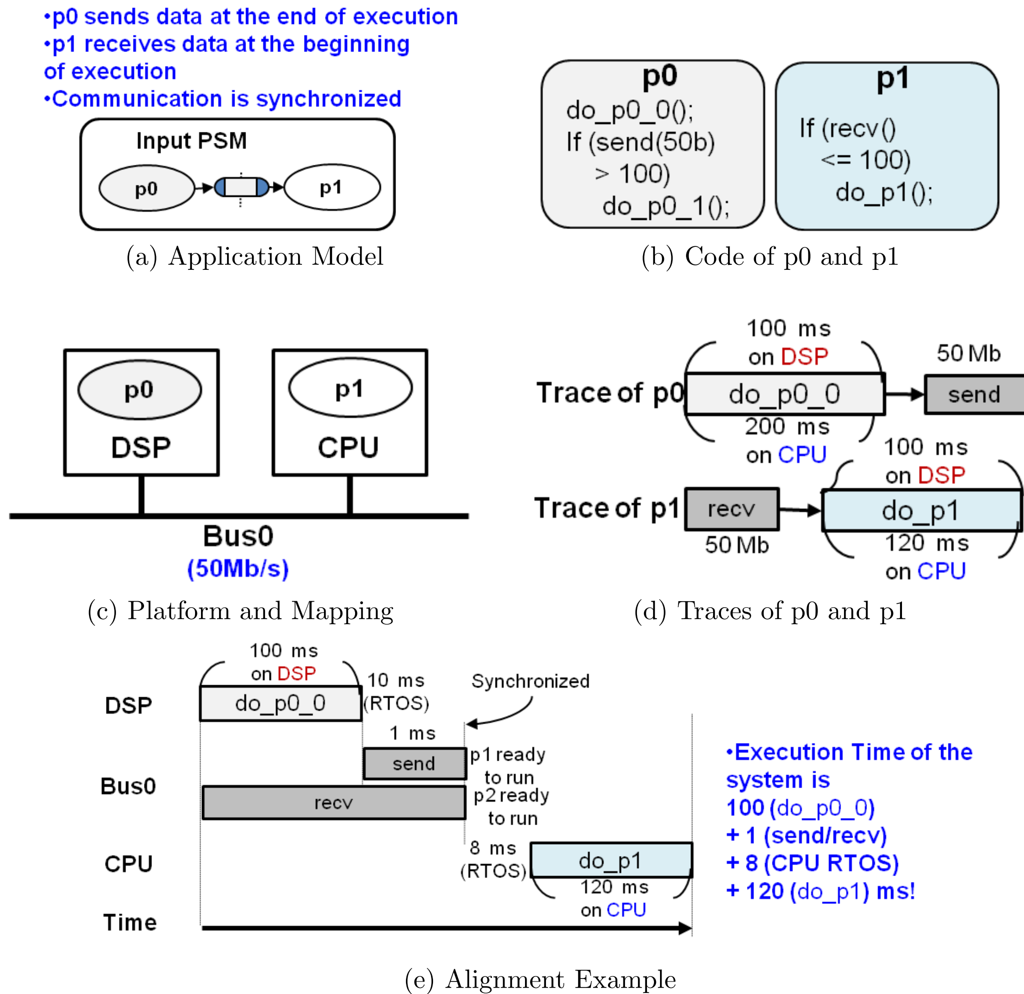


Figure 7.1: Alignment Example w/ RTOS Model and Bus Protocol Model



(e) Alignment Example

Figure 7.2: Trace-Driven Performance Estimation

7.1.3 Trace-Driven Performance Estimation

Figure 7.2 shows the basic idea of TDPE. Figures from 7.2a through 7.2c show the application, the code of each task, the platform and mapping. In Figure 7.2b, do_p0_1 is not called and do_p1 is called. Thus, the trace of each task is as it appears in Figure 7.2d. The alignment engine must place each event in the traces on the global time line. A trace defines the order of its events. Moreover, inter task communications defines the partial order of events in different traces. In this example, p1 executes **recv** first and yields CPU. CPU is now in idle state. Since the communication is assumed to be synchronized, **send** and **recv** must be end at the same time. **Send**

is called after completion of the execution event do_p0_0. Thus, the order of the execution events and communication events must be as in Figure 7.2e. Between do_p1 and **recv**, there may be RTOS overhead.

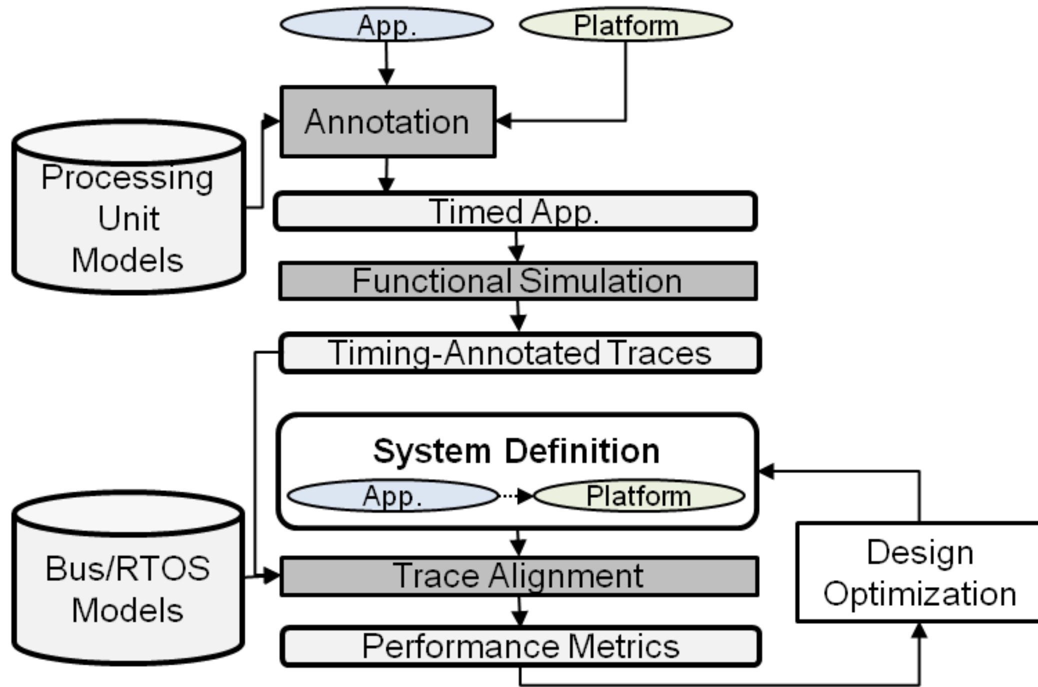


Figure 7.3: Design Flow with Trace-Driven Performance Estimation

Figure 7.3 shows the design flow with TDPE. Each basic block of the application source code is annotated with all possible delay estimates. During this annotation, small codes are instrumented to profile local memory accesses and branch operations. Our trace generation reuses the source code annotation described in [26]. Each basic block goes through the abstract data path of each PE. Following that, the optimistic scheduling delay estimate of each basic block is given. Performed with this timed application is a single functional simulation.Inside the communication APIs are recorded the traces and their performance metrics. After system definition, which is basically platform selection and mapping, the generated traces are aligned and the performance metrics given. Note that there is no simulation. Even if either the platform or mapping changes by design optimization, trace alignment provides performance metrics,

making simulation unnecessary.

7.1.4 Abstract RTOS, Memory Hierarchy, Processing Element and Bus Protocol Models

As Hwang et al [26] did, we compute the impact of cache based on statistics. Functional simulation profiles the amount of memory accesses. TDPE multiplies this by the cache miss rate. A PE can have such configurations as cache sizes and frequency. The alignment engine takes all of these into account when computing the length of each execution event.

Figure 7.1 describes RTOS models and bus protocol models. Figures 7.1a and 7.1b show the platform, mapping, and traces. Note that c_{p1} and c_{p2} are read/write operations to the global memory MEM. The core of abstract RTOS models of TLM is the abstract RTOS operations such as scheduling with the proper RTOS overheads [25]. Likewise, the core of the abstract bus models in TLM [61] is abstract bus operations and their overhead estimates. In our TDPE, those are emulated during alignment. In this example, the RTOS on DSP is preemptive and priority based. p1 is assumed to have the highest priority. Thus, at T0, the first event of the task p1 is selected. Once p1 yields DSP at T1 for communication, p0 is selected by the scheduler and thus the first event of p0 $e_{p0,0}$ is placed on the global time line following $e_{p1,0}$. Note that a proper RTOS overhead is prepended to $e_{p0,0}$ so $e_{p0,0}$ is placed at T3 instead of at T2. The alignment engine can also take into account preemption. At T4, the communication event c_{p1}— which is a communication API call by p1—is finished. However, $e_{p0,0}$ is not completed yet. Thus, the alignment engine shortens $e_{p0,0}$ and preempts. The first of the remaining events of p1 is $e_{p1,1}$ and the event is placed at T6. The rest of $e_{p0,0}$ is placed at T8 after $e_{p1,1}$ is finished.

Table 7.1: Comparison in Estimation Time

Target Platform	CAM Estimation	TLM Estimation Annotation+ Simulation	TDPE Estimation Annotation+ Simulation
SW	15.93h	31.262s+0.004s	x+0.076s
SW+1	17.56h	49.986s+0.217s	x+0.077s
SW+2	17.71h	47.290s+0.254s	x+0.081s
SW+4	18.06h	71.131s+0.357s	x+0.084s

$x = \frac{93.56}{N} sec$ is the average estimation time for a platform selection&mapping if we need N platform selection&mapping to find the solution that meets all design constraints.

Table 7.2: Accuracy Results: Error % against TLM Estimation (SW, SW+1)

I/D Cache	SW			SW+1		
Size	TLM	TDPE	Error	TLM	TDPE	Error
0k/0k	291.05ms	290.19ms	0.29%	308.88ms	308.44ms	0.14%
2k/2k	104.58ms	100.17ms	4.22%	93.07ms	92.75ms	0.33%
8k/4k	65.34ms	63.73ms	2.47%	59.47ms	59.28ms	0.31%
16k/16k	58.64ms	57.39ms	2.14%	50.25ms	48.95ms	2.58%
32k/16k	58.46ms	56.36ms	3.60%	58.09ms	57.27ms	1.40%

The application is an MP3 decoder. MicroBlaze has a priority-based non preemptive RTOS.

Table 7.3: Accuracy Results: Error % against TLM Estimation (SW+2, SW+4)

I/D Cache	SW			SW+1		
Size	TLM	TDPE	Error	TLM	TDPE	Error
0k/0k	253.05ms	248.25ms	1.89%	225.92ms	221.55ms	1.94%
2k/2k	82.31ms	81.20ms	1.36%	83.16ms	81.48ms	2.02%
8k/4k	267.70ms	266.99ms	0.26%	52.53ms	51.91ms	1.18%
16k/16k	51.45ms	50.95ms	0.97%	49.41ms	49.03ms	0.77%
32k/16k	52.92ms	51.67ms	2.34%	48.02ms	46.97ms	2.19%

The application is an MP3 decoder. MicroBlaze has a priority-based non preemptive RTOS.

The alignment engine emulates the abstract bus protocol models. In this example, p1 and p2 request, at the same time, the global memory access through Bus0. In the bus protocol model, these requests must be sequentialized according to the arbitration policy. Since DSP has the highest priority, p1 on CPU gets the bus first at T2. Note that there is an arbitration delay between T2 and T1. The length of each communication event is equal to 20 ms. The length is computed by the alignment

engine according to the bus protocol model.

7.2 Case Study

We compared TDPE to TLM estimation [26]. In this case study, we reused the application, platform, mapping and configurations used by the authors. The application is MP3 decoder. In the platform, there are 3 types of PEs: MicroBlaze processor, custom HW and IMDCT. SW, SW+1, SW+2 and SW+4 are the platform with 0, 1, 2 and 4 hardware components. Instruction/Data cache size of MicroBlaze processor was configurable. The experiments with CAM were taken from [26].

TABLE 7.1 compares, in terms of estimation time, TLM Estimation, estimation using CAM and TDPE. Estimation using CAMs is accurate but, compared to the rest, orders of magnitude slower. TLM Estimation time is the sum of annotation time and simulation time. Accordingly, TDPE time is the sum of annotation time and alignment time. As expected, alignment time was much shorter than simulation time. Moreover, if N-system definitions are required for an optimal design to be found, annotation time of TDPE is shortened inversely proportional to N since annotation is conducted once regardless of N. Therefore, TDPE time was 18.84% than TLM Estimation when N is 10 and 2.03% when N was 100.

We are grateful to the authors of [26, 25], who allowed us access to the internals of their implementation. We implemented TDPE so that it emulated the authors' framework. During alignment, TDPE emulated the RTOS operations with their overhead values and bus operations with their overhead values. The operations and overhead values were almost the same as the framework of [26]. TDPE reused the authors' annotation engine to compute the optimistic scheduling delays. Once the

platform was selected and mapping given, the right optimistic scheduling delay was chosen. Following that, TDPE computed the branch prediction and cache miss delay in the same way as [26] and added the computed delay to the optimistic scheduling delay. Therefore, as TABLEs 7.2 and 7.3 show, the accuracy of TDPE is almost the same as that of the TLM Estimation.

Chapter 8

Conclusion

8.1 Summary

To cope with the dual obstacles of design complexity and the pressure of a short time-to-market, the paradigm of embedded system design has shifted to platform-based design. The platform-based design is able to maximize post-fabrication reuse of the verified components. Moreover, this platform-based design tends to be software-centric. Otherwise, the design process would not be able to handle the high design complexity. However, software-centric design may encounter performance issues. To cope with this design challenge, designers have traditionally resorted to frequency scaling. Frequency scaling, however, has become infeasible. Indeed, power consumption is proportional to the square of frequency; embedded systems often have power constraints, and battery technology has advanced only gradually. Thus, the de facto standard for tackling the performance issues in software-centric platform-based design

is the use of multi-core/processor platforms. These multi-core/processor platforms, however, burden embedded system design with more design challenges. Heightening even further the design complexity of embedded systems have been ever-increasing user demand and the exponentially increasing rate of chip density. Thus, every major technological roadmap addresses the productivity gap.

Researchers have proposed as a way to close this productivity gap Transaction Level Model-based design methodology. Transaction Level Model-based design defines three different models: Transaction Level, Pin-Cycle Accurate, and the Behavioral model, which is purely functional. At Behavioral Level, the design space is given with regard to hardware-software partitioning, platform selection, and mapping. Designers explore the design space to make such design decisions. Based on the design decisions, TLM is automatically generated so as to evaluate them. Once the design constraints are met, PCAM is generated from the TLM. In Transaction Level Model-based design, major design decisions have been moved to the system level, where the design complexity is much lower than any other level of abstraction.

Although the design space is dramatically narrowed in Transaction Level Model-based design, it is still a must that there be automation in design space exploration. Contemporary embedded systems are already too complex, making the design space too broad. Manual design space exploration is fast becoming infeasible.

In automatic design space exploration, there exists a conflict that gives rise to a "chicken-or-the-egg" type of dilemma: the design decisions need estimation, yet accurate estimation cannot be fed to the input of the design process. Previous works mostly assume that estimation is given. However, such estimations cannot be accurate. Cycle-approximate estimations such as Transaction Level Model estimation can hardly be coupled with existing automatic design space algorithms. Hence, a new approach to automation in design space exploration is called for.

This dissertation completes Transaction Level Model-based design methodology by adding new approaches to automatic design space exploration. The basic idea is to divide the design space exploration process into two phases: initial design decision-making based on rough estimation and iterative improvements based on Transaction Level Model estimation. In addition, as the iterative improvements phase depends on TLM estimation, it is crucial to provide fast, accurate, and general TLM estimation.

Thus, the problems that are solved by this dissertation can be summarized as follows:

- algorithms for the initial design decision-making phase
- algorithms for the iterative improvements of the given design
- fast, accurate, and general TLM estimation

Any automatic design space algorithm is based on the computation model. In selecting computation models, this dissertation considered the following two tenets:

- Computation models should be complete so as to describe the entire system.
- Computation models should serve not only for simulation but also for synthesis.

A general computation model such as a PSM is complete. There exist well-established design methodologies and tools that offer synthesis flow for the computation model. Thus, in this dissertation, general models are used as computation models.

8.2 Initial Mapping

As a portion of initial mapping algorithms for general computation models, this dissertation addresses mapping for pipelined applications–in multimedia applications. It

is crucial here to balance each stage's execution time especially so as to minimize that of the whole system. Previous work assumed each period of the pipelined execution could be flattened to an acyclic directed graph. However, a period in general models can hardly become an acyclic directed graph. Therefore, this dissertation proposes Hierarchy-Aware mapping. This type of mapping also divides large stages/tasks into pieces while being aware of complex hierarchy in the stages/tasks. The case study is performed with JPEG encoder and Canny Edge Detector to compare Hierarchy-Aware mapping to the optimal pipeline-aware mapping that is unaware of hierarchy: the exhaustive hierarchy-unaware mapping. The execution time is decreased on average by 23.3%.

It is also valuable to address the impact of process scheduling. As ways to solve the problem, several previous works have already offered their own algorithms. However, since process scheduling is less predictable in general models, these algorithms are not applicable. This dissertation presents N-Way Clustering and Mapping. The optimization goal is to minimize the execution time of the system under all other design constraints such as cost. NWCM starts a greedy algorithm-based clustering and the closeness function takes as parameters process scheduling as well as communication and computation. Clustering is followed by one-to-one mapping between N clusters and N PEs based on execution delays and speeds of PEs. NWCM is compared to Load Balancing Algorithm, Longest Processing Time algorithm, and Strength Pareto Evolutionary Algorithm. The case study performed with a computation-intensive multimedia application running an MP3 decoder and JPEG encoder at the same time shows that NWCM outperforms the competitors by at least 24.4%

8.3 Iterative Improvement

In Transaction Level Model-based design, iterative improvements based on TLM estimation is crucial. This dissertation proposes Cycle-Approximate Estimation-Based Mapping (CAEBM) to address the problem. With an initial mapping given by any previous work and its cycle-approximate estimate, CAEBM, to meet the design constraints, conducts a local search by using iterative cycle-approximate estimation and heuristics. The optimization goal is to minimize execution time. The case study is performed with a multimedia application running an MP3 decoder and JPEG encoder. According to the case study, when we compare CAEBM to Strength Pareto Evolutionary Algorithm and Load Balancing Algorithm [9], CAEBM improves the design (i.e. execution time) by 36.3%.

8.4 Transaction Level Model Estimation

The speed of Transaction Level Model estimation limits the design space that can be explored during the iterative improvement phase. Although existing simulation-based TLM estimation is fast, cycle-approximate, and general, there is still room for improvement especially regarding speed. This dissertation presents Trace-Driven Performance Estimation (TDPE). TDPE is complementary to simulation-based TLM estimations. For a portion of applications, TDPE provides TLM estimation that is faster by orders of magnitude than simulation-based TLM estimation. And yet TDPE is still as accurate as TLM estimation and as general.

TDPE does not simulate the entire platform model from scratch when there is any change in mapping and/or the platform. Instead, TDPE generates traces once at Transaction Level and places the traces on the global timeline. The alignment phase

takes into account mapping, the data path of each PE, abstract RTOS, bus protocol, and memory hierarchy models. The case study performed with the MP3 decoder and four different platforms with five different configurations shows that TDPE is, without losing accuracy, 49.34 times faster than TLM Estimation when the number of required mapping and platform selection is 100.

Chapter 9

Future Work

In this dissertation, automatic design space exploration focuses on optimization of execution time. Recently, in many embedded system designs, power consumption is no more a secondary issue. Therefore, in the long run, automatic design space exploration algorithms must take into consideration power consumption. In the framework of TLM-based design methodology, early power estimation at Transaction Level is mandatory. The speed of power estimation will also limit the design space that can be explored. What is thus needed is fast power estimation at the Transaction Level.

This dissertation mainly focuses on mapping. However, for the system-level synthesis, platform selection also has to be, as much as possible, automated. Therefore, future work also should include automation in platform selection.

Bibliography

[1] Moore's law. http://www.technewsdaily.com/17450-moores-law.html, May., 27 2014.

[2] Frank Vahid and Tony Givargis. *Embedded system design - a unified hardware / software introduction.* Wiley-VCH, 2002.

[3] K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincetelli. System-level design : Orthogonalization of concerns and platform-based design. *Computer Aided Design of Integrated Circuits, IEEE Transactions on*, 2000.

[4] N.Z. Haron and S. Hamdioui. Why is cmos scaling coming to an end? In *Design and Test Workshop, 2008. IDT 2008. 3rd International*, pages 98–103, Dec 2008.

[5] G. Martin. Overview of the mpsoc design challenge. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 274–279, 2006.

[6] Wayne Wolf. The future of multiprocessor systems-on-chips. In *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, pages 681–685, New York, NY, USA, 2004. ACM.

[7] International technology roadmap for semiconductor, 2011.

[8] Hoeseok Yang and Soonhoi Ha. Pipelined data parallel task mapping/scheduling technique for MPSoC. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 69 –74, april 2009.

[9] Daniel D. Gajski, Samar Abdi, Andreas Gerstlauer, and Gunar Schirner. *Embedded System Design: Modeling, Synthesis and Verification.* Springer Publishing Company, Incorporated, 1st edition, 2009.

[10] Axel Jantsch. Models of embedded computation. In *EMBEDDED SYSTEMS HANDBOOK*. CRC Press, 2005.

[11] John E. Savage. *Models of Computation: Exploring the Power of Computing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1997.

[12] J. M. Paul and D. E. Thomas. *Models of Computation for Systems-on-Chip*. Morgan Kaufman Publishers, Boston, MA, USA, 2004.

[13] E.A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(12):1217–1229, Dec 1998.

[14] Lukai Cai and Daniel Gajski. Transaction level modeling: An overview. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/-Software Codesign and System Synthesis*, CODES+ISSS '03, pages 19–24, New York, NY, USA, 2003. ACM.

[15] Mark Thompson, Hristo Nikolov, Todor Stefanov, Andy D. Pimentel, Cagkan Erbas, Simon Polstra, and Ed F. Deprettere. A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '07, pages 9–14, New York, NY, USA, 2007. ACM.

[16] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: toward composable multimedia mp-soc design. In *Proceedings of the 45th annual Design Automation Conference*, DAC '08, pages 574–579, New York, NY, USA, 2008. ACM.

[17] S. Ha, C. Lee, Y. Yi, S. Kwon, and Y. P. Joo. The peace: Hardware-software codesign of multimedia embedded systems. In *Embedded and Real-Time Computing Systems and Applications, 1982. 12th International Conference on*, pages 207 –214, 2006.

[18] Cagkan Erbas, Selin C. Erbas, and Andy D. Pimentel. A multiobjective optimization model for exploring multiprocessor mappings of process networks. pages 182–187. ACM, 2003.

[19] F. Ferrandi, C. Pilato, D. Sciuto, and A. Tumeo. Mapping and scheduling of parallel c applications with ant colony optimization onto heterogeneous reconfigurable mpsocs. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 799–804, Jan.

[20] T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67, Feb 2002.

[21] Moo-Kyoung Chung, Sangkwon Na, and Chong-Min Kyung. System-level performance analysis of embedded system using behavioral c/c++ model. In *VLSI Design, Automation and Test, 2005. (VLSI-TSA-DAT). 2005 IEEE VLSI-TSA International Symposium on*, pages 188–191, April 2005.

[22] J.T. Russell and M.F. Jacome. Architecture-level performance evaluation of component-based embedded systems. In *Design Automation Conference, 2003. Proceedings*, pages 396–401, June 2003.

[23] R. Plyaskin, A. Masrur, M. Geier, S. Chakraborty, and A. Herkersdorf. High-level timing analysis of concurrent applications on mpsoc platforms using memory-aware trace-driven simulations. In *VLSI System on Chip Conference (VLSI-SoC), 2010 18th IEEE/IFIP*, pages 229–234, Sept 2010.

[24] K. Lahiri, A. Raghunathan, and S. Dey. System-level performance analysis for designing on-chip communication architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(6):768–783, Jun 2001.

[25] Y. Hwang, S. Abdi G. Shirner, and D. D. Gajski. Accurate timed rtos model for transaction level modeling. In *DATE*, 2008.

[26] Y. Hwang, S. Abdi, and D. D. Gajski. Cycle-approximate retargetable performance estimation at the transaction level. In *DATE*, Munich, Germany, April 2008.

[27] L. Gao, K. Karuri, Stefan Kraemer, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Multiprocessor performance estimation using hybrid simulation. In *Proceedings of the Design Automation Conference (DAC '08)*, Anaheim, CA, USA, jun 2008.

[28] Preeti Ranjan Panda. Systemc: A modeling platform supporting multiple design abstractions. In *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS '01, pages 75–80, New York, NY, USA, 2001. ACM.

[29] L. Yu. Lochi, S. Abdi, and D. D. Gajski. *Estimation of Communiaction in SystemC Transaction Level Models*. CECS Technical Report, University of California Irvine, May, 2009.

[30] Rainer Dömer. The specc system-level design language and methodology, part 1. In *Parts1 2, Embedded Systems Conference*, 2001.

[31] S.S. Bhattacharyya, P.K. Murthy, and E.A. Lee. *Software Synthesis from Dataflow Graphs*. Springer, 1996.

[32] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[33] Charles Antony Richard Hoare. *Communicating sequential processes*, volume 178. Prentice-hall Englewood Cliffs, 1985.

[34] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[35] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[36] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill, Inc., New York, NY, USA, 1st edition, 1998.

[37] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara, editors. *Hardware-software Co-design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[38] Rainer Dömer, Andreas Gerstlauer, Junyu Peng, Dongwan Shin, Lukai Cai, Haobo Yu, Samar Abdi, and Daniel D. Gajski. System-on-chip environment: A specc-based framework for heterogeneous mpsoc design. *EURASIP J. Embedded Syst.*, 2008:5:1–5:13, January 2008.

[39] Jing Lin, A. Srivatsa, A. Gerstlauer, and B.L. Evans. Heterogeneous multiprocessor mapping for real-time streaming systems. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 1605–1608, 2011.

[40] A. Benoit, P. Renaud-Goud, and Y. Robert. Performance and energy optimization of concurrent pipelined applications. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, 2010.

[41] A. Benoit, L. Marchal, Y. Robert, and O. Sinnen. Mapping pipelined applications with replication to increase throughput and reliability. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on*, pages 55–62, 2010.

[42] H. Javaid and S. Parameswaran. A design flow for application specific heterogeneous pipelined multiprocessor systems. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 250–253, 2009.

[43] Embedded system environment. http://www.cecs.uci.edu/˜ese, Dec., 23 2011.

[44] D. Densmore and R. Passerone. A platform-based taxonomy for esl design. *Design Test of Computers, IEEE*, 23(5):359–374, May 2006.

[45] Ptolemy ii. http://ptolemy.eecs.berkeley.edu/ptolemyII, May., 26 2014.

[46] S. Ha. Hopes. `http://peace.snu.ac.kr/hopes/korean/index.php`, May 2012.

[47] Seongnam Kwon, Yongjoo Kim, Woo-Chul Jeun, Soonhoi Ha, and Yunheung Paek. A retargetable parallel-programming framework for mpsoc. *ACM Trans. Des. Autom. Electron. Syst.*, 13(3):39:1–39:18, July 2008.

[48] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm. Technical report, 2001.

[49] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *SIGARCH Comput. Archit. News*, 34(5):151–162, October 2006.

[50] A. Bonfietti, L. Benini, M. Lombardi, and M. Milano. An efficient and complete approach for throughput-maximal sdf allocation and scheduling on multi-core platforms. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 897–902, 2010.

[51] Hyunok Oh and Soonhoi Ha. A hardware-software cosynthesis technique based on heterogeneous multiprocessor scheduling, 1999.

[52] M. Girkar and C.D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *Parallel and Distributed Systems, IEEE Transactions on*, 3(2):166–178, 1992.

[53] Lochi Yu. *Automatic Generation and Verification of Transaction Level Modeling.* Ph. D. Dissertation, University of California Irvine, 2009.

[54] S. Abdi and D. Gajski. *A Universal Bus Channel for Transaction Level Modeling.* CECS Technical Report, University of California Irvine, 2006.

[55] K. Kim. *Embedded System Environment : Overview.* CECS Technical Report, University of California Irvine, December 2012.

[56] K. Kim. *TLM Generation with ESE.* CECS Technical Report, University of California Irvine, July 2010.

[57] H. Yu A. Gerstlauer and D. D. Gajski. Rtos modeling for system-level design. In *DATE*, Munich, Germany, March 2003.

[58] Seng Lin Shee, A. Erdos, and S. Parameswaran. Heterogeneous multiprocessor implementations for jpeg:: a case study. In *Hardware/Software Codesign and System Synthesis, 2006. CODES+ISSS '06. Proceedings of the 4th International Conference*, pages 217–222, 2006.

[59] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-8(6):679 –698, nov. 1986.

[60] Xu Han, Yasman Samei, and Rainer Dömer. System-level modeling and refinement of a canny edge detector. Technical Report # 12-14, Center for Embedded Computer Systems, November 2012.

[61] S. Abdi, Yonghyun Hwang, Lochi Yu, Hansu Cho, I. Viskic, and D.D. Gajski. Embedded system environment: A framework for tlm-based design and prototyping. In *Rapid System Prototyping (RSP), 2010 21st IEEE International Symposium on*, pages 1 –7, june 2010.

[62] K. Gruttner and W. Nebel. Modeling program-state machines in SystemC^{TM}. In *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*, pages 7–12, 2008.

[63] Xu Han and Gunar Schirner. Real-time mp3 decoding on fpga: a case study of system model features. Technical report, Center for Embedded Computer Systems, July 2009.

[64] Samar Abdi Yongjin Ahn. Process network modeling and tlm generation for h.264 codec design. Technical report, Center for Embedded Computer Systems, October 2008.

[65] Zhengting He and A. Mok. Fast cosimulation of transformative systems with os support on smp computer. In *Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004. International Conference on*, pages 164–169, 2004.

[66] Frank Vahid and Daniel D. Gajski. Clustering for improved system-level functional partitioning. In *in International Symposium on System Synthesis*, pages 28–33, 1995.