Improving Scalability of Chip-MultiProcessors with Many HW ACCelerators

A Dissertation Presented
by
Nasibeh Teimouri
to
The Department of Electrical and Computer Engineering
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Northeastern University
Boston, Massachusetts

November 2017
Dissertation Title: Improving Scalability of Chip-MultiProcessors with Many HW ACCelerators.

Author: Nasibeh Teimouri.

Department: Electrical and Computer Engineering.

Approved for Dissertation Requirements of the Doctor of Philosophy Degree

Dissertation Advisor: Dr. Gunar Schirner

Dissertation Reader: Dr. David Kaeli

Dissertation Reader: Dr. Yunsi Fei

Dissertation Reader: Dr. Hamed Tabkhi

Department Chair: Dr. Miriam Leeser

Director of the Graduate School:

Dean: Dr. Thomas C. Sheahan
To my devoted parents, my beloved husband, and my adorable siblings.
# Contents

List of Figures vi
List of Tables ix
List of Acronyms x
Acknowledgments xii
Abstract xiii

1 Introduction 1

1.1 Chip Multi Processor .............................................. 4
1.2 ACCelerator-based CMP (ACMP) .................................. 5
  1.2.1 Trend Toward Integration of Many ACCs on a Chip ........ 7
  1.2.2 Processor-Centric Architectures for Integration of ACCs on Chips .... 9
  1.2.3 Scalability Limitations of Processor-Centric ACMP Architectures ...... 11
1.3 Problem Definition ............................................. 12
1.4 Dissertation Overview .......................................... 13
1.5 Contributions .................................................. 16
1.6 Related Work .................................................. 18
  1.6.1 Processor-centric ACMPs with Slave ACCs ................. 18
  1.6.2 Peer View between ACCs and Processor ................ 20

2 Background 21

2.1 Single ACCelerator Design ....................................... 21
  2.1.1 Application Specific Integrated Circuit (ASIC) ............ 24
  2.1.2 Field Programmable Gate Array (FPGA) .................... 24
  2.1.3 Application Specific Instruction-based Processors (ASIP) ........ 24
  2.1.4 Graphics Processing Unit (GPU) ........................... 25
  2.1.5 Function Level Processor (FLP) ............................ 25
  2.1.6 Coarse Grain Reconfigurable Array (CGRA) ................ 26
  2.1.7 Hybrid Designs ........................................... 27
2.2 ACC Integration .............................................. 27
  2.2.1 Architecture Support to Reduce the ACCs’ Orchestration Load .... 29
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.2 Architecture Support to Relieve the ACCs’ Data Storage Load</td>
<td>31</td>
</tr>
<tr>
<td>2.2.3 Architecture Support to Relieve the ACCs’ Data Transfer Load</td>
<td>32</td>
</tr>
<tr>
<td>2.3 Streaming Applications</td>
<td>34</td>
</tr>
<tr>
<td>3 Scalability Implication on ACMP Performance</td>
<td>36</td>
</tr>
<tr>
<td>3.1 Resources Bottlenecks: Analytical Investigation</td>
<td>36</td>
</tr>
<tr>
<td>3.1.1 Origins of the Resources Bottlenecks</td>
<td>36</td>
</tr>
<tr>
<td>3.1.2 Resources Utilization: Analytical Modeling</td>
<td>40</td>
</tr>
<tr>
<td>3.1.3 Resources Utilization: Analytical Results</td>
<td>41</td>
</tr>
<tr>
<td>3.2 ACMPPerf: Analytical Performance Model of ACMP</td>
<td>43</td>
</tr>
<tr>
<td>3.2.1 Comp Model</td>
<td>46</td>
</tr>
<tr>
<td>3.2.2 Comp+Comm Model</td>
<td>47</td>
</tr>
<tr>
<td>3.2.3 Comp+Comm+Orch Model</td>
<td>47</td>
</tr>
<tr>
<td>3.3 ACMP Architecture: Analytical Results</td>
<td>48</td>
</tr>
<tr>
<td>3.4 Summary</td>
<td>51</td>
</tr>
<tr>
<td>4 ACC Communication Semantics</td>
<td>52</td>
</tr>
<tr>
<td>4.1 ACC Communication Aspects</td>
<td>52</td>
</tr>
<tr>
<td>4.1.1 Single ACC</td>
<td>57</td>
</tr>
<tr>
<td>4.1.2 Communicating ACCs</td>
<td>57</td>
</tr>
<tr>
<td>4.2 ACC Communication Semantics</td>
<td>59</td>
</tr>
<tr>
<td>4.3 Summary</td>
<td>60</td>
</tr>
<tr>
<td>5 Transparent Self-Synchronizing (TSS) ACCs</td>
<td>61</td>
</tr>
<tr>
<td>5.1 Single ACC in TSS</td>
<td>62</td>
</tr>
<tr>
<td>5.2 Communicating ACCs in TSS</td>
<td>63</td>
</tr>
<tr>
<td>5.3 Peer Processor Communication</td>
<td>65</td>
</tr>
<tr>
<td>5.4 TSS System Integration</td>
<td>67</td>
</tr>
<tr>
<td>5.5 TSS: Analytical Evaluation</td>
<td>68</td>
</tr>
<tr>
<td>5.6 Summary</td>
<td>70</td>
</tr>
<tr>
<td>6 Experimental Results: TSS vs. ACMP</td>
<td>71</td>
</tr>
<tr>
<td>6.1 Experimental Setup</td>
<td>71</td>
</tr>
<tr>
<td>6.2 Constant Computation Coverage</td>
<td>72</td>
</tr>
<tr>
<td>6.2.1 Performance Evaluation</td>
<td>73</td>
</tr>
<tr>
<td>6.2.2 Energy Consumption</td>
<td>75</td>
</tr>
<tr>
<td>6.2.3 Area Overhead</td>
<td>76</td>
</tr>
<tr>
<td>6.3 Increasing Computation Coverage by ACCs</td>
<td>77</td>
</tr>
<tr>
<td>6.3.1 Performance Evaluation</td>
<td>78</td>
</tr>
<tr>
<td>6.3.2 Energy Consumption</td>
<td>79</td>
</tr>
<tr>
<td>6.3.3 Relative Overhead</td>
<td>81</td>
</tr>
<tr>
<td>6.4 Relative and Absolute Accuracy Investigation of ACMPPerf</td>
<td>82</td>
</tr>
<tr>
<td>6.5 Increasing Communication Coverage by ACCs</td>
<td>83</td>
</tr>
<tr>
<td>6.6 Summary</td>
<td>85</td>
</tr>
</tbody>
</table>
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The trends of transistor density, power, clock frequency and number of cores over the last decades [48].</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Block diagrams of two homogenous CMPs.</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>The block diagram of the OMAP5430 [110].</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>Convolution Neural Network (CNN) with different convolution/computation layers [79].</td>
<td>7</td>
</tr>
<tr>
<td>1.5</td>
<td>Examples of ACC-based implementations for running simple/moderate tasks.</td>
<td>7</td>
</tr>
<tr>
<td>1.6</td>
<td>Apple SoC’s board toward more customization [5].</td>
<td>8</td>
</tr>
<tr>
<td>1.7</td>
<td>Heterogeneous ACC-based CMP (ACMP).</td>
<td>10</td>
</tr>
<tr>
<td>1.8</td>
<td>Events sequence of one ACC transaction.</td>
<td>10</td>
</tr>
<tr>
<td>1.9</td>
<td>Processor-centric ACMP with increasing number of ACCs and computation load covered by the ACCs.</td>
<td>12</td>
</tr>
<tr>
<td>1.10</td>
<td>The dissertation overview: problem analysis to solution and evaluation.</td>
<td>14</td>
</tr>
<tr>
<td>1.11</td>
<td>Direct ACC-to-ACC connection realization in conventional ACMP platforms.</td>
<td>15</td>
</tr>
<tr>
<td>2.1</td>
<td>Trade-off between flexibility and energy efficiency.</td>
<td>22</td>
</tr>
<tr>
<td>2.2</td>
<td>Landscape of processing in view of flexibility, and energy efficiency.</td>
<td>23</td>
</tr>
<tr>
<td>2.3</td>
<td>Processor pipeline with DySER Datapath and DySER elements [49].</td>
<td>26</td>
</tr>
<tr>
<td>2.4</td>
<td>Models of coupling ACC with a general purpose processor on a chip [19].</td>
<td>29</td>
</tr>
<tr>
<td>2.5</td>
<td>ACCelerator Rich CMP(ARC).</td>
<td>30</td>
</tr>
<tr>
<td>2.6</td>
<td>CHARM Architecture [14].</td>
<td>30</td>
</tr>
<tr>
<td>2.7</td>
<td>Accelerator store [64].</td>
<td>32</td>
</tr>
<tr>
<td>2.8</td>
<td>Sharing SRAM in SoCs [30].</td>
<td>32</td>
</tr>
<tr>
<td>2.9</td>
<td>GPU Direct on the same PCIe bus [52].</td>
<td>33</td>
</tr>
<tr>
<td>2.10</td>
<td>Examples of streaming applications.</td>
<td>35</td>
</tr>
<tr>
<td>2.11</td>
<td>DF model of MP3 decoder; circles represent the computation kernel and solid lines represent the streaming data.</td>
<td>35</td>
</tr>
<tr>
<td>3.1</td>
<td>Ideal and real time-line execution of ACMP with 3 ACCs (chained in producer-consumer fashion).</td>
<td>37</td>
</tr>
<tr>
<td>3.2</td>
<td>ACCs’ loads on the communication fabric/DMA and memory with fixed job size.</td>
<td>38</td>
</tr>
<tr>
<td>3.3</td>
<td>Injection of input problem as input jobs.</td>
<td>38</td>
</tr>
<tr>
<td>3.4</td>
<td>Limited memory and smaller job size.</td>
<td>39</td>
</tr>
</tbody>
</table>
3.5 ISR exposed to the processor ........................................... 39
3.6 Processor utilization and communication BW with increasing number of ACCs, ........................................... 42
3.7 ACC utilization with increasing number of ACCs for different constant amount of resources, ........................................... 43
3.8 Arch. model, App. model, and Map. functions ........................................... 44
3.9 ACMP architecture-aware DF ........................................... 45
3.10 Synthetic application captured in DF ........................................... 48
3.11 Comp Model ........................................... 49
3.12 Comp+Comm Model ........................................... 49
3.13 Comp+Comm+Orch Model ........................................... 50
4.1 Sequence of data access and synchronization in single buffer ........................................... 53
4.2 Sequence of data access and synchronization in double buffer ........................................... 54
4.3 Sequence of data access and synchronization in triple buffer ........................................... 55
4.4 Sequence of data access and synchronization in a FIFO with the length of n ........................................... 56
4.5 Single ACC with the extended FIFO semantics ........................................... 57
4.6 Communicating ACCs with the extended FIFO semantics ........................................... 58
4.7 Granularity management ........................................... 58
5.1 Single ACC communication in TSS ........................................... 62
5.2 ACC-to-ACC communication in TSS ........................................... 62
5.3 Nodes with multiple input/output ports ........................................... 63
5.4 Gateway Architecture ........................................... 65
5.5 Multiple Chains in TSS to Accelerate the Execution of the Mapping-aware Synthetic DF ........................................... 66
5.6 TSS system integration ........................................... 68
5.7 Comp+Comm+Orch Model ........................................... 68
5.8 TSS relative improvement over ACMP ........................................... 69
6.1 Virtual platform generation ........................................... 72
6.2 System Performance and Relative Speedup ........................................... 74
6.3 Demand on the shared resources ........................................... 75
6.4 Energy improvement ........................................... 76
6.5 Relative throughput of TSS over ACMPs WRT. computation coverage using VPs ........................................... 78
6.6 Energy consumption reduction of TSS over conventional ACMP WRT. computation coverage ........................................... 79
6.7 Relative Energy Delay Product (EDP) in TSS vs. ACMP WRT. computation coverage ........................................... 80
6.8 Relative overhead of TSS over ACMPs WRT. computation coverage ........................................... 81
6.9 Relative throughput of TSS over ACMPs WRT. computation coverage using ACM-Perf and VP ........................................... 83
6.10 Relative throughput of TSS over ACMPs WRT. communication coverage using ACMPerf ........................................... 84
6.11 Relative overhead of TSS over ACMPs WRT. communication coverage ........................................... 85
7.1 Speed accuracy trade-off ........................................... 89
7.2 NoC modeling granularity. .................................................. 90
7.3 The router architecture in the proposed NoC abstract models. .... 92
7.4 Performance validation of the BFM (8x8 mesh, 40%-hot spot). .... 96
7.5 Comparing simulation time of the BFM and the TLM (8x8 mesh, 40%-hot spot). 98
7.6 Cumulative probability of accuracy error. ............................. 99
## List of Tables

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Performance Contributors in Incremental Models</td>
</tr>
<tr>
<td>4.1</td>
<td>Access rights in single buffering approach</td>
</tr>
<tr>
<td>4.2</td>
<td>Access right in double buffering approach</td>
</tr>
<tr>
<td>4.3</td>
<td>Access rights in triple buffering approach</td>
</tr>
<tr>
<td>6.1</td>
<td>Applications characteristics (used in experiments with constant computation coverage by ACCs)</td>
</tr>
<tr>
<td>6.2</td>
<td>Applications characteristics (used in experiments with increasing computation coverage by ACCs)</td>
</tr>
<tr>
<td>7.1</td>
<td>NoC abstraction overview</td>
</tr>
<tr>
<td>7.2</td>
<td>Contention events and arbitration points</td>
</tr>
<tr>
<td>7.3</td>
<td>TLM vs. BFM</td>
</tr>
</tbody>
</table>
List of Acronyms

ACC  Hardware ACCELERator.
Dedicated hardware core to execute a function in a more power and performance efficient way than general purpose processors.

ACMP  ACC-based Chip Multi-Processor.
A CMP that integrates specialized HW ACCs on to the chip.

AHB  Advanced High-performance Bus.
System bus definition within the AMBA 2.0 specification defining a high-performance bus including pipe-lined access, bursts, split and retry operations.

BFM  Bus Functional Modeling.
A non-synthesizable software model of an integrated circuit elements with one or more external buses.

CMP  Chip Multi-Processor.
The integration of multiple cores on a chip as well as off-chip memory and a hierarchy of on-chip memory, all connected through an on-chip interconnect.

DFM  Data Flow Model.
A programming language that models the application as a directed graph of operations with stream of data flowing across the operations.

DMA  Direct Memory Access.
A feature of computer systems that allows certain hardware subsystems to access the main memory independently of the processor core.

GPP  General Purpose Processor. A microprocessor optimized to execute a wide range of applications.

ICM/OCM  Input/Output Control Management.
Management units added per ACC’s input/output ports inside the TSS to realizes the last three aspects of ACC communication semantic including (1) data granularity management, (2) data marshalling, and (3) synchronization.
**SPM** Scratch Pad Memory.
Local fast memory per ACC, addressable by programmers and part of on-chip memory.

**ML-AHB** Multi Layers of AHBs.
Parallel set of AHBs to provide higher throughput and bandwidth.

**NoC** Network on Chip.
Parallel communication interconnect on a chip to connect cores in a System on Chip (SoC).

**SCE** System on Chip Environment.
A tool released by the Embedded Systems Methodology group in UCI as ‘an implementation of the concept of the exploration engine as a step in their overall SoC design flow research’.

**SDL** System Description Language.

**SLDL** System-level Design Language.

**SoC** System on Chip.
An integrated circuit (chip) that integrates all components of a computer and other electronic systems.

**SpecC** Specification C.
An SDL/SLDL as an extension of the ANSI C programming language to aid the design and specification of digital embedded systems.

**TLM** Transaction Level Modeling.
A high-level approach to model digital systems with implementation separation of the communication details from the functional units or the communication architecture.

**TSS** Transparent Self-Synchronizing ACCs.
An architecture template to efficiently realize the communication semantic aspects of ACC-to-ACC connections with the goal of reliving the load on the shared architectural resources.
Acknowledgments

First, I would like to thank my adviser, Dr. Gunar Schirner, and then Dr. Hamed Tabkhi for guiding me during the last four years. Dr. Schirner and Dr. Tabkhi have been excellent sources of embedded systems knowledge throughout my graduate study. It is also my pleasure to thank Dr. David Kaeli for all his nice technical and nontechnical supports and Dr. Yunsi Fei for her guidance.

More personal, I would like to thank my spouse, Mohammad, who has been always warming my heart over years. Then, I feel very grateful to my parents, and my siblings for their unconditional loves and supports when I was away from them studying.

This material is based upon work partially supported by the National Science Foundation under Grant No. 1319501.
Abstract

Improving Scalability of Chip-MultiProcessors with Many HW ACCelerators

by

Nasibeh Teimouri

Doctor of Philosophy in Computer Engineering
Northeastern University, November 2017

Dr. Gunar Schirner, Adviser

Breakthrough streaming applications such as virtual reality, augmented reality, autonomous vehicles, and multimedia demand for high-performance and power-efficient computing. In response to this ever-increasing demand, manufactures look beyond the parallelism available in Chip Multi-Processors (CMPs), and more toward application-specific designs. In this regard, ACCelerator (ACC)-based heterogeneous CMPs (ACMs) have emerged as a promising platform.

An ACM combines application-specific HW ACCelerators (ACCs) with General Purpose Processor(s) (GPP) onto a single chip. ACCs are customized to provide high-performance and power-efficient computing for specific compute-intensive functions and GPP(s) runs the remaining functions and controls the whole system. In ACM platforms, ACCs achieve performance and power benefits at the expense of reduced flexibility and generality for running different workloads. Therefore, manufactures must utilize several ACCs to target a diverse set of workloads within a given application domain.

However, our observation shows that conventional ACM architectures with many ACCs have scalability limitations. The ACCs benefits in processing power can be overshadowed by bottlenecks on shared resources of processor core(s), communication fabric/DMA, and on-chip memory. The primary source of the resources bottlenecks stems from ACCs data access and orchestration load. Due to very loosely defined semantics for communication with ACCs, and relying upon general platform architectures, the resources bottlenecks hamper performance.

This dissertation explores and alleviates the scalability limitations of ACMs. To this end, the dissertation first proposes an analytical model to holistically explore how bottlenecks emerge on shared resources with increasing number of ACCs. Afterward, it proposes ACMPerf, an analytical model to capture the impact of the resources bottlenecks on the achievable ACCs’ benefits.
Then, to open a path toward more scalable integration of ACCs, the dissertation identifies and formalizes ACC communication semantics. The semantics describe four primary aspects: data access, synchronization, data granularity, and data marshalling.

Considering our identified ACC communication semantics, and improving upon conventional ACMP architectures, the dissertation proposes a novel architecture of Transparent Self-Synchronizing ACCs (TSS). TSS efficiently realizes our identified communication semantics of direct ACC-to-ACC connections often occurring in streaming applications. The proposed TSS adds autonomy to ACCs to locally handle the semantic aspects of data granularity, data marshalling and synchronization. It also exploits a local interconnect among ACCs to tackle the semantics aspect of data access. As TSS gives autonomy to ACCs to self-synchronize and self-orchestrate each other independent of the processor, thereby enabling finest data granularity to reduce the pressure on the shared memory. TSS also exploits a local and reconfigurable interconnect for direct data transfer among ACCs without occupying DMA and communication fabric.

As a result of reducing the overhead of direct ACC-to-ACC connections, TSS delivers more of the ACCs’ benefits than that of conventional ACMP architectures: up to 130x higher throughput and 209x lower energy, all as results of up to 78x reduction in the imposed load to the shared resources.
Chapter 1

Introduction

Rapid technology scaling has enabled to increase CMOS transistors integration density on a chip (doubled every 18 to 24 months) as Moore’s law\cite{67} states. With Dennard observation \cite{23}, as transistor shrinks, its voltage and capacitance scale down and operating frequency increases accordingly. Therefore, the power density stays the same from each technology node to a subsequent one.

In Dennard scaling era (from the beginning of silicon integration up to the mid-2000s), increasing frequency enabled by technology scaling had the major influence on the performance of single processors. Moreover, architectural evolution through exploiting Instruction-Level Parallelism (ILP) and Simultaneous Multi-Threading (SMT) have helped to boost performance of a single core processor. ILP refers to groups of independent instructions that can be executed in the same clock cycle, and SMT is a processor design for multiplexing resources among parallel threads within a single cycle to increase the processor utilization. However, programs often have limited amount of inherent parallelism, and with more complex cores to support parallelism, area increases quadratically, so does the delay and power consumption \cite{25, 114, 71, 28, 95, 82, 76, 95, 6, 46}.

In post Dennard scaling era (2004-current), power stopped scaling. Consequently, the operating frequency of processors stopped to increase due to increasing power consumption and thermal issues \cite{113}. To keep increasing the performance while keeping the same power envelope as before, the paradigm shifted toward multiple processor cores on the same chip, which is known as Chip Multi Processor (CMP). A CMP supports parallel execution of applications on multiple cores to improve the throughput (as an indicator of performance) without hitting the power wall \cite{71}.

Figure 1.1 illustrates the trends in transistor density, power, frequency and number of cores in both Dennard scaling and post Dennard scaling eras. In post Dennard scaling era, the frequency
CHAPTER 1. INTRODUCTION

Figure 1.1: The trends of transistor density, power, clock frequency and number of cores over the last decades [48].

...and power consumption of processors stay the same, while the number of cores starts to increase.

With more number of cores integrated on a chip, power scaling challenge stays as a serious constraint, and only a portion of cores can be actively working. The implication of power constraint has led to new challenges for computer architects and system designers. The International Technology Road map for Semiconductors (ITRS) [107] estimates that 75%-88% of a chip in an 8-nm technology should be un-powered, that is called dark silicon phenomenon or utilization wall. Dark silicon is the direct impact of stopping power scaling that was used to take for granted for several technology generations.

To cope with dark silicon effects while improving the performance, the main focus has been beyond parallelism and shifted toward application-specific ACCelerators (ACCs) [40]. An ACC is a design with reduced generality and more customization to provide power and performance efficient computing for a specific (or set of) workload(s). As an ACC is customized to only be switched on to provide highly power-performance optimized computing for the specific (or set of) workload(s) for which it was designed for, the impact of dark silicon can be relieved.

Customization has been widely used in embedded systems, and dark silicon pushes computer architects to exploit customization in general purpose computing as well. Several ACCs are designed to improve power and performance by orders of magnitude as compared to general purpose processors. For example, neuro vector engine ACC is efficiently designed for Convolution Neural Networks (CNN) computation [79], Sonic Millip3De [85] is designed to accelerates audio processing, and database processing units (DPUs) harnesses a several number of ACCS to boost...
CHAPTER 1. INTRODUCTION

database operations.

The power and performance efficiency of single ACCs, and effectively pushing customization into general purpose computation are two main reasons of integrating many ACCs on a chip to keep up with the emergence of new applications with increasing demand for both power and performance efficient computation on massive amount of data [91] [58] [17] [15] [50]. [15] [17] are examples of accelerator rich architectures which integrate many number of ACCs to accelerate biomedical applications. [40] proposes to integrate many ACCs to accelerate the electronic design automation (EDA) algorithms. [7] has integrated more than 20 ACCs to accelerate vision processing. Apple [5] as a more tangible example in human lives has increased the number of ACCs up to 4x from its A4 generation to its A8 generation.

However, integration of many ACCs on a chip using conventional architectures poses scalability challenges and diminishes benefits of ACCs. Conventional architectures treat ACCs as slaves devices requiring the shared resources of communication fabric, DMA, memory, and especially processor core(s). This processor-centric view limits the ACC benefits in boosting performance and energy efficiency.

The focus of this dissertation is on investigation and alleviation of scalability limitations of conventional processor-centric ACC-based CMP (ACMP) architectures. An ACMP is a CMP that integrates multiple ACCs. We investigate the sources of scalability limitations, then propose and evaluate an ACC-centric architecture to address the inefficiencies of conventional processor-centric architectures. In contrary to conventional ACMPs, our architecture realizes a system architecture in which ACCs are not seen as slaves devices, rather they provide a good level of autonomy with flexible and configurable connections.

The rest of this chapter is organized as follows. First, we discuss Chip Multi Processor (CMP) as the underlying architecture to integrate ACCs. Then, we introduce ACCelerator-based CMP (ACMP) as a new design for energy-efficiency in general purpose computing. We describe inefficiencies associated with conventional integration of ACCs and highlight the scalability limitations of conventional processor-centric architectures. Next, we define the problems, and then present the overview and contributions of the dissertation. Finally, we review closely related work and describe the differences of our approach as compared to them.
CHAPTER 1. INTRODUCTION

1.1 Chip Multi Processor

Limited inherent parallelism in applications, design complexity of an energy efficient single core, and power constraint through technology nodes, all pushed processor vendors to exploit the additional transistors for multiple processor cores on the same chip, which is known as Chip Multi Processor (CMP). A CMP design integrates multiple independent cores into one processor chip to execute multiple instructions (within an application or different applications), and enhance the system throughput while keeping the same power envelope as before.

At the beginning, homogenous CMPs such as Intel’s Core 2 Duo T7600 with two processors on chip, Intel’s Core 2 Quad Q6700 with two dies each one with two processors, and Intel’s Core 2 Extreme X7900 have been adopted by the industry successfully and quickly [1]. With a homogenous CMP that consists of several numbers of cores of the same architectural characteristics such as issue width, cache size, and clock frequency, manufacturing process cost as well as system verification cost are amortized over the number of cores.

Figure 1.2 shows the block diagrams of two homogenous CMPs. Figure 1.2a shows the block diagram of the RAW MIT processor as a set of 16 programmable identical and small tiles that allow traveling across the tile in one clock cycle. The tiles are connected together via a four 32-bits full duplex networks. Each core includes eight stages, single issue and in-order MIPS style processor supplied with caches and floating point units [105]. Similarly, Figure 1.2b shows the block diagram of the IBM powerCell as it has a homogenous set of Synergistic Processing Elements (SPEs) combined with a Power Processing Element (PPE) to control the SPEs [55]. Each SPE has a Direct Memory Access (DMA) unit for input/output data transfer.

Homogeneous CMPs consist of one-size-fits-all cores to provide enough resources to execute a wide range of workloads. Providing more than enough resources in homogenous CMPs results in excessive energy consumption in various workloads. Therefore, heterogeneous CMPs are proposed as a key enabler to boost energy-efficiency [12, 101, 102, 92, 17]. For instance, in big.LITTLE architecture [51], high-performance cores are used for workloads that performance is required, while low-power cores are utilized whenever energy saving is important.

A heterogeneous CMP is composed of different types of cores (in view of micro-architectural details) with different set of resources, and some resources shared among the cores to avoid area constraint. The cores are designed with different capabilities to satisfy the performance demand for specific range of workloads, and otherwise powering off or working at lower frequency to reduce power dissipation. Thus, heterogeneous CMPs offer better area and power efficient coverage for its
CHAPTER 1. INTRODUCTION

(a) RAW MIT processor [105]  (b) IBM PowerCell [55]

Figure 1.2: Block diagrams of two homogenous CMPs.

target workloads as compared to homogenous CMPs [95, 82].

Intel QuickIA [12], in which two Xeon cores are integrated with two Atom cores, ARM big.LITTLE architecture that integrates energy-efficient cores with high-performance cores and widely used in mobile platforms are a few examples of heterogeneous CMPs.

Despite the fact that heterogeneous CMPs improve power efficiency, the power constraint is still a serious challenge. Dark silicon or utilization wall prevents to harness all the potential of on-chip transistors [28]. The threatening dark silicon forces several number of cores to be underutilized at any given time that will diminish both performance and throughput. To fight against the dark silicon through technology generations and keep enhancing both performance and throughput, the recent focus has moved beyond parallelism and shifted toward application-specific designs.

Application-specific design is a design with reduced generality and more customization for one/set of specific workload(s), and delivers orders of magnitude performance and power benefits as compared to general-purpose solutions [92].

1.2 ACCelerator-based CMP (ACMP)

Breakthrough streaming applications such as virtual reality, augmented reality, autonomous vehicles, and multimedia that demand for high-performance and power-efficient computing necessitate the use of Application specific ACCelerators (ACCs). ACCs have come into the scene to provide orders of magnitude enhancement in both power and performance efficient computing over massive amount of data as compared to general purpose designs [123].

Although application specific ACCs significantly improve performance/power efficacy, they reduce re-usability across different applications and workloads domains. In addition, with
application specific designs, the overall design time and cost increase in contrary to general-purpose designs that amortize design time and cost over many applications and workloads domains. Therefore, a combination of both general purpose and application specific designs is preferable in view of power, performance, and flexibility to execute different applications and workloads domains as well as design time/cost [122].

The integration of application specific HW ACCs and heterogeneous CMP realizes ACCELERATOR-BASED CMP (ACMP). ACMPs can improve energy efficiency by 10-100X over fully general-purpose designs [15]. Example of ACMP architectures include the Intels Atom E6x5C processor that uses multiple RISC cores along with a Fine Grained Programmable Array (FPGA) for customization [1] and TI OMAP5430 which is used in mobile processors [100]. Figure 1.3 shows the block diagram of OMAP5430 [110] as a heterogeneous set of different ACCs for general computation, audio/video processing, digital signal processing, and 2D/3D graphic processing.

1 Atom and FPGA fabric are not on a same die but on a single package
CHAPTER 1. INTRODUCTION

Figure 1.4: Convolution Neural Network (CNN) with different convolution/computation layers [79].

Figure 1.5: Examples of ACC-based implementations for running simple/moderate tasks.

1.2.1 Trend Toward Integration of Many ACCs on a Chip

The rapid growth in data volumes, and accordingly ever-increasing computing needs can no longer be satisfied using integrating more general purpose computing cores on a chip without hitting the power wall. As a consequence, designers are developing heterogeneous systems with more hardware ACCs to provide power and performance efficient computing and effectively overcome the dark silicon problem.

ACCs vary in flexibility, efficiency, and types. Some of ACCs are only tailored to execute a particular application efficiently such as convolutional neural network ACC (Figure 1.4) [79, 124, 119], or H.264 video codec (encoder/decoder) ACCs [61, 42, 10]. Despite being high performance and energy efficient, the use of application-specific ACCs are limited to one application.

To increase the generality and flexibility, some research work such as Conservation cores (Figure 1.5a) [116] and Quasi-specific Cores (Figure 1.5b) [117] have proposed general-purpose ACCs. Reducing the customization to improve generality reduces the efficiency as compared to application-specific ACCs. Domain-specific ACCs are the middle ground solution where they try
CHAPTER 1. INTRODUCTION

Figure 1.6: Apple SoC’s board toward more customization [5].

to have the broad applicability of general-purpose ACCs and high efficiency of application-specific ACCs.

Domain-specific designs provide the capability of adapting the architectures to set of applications to achieve orders of magnitude improvement in power and performance efficiency over general purpose designs [99, 15]. For instance Q100 [121] is an example of domain-specific accelerators. Q100 is composed of many heterogeneous ASIC tiles that is called Database Processing Units (DPUs), and is able to efficiently handle database applications. Q100 processes streams of database queries with utilizing direct communication between producer and consumer kernels.

Function Level Processor (FLP) [99] is another example of domain-specific ACCs that targets streaming applications. FLP boosts performance and power efficiency by customizing the frequently used functions in a target domain and allows data path optimization per function block. Similarly, Memory Accelerated Library (MEALib) [41] is another domain-specific ACC design that performs at the granularity of functions. MEALib aims to improve efficiency of high performance software libraries with well-defined application programming interfaces (APIs). The limited configurability of APIs simplifies the design of ACCs significantly, while guarantees the wide applicability of ACCs for high performance computation in the domain.

The natural evolution of this trend will be integration of many diverse and customized ACCs in future designs [91, 57], as ITRS predicts it will reach to hundreds and thousands ACCs by 2022 [106]. There is also a significant body of research at different fields of study that call for increasing number of ACCs on chips [91, 58, 17, 15, 50, 40, 2, 57, 121, 116, 117, 79, 124, 119, 42, 103, 10, 99].

We pick Apple SoC die as a more tangible example, analysis of its die photos from Apples A6 (iPhone 5), A7 (iPhone 5s), and A8 (iPhone 6) systems on chips (SoCs) shown in Figure 1.6a.
implies that more than half of the die area is dedicated to specialized IP blocks. We also observed a consistent trend of an increasing number of specialized IP blocks, 4x from A4 to A8, across generations of Apple's SoCs as Figure 1.6b shows. Other examples of using many numbers of ACCs is Myriad 2 vision processing unit (VPU) that integrates more than 20 ACCs to accelerate vision processing.

However, current architectures have processor-centric view as they were built upon the assumption of sparse integration of ACCs. With processor-centric view, ACCs are observed as slaves devices requiring mostly processor cores and other shared resources such as communication fabric/DMA and memory for every single transaction. The processor-centric architectures can be sufficient with sparse integration of ACCs, but suffer from scalability limitations for dense integration of ACCs.

First, we discuss the integration of ACCs in processor-centric ACMP architectures. We describe the sequence of events for each ACC transaction and highlight shared resources involvement. Next, we show the scalability limitations of processor-centric ACMP architectures when the number of ACCs increases. Then, we define the problems and give a high level overview of our contributions to address the problems.

1.2.2 Processor-Centric Architectures for Integration of ACCs on Chips

Figure 1.7 outlines a processor centric ACMP architecture that integrates ACCs with a host processor core(s) on a chip. In this architecture, ACCs are customized to provide energy efficient computing for specific compute-intensive functions/workloads, and the processor core(s) runs the remaining functions and controls the whole system.

This architecture integrates a private memory for each ACC that is called Scratch Pad Memory (SPM), Shared memory, a multi-layer communication fabric, and multiple Direct Memory Access (DMA) channels. The SPM contains local data as well as intermediate data in each one of ACCs, and the shared memory holds the data shared across the ACCs and the processor core(s). It is the responsibility of the DMA(s) to transfer data to/from the shared memory from/to the ACCs through the communication fabric. The host processor(s) is responsible for system orchestration that includes synchronization and control of ACCs and the DMA(s) through the control bus. The processor core(s) receives the orchestration requests for any ACC transaction via the interrupt line.

Figure 1.8 illustrates 9 steps of a single ACC transaction and the resources involvement:

1) The processor initializes the DMA for data transfer. (2) The DMA transfers data from the shared
CHAPTER 1. INTRODUCTION

Figure 1.7: Heterogeneous ACC-based CMP (ACMP).

memory to the ACC’s SPM through the communication fabric. (3) The DMA notifies the processor (signal *Mem copy done*) that data transfer is done. (4) The processor initializes the ACC for data processing. (5) The ACC starts processing on the input data. (6) The ACC notifies the host processor (signal *Processing done*) upon processing completion. (7) The processor configures the DMA for data transfer. (8) The DMA begins transferring the processed data from ACC’s SPM to the shared memory. (9) The DMA notifies the host processor about data transfer completion. Having all shared resources of processor, communication fabric, DMA, and memory involved for an ACC transaction can create bottlenecks as the number of ACCs increases.

Figure 1.8: Events sequence of one ACC transaction.

Moreover, streaming data is too large to fit in the ACC’s SPM and be processed in one transaction. Therefore, the input data is split into smaller pieces, called *job*, and its size is determined by the ACC’s SPM size. Total memory is fixed due to area and power constraints. Thus, with more ACCs, the job size reduces to fit the SPMs resulting even more transactions. This compounds the pressure on shared resources.
CHAPTER 1. INTRODUCTION

1.2.3 Scalability Limitations of Processor-Centric ACMP Architectures

Due to shared resources (processor core(s), communication bandwidth and memory) involvement for every single ACC transaction in conventional processor-centric architectures, integration of many ACCs results in increasing load on shared resources. With fixed shared resources, there will be significant arbitration overhead for shared resources that are overwhelmed with ACCs’ load. As a result, each ACC has to wait for shared resources that adversely affect energy efficiency and utilization of ACCs.

To experiment the impact of increasing ACCs on shared resources, this section explores the whole design space of a synthetic streaming application running on a conventional processor-centric ACMP architecture. For the explorations, we derived a first-order analytical model of conventional ACMP architecture with one processor core, 8MB on-chip memory, 4-layered interconnect and we increased the number of ACCs from 0 to 14 in order to offload 0% to 100% of total computation load on the ACCs (the details of the application and model are presented in Chapter 3).

Figure 1.9a shows the synchronization load of ACCs on the host processor core, and Figure 1.9b demonstrates the ACCs’ load on the communication fabric/DMA(s) and memory for data transfer. Both of the synchronization and data transfer load of ACCs on the shared resources grow with an increasing number of ACCs and increasing amount of computation load covered by the ACCs. As a result, shared resources saturate, thereby ACCs have to wait and significantly under-utilize as Figure 1.9c shows.

To evaluate the performance impact of idle ACCs waiting for shared resources, we estimated system throughput as an indicator of system performance. As Figure 1.9d shows, the system throughput is not always increasing with increasing number of ACCs. At some point (in our example, where a 75% of total computation are covered by ACCs), the throughput drops. This increasing-decreasing trend (when the number of ACCs increases) is also observed for ACC utilization in Figure 1.9c.

The explorations above show that conventional processor-centric ACMP architectures suffer from scalability limitations. With an increasing number of ACCs, the ACCs’ benefits are not always increasing, and at some point, the benefits diminish due to bottlenecks emerging on shared resources. Increasing number of ACCs and increasing load on shared resources drive ACCs to under-utilize.
CHAPTER 1. INTRODUCTION

(a) Synchronization load  
(b) Data transfer load  
(c) ACC utilization  
(d) System throughput  

Figure 1.9: Processor-centric ACMP with increasing number of ACCs and computation load covered by the ACCs.

1.3 Problem Definition

The focus of this dissertation is on relieving scalability limitations of conventional ACMP architectures to provide efficient integration of ACCs and achieve maximum benefits. This dissertation addresses the following problems.

1. Lack of holistic view of the origins of conventional ACMP scalability limitations

   A holistic approach is required to provide the sources of inefficiency and scalability limitations in conventional processor-centric ACMP architectures as more ACCs are integrated, while all shared resources are fixed.

2. Lack of a through analysis of the scalability limitation impact on ACMP performance
CHAPTER 1. INTRODUCTION

Increasing ACCs results in simultaneous demand for all shared resources, and thereby overshadows ACCs’ performance efficiency. A thorough investigation is demanded to capture the impact of scalability limitations on the maximum achievable performance benefits of ACCs.

3. **Lack of semantics for ACC communication**

With increasing number of ACCs and the likelihood of direct ACC-to-ACC connections, less load is expected on shared resources as ACCs should directly communicate with each other. However, even for direct ACC-to-ACC connections, all shared resources are still involved due to lack of precise semantics for ACC communication and relying upon general architectures to connect many ACCs on a chip.

4. **Lack of efficient realization of direct ACC-to-ACC connections**

A novel architecture is required to efficiently realize direct ACC-to-ACC connections (that increases with integration of more ACCs) without posing any overhead to shared resources.

A holistic approach is required to address these challenges and help us through designing a scalable architecture tailored for many ACCs.

1.4 **Dissertation Overview**

This dissertation addresses scalability limitations of conventional ACMP architectures with many ACCs integrated on a chip. Figure 1.10 visualizes the remainder of this dissertation.

Single ACC design and conventional architectural approaches to integrate many ACCs is the focus of Chapter 2. Section 2.1 studies ACC design in view of energy efficiency and flexibility to cover wider range of workloads. Following that, Section 2.2 inquires into the architectural approaches that partially resolve scalability limitations.

Chapter 3 focuses on the origins of the scalability limitation, and its impact on system performance. First, Section 3.1.2 proposes an analytical model to estimate the load on shared resources (processor, communication fabric/DMA, and memory) as more ACCs are integrated on a chip. With more ACCs, scalability challenges appear that limit ACCs’ benefits. Later on, Section 3.2 proposes ACMPerf, an analytical performance model of ACMP architecture capturing ACCs’ benefits and the impact of ACCs’ load on shared resources. Using ACMperf, designers can quickly explore the whole design space with respect to the number of ACCs and computation covered by ACCs to find the most efficient design prior the real implementation.
CHAPTER 1. INTRODUCTION

Figure 1.10: The dissertation overview: problem analysis to solution and evaluation.

To lay the foundations for improving the efficiency of ACCs integration, Chapter 4 identifies and formalizes the ACC communication semantics. The semantics includes synchronization of ACCs, putting ACCs’ data in correct size and representation, as well as data transfer among ACCs. At the end, it summarizes how the semantics realization on processor core(s), and other shared resources lead to scalability limitations in conventional ACMP architectures with many ACCs.

With sparse integration of ACCs where there are only processor-to-ACC connections (a processor produces data to be consumed by an ACC) or ACC-to-processor connections (an ACC produces data to be consumed by a processor), it is inevitable that shared resources are involved. With increasing ACCs, the likelihood of direct ACC-to-ACC connections (an ACC produces data...
to be consumed by another ACC) increases, while all shared resources are still involved. As we demonstrate in Figure 1.11, the connection from a producer ACC (ACC\textsubscript{P}) to a consumer ACC (ACC\textsubscript{C}) is realized as a connection from ACC\textsubscript{P} to the processor, and then, from the processor to ACC\textsubscript{C}. In both connections, the memory is involved to store the data that is transferred through the communication fabric and via DMA under the control and synchronization of the processor. It is also the responsibility of the processor to put the data produced by the ACC\textsubscript{P} into a correct format/size acceptable by the ACC\textsubscript{C}.

Chapter 5 proposes a novel architecture Transparent Self-Synchronizing ACCs (TSS) that provides equal view between ACC and processors to support direct connections among ACCs without imposing any overhead to shared resources. TSS adds autonomy to ACCs to self-synchronize each other, and handle the size and representation of data by themselves. Furthermore, TSS exploits a local MUX-based interconnect across ACCs for direct data transfer. Thus, the proposed TSS internally and without imposing any overhead to shared resources realizes communication semantics of all direct connections in chains. A chain is a set of direct ACC-to-ACC connections in which a consumer ACC in one direct connection is a producer ACC in another direct connection.

However, the first and last ACC of a chain have to communicate with processor. To keep the equal view between ACC and processor, TSS adds a gateway as the interface between TSS (a chain of ACCs) and processor(s). TSS feeds chains (to the first ACC in a chain), and then collects chains’ resultant data (from the last ACC of a chain). Additionally, the gateway is responsible to configure the MUX-based local interconnect to setup all required direct paths among pairs of producer and consumer ACCs.

To provide more flexibility for directly connecting ACCs to each other, Chapter 7 proposes TSS plus that aims at replacement of the MUX-based interconnect with a Network on Chip (NoC).
CHAPTER 1. INTRODUCTION

The main focus of this chapter is on abstract and yet, accurate modeling of NoC to enable quick evaluation of TSS plus with changing NoC parameters based on applications. This chapter, proposes a Transaction Level Model (TLM) of NoC with almost 10x faster than RTL-implemented NoC, with 10% to 20% accuracy loss on average. The integration of the proposed TLM is defined as the future work of this dissertation.

1.5 Contributions

The main contributions of this dissertation are the following:

1. **Holistic analysis of the scalability implication on ACMP performance**

   We proposed an analytical model that provides an overview of the ACMP scalability limitation and its impact on the maximum achievable ACCs’ benefits.

   - **Holistic investigation of the origins of ACMP scalability limitations**
     
     We proposed an analytical model to investigate the impact of increasing ACCs on shared resources (processor, communication fabric/DMA, and memory). The model estimates how shared resources are over-utilized due to ACCs’ communication and orchestration load (published in [108] and details are provided in Section 3.1 of Chapter 3).

   - **Through investigation of the scalability limitation impact on ACMP performance**
     
     We proposed ACMPerf, a first order analytical performance model of conventional ACMP architectures that quantifies the impact of scalability limitations on ACMP performance. Our ACMPerf captures the main performance contributors that include the processing acceleration benefits of ACCs and ACCs’ load on shared resources (details are provided in Section 3.1.3 of Chapter 3).

2. **Identification/formalization of ACC communication semantics**

   We formalized the semantics of ACC communication that include: data access model, synchronization, data marshalling, and data granularity (published in [109] and details are provided in Chapter 4)

3. **The proposed Transparent Self-Synchronizing ACCs (TSS)**
CHAPTER 1. INTRODUCTION

We proposed a novel architecture of Transparent Self-Synchronizing (TSS) ACCs that internally realizes all direct ACC-to-ACC connections without imposing any overhead to shared resources.

- **Efficient realization of the semantics for direct ACC-to-ACC connections**
  Our proposed TSS has added autonomy to single ACC to handle the semantics aspects of synchronization, data granularity, and data marshalling, as well as a local interconnect (which is MUX-based) to handle the semantics aspect of data access model, all locally inside the TSS. Consequently, TSS realizes all semantics aspects involved for direct ACC-to-ACC connections without posing any overhead to shared resources and alleviates the scalability limitation (published in [109] and details are provided in Chapter [5]).

- **Automatic generation of virtual platforms for TSS evaluation vs. ACMPs**
  To evaluate the proposed TSS versus conventional processor-centric ACMP architectures, we used SpecC to model the architectures (both conventional and TSS), and then refined the models in System-On-Chip Environment (SCE) in different views of architecture, scheduling, network and communication to generate Virtual Platforms (VP). All steps from specification to VP generation and evaluation are automated allowing us to explore large design space and compare different design metrics effectively (published in [108, 109] and details are provided in Chapter [5]).

- **TSS plus: TSS with an abstract model of NoC for integration of ACCs**
  To better exploit TSS with the same number of ACCs, we proposed TSS plus. TSS plus has all the TSS features, but NoC as a local interconnect across ACCs to provide more flexibility for directly connecting ACCs together. The novelty of this work is the abstract and accurate modeling of NoC that allows us to easily change the NoC parameters and evaluate their impact on achievable ACCs’ benefits. The integration of the proposed model of NoC inside TSS is defined as one of the future work (published in [44] and details are provided in Chapter [7]).

After analyzing the scalability limitations of processor-centric ACMP architectures, and defining the semantics of ACC communication, we will propose our Transparent Self-Synchronizing Accelerators (TSS) architecture that is tailored for integration of many ACCs on a chip. TSS internally realizes the semantics of direct ACC-to-ACC connections without imposing any load to shared resources, and this is the difference of TSS with all previous work. TSS solves scalability
CHAPTER 1. INTRODUCTION

limitations by reducing the ACCs’ load on all shared resources, while previous work either avoids scalability limitations or attempts to partially solve it by focusing on one resource and overlooking the other resources.

1.6 Related Work

1.6.1 Processor-centric ACMPs with Slave ACCs

In conventional ACMP architectures, ACCs are treated as slave devices occupying processor core(s), and many other shared resources of communication fabric, DMA unit(s), and shared memory for every single transaction. This processor-centric view and ACCs’ load imposed on many shared resources has led to ACMP scalability limitations and diminishing ACCs’ benefits. Most literature acknowledges the ACMP scalability limitation by avoiding it or partially solving it through relieving the load on only one shared resources.

1.6.1.1 ACMPs with Limited Number of ACCs

There are comparatively few work \cite{73,3} that limit the number of ACCs to avoid scalability limitations.

Nilakantan et.al \cite{73} and Zidenberg et al. \cite{126} clearly appreciate the resources bottlenecks when it comes to integrating many ACCs. Following that, they propose analytical approaches to find the optimum set of ACCs either with respect to communication limitations \cite{73} or area limitations \cite{3}.

\cite{73} uses a mathematical method to profile ACCs’ data access load. Then, based on the profiling information, and with the aim of overlapping ACCs’ communication and computation, it finds the optimum sets of ACCs.

Comparably, the authors in \cite{126,68} propose an analytical optimization framework that receives available system resources (area and power), system components, characteristics of all components, and design goals. Then, the framework allocates a particular set of ACCs and assigns resources to ACCs in order to optimize the design goals. The framework can be extended to consider different constraints rather than the area and power such as SoC partitioning, bandwidth, and energy consumption.

Conventional ACMP architectures have processor-centric view. With the processor-centric view, ACCs are treated as slaves devices requiring all shared resources, and more particularly the
CHAPTER 1. INTRODUCTION

processor core(s) for every single transaction. To mitigate this dependency of ACCs to shared resources, literature attempts to relieve the ACCs’ load primarily on one of shared resources without bringing a holistic solution to reduce the load on all shared resources.

1.6.1.2 ACMPs with Relieving Load on One Shared Resource

This section categorizes ACMP architectures in three categories based on the shared resource that they focus on to partially resolve the ACMP scalability limitation.

- **ACMP with reduced load on processor core(s):** CAMEL [16] and CHARM [14] identified significant synchronization and control load on the host processor(s) in conventional ACMP architectures. To reduce that load, [16] and [14] propose Accelerator Block Composer(s) –ABC(s)– that is a replacement for processor(s) per a cluster of ACCs and responsible for all inter-cluster’s synchronization and control load. However, the functionality of ABC(s) is limited, and it is also not clear how the ABC(s) replaces the processor(s) for all synchronization and control tasks.

- **ACMP with reduced demand for the shared memory:** [20], [50], and [64] have observed the issue of large memory requirements to hold local data per ACCs. They aim at reusing the memory in order to minimize the available buffers per ACC, and reduce total memory usage. The accelerator-store architecture [64] proposes a centralized buffer shared among ACCs. Similarly, [20] and [50] propose sharing ACCs’ buffers with processor core(s). Thus, the memory can be reused by both ACCs and processor core(s) according to run-time needs. Despite reduced load on the memory, the processor core(s) is still involved to manage the memory sharing among ACCs and processor core(s). In addition, ACCs’ input/output data is transferred through the communication fabric and as a result, a bottleneck still appears on the communication fabric with integrating many ACCs on a chip.

- **ACMP with reduced load on communication fabric:** [80], [100], and [96] show that the achievable ACC benefits are bounded by the latency of the communication fabric; with more ACC communication demand (amount of input/output data from/to ACCs), system performance exponentially reduces. To relieve the load on the communication fabric, both [80] and [100] cluster the ACCs, and propose a two-level hierarchical interconnect architecture: the first level localizes the intra-cluster communication demand through a NoC interconnect, and the second level provides a bus for transferring inter-cluster communication demand.
Similarly, the feature GPU-Direct \[81\] offered by CUDA aims at enabling efficient data movement among GPUs and other network devices. Nonetheless, there is no change in the load on the memory and processor core(s).

1.6.2 Peer View between ACCs and Processor

One recently published study \[112\] hints about shifting away from processor centric view to a more view between ACC and processor. This work proposes a SW-based system level linker that allows functions (independent of in HW or SW) to call each other, and to be linked together. By linking either post-compilation components (SW object files) and post-synthesis components (HW object files), programmers can transparently move functions between devices simply.

However, this work only creates an illusion of equal view of SW and HW for programmers, the underlying ACMP architecture is still the same processor-centric ACMP architecture. Therefore, there is still a need toward architecture platforms that support the equal view of ACCs and processor core(s).

To sum up, the literature lacks a holistic view to solve the scalability limitations of ACMPs considering the load on all shared resources simultaneously. Therefore, there is a need for more efficient architectures with equal view of ACC and processor to relieve the ACCs’ load on all the shared resources when more ACCs are integrated.
Chapter 2

Background

In this chapter, we describe the terminologies related to ACMP architectures, and discuss supplementary materials that are required to better articulate our proposed ACMP architecture. First, we provide a taxonomy for single ACC design that helps to qualitatively compare different ACC designs in views of flexibility and energy efficiency. Next, we focus on compositions and integrations of ACCs on chips. Considering the involvement of all shared resources (communication fabric, DMA, memory and processor) for every single ACC transaction in conventional processor-centric ACMP architectures, this chapter surveys the literature that attempts to (partially) address integration of ACCs.

Finally, we discuss streaming applications that are the focus of this dissertation. We chose streaming applications as they are revolutionizing our lives in different views of education, communication, entertainment, etc. Many practical algorithms such as deep-learning, image processing (e.g., object detection, background subtraction), and multimedia (e.g., video encoding/decoding) have streaming nature. In addition, streaming applications follow dataflow programming paradigm, with inherent pipeline parallelism over considerable amount of data which make them as a good candidate for acceleration.

2.1 Single ACCelerator Design

The primary motivation to exploit ACCs is to remove the overhead of general purpose processing, and thus improving energy and power efficiency. The main challenge of ACC design, and generally all processing elements is the trade-off between power-performance (energy) efficiency and flexibility as shown in Figure 2.1. The more a processing element is customized to execute
a specific workload(s), the more it provides energy efficiency. Conversely, the more general a processing element is to execute wider range of workloads, the less time and cost is needed for its design/verification at the expense of energy efficiency. To provide a landscape for processing elements with respect to both flexibility and energy efficiency, we first define two aspects of flexibility: \textit{Processing flexibility}, and \textit{Communication flexibility}.

Processing flexibility defines the granularity of programming at run-time, and communication flexibility defines the granularity at which the communication can change at run-time.

The granularity of programming ranges form (high to low) as: (1) \textit{Bit}, and its primary example is Field Programmable Gate Array (FPGA) that can be programmed bit-by-bit via logically connecting the basic gates, (2) \textit{Instruction}, and its main example is processor cores that can be programmed for each instruction. (3) \textit{Kernel}, and one example is Functional Level Processor (FLP) \[99\] that is re-configured to execute specific kernels, and (4) \textit{Application}, and the main example is Application Specific Integrated Circuit (ASIC) that is fixed at design time to execute a specific application. The granularity of communication is similarly defined from high to low: high refers to any-to-any communication among processing units. Conversely, low refers to fixed communication among processing units. Processor core is an example of processing elements with high communication flexibility, in which the output(s) of any instruction(s) can be passed as input(s) to any other instructions using shared memory or last level caches among the processing cores. FPGA is an example of processing elements with the highest communication flexibility to connect the output(s) of any gate(s) to the input(s) of any other gate(s).

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{energy_vs_flexibility.png}
\caption{Trade-off between flexibility and energy efficiency.}
\end{figure}
CHAPTER 2. BACKGROUND

Figure 2.2: Landscape of processing in view of flexibility, and energy efficiency.

Figure 2.2a categorizes processing in view of flexibility with using examples. The Y-axis presents the communication units, and the x-axis lists the processing units. As this figure shows, at one extreme, FPGA has the highest processing granularity (bit), and highest communication granularity, as every bit can communicate to every other bit. At another extreme, there is ASIC that is designed to execute a specific application, and its communication is fixed at design time.

All GPP, GPU, ASIP, and CGRA have the same processing flexibility, and they are programmed at instruction level. However, GPP and ASIP have higher communication flexibility as processing cores can communicate to each other via shared memory (or last level caches). In GPU, the cores communicate together via the SPM in streaming multi-processing or main memory, therefore, the communication flexibility of GPU is limited as compared to GPP and ASIP. Coarse Grained Reconfigurable Array (CGRA) has the least communication flexibility among them as in CGRA, each core can communicate only with its neighbors in a mesh-based interconnection topology.

In FLP, each functional units (FU) is fixed to execute specific kernel, but there is flexibility to connect any FUs to any other FUs. By changing the composition of FUs, FLP can be re-configured to execute different kernels in the same domain that is designed for.

To compare the examples designs in view of energy efficiency, we combined both flexibility aspects and show them as one metric in X-axis of Figure 2.2b. The ASIC with lowest degree of flexibility has highest energy efficiency. Then, FLP and CGRA have highest level of energy efficiency, while provide more flexibility than ASIC.

FPGA provides highest flexibility, however, the overhead of find-grained routing reduces
CHAPTER 2. BACKGROUND

its energy efficiency. GPU and ASIP, assuming running the right class of applications that they are
designed for can provide better energy efficiency as compared to GPP. Figure 2.2b shows the same
trend of energy-flexibility trade-off shown in Figure 2.1. With more flexibility, less energy efficiency
is achieved.

In the following, we study how literature designed different processing elements to enhance
energy by focusing on a specific workload(s) and improve performance for that workload(s).

2.1.1 Application Specific Integrated Circuit (ASIC)

ASICs are designed and highly customized for a specific application. ASICs have energy
efficient execution as there is no need to go through the normal processor pipeline stages of fetch,
decode, commit and write-back, but directly execution stage. However, they lack re-useability and
need to be re-designed and re-customized for a new application.

Hameed et al. [42] designed an ASIC for H.264/AVC encoder on the silicon core size of
7.68\texttimes 4.13 \text{mm}^2 and 180 nm CMOS technology. Another example is Neuro Vector Engine (NVE)
designed by Peemen et al. [79]. NVE is an ASIC for visual object classification in convolution
networks that targets portable and wearable devices. It achieves 30 Gops with a power envelope of
54 mW and only 0.26 \text{mm}^2 silicon footprint at TSMC 40 nm technology.

2.1.2 Field Programmable Gate Array (FPGA)

These designs are implemented using programmable fabric (e.g., FPGAs). The HW
components to make these designs are often Lookup tables (LUTs) and gates, and different fusing of
the LUTs results in different designs with high flexibility. [116] and [45] are a few examples using
FPGA designs besides the general purpose core(s) to accelerate frequent regions of the programs.

Fusing LUTs makes these designs to be re-designed easily, but with considerably larger
area and power dissipation, as well as lower speed in comparison with the ASICs. [59] reports that
on average an FPGAs is 40X larger and 3.2X slower, with 12X higher dynamic power dissipation
than ASICs.

2.1.3 Application Specific Instruction-based Processors (ASIP)

ASIP processor core(s) with specialized Instruction Set (ISA) is tailored to benefit a specific
application domain (usually radio base-band, image and video signal processing [62]). It means
that, they are general enough to execute different application programs (instruction by instruction
CHAPTER 2. BACKGROUND

or a region of code through normal pipeline stages) in that domain they are customized for, but at a cost of reduced energy efficiency as compared to ASIC [74]. In addition, the ASIP can have reconfigurable components (fine grain on programmable logic) to add new instructions.

[72] developed a new biochip platform for autonomously sequencing and aligning biological sequences. This platform uses a shared memory model and multiple instantiations of a novel ASIP (with both general and special purpose single-instruction multiple-data instructions) to achieve 30x performance improvement with dissipating small power as compared to sequential algorithmic implementations on GPPs.

2.1.4 Graphics Processing Unit (GPU)

GPUs are processing core(s) that were initially aimed at acceleration of the memory-intensive work of texture mapping and rendering polygons [74]. In recent developments, GPUs are configured as more general purpose cores to improve the performance of applications with embarrassingly parallel nature. It is the responsibility of the host processor(s) to re-configure GPU core(s) to accelerate a kernel with massive parallelism.

[125] proposes an optimized GPU implementation of Mixture of Gaussians (MoG) to enhance the performance over a sequential implementation. The GPU implementation with general optimization such as memory coalescing, and computation/communication overlapping outperforms the sequential implementation on GPPs by 57x. The authors showed up to 100x enhancement on performance in their GPU-implemented algorithm by applying algorithm-specific, and windowed optimizations.

2.1.5 Function Level Processor (FLP)

The Function Level Processor (FLP) is a complete processor consisting of individual functional blocks (FB). The FBs are optimized to execute specific functions for a target domain. Without any need for the host processor(s), it has an autonomous programmable architecture for application (in the same domain) composition out of FBs.

FLP demonstrated on an industry example of the Pipeline-Vision Processor (PVP) with six embedded vision applications. It can offer up to 22.4 GOPs/s performance with 17x less power dissipation as compared to general purpose processors.
CHAPTER 2. BACKGROUND

2.1.6 Coarse Grain Reconfigurable Array (CGRA)

These designs are composed of a set of smaller ASIC designs (that could be observed as cores or FUs) with the aim of HW re-usability. Through different compositions of smaller ASIC designs in order to adapt the data-path at run-time to the application, CGRAs provide flexibility to run wide range of workloads \[14, 78, 13, 120\]. However, a CGRA design still suffers from being limited to the set of its smaller ASIC designs, when facing new algorithmic innovations or being used in the other domains different than the one its smaller ASIC designs were originally designed for \[31\].

The semi-programmable loop \[31\] and Veal \[13\] are both examples of CGRAs aim to accelerate the innermost loops that are frequently occurred in programs. In the semi-programmable loop, it is the responsibility of the compiler to generate instructions to be executed on the ASIC cores. However, to improve the code portability, Veal exploits virtual translator to maps the instructions to the ASIC core at run-time. The semi-programmable loop achieves up to 34x power improvement. Veal also provides a mean speedup of 2.66x as compared to a single-issue GPP.

Figure 2.3: Processor pipeline with DySER Datapath and DySER elements \[39\].

The Dynamically Specialized Execution Resource (Dyser) \[39\] is another well-known example of CGRAs. Dyser integrates a sufficiently general purpose and flexible CGRA units into a processor pipeline as Figure 2.3 shows. The CGRA unit consists of FUs connected together via switches. It is the responsibility of the co-designed compiler to adapt the CGRA to execute any
CHAPTER 2. BACKGROUND

applications. With Dyser, an extra issue slot is added to the execution stage. This slot contains a FIFO structure to make a vector from sequential instructions, and then feeds these to the array of FUs. With integration of 64-FU DySER that supposed to cover 12% to 100% of the dynamically executed instruction stream, the architecture results in 1.15X to 10X speedup, with geometric mean energy reduction of 40% to 70%.

2.1.7 Hybrid Designs

These designs are a composition of designs with different energy efficiency and flexibility for covering wider range of workloads. For example, CAMEL \[16\] exploits CGRA-like building blocks for energy efficiency. CAMEL also has a programmable fabric (PF) to extend the use of composable ASICs in supporting algorithms that are beyond the scope of the original domain that the platform is designed for. Using a combination of HW extensions and compiler support, CAMEL achieve on average 11.6x performance improvement and 13.9x energy savings across benchmarks that deviate from the original domain.

In this dissertation, we propose a novel architecture to efficiently integrate ACCs on a chip to enhance both flexibility and efficiency. The ACCs are designed in ASIC for its highest energy efficiency, and the architecture efficiently integrates and composes ACCs to (1) internally realize all ACCs composition without imposing any load to shared resources, (2) covers a wide range of workload for which the ACCs are designed. In the following we will study previous architectures for ACCS integration that attempted to partially reduce the load of ACCs integration on shared resources.

2.2 ACC Integration

Based on how ACCs are integrated with general purpose processor(s) on a chip, ACCs can be categorized into two classes of i) Tightly Coupled ACCs (TCA), and ii) Loosely Coupled ACCs (LCA).

As Figure 2.4a shows, TCA is a functional unit that is attached to a specific core through its datapath. The core manages the TCA with customized instructions that have been added to the core’s ISA \[54\]. The ISA expansion is usually diffused through the software via the compiler or the low-level libraries, and memory hierarchy is shared between TCA and core. Most of the microprocessors use TCAs for acceleration of multimedia applications, and their ISA is extended with customized Single-Instruction Multiple-Data (SIMD) to access the TCAs. The Intel’s MMX and SSE extensions
added into the x86 ISA are examples of two generations of customized instructions. Tensilica ASIP \[29\] is also consists of reconfigurable logics allowing implementation of tightly-coupled instruction set extensions.

The run-time overhead to invoke TCAs is negligible as it is done through execution of some instructions. However, integrating a TCA involves a few disadvantages. First, TCA is integrated with a specific core, and is not accessible by the other cores. This design make core design and verification more complex. Second, the TCA’s parallelism potential is confined by the unavoidable low number of ports of L1 caches, and the data granularity of TCA is also the same as the cache line size. Last, but not least, the TCAs have limited portability among different system designs as they are necessarily adapted to work with a specific core \[19\].

Alternatively, the LCA is a distinct component that can be shared and reused among multiple cores through an on-chip or off-chip interconnect. To integrate an LCA, it is sufficient to have a thin hardware wrapper that interfaces the LCA’s configuration registers and DMA controller with the interconnect. By separating processing core and ACC in an LCA design, the core(s) is free to run other tasks or to be turned off to save energy when LCA is running other tasks. LCAs have only access to the last level cache of the core or DRAM through the DMA as shown in Figure 2.4b.

The interaction between a core and an LCA happen via an interrupt-based system and in the form of bulk memory transfers between the core’s and LCA’s separate memory spaces. The user application prepares the data in memory and invokes a system call for passing the physical addresses of the memory to the LCA’s DMA controller. The user application (specifically, its execution thread that caused invoking the LCA) sleeps until an interrupt from the LCA arrives \[19\]. In the IBM CELL processor \[55\], the SPEs are examples of LCAs. More commonly, the GPUs are another examples of LCAs connected to a general purpose core(s).

To sum up, LCAs have some characteristics that can resolve the limitations of TCAs. LCAs are designed independently from the cores and can be reused by different cores. An LCA has its own private memory which is known as Scratch Pad Memory (SPM) and stores the local and temporal data of LCAs. However, LCAs have also some limitations. First, being shared across different cores necessitates some forms of arbitration and scheduling. Second, Direct Memory Access (DMA) to the Last-Level Cache (LLC) and to the DRAM (DRAM-DMA) cause bandwidth saturation and create a bottleneck. Last, but not least, with more number of LCAs, the memory is highly demanded to provide the SPMs and increase energy consumption.

From a system integration viewpoint, LCA’s re-usability as well as independence from the cores offer better design-reuse opportunities and simplify porting LCAs across different technology
CHAPTER 2. BACKGROUND

(a) Tightly Coupled ACC

(b) Loosely Coupled ACC

Figure 2.4: Models of coupling ACC with a general purpose processor on a chip [19].

processes [19]. The focus of this dissertation is on the integration of LCAs similar to the previous work [18,94,19].

However, current ACMP architectures have processor-centric view, i.e., all ACCs are dependent on the processor (and particularly the operating system running on top of the processor) to execute their functions. In addition, ACCs have I/O complexity and loads on the communication fabric, DMA and memory [64] that can adversely impact the expected efficiency gains out of ACCs. In the following, we discuss the architectures that try to relieve the ACCs’ loads on shared resources.

2.2.1 Architecture Support to Reduce the ACCs’ Orchestration Load

Global ACC Management (GAM) [11], Composeable Heterogeneous ACCelerator-Rich CMP (CHARM) [14], and Composable Accelerator-rich Microprocessor Enhanced for Longevity (CAMEL) [16] are the architectures that try to share ACCs among all cores and improve the re-usability of ACCs using virtualization and ACC combination.

Figure 2.5a shows the proposed architecture that exploits GAM. This architecture is composed of cores, ACCs, GAM, shared L2 cache banks among the cores, and all connected through Network on Chip (NoC) routers. There is a dedicated DMA controller, SPM per individual ACC, as well as a small translation look-aside buffer (TLB) locally per ACC for virtual to physical address translation. GAM as a hardware resource management enables sharing a common set of ACCs among multiple cores. It also keeps track of the number of ACCs of each type, the jobs currently running on the ACCs, and the jobs waiting to be processed by ACCs. Therefore, it hides the ACCs orchestration and composition.

Figure 2.5b shows the steps for giving grant for accessing an ACC to a core. (1) core sends an access request to GAM, (2) based on the availability of ACCs, GAM sends the access grant to the core. When the core gets the grant, (3) it places data in the memory and passes its address (virtual address) to the ACC. The cores’ ISAs are expanded to have instructions to initiate ACCs. Finally,
CHAPTER 2. BACKGROUND

(a) Architectural Elements [17]  
(b) Communication among GAM, ACC and memory [17]

Figure 2.5: ACCelerator Rich CMP(ARC).

Figure 2.6: CHARM Architecture [14].

(4) the core is notified when the ACC is done, or there is a TLB miss to find the relevant physical address, through a lightweight interrupt between the processor and GAM.

GAM in this architecture supports composition of smaller ACCs to a bigger ACCs and hides all management of composed ACCs from the cores. However, in this architecture, the ACCs might not be well utilized. CHARM architecture [14] addresses utilization problem and improves energy efficiency by exploiting fine-grained ACCs, and avoiding duplication of resources such as DMA and SPM for individual ACC. CHARM exploits ACC Building Blocks (ABBs) implemented as ASIC for computation of frequently executed code in a domain.

As Figure 2.6 shows, the architecture includes islands of ABBs. Each island contains ABBs, one DMA and one SPM shared across all ABBs. The GAM is also replaced with ACC Block Composer (ABC). Identification of hotspots in programs and selection of ABBs, and their composition as LCAs in islands, arbitration over requests for ABBs and load balancing among available compute resources are all the responsibilities of ABC. Due to HW-reuse, CHARM achieves
CHAPTER 2. BACKGROUND

up to 2.1x speedup for running medical imaging benchmarks as compared to GAM and reduces energy consumption by 2.4x.

However, both GAM and CHARM architectures are limited to the workload that the ACCs are designed for. CAMEL \cite{16} introduces better flexibility by covering a wider range of workload. It is supplied with a reconfigurable program logic to implement virtual ABBs. The ABBs implemented on ASIC are called real and the ones on programmable logic is called virtual. It is the responsibility of the compiler map applications to real and virtual ABBs. Replacing 50% of the ASIC ABBs with programmable logic achieves on average 11.6x performance improvement, and 13.9x energy savings over CHARM.

2.2.2 Architecture Support to Relieve the ACCs’ Data Storage Load

As the number of ACCs increases, there is a demand for high capacity and high bandwidth memory. However, it is not efficient to increase the size of memory due to the power wall and cost issues \cite{104, 9}. Not all the ACCs are powered-on at the same time during application execution, thereby memory can be shared among ACCs as ACC-store architecture \cite{64} does. Similarly, Buffer-Integrated-Cache(BiC) \cite{30} and buffer-in-NUCA (BiN) schemes address memory sharing \cite{50}.

The ACC-store \cite{64} shown in Figure \ref{fig:2.7} shares a pool of buffer among ACCs, where each ACC has a store port (ASPort) to access the data, instead of a bus to avoid contentions. Inside the ACC store, there are three main components: (1) the priority table for arbitrating memory requests from ACCs, (2) the handle table for translating these demands from ACCs into SRAM accesses, (3) and the SRAM collection for storing data. When the SRAM collection is all occupied, the ACC state can be temporarily paged out from the ACC store to system memory.

However, ACC-store uses a contiguous space to each buffer to simplify buffer access. However, when requested buffers have unpredictable space demand and come in dynamically, fragmentation will happen. The BiC architecture \cite{30} re-uses ACCs’ buffer by cores when the buffer is not used. As Figure \ref{fig:2.8a} shows, the cache area is separate from the ACCs’ buffers which are dedicated for ACCs. The cache and ACCs’ buffer are in the SRAM, the BiC enables cores to view portions of the SRAM as cache while ACCs access other portions as private buffers. The BiC has a substrate that is flexible by allowing SRAM to be dynamically re-purposed as buffer or cache.

In a similar work, BiN architecture \cite{50} shares the SRAM among ACCs and processor cores (as cache) in order to reduce required memory demand. In addition, it resolves the space fragmentation in two ways: (1) with a small page table per ACCs and a flexible paged buffer
2.2.3 Architecture Support to Relieve the ACCs’ Data Transfer Load

The data transfer across ACCs, general purpose processor(s) and memory through the communication fabric is one of the loads that can significantly overshadow the ACC benefits. The communication latency is responsible for 60% to 70% of total latency [86]. Stillwell et al. [96] shows that with increasing the number of ACCs and their communications, the performance is reduced exceptionally. Nilakantan et al. [73] considers the communication latency as an important factor that impacts the performance drastically and proposes an analytical model to find the optimum number of ACCs while the computation is overlapped with the communication.

One approach to reduce the impact of ACCs’ communication is to increase the system
CHAPTER 2. BACKGROUND

communication bandwidth. For example, in PowerCell [55], there is a central communication channel that is consists of four 128-bit wide concentric rings to provide point to point connections across the SPEs. Additional nodes (e.g. SPEs) can be added easily with only increasing the maximal latency of the ring.

Other approach is to localize ACCs’ communication. Tan et al. [100] proposes a hybrid communication infrastructure using both standard bus and area-efficient and delay-optimized Network on Chip (NoC) for an ACMP. The bus based interconnect is very area efficient for systems with a small number of cores. It transfers data across the ACCs through an NoC and data across cores and ACCs through the bus. Therefore, with increasing number of ACCs and more number of direct ACC-to-ACC connections, the NoC is more utilized. In addition, the bus is free to transfer data and control/interrupt information from/to the processor to/from the ACCs.

There are also some prestigious work to improve the communication latency specifically for GPUs. The GPU communication has resided out on PCIe as a discrete device, thus the performance of GPU applications can be bottlenecks by data transfers between the CPU and GPU over PCIe. The CPU has to be involved for any interactions between two GPUs. In this regard, CUDA has offered the GPU Direct [52] to relieve the data movement load of CPU/GPU over PCIe. Figure 2.9 shows direct access and data transfer for GPUs. A GPU has direct access to the memory of another GPU one the same PCIe Figure 2.9a and data can be directly transferred between memories of two GPUs on the same PCIe Figure 2.9a, which results in 6x faster GPU to GPU interactions. Similarly, AMD Fusion fuses the functionality of GPU and CPU on the same die. In this architecture, the x86 CPU cores and the GPU cores share a common path to system memory to assist data movement between the x86 and GPU cores and improves application performance.

![Figure 2.9: GPU Direct on the same PCIe bus](image-url)

(a) P2P direct access
(b) P2P direct transfer
CHAPTER 2. BACKGROUND

Overall, we observed that all partially-efficient architectures for integration of many ACCs can experience diminishing benefits of ACCs. Specifically, for compute-intensive applications such as streaming applications with massive amount of data, it happens earlier. Next, we study streaming applications as the main focus of this dissertation.

2.3 Streaming Applications

Streaming processing has appeared pervasive in different streaming application domains such as multimedia, augmented reality, virtual reality, autonomous vehicle, and vision (shown in Figure 2.10). A streaming processing is usually expressed as a Data Flow (DF) programming model \[27\], or Kahn Process Network (KPN) \[22\] that is composed of concurrent computation kernels with streams of data flowing across the kernels \[27\]. A kernel is a function that is replayed for each subsequent elements in its input streams to produce output streams and feeds the next kernels \[83\]. The kernel that produces a stream(s) is called producer kernel and the one that is fed with a stream(s) is called consumer kernel.

Figure 2.11 shows the data flow model of the MP3 decoder as one of the multimedia applications. In this example, computation kernels are represented as circles (nodes) and data streams flowed across the kernels are represented as directed lines (edges). The direction of an edge distinguishes the producer kernel from the consumer kernel on that edge. Each circle is labeled with a computation load (calculated according to the number of computation operations of a kernel) and each directed line is labeled with a communication load (according to the size of data) flowed from a producer kernel to a consumer kernel. As an example in Figure 2.11 the Req0 produces data for the Reorder0.

To deal with intensive computation demand on massive amount of data, the streaming processing requires to be accelerated on power efficient platforms to meet the requirements of real time, embedded and mobile systems. Meeting these requirements necessitates to exploit the characteristics of the streaming processing: locality and parallelism.

Both kernel and producer-consumer localities are available in the streaming processing \[111\]. The kernel locality means that a kernel only accesses to its local/temporal data as well as global input/output data and there is no need for accessing the other kernels’ local/temporal data. The producer-consumer locality indicates that one kernel produces stream elements to be consumed by another kernel(s) in sequence \[38\]. Therefore, the streaming processing comprises a bandwidth hierarchy: (1) Temporary/local data that is accessed more often and quickly by the same kernel, (2)
CHAPTER 2. BACKGROUND

Figure 2.10: Examples of streaming applications.

Figure 2.11: DF model of MP3 decoder; circles represent the computation kernel and solid lines represent the streaming data.

Intermediate streams are distributed between kernels (from producer kernel(s) to consumer kernel(s)), and (3) Global input/output data is accessed infrequently.

The producer-consumer parallelism, and easy extraction of data dependencies across all kernels as inherent characteristics of streaming processing allow each element of an input stream (an audio frame in case of MP3 decoder) to be processed in parallel [参阅].

The locality and parallelism described above make streaming applications a good candidate to be accelerated on ACMP platforms, with parallel cores and hierarchical BW for intra/inter core data transmission.
Chapter 3

Scalability Implication on ACMP Performance

This chapter holistically explores and analyzes conventional processor-centric ACMP architectures in view of scalability limitations. To this end, this chapter first estimates the impact of increasing ACCs on shared resources (processor, communication fabric/DMA, and memory). It explores how shared resources are over-utilized with increasing ACCs, and as a result dim ACCs’ benefits.

Then, to quickly explore the whole design space with respect to the number of ACCs and find the most efficient design in view of maximum ACCs’ benefits, it proposes ACMPerf. ACMPerf is a first-order analytical model (inspired by [60]) of ACMP performance capturing primary system-level performance contributors to estimate the achievable ACCs’ benefits without going through time-consuming simulations.

3.1 Resources Bottlenecks: Analytical Investigation

3.1.1 Origins of the Resources Bottlenecks

In the past, we have seen ACMPs in which software-based computation was interspersed with hardware acceleration. With sparse integration of ACCs, ACCs were not first-class citizens in designs, and processor-centric view was dominant in architectures. With a processor-centric view, host processor core(s) is responsible for synchronization and control of ACCs. Looking back to Figure [1.7] in Section [1.2.2] the processor core(s) configures the DMA for each ACC’s input/output
CHAPTER 3. SCALABILITY IMPLICATION ON ACMP PERFORMANCE

Figure 3.1: Ideal and real time-line execution of ACMP with 3 ACCs (chained in producer-consumer fashion)

data transfer from/to the shared memory occupying the communication fabric. Also, the processor core(s) puts data in appropriate type/size acceptable for each ACC, and synchronizes ACCs to start processing on their new input data.

The continuous growth in algorithm complexity, while needing highly power-efficient computing leads to an increase in hardware acceleration. Designers increase ACCs to execute more amount of total computation on ACCs, and provide energy-efficient computing \[91, 58, 15, 50]\]. However, shared resources are fixed, and thus they can be overwhelmed with ACCs’ data access and synchronization load. As a result, ACCs have to wait and serially execute.

Figure 3.1 shows an ideal and real execution time line of a Data Flow (DF) model mapped on an ACMP architecture. The DF consists of three kernels chained together and work in a producer-consumer pipeline fashion. The ACMP architecture includes three ACCs that means one individual ACC for each kernel.

Ideally, with infinite shared resources, all ACCs are fully utilized. As Figure 3.1 shows, by exploiting double buffering in each ACC, there will be overlapped of data processing by ACC (\(P\)) with receiving input data from DMA (\(R\)) and sending output data to DMA (\(S\)). Double buffering for read and write channels per ACC helps to overlap computation with communication in order to improve the overall throughput at the cost of larger SPMs (almost double). However, in reality with limited shared resources, ACCs execute serially as it is zoomed for stage 7 in Figure 3.1. Simultaneous demand of communication and synchronization of ACCs imposed to shared resources lead to highly utilized shared resources, and under-utilized ACCs waiting for shared resources.

Assuming fixed shared resources, and increasing ACCs in a chain (each ACC performs
computation on 64KB; 1B per its cycle), with 1GB total data to be processed only on ACCs, we observe increasing load on shared resources as follows:

The communication load on the communication fabric/DMA/memory: With increasing ACCs, both communication fabric and DMA have to handle more communication load and will face more contentions. Assuming fixed job size for each ACC as Figure 3.2a shows, there is a linear increase of communication load on the communication fabric and DMA as more ACCs are integrated. In addition, as Figure 3.2b shows, with integrating more ACCs (with different sizes of local SPMs), a larger size of memory is required.

![Communication Load](image1.png) ![Required Memory](image2.png)

**Figure 3.2:** ACCs’ loads on the communication fabric/DMA and memory with fixed job size.

![Injection of Input Problem](image3.png)

**Figure 3.3:** Injection of input problem as input jobs.

However, due to power and area concerns, the size of memory is limited. Therefore, with more ACCs, the memory is shared across more ACCs, thereby smaller SPM is allocated per ACC. As the job size is determined by the size of SPM, with smaller SPM, the job size decreases, and the same input problem has to be split into more jobs as illustrated in Figure 3.3.
CHAPTER 3. SCALABILITY IMPLICATION ON ACMP PERFORMANCE

We evaluate the impact of shared memory on the possible job size and number of ACCs with two configurations: (1) 1MB shared memory (observed in current state-of-the-art embedded platforms), and (2) 4MB shared memory (assumed for future platforms). In both configurations, each ACC performs computation on every 64KB data (it is mentioned as required LSPM size). Figure 3.4 shows the correlation between maximum job size and number of ACCs. The job size exponentially drops as more ACCs are integrated.

Figure 3.4: Limited memory and smaller job size.

Figure 3.5: ISR exposed to the processor

The synchronization load on the processor: Looking back to the Figure 1.8 in Section 1.2.2 the processor is involved for 6 steps of 9 steps of a single ACC transaction. It means linear increase
CHAPTER 3. SCALABILITY IMPLICATION ON ACMP PERFORMANCE

(6x) of load on the processor with a linear increase in number of ACCs. This increasing load
of synchronization on the processor gets heavier (exponentially) due to limited size of on chip
memory and more number of smaller jobs as Figure 3.5 shows.

Linear increase of load on the communication fabric, DMA, and memory and exponential
increase of load on the processor core result in highly utilized resources as more ACCs are integrated.
With highly utilized resources, ACCs have to wait and can not be well-utilized.

3.1.2 Resources Utilization: Analytical Modeling

According to the impact of increasing ACCs on the limited shared resources, we have
derived a first-order mathematical model to estimate the resources utilization in conventional ACMP
architectures.

For the sake of simplicity, we assume an equal job size and processing duration for each
ACC. In addition, we assume a single processor core operating at $Freq_{Proc}$ with a light-weight OS
with $Latency_{ISR}$ to coordinate the whole system (no computation). In our assumptions, each ACC
performs computation on 64KB data (1B per cycle at $Freq_{ACC}$) and have double buffer SPMs to
overlap its computation with communication. We also assume that the communication fabric is a
multi-layer AMBA AHB with 4 or 8 channels of 32bit width. Each communication channel has a
dedicated DMA. The SPM size and job size vary with number of ACCs.

Given the assumptions about the architecture and with different number of ACCs ($Num_{ACC}$),
the latency of a single stage $Latency_{Pipe}$ of the execution pipeline is calculated by (3.1).

$$Latency_{Pipe} = \max(P, S, R) + Latency_{Synch} + Latency_{Arb} \quad (3.1)$$

$$Latency_{Synch} = Num_{ACC} \times 3 \times Freq_{Proc} \times Latency_{ISR} \quad (3.2)$$

$Latency_{Pipe}$ is calculated as summation of the maximum between processing ($P$), data
sending ($S$), data receiving ($R$) – as all happen in parallel –, the latency of bus arbitration ($Latency_{Arb}$),
and the synchronization latency in the processor core ($Latency_{Synch}$). For simplicity, we assume
a constant $Latency_{Arb}$ (note: we consider contention for $S$ and $R$). $Latency_{Synch}$ represents
synchronization overhead on the processor(s), estimated by (3.2). It depends on the processor’s
frequency ($Freq_{Proc}$), ISR latency ($Latency_{ISR}$), and the number of simultaneous interrupt
requests. We assume a constant $Latency_{ISR}$ and 3 interrupts per each job (as illustrated in Figure 1.8).
Following the pipeline fashion, (3.3) calculates the system execution time.
CHAPTER 3. SCALABILITY IMPLICATION ON ACMP PERFORMANCE

\[ Time_{Exec} = (Num_{Jobs} + Num_{ACC} - 1) \times Latency_{Pipe} \]  

(3.3)

\[ Latency_{R/S} = Job_{size} \times \left( \frac{Bus_{layers}}{Bus_{freq}} + \frac{Mem_{ports}}{Mem_{freq}} \right) \]  

(3.4)

\[ Latency_{P} = \left( \frac{Job_{size}}{ACC_{freq}} \right) \]  

(3.5)

The latencies of \( R \) and \( S \) depend on the number of memory ports (\( Mem_{ports} \)) and bus layers (\( Bus_{layers} \)) as calculated in (3.4) (symmetric for both \( R \) and \( S \)). In contrast to communication, computation latency is fairly constant and only depends on the \( Job_{size} \) for a fixed frequency of \( ACC \) (highlighted in (3.5)).

We assume the same \( Job_{size} \) for all \( ACCs \), only depending on the available SPM. The total on-chip memory is equally distributed across all \( ACCs \). Since the maximum on-chip memory size is limited, and each \( ACC \) needs an own SPM, the SPM size shrinks as more \( ACCs \) are integrated.

To complete the model, our model calculates the processor utilization (\( Proc_{Util} \)), communication bandwidth (\( CommBW \)), and \( ACC_{Util} \). The processor utilization is estimated based on its activity defined as the time to serve the ISR as (3.6).

\[ Proc_{Util} = \frac{Latency_{Sync}}{Latency_{Pipe}} \times 100 \]  

(3.6)

The communication bandwidth is estimated according to the amount of \( ACCs \)'s data transferred over the communication fabric as (3.7), and the \( ACC_{Util} \) is also calculated based on the average computation load on \( ACCs \) as (3.8).

\[ CommBW = \frac{TotalComm}{Time_{Exec}} \]  

(3.7)

\[ ACC_{Util} = \frac{P}{Latency_{Pipe}} \times 100 \]  

(3.8)

3.1.3 Resources Utilization: Analytical Results

Assuming a conventional ACMP architecture with 1GHz for \( Freq_{Proc} \), 20000 cycles as \( Latency_{ISR} \), 200MHz for \( Freq_{ACC} \), and 1GB total data to process, Figure 3.6 plots the processor utilization and communication bandwidth. We experiment two configurations for total memory sizes includes 1M and 16M, and two interconnect configurations include Multi-Layer AHB with 4 and 8 layers.
Figure 3.6: Processor utilization and communication BW with increasing number of ACCs.

The core utilization significantly increases with more ACCs as Figure 3.6a shows. The utilization is higher in the 1MB configuration, as it results in a smaller job size and more jobs. The utilization levels off with more than 10 ACCs as the processor core becomes the bottleneck. The synchronization latency \( \text{Latency}_{\text{Synch}} \) increases and the synchronization requests are being serialized. The \( \text{Mem:16MB, 4-AHB} \) is more linear and does not saturate the processor indicating that a different resource is the bottleneck.

Figure 3.6b plots the communication bandwidth (BW). The configuration \( \text{Mem:16MB, 4-AHB} \) saturates the whole fabric having a flat BW of 2400MB/s with any number of ACCs. Even 8 layers saturate at 4800MB/s. However, with more than 12 ACCs, processor serialization becomes more dominant and bandwidth drops. In the 1MB configurations (both 4 and 8 layer), the communication fabric is less utilized as the processor core is drowned in synchronization overhead due to the smaller jobs.

With over-utilized processor core and saturated communication fabric, we expect to have ACCs under-utilized as ACCs have to wait for processor and communication fabric. Figure 3.7 plots ACC utilization over increasing number of ACCs. The ACC utilization drops for all configuration significantly as the number of ACCs increases beyond a few. The reduction is more pronounced for the 1MB configuration and 4-AHB communication channels. ACCs do not receive their data in time for processing most of the time and stay idle. Utilization is bounded by system resources limitations: (a) only 4 or 8 simultaneous transfers through the communication fabric, and (b) interrupts requests serialization in the host processor. With larger SPMs in the 16MB setting, ACCs synchronize less.
often, thus the effect of (b) is less pronounced.

![Graph showing ACC utilization with increasing number of ACCs for different constant amount of resources.](image)

Figure 3.7: ACC utilization with increasing number of ACCs for different constant amount of resources.

Our analytical explorations reveal the ACMP scalability challenge. As more ACCs are integrated, three main bottlenecks appear: (1) Significant load to the host processor for synchronizing/scheduling ACCs, (2) Large volume of redundant communication traffic for exchanging the streaming data across the ACCs, and (3) Large on-chip memory dedicated for ACC’s SPM and shared memory space to hold the streaming data under processing. Thereby, with increasing ACCs, shared resources over-utilize and overshadow ACCs’ benefits.

Toward an efficient design that achieves maximum benefits of ACCs, there is a need for a design space exploration (DSE) of all possible designs with respect to the number of ACCs and kernels mapping on ACCs. The size of the design space increases exponentially with the number of ACCs and number of kernels. Therefore, an analytical performance model of ACMPs is required to explore the whole design space effectively. To this end, next section presents ACMPerf, an analytical performance model of ACMP.

### 3.2 ACMPerf: Analytical Performance Model of ACMP

ACMPerf is an analytical performance model of processor-centric ACMP architectures capturing primary system-level contributors. The primary ACMP system-level performance contributors include: (1) ACCs’ acceleration of processing, (2) ACCs’ data access to the communication fabric, DMA, and memory, and (3) ACCs’ synchronization and control by the processor(s). (2)
and (3) are architectural overhead as they are imposed by the platform architecture. They could overshadow the ACCs’ acceleration of processing.

In preparation for analysis, we introduce definitions for application, architecture, and mapping (illustrated in Figure 3.8).

**Application:** A streaming application is captured in DF model and represented as a graph $G(P,E)$ (top of Figure 3.8). $P$ is the set of processing kernels ($\{P_1, P_2, ..., P_V\}$), each one with its own computation demand $\text{Comp}_{P_i}$, quantifying the amount of operations to execute that kernel. $E$ is the set of communication edges (indicating data dependency) across the kernels ($\{\bigcup_{i,j<V} e_{ij}\}$). The edge $e_{ij}$ from kernel $P_i$ to kernel $P_j$ has a communication demand $\text{Comm}_{e_{ij}}$, quantifying the amount of data (in bytes) produced by $P_i$ and consumed by $P_j$.

**Architecture:** An ACMP architecture is a set of processing elements ($PE$s) and a set of communication elements ($CE$s) (bottom of Figure 3.8). $PE$s include $N$ HW ACCs ($PE_{HW} = \{PE_{HW_1}, ..., PE_{HW_N}\}$) and $K$ SW cores ($PE_{SW} = \{PE_{SW_1}, ..., PE_{SW_K}\}$). All the SW cores are homogenous, and computation latency of each SW core is calculated based on the working frequency ($\text{Freq}_{PE_{SW}}$). We assume that the execution of a kernel on a $PE_{HW}$ is 20x faster than on a $PE_{SW}$. The system communication fabric is captured as a set of multiple ($M$) layers ($CE = \{CE_1, CE_2, ..., CE_M\}$). All buses have the same frequency ($\text{Freq}_{CE}$) and bandwidth ($\text{BW}_{CE}$).

**Mapping:** A mapping of an application to an ACMP platform is defined by two mapping functions: computation ($\text{Map}_{\text{Comp}} : (P, PE) \to (0 \text{ or } 1)$) and communication ($\text{Map}_{\text{Comm}} : (E, CE) \to (0 \text{ or } 1)$) to indicate if a kernel($P$)/communication edge($E$) is mapped to the given $PE/CE$ (bright dashed lines in Figure 3.8). When more than one kernel/edge is mapped to an element of $PE/CE$, the kernels/edges are executed/transferred serially.
A DF is executed in a pipeline fashion. Since the processing time for each actor may vary, a consumer actor cannot start execution until its producer actor provides the input data. Therefore, the actor with the longest execution time creates slack time for all other actors in pipeline, and dictates overall performance [87].

Considering pipelined execution, and given the definitions for application, architecture, and mapping, ACMPerf calculates the throughput based on the job latency. Job latency is the latency between two consecutive jobs produced and delivered. Throughput is calculated based on the job size (the amount of data) produced at each job latency.

The throughput is calculated through three incremental models: Comp, Comp+Comm, and Comp+Comm+Orch. They allow a differentiated view onto the performance contributors’ impact on throughput. Figure 3.9 shows how each performance contributors is modeled in the DF. The Comp model assumes an ideal architecture with zero architectural overhead and calculates throughput according to the ACCs’ acceleration of processing. Architectural overhead refers to the overhead for communication, synchronization, and control that results from the architectural restriction. The Comp+Comm model considers the impact of communication overhead on the throughput achieved by Comp model. The Comp+Comm+Orch model additionally captures the impact of synchronization and control overhead. Table 3.1 lists the main performance contributor(s) that each model captures to compute the throughput.
CHAPTER 3. SCALABILITY IMPLICATION ON ACMP PERFORMANCE

Table 3.1: Performance Contributors in Incremental Models

<table>
<thead>
<tr>
<th>Model</th>
<th>Acceleration Effects</th>
<th>Comm. Load</th>
<th>Orch. Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Comp+Comm</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Comp+Comm+Orch</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

3.2.1 Comp Model

The Comp model only considers processing. It assumes an ideal architecture where all data transfers, synchronization, and control occur simultaneously, and instantly. As Figure 3.9a shows, this model uses the original DF annotated with mapping-specific processing delays, without any architectural restrictions. As a result it yields the lower bound of job latency and the upper bound of throughput. The job latency depends on critical PE, which takes the longest execution time determining the performance. The execution time (latency) of each \( PE_{HW} \) and each \( PE_{SW} \) is estimated based on the computation demand of the kernels mapped to them as per (3.9) and (3.10).

Then, the critical HW ACC or the critical SW core, each one that takes longer determines the job latency as (3.11). Following the estimation of job latency, and based on the job size, (3.12) calculates the throughput. The average ACC utilization is computed as the ratio of average ACCs’ processing time to job latency as (3.13).

\[
L_{PE_{HW}} = \frac{\sum_{i=1}^{V} Comp_{P_{i}} \times Map_{Comp}(P_{i}, PE_{HW_{n}})}{Freq_{SW} \times 20} \tag{3.9}
\]

\[
L_{PE_{SW}} = \frac{\sum_{i=1}^{V} Comp_{P_{i}} \times Map_{Comp}(P_{i}, PE_{SW_{k}})}{Freq_{SW}} \tag{3.10}
\]

\[
L_{ACMP} = \max(L_{PE_{HW}}, L_{PE_{SW}})
\]

\[
L_{PE_{HW}} = \max_{i=1}^{N} L_{PE_{HW_{i}}} \tag{3.11}
\]

\[
L_{PE_{SW}} = \max_{i=1}^{K} L_{PE_{SW_{i}}}
\]

\[
Throughput_{ACMP} = \frac{\text{job size}}{L_{ACMP}} \tag{3.12}
\]

\[
Util_{ACC} = \frac{\sum_{i=1}^{N} L_{PE_{HW_{i}}}}{L_{ACMP}} \tag{3.13}
\]

46
CHAPTER 3. SCALABILITY IMPLICATION ON ACMP PERFORMANCE

3.2.2 Comp+Comm Model

This model adds communication overhead to the \textit{Comp} model to see how it reduces the throughput as compared to the upper bound. The communication overhead is modeled as new nodes (in blue) in Figure 3.9b. Each communication exposed to the system communication fabric is modeled by two added blue communication nodes. One for transfer into the ACC’s SPM and one for transfer out of the ACC’s SPM. Communication edges that cross HW/SW boundary (between $PE_{SW}$ and $PE_{HW}$) are exposed, as well as edges between HW mapped actors (between $PE_{HW}$ and $PE_{HW}$). The edges across SW kernels do not impose any overhead on the communication fabric.

The latency of each \textit{CE} is calculated based on the communication demand of the edges mapped to them, as well as \textit{Freq}_{CE} and \textit{BW}_{CE} as \ref{eq:3.14}.

\[
L_{CE_m} = \frac{\sum_{i=1}^{E}(Comm_{e_i} \ast Map_{Comm}(e_i, CE_m))}{BW_{CE} \ast \textit{Freq}_{CE}} \tag{3.14}
\]

The slowest $PE_{SW}$, $PE_{HW}$, and $CE$ determines job latency as \ref{eq:3.15}. The system throughput (\ref{eq:3.12}) and average ACC utilization (\ref{eq:3.13}) are calculated based on job latency.

\[
L_{ACMP} = \max(L_{PE_{HW}}, L_{PE_{SW}}, L_{CE}) \tag{3.15}
\]

\[
L_{CE} = \max_{j=1}^{M} L_{CE_m} \tag{3.15}
\]

3.2.3 Comp+Comm+Orch Model

This model adds the synchronization and control overhead of ACCs which we term as \textit{orchestration}. The orchestration nodes (small white) represent 3 interrupts based on \ref{Figure 1.8 in Section 1.2.2}: Two interrupts for \textit{Mem copy done} (depending on direction) and one interrupt for \textit{processing finished}.

\ref{eq:3.16} calculates total orchestration latency ($L_{Orch}$) based the number of ingoing and outgoing edges of each $PE_{HW_i}$ ($InEdge_{PE_{HW_i}}$ and $OutEdge_{PE_{HW_i}}$, respectively), and $L_{ISR}$. $L_{ISR}$ denotes the average latency for interrupt service routine (ISR).

\[
L_{Orch} = (\Sigma_{i=0}^{N-1}(1 + InEdge_{PE_i} + OutEdge_{PE_i})) \ast L_{ISR} \tag{3.16}
\]

\[
L_{PE_{SW_k}} = \frac{L_{Orch}}{K} + \frac{\sum_{i=1}^{N} \max{Comp_{P_i} \ast Map_{Comp}(P_i, PESW_k)}}{Freq_{SW}} \tag{3.17}
\]
CHAPTER 3. SCALABILITY IMPLICATION ON ACMP PERFORMANCE

We assume that the orchestration overhead is evenly distributed across all $PE_{SW}$. With this, the latency of each $PE_{SW}$ is updated as (3.17). This impacts the overall job latency ((3.15)), throughput ((3.12)) and ACC utilization ((3.13)).

3.3 ACMP Architecture: Analytical Results

![Figure 3.10: Synthetic application captured in DF.](image)

This section explores the impact of architectural overhead on the achievable ACC benefits for one example synthetic streaming application (shown in Figure 3.10).

The computation demand of each node is denoted as percentage of total demand (10 MCycle). The communication demand of each edge is a percentage of the problem size (4 MBytes). Primary resource constraints are 8 MB on-chip memory working at 200Mhz, Multi-Layer (ML) AHBs with four parallel 32-bit read/write channels working at 200MHz, ACCs operating at 100MHz, with 20x faster compared to processor core (for simplicity, it only contains one processor core) working at 500MHz.

To investigate the impact of scalability limitations, we increase the number of ACCs from 0 to 14. Considering all mappings of kernels to ACCs yields $2^{14}$ design points. To aggregate the results, we sort the design points by $ACC Comp Coverage$. $ACC Comp Coverage$ is the ratio of processing performed in ACCs over the total processing defined in (3.18).

$$ACC Comp Coverage = \frac{\sum_{i=1}^{J=N} \sum_{j=1}^{J=V} (Comp_{P_j} \cdot Map_{Comp}(P_j, PE_i))}{\sum_{v=1}^{V} Comp_{P_v}}$$  (3.18)

Figure 3.11, Figure 3.12, and Figure 3.13 show the $ACC utilization$ and $throughput$ of the ACMP architecture running the synthetic application for three models.
CHAPTER 3. SCALABILITY IMPLICATION ON ACMP PERFORMANCE

Figure 3.11: Comp Model

Figure 3.12: Comp+Comm Model

Figure 3.11a and Figure 3.11b show the upper bound of ACC utilization and throughput as the Comp model includes no architectural overhead. With increasing ACC computation coverage, the ACC utilization increases exponentially by 30x on the average, so does the system throughput (performance). The ACC(s) by itself is 20x faster than the processor core. Additionally, increasing number of ACCs and computation parallelism alleviate the impact of Amdahl’s law on the system throughput. In 0% ACC computation coverage, all computation is executed serially on the processor core. By increasing ACC computation coverage and offloading more computation to be executed on ACCs in parallel, less computation is executed on the processor core boosting the system performance.

Figure 3.12a and Figure 3.12b add the communication overhead (Comp+Comm model). The data movement over the communication fabric enforces ACCs to wait, and underutilized. The
CHAPTER 3. SCALABILITY IMPLICATION ON ACMP PERFORMANCE

ACC utilization dramatically drops by 8x (from 32% to 4%) and results in reduced throughput down to about 8% of the upper bound (from 1600 MB/s to 130 MB/s). The communication fabric saturates and becomes the bottleneck starting with 70% ACC computation coverage, and throughput drops.

Taking into account the ACCs’ orchestration overhead (Comp+Comm+Orch model), the ACC utilization is more reduced and will reach to 3 MB/s as Figure 3.13a shows. Accordingly, throughput is more reduced from 130 MB/s to 78 MB/s (Figure 3.13b). With more ACCs, the size of SPMs per ACC decreases that leads to more number of jobs and more orchestration load. The orchestration load shifts the saturation point from 70% (where throughput drops in Comp+Comm model) to 65%.

The results show that the conventional ACMPs may suffice for small ACC computation coverage and sparse integration of ACCs, while not for denser integration of ACCs. Even though in a case when all the computation is mapped to ACCs (100% ACC computation coverage), not much of the acceleration is achieved in overall application throughput. This roots in the underlying platform, and not the kernel acceleration itself. In order to improve scaling with larger number of ACCs, new platforms are necessary.

In sparse integration of ACCs, there exist only connections of (mostly) processor-to-processor, ACC-to-processor, and processor-to-ACC. In processor-to-processor connections, both producer and consumer kernels are mapped to processor(s). In ACC-to-processor connections, the producer kernel is mapped to an ACC, and the consumer kernel is mapped to a processor, and vice versa in processor-to-ACC connections. Conversely, with more ACCs, the likelihood of direct ACC-to-ACC connections increases. In a direct ACC-to-ACC connection, both producer and consumer
CHAPTER 3. SCALABILITY IMPLICATION ON ACMP PERFORMANCE

kernels are mapped to individual ACCs. For instance, in the experiment above, with increasing number of ACCs from 0 (no kernels on ACCs) to 14 (dense ACC computation coverage as all kernels are mapped to ACCs), the number of direct ACC-to-ACC connections increases from 0 to 18. Increasing number of direct ACC-to-ACC connections, and still increasing architectural overhead indicate that conventional ACMP architectures do not support direct communication across ACCs.

The primary source of this challenge is lack of precise semantics to communicate with ACCs, and relying upon general platform to connect many ACCs on a single chip. To address this problem, first we need to identify the semantics of communication and understand how these semantics are realized in conventional platforms. Then, we can revisit the deficiencies of semantics realization in conventional ACMPs and propose a platform that is tailored to integrate many ACCs effectively.

3.4 Summary

This chapter addressed the scalability limitation challenges in conventional processor-centric ACMP architectures. With a processor-centric view, ACCs are treated as slaves devices requiring all shared resources (processor core, communication fabric, DMA, and memory) for every single transaction. Thus, conventional processor-centric architectures may suffice for sparse integration of ACCs, but not for denser integration of ACCs due to increasing load on shared resources. As more ACCs are integrated on a chip, processor is over-utilized to synchronize ACCs, and communication fabric is saturated to transfer data among ACCs. Moreover, the pressure on shared memory increases as each ACC needs an own private memory.

This chapter first provided an analytical model to estimate the impact of increasing ACCs on shared resources, and how shared resources over-utilize to handle ACCs’ communication. Then, it provides ACMPerf, an analytical performance model of ACMPs to estimates how the resources (over)utilization can overshadow ACCs’ benefits. Using ACMPerf, we observed diminishing ACCs’ benefits with more ACCs, and even with more number of direct ACC-to-ACC connections that are supposed to happen directly without imposing any overhead to shared resources. We observed lack of precise semantics for ACC communication, and accordingly lack of efficient realization of the semantics to support direct connections among ACCs.
Chapter 4

ACC Communication Semantics

The goal of this chapter is to identify and formalize the semantics of ACC communication, and give insights to system designers on how to re-architect ACMP platforms to achieve more benefits of ACCs. To this end, this chapter first studies different aspects of communication with ACCs and then, formalizes the semantics of ACC communication. At the end, and to shed light upon the sources of ACMP scalability limitations, it will discuss how inefficiently the conventional ACMP architectures implement the ACC communication semantics that lead to resources bottlenecks and scalability limitations.

4.1 ACC Communication Aspects

Classic for ACCs, we assume that at each point of time, processing inside an ACC is isolated to the current job. The aspects of ACC communication to receive the input job and sends the resultant job after processing can be primarily observed as data access model and synchronization by looking back to the current realization in Figure 1.8 (Section 1.2.2).

Each ACC has separate input (I) and output (O) buffers that determine the job size. The provided buffers should not be smaller than the granularity of ACC’s specific functionality.

Figure 4.1 outlines the sequence hinting to a formalization. For simplicity, single buffering (using $I_0$, $O_0$) is assumed. Table 4.1 illustrates the access rights in two phases. In $Phase_0$, the ACC has exclusive accesses to $I_0$ and $O_0$ to read input and write output of current processing (current job). Then ACC is idle in $Phase_1$, and $I_0$ and $O_0$ are accessed by bus for data copy/transfer.

To start processing in $Phase_0$, the ACC needs to receive both $IReady$ (indicating that a new buffer with input data is available) and $ORead$ (indicating that an empty output buffer is available).
At this point, ACC gains exclusive random access (RA) to $I_0$ and $O_0$ for processing. When finished, the ACC issues a finished signal, and Phase_0 switches to Phase_1. In Phase_1, the output data of current processing is copied from $O_0$ to the shared memory, and $I_0$ is filled with the input data for next processing. In Phase_1, the ACC is no longer allowed to access either $I_0$ and $O_0$. Consequently, the ACC is idle and waiting for $IReady$ and $ORead$ before starting the next processing phase.

Table 4.1: Access rights in single buffering approach

<table>
<thead>
<tr>
<th>Phase</th>
<th>InBuf 0</th>
<th>OutBuf 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase 0</td>
<td>EX Access</td>
<td>EX Access</td>
</tr>
<tr>
<td>Phase 1</td>
<td>Bus Access</td>
<td>Bus Access</td>
</tr>
</tbody>
</table>

Table 4.2: Access right in double buffering approach

<table>
<thead>
<tr>
<th>Phase</th>
<th>InBuf 0</th>
<th>InBuf 1</th>
<th>OutBuf 0</th>
<th>OutBuf 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase 0</td>
<td>EX Access</td>
<td>Bus Access</td>
<td>EX Access</td>
<td>Bus Access</td>
</tr>
<tr>
<td>Phase 1</td>
<td>Bus Access</td>
<td>EX Access</td>
<td>Bus Access</td>
<td>EX Access</td>
</tr>
</tbody>
</table>

To reduce the ACC’s idle time and overlap data transfer/copy and processing, double buffering is employed. While the ACC has exclusive access to one buffer set, the other buffer set is emptied/filled simultaneously. When the ACC is finished with processing on one buffer set, and the other buffer set is filled / emptied, the buffer access is swapped.
Table 4.2 illustrates the access rights in two phases. In Phase0, the ACC has exclusive accesses to I0 and O0 to read input and write output of current processing (current job). At the same time, I1 and O1 are accessed by bus for data copy (input data for next processing and output data of previous processing). Conversely, in Phase1, ACC has exclusive access to I1 and O1, while I0 and O0 are accessed by bus. The phases switch when processing and bus access (read/write accesses) all are done. Figure 4.2 visualizes the phases with signaling and parallel actions when double buffering is exploited. Here, data transfer and processing occur concurrently (in contrast to single buffering, Figure 4.1 operating sequentially).

With double buffering when the processing of the current input in one buffer set is finished, the ACC has to wait until the data copy/transfer from the other buffer set finishes. This waiting period can be considerable and in the meantime the buffers cannot be touched. To reduce this waiting time and improve the performance, triple buffering (three sets of buffer) can be employed. When the ACC finishes processing on one of the first buffer set, there is no need to wait for the copy/transfer completion of the second buffer set and it can start processing on the third buffer set.

Table 4.3 illustrates the access rights in three phases. In each phase, the ACC has exclusive accesses to one of the buffer sets, while the other two buffer sets are accessed by bus. Figure 4.3 visualizes the phases with signaling and parallel actions when triple buffering is exploited. Here, when the ACC finishes processing of the current input data in one buffer set, it can switch to
### Table 4.3: Access rights in triple buffering approach

<table>
<thead>
<tr>
<th>Phase</th>
<th>InBuf 0</th>
<th>InBuf 1</th>
<th>InBuf 2</th>
<th>OutBuf 0</th>
<th>OutBuf 1</th>
<th>OutBuf 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase 0</td>
<td>EX Access</td>
<td>Bus Access</td>
<td>Bus Access</td>
<td>EX Access</td>
<td>Bus Access</td>
<td>Bus Access</td>
</tr>
<tr>
<td>Phase 1</td>
<td>Bus Access</td>
<td>EX Access</td>
<td>Bus Access</td>
<td>Bus Access</td>
<td>EX Access</td>
<td>Bus Access</td>
</tr>
</tbody>
</table>

processing the next input data without waiting for the previous output data copy completion (in contrast to double buffering in Figure 4.2 which can lead to ACC under-utilization).

![Figure 4.3: Sequence of data access and synchronization in triple buffer.](image)

With triple buffering, since one of the buffer set is always complete, the ACC never has to wait for the data transfer completion. Consequently, the ACC and the DMA (for data transfer) are completely independent and can run at their own paces. After the ACC has finished processing two buffers, it should wait until the DMA is finished with transferring the first buffer’s data. It is similar to First In, First Out (FIFO) queue with the length of three. If the rate mismatch of DMA and ACC’s processing time is in the order that cannot be fully overlapped together, more buffers can be
Figure 4.4: Sequence of data access and synchronization in a FIFO with the length of n.

introduced to the FIFO queue to reduce the waiting time.

Figure 4.4 visualizes the more general case when there is n-buffer sets in a FIFO. In this way, the ACC and DMA have more freedom to work on their own paces. There are n phases, and in each phase, the ACC is processing the data in one buffer set. Therefore, there are always \( n - 1 \) buffer sets for the ACC to process until the DMA is finished with the data transfer in one buffer set.

The analysis up to now highlighted the importance of data access model and synchronization. The data access model refers to data storage (in shared memory and SPMs), and transfer (through the communication fabric). The synchronization signals schedule processing of ACC and its data transfer: initializes ACC computation and communication, as well as interrupts indicating ACC computation or communication tasks completion.
4.1.1 Single ACC

In the context of single ACC, data storage as part of data access model and synchronization are required. For data storage, double buffers are used that is more generalized as an extended FIFO with \(N\) elements (Figure 4.5). An extended FIFO with 1 element yields single buffer semantics. An extended FIFO with 2 elements yields double buffer semantics. Deeper FIFOs are also possible. The data granularity of each FIFO element is the size of one job. In deviation from a standard FIFO, the FIFO’s head element allows random read and write access through the synchronization signals \(I\text{Ready}\), \(O\text{Read}\), and \(\text{Finished}\).

Random Access (RA) to the head element (i.e. \(I_0\)) is granted through the \(I\text{Ready}\). The ACC signals that it has finished consuming \(I_0\) through the \(\text{Finished}\), upon which the FIFO can produce a new head element. Symmetrically, the same holds true for the output path. The tail element (\(O_0\)) has random access. The \(O\text{Read}\) signals the availability of an empty tail, and the \(\text{Finished}\) commits the tail element. After receiving both \(I\text{Ready}\) and \(O\text{Read}\), the ACC has exclusive access to the pair of input (\(I_0\)) and output (\(O_0\)) buffers for processing. Upon processing completion, the ACC consumes with the \(\text{Finished}\) \(I_0\) and commits \(O_0\).

4.1.2 Communicating ACCs

Considering two communicating (producer/consumer) ACCs, there is a need for data granularity adjustment, and data type might need to be adjusted. Figure 4.6 shows the communication between a producer ACC (\(ACC_P\)) and a consumer ACC (\(ACC_C\)) with conceptual signals.

The output granularity of \(ACC_P\) may differ from the input granularity of \(ACC_C\). As a result, granularity translation is required. In the simplest case (assuming identical output and input data types, and contiguous data placement), granularity adjustment could be realized when assuming word-wise transfer and counting of received bytes. Then, the input FIFO can announce a new input element once sufficient bytes have been received. Granularity adjustment enables splitting/combining one/multiple data element(s) into multiple/one element(s). Figure 4.7 visualizes
CHAPTER 4. ACC COMMUNICATION SEMANTICS

Figure 4.6: Communicating ACCs with the extended FIFO semantics.

(a) Multiple smaller jobs combined out as a bigger job

(b) A big job split into multiple smaller jobs

Figure 4.7: Granularity management

granularity adjustment: Figure 4.7a for the case of combining multiple smaller job to make a bigger job and Figure 4.7b for the case of splitting a big job to multiple smaller job.

The data type translation is also needed, if $ACC_P$’s output data type differs from $ACC_C$’s input data type (referring to Figure 4.6). This data type translation can be considered similarly to marshalling in a networking protocol. $ACC_P$ and $ACC_C$ have to agree upon a common data representation/type. Depending on differences between data representation/type, additional storage may be needed, e.g. to account for differently stridden accesses, or varying order of parameters in the data stream. System-level design decisions can be made for the realization of marshalling.

The simplest design is removing the marshalling and enforcing a common network format, for instance byte. It means that ACCs have to produce data in the common network format and expect to receive data in that format. Thereby, ACCs lack portability and cannot work in other chip that has another networking format.

To provide portability to ACCs, there is a need to change the interfaces of ACCs and supply
them with marshalling units. Thereby, no matter of what the network format is, the marshalling unit in ACCs prepares the data in the format that ACCs expects to produce or consume, at the cost of area and power constraints. This area and power overhead can be reduced for the case that the output of an ACC is going to be consumed by another ACC. This way, the producer ACC can skip marshalling on the produced data and leave it for the consumer ACC to do marshalling when data is received.

4.2 ACC Communication Semantics

Based on the realization of a single ACC and the communication between a pair of producer-consumer ACCs in current ACMP architectures, this article defines the semantics of ACC communication as bellow:

- **Data Access Model** defines how/where data can be stored and transferred (i.e. to enable overlapping between data processing and data transfer).
- **Synchronization** defines when data needs to be accessed regarding the start and finish of processing.
- **Data Granularity** defines the minimum amount/size of data required for processing. The granularity depends on the specific ACC’s functionality.
- **Data Marshalling** refers to input and output data representation. The communication type/representation may differ from the processing type/representation, which then requires conversion.

Current ACMPs realize the semantics in a very simple and consequently inefficient way. Current ACMP architectures often use a *data access model* of double buffering and shared communication fabric (data storage and transfer respectively). *Synchronization* is also realized through the interrupt signals of DMAs, a memory mapped register(MMR) write to start ACC processing, and another interrupt indicating processing finished. Less attention is placed on *data granularity* and *data marshalling* as they are implemented on the processor. Therefore, all the shared resources, and specially processor are involved to realize the semantics. This processor-centric view leads to bottlenecks on the shared resources, and dramatic scalability issues as the number of ACCs increases. Therefore, there is a need to shift away from processor-centric architecture, and toward a more equal view between ACC and processor.
CHAPTER 4. ACC COMMUNICATION SEMANTICS

4.3 Summary

This chapter formalized the ACC communication semantics that include: synchronization, data granularity, data marshalling, and data access model. It identified inefficient realizations of the ACC communication semantics as origins of the resources bottlenecks and scalability limitations in conventional ACMP architectures. The conventional ACMP architectures have processor-centric views as they realize three semantics aspects of synchronization, data granularity, and data marshalling on processor, and the aspect of data access model on communication fabric/DMA and memory.

Toward a more scalable architectures tailored for many ACCs, there is a need for an equal view between ACC and processor making ACCs more autonomous to relieve the load on shared resources and relieve scalability limitations.
Chapter 5

Transparent Self-Synchronizing (TSS) ACCs

Current ACMP architectures have processor-centric view as they are built upon the assumption of sparse integration of ACCs. With processor-centric view, all shared resources, and especially processor are involved for any ACC communication.

However, the number of ACCs increases, so does the probability to have more number of direct ACC-to-ACC connections. For instance, in the experiment in Section 3.3, with increasing number of ACCs from 0 (no kernels are mapped to ACCs) to 14 (all kernels are mapped to ACCs), the number of direct ACC-to-ACC connections increases from 0 to 18. With the processor-centric view, there is no architectural support for direct ACC-to-ACC connections, and all shared resources will be still involved and limit the ACCs’ benefits.

To relieve the load on shared resources, there is a need to shift away from processor-centric view to more equal, peer view between ACCs and processor core(s). This chapter first introduces our proposed architecture template, Transparent Self-Synchronizing (TSS) ACCs. TSS adds autonomy to ACCs, and provides a local interconnect across autonomous ACCs to realize direct ACC-to-ACC connections independent of shared resources. Thus, TSS relieves the scalability limitations with more ACCs and more number of direct ACC-to-ACC connections.

To explore the improvement of TSS over conventional architectures, this chapter then uses ACMPerf (presented in Section 3.2), with excluding the load of direct ACC-to-ACC connections from the application model. As direct ACC-to-ACC connections are realized internally, without imposing any overhead to shared resources, the proposed TSS achieves more ACCs’ benefits than...
CHAPTER 5. TRANSPARENT SELF-SYNCHRONIZING (TSS) ACCS

that of conventional ACMP architectures.

5.1 Single ACC in TSS

![Diagram of Single ACC communication in TSS](image1)

Figure 5.1: Single ACC communication in TSS.

![Diagram of ACC-to-ACC communication in TSS](image2)

Figure 5.2: ACC-to-ACC communication in TSS.

Figure 5.1 outlines our TSS in the context of a single ACC. Processing is decoupled from communication in order to allow ACC designers to focus mainly on realization of the processing, and ignore the tedious and error-prone tasks of outside communication. The ACC itself only realizes the processing, standard modules realize communication semantics. These modules include an Input Control Management (ICM) unit for each input port, and Output Control Management (OCM) unit for each output port. ICM and OCM units provide double buffers and synchronization units to realize the semantics aspects of data access model and synchronization, respectively.

**Data access model** is internally realized through double input ($I_0, I_1$) and output ($O_0, O_1$) buffers to store ACC’s data. We used double buffering for easier comparison w current ACMPs.

**Synchronization** unit coordinates processing of jobs based on the double buffering semantics (for the $I\text{Ready}$ and the $O\text{Read}$ signals). It receives the finished signal from the ACC and issues the $I\text{Ready}$ and $O\text{Read}$ signals to notify the ACC about the availability of data in the buffers.
CHAPTER 5. TRANSPARENT SELF-SYNCHRONIZING (TSS) ACCS

Figure 5.3: Nodes with multiple input/output ports.

5.2 Communicating ACCs in TSS

To support direct ACC-to-ACC connections without imposing any load to shared resources, TSS includes a local interconnect, and additional units for data granularity and data marshalling adjustment inside the communication modules.

Data access model additionally includes a local interconnect to connect the OCMs of producer ACCs to the ICMs of their consumer ACCs. Figure 5.2 shows the interconnect as a cloud that could be designed differently to provide direct paths between pairs of producer-consumer ACCs.

Options for this interconnect network range from a bus-based communication to Network-on-Chip (NoC) with a trade-off between performance-power efficiency and flexibility. Our first interconnect option is a MUX-based interconnect; less flexible, less power consuming, and more utilized than a general NoC. A general NoC provides a complete connectivity across all ACCs, while not all of the connections lead to meaningful compositions of ACCs (ACCs to ACCs).

Our MUX-based interconnect inspired by industrial products, e.g. Analog Devices Pipelined Vision Processor [8] supports a sparse connectivity based on application requirements. For instance, in Figure 3.10, P4 and P7 do not communicate directly, hence no direct point-to-point connections is needed. Conversely, P3 communicates to both P1 and P7 and the interconnect should provide the point-to-point connections.

Synchronization units in producer and consumer sides connect together to issue the OReady and the IReady signals accordingly. As a result, the communicating ACCs self-synchronize each other without any need for processor core(s).
CHAPTER 5. TRANSPARENT SELF-SYNCHRONIZING (TSS) ACCS

Data granularity management units in both ICMs (on consumer side) and OCMs (on producer side) make fine tune job sizing based on both possible finest granularity of functionality, and the minimum job sizes of consumer ACCs and producer ACCs. In contrast to current ACMP architectures, the size of buffers in ICMs and OCMs is much smaller equal to minimal streaming data (job) required for processing. The ACCs are self-synchronizing and there is no need to processor(s) for synchronization. Therefore, there is no increasing orchestration load on processor(s) with going to the finest job size and consequently more number of transactions.

Data marshalling units on the producer sides (inside OCMs) serialize a filled buffer in the output format of the producer ACC into a flat byte stream. Similarly, the marshalling units on the consumer sides (inside ICM) receive the flat byte stream and creates a filled buffer in the consumer ACCs’ desired input format. The marshalling unit splits/collcts and reorders the data in bytes.

Overall, using ICMs/OCMs and the decentralized realization of synchronization, data granularity, and data marshalling eliminate processor(s) from direct ACC-to-ACC connections. Thus, ACCs become more independent of host processor(s), and the processor-centric view shifts toward a peer-controlled, non-discriminatory view. In addition, the local interconnect provides direct paths among ACCs (all potential paths from OCM units of producer ACCs to ICM units of consumer ACCs) to localize ACCs from the communication fabric and DMA.

To keep the design of OCMs and ICMs general, for each producer-consumer relation, one pair of OCM/ICM have to be supplied. Therefore, for multi-input nodes (nodes with more than one input) and multi-output nodes (nodes with more than one output), multi ICMs and multi OCMs are required, respectively. Multi-input nodes and multi-output nodes are frequently occur in vision algorithms such as the algorithms in the library OpenVX [37].

P3 is an example of multi-output nodes in Figure 5.10. P3 produces data for two consumers P1 and P7. Therefore, there is a need to have two OCMs for the ACC that runs P3. One OCM is connected to the ICM of the ACC that runs P1, and the other OCM is connected to the ICM of the ACC that runs P7. Conversely, P5 is is an example of multi-input nodes in Figure 5.10. P5 consumes data produced by P6 and P4. Thus, the ACC that runs P5 needs two ICMs to connect to the OCMs of ACCs that run P6 and P4. Figure 5.3a and Figure 5.3b show how a multi-output node and a multi-input node are realized with more OCMs/ICMs in TSS.
CHAPTER 5. TRANSPARENT SELF-SYNCHRONIZING (TSS) ACCS

5.3 Peer Processor Communication

With distributed communication modules (ICMs/OCMs) per ACC, and the local interconnect, producer ACCs internally communicate to their consumer ACCs composing a *chain*. A *chain* is a set of ACCs working in producer-consumer fashion in which the consumer in a direct ACC-to-ACC connection is the producer in another direct ACC-to-ACC connection. In TSS, all direct ACC-to-ACC connections are transparent to processor(s) and other shared resources.

![Gateway Architecture](image)

Figure 5.4: Gateway Architecture.

However, the first and last ACC of a chain have to communicate with processor(s). To avoid those ACCs’ communication with processor(s), we add the *gateway*. The *gateway* as the interface between TSS (a chain of ACCs) and shared resources feeds the chain, and then collects its resultant data. Additionally, the gateway is responsible to configure the MUX-based local interconnect to setup all required direct paths among pairs of producer and consumer ACCs.

Figure 5.4 shows the gateway: The bottom side is the interface of the gateway to the shared resources, and the top side is the interface of the gateway to the ACCs.

**BIF, MMR, Interrupt and control line:** The gateway has bus interfaces (BIF), an interrupt line to the processor(s), as well as a control/configuration unit with Memory-Mapped Registers (MMRs) to provide outside communication of the gateway with the processor(s) in a very similar way as described for ACCs in current ACMPs (Figure 1.8): DMA in, MMR write to start, interrupt signaling done, and DMA out.

In addition, once at the beginning, the processor(s) writes the configuration information
CHAPTER 5. TRANSPARENT SELF-SYNCHRONIZING (TSS) ACCS

Figure 5.5: Multiple Chains in TSS to Accelerate the Execution of the Mapping-aware Synthetic DF.

(a) Mapping-Aware DF

(b) Multiple Chains in TSS

(for MUXs) into the MMRs, and then the gateway configures the MUXs based on those information. For simplicity, we assume a constant configuration during application execution.

Gateway SPM: The gateway has input and output SPMs to store the input data for feeding the chain(s), and the resultant data from the chain(s), respectively. Similar to conventional ACMP architectures, the data transfers between memory and both input and output SPMs are performed via the communication fabric. However, these are the only visible data transfers to shared resources, and all the data transfers of direct ACC-to-ACC connections are transparent.

G-ICMs and G-OCMs: The gateway contains gateway OCMs (G-OCM) and gateway ICMs (G-ICM) enabling the gateway to talk to ACCs. G-ICMs and G-OCMs have double buffering; the double buffers inside G-OCMs feed the chain(s), and the double buffers inside G-ICMs gets resultant data of the chain(s). This is the responsibility of the synchronization unit to order feeding the chain(s) and collecting the resultant data of the chain(s).

The gateway also performs marshalling management and granularity adjustment on output (through OCMs) and input (through ICMs). A larger outside job (stored in input SPM) can be broken up into many smaller internal jobs to feed the double buffering inside the G-OCMs(s). Accordingly,
CHAPTER 5. TRANSPARENT SELF-SYNCHRONIZING (TSS) ACCS

the output SPM collects the smaller internal jobs out of the double buffering inside the G-ICMs(s).

This granularity adjustment is different from current ACMP platforms. Current ACMP platforms often have to operate at large job sizes to reduce the orchestration load on host processor(s). In TSS, the ACCs synchronize directly with each other thus allowing much smaller job sizes, reducing memory size (per ACC) requirements and orchestration load on processor(s).

For a chain, regardless of the number of its direct ACC-to-ACC connections, only the data transfers to/from gateway SPMs are exposed to shared resources. Therefore, a longer chain in TSS hides more number of direct ACC-to-ACC connections from shared resources and boosts power and performance efficiency. Additionally, TSS supports multiple chains of ACCs to support parallel execution. We provided four G-ICMs, four G-OCMs, four input/output gateway SPMs to support four chains within an application or multiple applications.

Figure 5.5b shows an example of multiple chains in one application with the given mapping of the synthetic application’s kernels on individual ACCs or processor in Figure 5.5a. The MUX-based interconnect plus OCM and ICM units placed for all potential producer-consumer relations, provide the point-to-point connectivity to realize up to four chains of ACCs as long as the chains do not conflict on the same ICMs or OCMs or MUXs’ active ports.

The multiple chains of ACCs with proper setting of MUXs inside TSS are as follows:

- Chain 1 realizes loop back, a multi-output node (ACC4) and a multi-input node (ACC1).
- Chain 2 realizes a multi-output node (ACC3).
- Chain 3 realizes a multi-output node (ACC3) and a multi-input node (ACC6).
- Chain 4 realizes a simple chain.

The benefits of TSS increase with small number of long chains: small number of chains to keep the data transfers between shared memory and input/output SPMs low, and longer chains to internally realize more number of direct ACC-to-ACC connections without imposing any load to shared resources. Conversely, without any ACC-to-ACC connections, TSS behaves identically to conventional ACMP architectures.

5.4 TSS System Integration

Figure 5.6 shows TSS integration to the host processor through the shared memory and communication fabric. TSS is practically only visible through the gateway. The gateway receives
CHAPTER 5. TRANSPARENT SELF-SYNCHRONIZING (TSS) ACCS

Figure 5.6: TSS system integration

the configuration information once from the processor(s), through the control bus, and sets up the
interconnect at the beginning. During the application execution, the gateway reads large input data
from shared memory, breaks it into smaller jobs to feed the chains. Conversely, it collects small
resultant data from the chains, and writes a larger resultant data to shared memory.

5.5 TSS: Analytical Evaluation

This section analytically estimates the TSS improvement over current ACMPs in view
of throughput and architectural overhead. For the purpose of comparison, the same formulation
presented in Section 3.2 is used, while the architectural overhead (data access and orchestration) for
direct ACC-to-ACC connections are masked. Figure 5.7 shows that the edges belong to \( E_{HW-HW} \)
(indicating direct ACC-to-ACC connections) are hidden in evaluation.

Figure 5.7: Comp+Comm+Orch Model

Figure 5.8a explores the TSS as compared to processor-centric ACMP architectures in
view of relative throughput for 0% to 100% ACC Computation Coverage. Relative throughput is
defined as TSS’s throughput improvement compared to processor-centric ACMPs. For small ACC
computation coverages, there are some design points with no improvement (relative throughput

68
CHAPTER 5. TRANSPARENT SELF-SYNCHRONIZING (TSS) ACCS

equals one). As the coverage increases by 100%, the throughput improves up to 10.8x. Both TSS and processor-centric ACMP architectures are the same in view of the first performance contributors (ACCs’ acceleration of processing). This improvement comes from improved relative architectural overhead. Relative architectural overhead is defined as TSS’s reduction in the data access and orchestration load compared to processor-centric ACMPs. \((5.1)\) and \((5.2)\) calculate the relative data access and orchestration load, respectively.

TSS internally realizes the load of \(E_{HW-HW}\). Thus, with increasing ACC computation coverage, as number of \(E_{HW-HW}\) increases, and number of \(E_{SW-HW}\) and \(E_{HW-SW}\) decrease, TSS outperforms conventional ACMPs more significantly.

\[
\text{Reduction}_{Comm} = \frac{Comm_{Total} - Comm_{E_{HW-HW}}}{Comm_{Total}} \\
Comm_{Total} = Comm_{E_{HW-HW}} + Comm_{E_{HW-SW}|SW-HW} \\
\text{Reduction}_{Sched} = \frac{|Edge_{Sched}| - |E_{HW-HW}|}{|Edge_{Sched}|} \\
|Edge_{Sched}| = |E_{HW-HW}| + |E_{HW-SW}|_{SW-HW} \\
\]

Figure 5.8b shows the relative architectural overhead of TSS as compared to processor-centric ACMPs. With small ACC computation coverage, when there are zero direct ACC-to-ACC connections, TSS imposes the same load to shared resources (relative architectural overhead equals 1), and behaves similarly to processor-centric ACMPs (relative throughput equals 1). With increasing ACC computation coverage to 100% (with 18 direct ACC-to-ACC connections), TSS exposes 14%
CHAPTER 5. TRANSPARENT SELF-SYNCHRONIZING (TSS) ACCS

of the total load that a processor-centric ACMP exposes to shared resources. Comparing Figure 5.8a and Figure 5.8b, TSS results in more improvement as its architectural overhead reduces.

5.6 Summary

To support direct ACC-to-ACC connections that have high occurring likelihood with integration of many ACCs, there is a need to shift away from the processor-centric view dominant in conventional ACMP architectures. Toward a more equal view between ACC and processor, this chapter proposed our novel architecture Transparent Self-Synchronizing ACCs (TSS). TSS added self-orchestration units to ACCs and local interconnect across ACCs to locally support communication semantics of all direct ACC-to-ACC connections.

With internal realization of direct ACC-to-ACC connections without posing any load to shared resources, TSS hides all ACCs from shared resources, and is visible to the outside as a gateway. The gateway gets the configuration information, the input data, and sends out the resultant data. Internally, based on the configuration information, TSS connects ACCs as chains together, and passes the input data through the first ACC of chains, and collects the resultant data from the last ACCs of chains and sends it to the memory. The gateway was designed to support more than one chain (within an application or multiple applications) of ACCs in order to more utilize ACCs.

To estimate the enhancement of TSS over conventional ACMPs, we used ACMPerf, but excluded the load of direct ACC-to-ACC connections in the application model. The results proved that with more number of direct ACC-to-ACC connections, less load is imposed to shared resources and more benefits are achieved by ACCs as compared to conventional architectures.
Chapter 6

Experimental Results: TSS vs. ACMP

This section evaluates the proposed TSS against conventional ACMP architectures. Both TSS and conventional ACMPs are architecture templates. By exploiting the same number of ACCs and same mapping of computation load to ACCs in both templates, the performance difference due to the architecture template can be measured.

As conventional ACMPs are built upon the assumption of sparse integration of ACCs, we first provide a set of experiments with moderate number of ACCs and constant computation coverage (about 50%) by ACCs. We will show that even with sparse integration of ACCs, the proposed TSS helps to realize more ACCs’ benefits than that of conventional ACMP architectures depending on the number of direct ACC-to-ACC connections and communication (and orchestration) load of the connections.

Then, we vary the number of ACCs and the computation coverage by ACCs (from 0% to 100%) to impose scalability limitations in conventional ACMP architectures. We will show significant TSS improvement over conventional ACMP architectures with increasing number of direct connections and communication and orchestration load on the connections. With more number of direct connections and communication load on the connections, TSS hides more load from shared resources, while conventional ACMPs increasingly overwhelm shared resources. Conversely, without any direct connections, TSS achieves the same benefits as conventional ACMPs.

6.1 Experimental Setup

For evaluation, we use Virtual Platforms (VPs) automatically generated by System-on-Chip Environment (SCE) [24] for both TSS and conventional ACMPs.
CHAPTER 6. EXPERIMENTAL RESULTS: TSS VS. ACMP

Through a python-based automation procedure, we specify both ACMP and TSS architecture models in SpecC system description language [33, 34]. The specification is based on applications characteristics in view of computation kernels and communication load passed across the kernels. Then, both models are refined in views of architecture, scheduling, network, and communication to generate VPs. Figure 6.1 abstractly visualizes the steps to generate a VP.

The general components of the VPs are: (1) an ARM9 core (running uCOS/II) simulated by an OVP ISS working at 500MHz clock frequency. (2) A multi-layer AMBA AHB (32 bit-width, 100MHz) with eight concurrent channels (4R and 4W). (3) Four DMA modules, one per one read/write channel. (4) A shared memory module with four access ports, the size of memory may differ based on the measurements’ goals and applications’ demand.

Figure 6.1: Virtual platform generation.

6.2 Constant Computation Coverage

This section investigates a constant computation coverage of about 50%. Here, the same amount of computation executes in software (processor) and hardware (ACC), respectively. We use eight streaming applications captured in abstract data flow (DF) models (generated by SDF3 [97]).
CHAPTER 6. EXPERIMENTAL RESULTS: TSS VS. ACMP

All DFs are acyclic, and they are communication intensive as the focus of this dissertation has been on efficient communication with ACCs.

Table 6.1 lists the properties of the DFs: (1) #Nodes, that is the number of computation kernels. (2) Comp[Min:Max], that defines the range of computation load of the kernels. (3) #Edges, that is the number of dependency edges among the nodes. (4) Comm[Min:Max], that defines the range of communication load passed among the nodes. (5) #ACCs, that is the number of allocated ACCs. (6) #direct ACC-to-ACC, that is defined as the number of edges whose both source and destination nodes are mapped to ACCs.

To define the mapping of computation kernels on hardware (ACC) or software (processor) for these experiments, we use the following strategy: (1) Computer-intensive kernels have highest priority to be executed on ACCs, (2) Ingoing / source and outgoing / destination kernels of communication-intensive edges have highest priority to be mapped to ACCs to have the maximum amount of communication load masked inside TSS. Note that other optimization strategies can be applied. However, as this dissertation focuses on the communication architecture template, automatic design space exploration is outside the scope.

Table 6.1: Applications characteristics (used in experiments with constant computation coverage by ACCs).

<table>
<thead>
<tr>
<th>App.</th>
<th>#Nodes</th>
<th>Comp[Min:Max]</th>
<th>#Edges</th>
<th>Comm[Min:Max]</th>
<th>#ACCs</th>
<th>#direct ACC-to-ACC</th>
</tr>
</thead>
<tbody>
<tr>
<td>H263Enc.</td>
<td>5</td>
<td>[6264:382419]</td>
<td>7</td>
<td>[384:38016]</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>MP3PB</td>
<td>4</td>
<td>[4:10000]</td>
<td>5</td>
<td>[4:4]</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Synthetic</td>
<td>23</td>
<td>[100:89112]</td>
<td>24</td>
<td>[100:1000]</td>
<td>13</td>
<td>13</td>
</tr>
</tbody>
</table>

6.2.1 Performance Evaluation

Figure 6.2 presents the absolute execution time in logarithmic scale (Figure 6.2a) as well as the relative speedup of TSS over the processor-centric ACMPs across all applications (Figure 6.2b). Overall, Figure 6.2b shows 3x performance improvement of TSS over the processor-centric ACMPs on average. The major sources of performance improvement are minimizing the interaction with the host processor(s), removing the unnecessary data copies between shared memory and local SPMs, and hiding ACC-to-ACC traffic from system communication fabric. Therefore, by increasing the
number of direct ACC-to-ACC links/edges over the total number of links/edges, TSS delivers higher speedups.

![Graph](image)

**Figure 6.2: System Performance and Relative Speedup.**

To show more insights about the sources of speedup in TSS, Figure 6.3 compares TSS and processor-centric ACMPs in view of load imposed on shared resources: memory, communication fabric, and processor. Figure 6.3a compares the memory demand between TSS and the processor-centric ACMPs. We assume a fixed but fairly large maximum on-chip memory: 2 MB on-chip memory. On average, TSS only requires 14% of original memory. TSS’s self-synchronization efficiently allows processing of small jobs without incurring high synchronization overhead. This enables ACCs to work on their minimal job sizes resulting in smaller SPMs per ACC. Figure 6.3b compares communication volume (in logarithmic scale) exposed to the system communication fabric. TSS significantly reduces the communication volume by hiding all ACC-to-ACC communications (e.g. TSS implementation of H263Encoder hides more than 95% of communication volume as compared to the processor-centric ACMP).

Figure 6.3c shows the number of interrupts (i.e. synchronization requests) sent to the host processor. On average, interrupt volume on TSS is about 3x less than processor-centric ACMPs. As the number of direct ACC-to-ACC increases, the fewer synchronization requests will be sent to the host processor. Figure 6.3d presents the execution time on the host processor dedicated for orchestration of ACCs (synchronization, granularity adjustment and marshaling). TSS significantly reduces orchestration time across all applications, 4.5x on average over the processor-centric ACMP (Figure 6.3e illustrates the relative comparison). The more TSS reduces the load on shared resources,
CHAPTER 6. EXPERIMENTAL RESULTS: TSS VS. ACMP

the more it improves performance as compared to processor-centric ACMPs.

However, there are some anomalies. For instance, in case of MP3 decoder, the speedup is very remarkable, while the load on shared resources has not reduced significantly. Similarly, in case of satellite, no significant speedup is achieved, while the load on shared resources are somehow reduced. In these cases, the impact of communication and orchestration load is dominated by the computation. In case of MP3 decoder, ACCs dominate the SW and high speedup is achieved, while in case of satellite, SW dominates the ACCs which is slower than ACCs.

6.2.2 Energy Consumption

For energy estimation, we construct an energy model that is driven by the VP statistics such as execution time in processor, ACCs, and data transfers. We use reported numbers in the literature [56, 65] including: 14 pJ per 8 bytes data transfer, 3.8pJ for each Kilo operation in the ACCs (as well as ICMs/OCMs in TSS), 300mW power for ARM9 running at 500 MHz and 30mW static power per each 100KB of on-chip shared memory.
CHAPTER 6. EXPERIMENTAL RESULTS: TSS VS. ACMP

Figure 6.6 shows both absolute energy consumption, and relative energy saving of TSS over processor-centric ACMPs. Overall, the energy saving is resulted from both performance and reduced load on shared resources. On average, TSS’s energy consumption is 8 times less than processor-centric ACMPs. The pronounced energy saving stems from faster execution time, significant reduction of the load on host processor, volume of data transfer on the system communication fabric, and on-chip memory. Therefore, for applications with higher speedup and lower load on shared resources, TSS saves more energy. For examples, TSS provides 22x energy saving for MP3Decoder with highest speedup, and up to 12x for H263Decoder with very small memory requirement and low load on processor.

![Energy Consumption and Energy Saving](image)

6.2.3 Area Overhead

As this work is based on VP simulation, it is too abstract to compare absolute area of both platform templates. Nonetheless, area considerations can be made based on memory and control structures. The local buffers in ICMs and OCMs to store streaming data under processing are primary area consumers in TSS. Conventional ACMP would also need SPMs for each ACC. With TSS’s self-synchronization, smaller jobs are possible without creating excessive orchestration load on the processor. Hence, buffers can be much smaller in TSS compared to conventional ACMPs. The interconnect for direct ACC-to-ACC connections is MUX-based (and sparse), it also consumes little area only. The gateway contributes somewhat to the area (mainly due to its bus interface), however it is still much less than a dedicated slave interface per ACC in a conventional ACMP. Overall, TSS leads to area saving due to reduction in on-chip memory as well as due to avoiding full bus interfaces.
CHAPTER 6. EXPERIMENTAL RESULTS: TSS VS. ACMP

for each ACC.

The experiments showed TSS’s improvement as compared to conventional ACMPs even with moderate number of ACCs where the impact of scalability limitations is low. TSS internally realizes all direct ACC-to-ACC connections, thereby ACCs’ allocation to have more number of direct ACC-to-ACC connections with high communication volume over the connections result in higher TSS’s improvement over conventional ACMPs.

6.3 Increasing Computation Coverage by ACCs

This section compares TSS against conventional ACMP architectures with increasing ACCs and computation coverage by ACCs to observe how TSS achieves more ACCs’ benefits than than of conventional ACMPs due to overcoming scalability limitations.

We used five streaming applications captured in DF models for evaluation: real applications include vision, LTE base band processing, MpegII, GMTII (all found in ([111], [93]), and the synthetic application introduced in Section 3.2. Figure 3.10. These applications are more complex with more intense communication load to measure the impact of scalability limitations on the maximum achievable performance benefits of ACCs. Table 6.1 lists the properties of the applications in view of: (1) number of processing kernels (#of Nodes), (2) number of edges/links among the kernels (#Edges), (3) range of computation load on kernels (Comp[Min:Max]), and (4) range of communication load on edges (Comm[Min:Max]).

Exhaustively simulating all possible mappings is prohibitive due to long simulation time. Using our analytical model, ACMPerf, we identify pareto optimal design choices as designs of interest. From those, we select six mappings with increasing ACC computation coverage from 0% to 100% for detailed simulation. Our metric to select the points was maximizing relative throughput of TSS over conventional ACMP architectures.

Table 6.2: Applications characteristics (used in experiments with increasing computation coverage by ACCs).

<table>
<thead>
<tr>
<th>App.</th>
<th>#Nodes</th>
<th>Comp[Min(M):Max(M)]</th>
<th>#Edges</th>
<th>Comm[Min(M):Max(M)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synth.</td>
<td>14</td>
<td>[0.2:2.4]</td>
<td>20</td>
<td>[0.04:0.32]</td>
</tr>
<tr>
<td>Vision</td>
<td>9</td>
<td>[0.02:51.84]</td>
<td>12</td>
<td>[0.02:6.22]</td>
</tr>
<tr>
<td>LTE BB. Proc.</td>
<td>10</td>
<td>[2e(-6):192e(-6)]</td>
<td>17</td>
<td>[2e(-6):12e(-6)]</td>
</tr>
<tr>
<td>MpegII</td>
<td>7</td>
<td>[3:1300]</td>
<td>10</td>
<td>[0.076:0.23]</td>
</tr>
</tbody>
</table>
CHAPTER 6. EXPERIMENTAL RESULTS: TSS VS. ACMP

6.3.1 Performance Evaluation

Figure 6.5 shows the relative throughput of TSS over ACMPs (TSS/ACMP) with increasing ACC computation coverage.

![Relative throughput of TSS over ACMPs](image)

Figure 6.5: Relative throughput of TSS over ACMPs WRT. computation coverage using VPs

Depending on the number of direct ACC-to-ACC connections (that is related to orchestration load), and the amount of communication (that is related to communication load), TSS outperforms ACMP architectures more significantly. In pure SW implementation with zero ACCs and zero direct ACC-to-ACC connections, there is no improvement. Conversely, in 100% ACC computation coverage with maximum number of direct ACC-to-ACC connections, TSS dramatically outperforms ACMPs. For instance, in case of GMTII with intensive communication (sixth row of Table 6.1), TSS improves the throughput up to 130x Figure 6.5(e). Similarly, in synthetic and vision applications with lighter communication demand, TSS improves throughput by 14x and 13x as Figure 6.5(a) and Figure 6.5(b) show, respectively.

However, increasing ACC computation coverage is not always leading to higher throughput improvement unless there is an increasing communication and orchestration load masked inside TSS. The application connectivity, the number of ACCs, and the kernels’ mapping on ACCs determine the communication and orchestration load that can be masked inside TSS. For instance, in case of vision (Figure 6.5(b)) and GMTII (Figure 6.5(e)), the relative throughput of TSS over ACMPs drops...
CHAPTER 6. EXPERIMENTAL RESULTS: TSS VS. ACMP

when computation coverage increases from 40% to 60%. Similarly, in case of LTE (Figure 6.5c) and MpegII (Figure 6.5d), the relative throughput drops when computation coverage increases from 60% to 80%. Relative throughput reduction is resulted from reduced number of direct ACC-to-ACC connections and reduced communication load masked inside TSS.

6.3.2 Energy Consumption

Using the same energy model as introduced in Section 6.2.2, Figure 6.6 shows energy consumption reduction of our TSS over conventional ACMPs when computation coverage increases.

![Energy Consumption Graphs](image)

Figure 6.6: Energy consumption reduction of TSS over conventional ACMP WRT computation coverage

In general with increasing ACC computation coverage, the conventional ACMP consumes more energy, while our TSS consumes less energy. The sources of energy saving in TSS are: (1) reduced load on shared resources, and (2) improved performance. With increasing ACCs and ACC computation coverage, conventional ACMPs face scalability limitations and cannot achieve most of the ACCs’ benefits. In addition, with more ACCs, shared resources are overwhelmed with ACCs’ load and dissipate a lot of energy. This is different in TSS, with more ACCs and higher likelihood of direct ACC-to-ACC connections, TSS increasingly reduces the load on shared resources and
improves both energy consumption and performance. For example in communication
intensive applications such as GMTII and LTE when maximum amount of communi-
cation and orchestration load is masked inside TSS in 100% computation coverage, TSS results in up to 158x and 78x
energy saving, respectively. Reduced orchestration load on processor and communi-
tication load on communication fabric and DMA lead to faster execution time.

To show the impact of both energy and performance, Figure 6.7 presents the relative Energy
Delay Product (EDP) of TSS as compared to conventional ACMP for the same set of applications
with increasing ACC computation coverage.

Figure 6.7: Relative Energy Delay Product (EDP) in TSS vs. ACMP WRT. computation coverage

Figure 6.7 shows that increasing ACCs and ACC computation coverage does not always
lead to more EDP improvement. EDP improvement depends on the number of direct ACC-to-ACC
connections and communication load of the connections. For instance, in case of MpegII, when
computation coverage increases from 60% to 80%, the number of direct connections reduces from 6
to 4, and the ratio of communication load that is masked inside TSS to total communication load
reduces from 83% to 43%.

To show more insights about the EDP improvement of TSS over conventional ACMPs,
next section measures the relative overhead of TSS as compared to conventional ACMPs.
CHAPTER 6. EXPERIMENTAL RESULTS: TSS VS. ACMP

6.3.3 Relative Overhead

With the same number of ACCs and same mapping of kernels on HW (ACCs) and SW (processor cores), both TSS and conventional ACMP architectures are identical in view of ACCs’ acceleration of processing. However, they are different in view of architectural overhead that is defined as the amount of communication and orchestration load imposed on shared resources. TSS internally realizes all direct ACC-to-ACC connections and as a result the communication volume and orchestration load of direct ACC-to-ACC connections will be hidden from shared resources.

![Graphs showing relative overhead of TSS over ACMPs WRT computation coverage](image)

Figure 6.8: Relative overhead of TSS over ACMPs WRT. computation coverage

Figure 6.8 illustrates the architectural overhead improvement of TSS over conventional ACMPs for all given five applications when computation coverage increase from 0% to 100%. In 0% computation coverage, when there is no ACCs, there is no overhead imposed on shared resources. Comparing Figure 6.5 and Figure 6.8, there is a strong correlation between relative throughput and relative overhead when computation coverage increases from 20% to 100%. The higher throughput is resulted from the lower architectural overhead. Conversely, with higher relative overhead, lower throughput is achieved.

However, there are some differences. For example, increasing computation coverage from 40% to 80% in case of GMTII results in higher relative overhead (Figure 6.8e), while the relative throughput is also increasing (Figure 6.5e). For these points, the detailed VP simulation indicated
that acceleration of processing dominates the other system performance contributors (communication and orchestration load). For these cases, the amount of communication and orchestration load that is masked inside TSS makes the processing acceleration dominant as compared to other performance contributors.

There is an additional reasoning. Previous measurements have shown anomalies as the computation coverage increases. As TSS benefits stem from communication and orchestration load improvements, the next set of experiments investigates the performance improvement over communication. For this purpose, we propose communication coverage defined as the amount of communication load of ACCs masked inside TSS to total communication load of ACCs. However, to avoid time consuming VP simulation, we use ACMPerf (in Section 3.2). To justify using the high level model, we first investigate the relative and absolute accuracy of our ACMPerf. Then we use the same set of applications in Table 6.1 and compare TSS against ACMPs with respect to the communication coverage.

6.4 Relative and Absolute Accuracy Investigation of ACMPerf

First-order analytical models such as our proposed ACMPerf (Section 3.2) in general are abstract descriptions of systems to provide fast evaluation. ACMPerf simplifies fast evaluation of both current ACMP and TSS architectures to provide design space exploration at very short time. However, due to the high level abstraction, it does not capture all dynamic events. For instance, it assumes constant arbitration time over the shared resources, or, similarly, constant arbitration time for accessing the gateway and local interconnect inside TSS.

To validate the accuracy of our proposed ACMPerf, we repeated the same experiments in Section 6.3 using the proposed ACMPerf. The results of ACMPerf are shown along with the results of VPs simulation in Figure 6.9.

In view of absolute accuracy that is defined as the average difference between simulation-result and analytical-result for each design point, our model is only 52% accurate. Since for each design point we have the relative throughput improvement of TSS over ACMPs, the correctness of a relative comparison that is defined as fidelity (relative accuracy) is sufficient. Relative comparison of simulation-result and analytical-result of any two design points are shown in Figure 6.9. The figures show that there is the same trend in relative throughput for both simulation-result and analytical-result when computation coverage increases from any x% to any y%, except for 60% to 80% in
CHAPTER 6. EXPERIMENTAL RESULTS: TSS VS. ACMP

Figure 6.9: Relative throughput of TSS over ACMPs WRT. computation coverage using ACMPerf and VP

LTE. Therefore our model has 98% fidelity, and it is suitable for a relative comparison of TSS over ACMPerf.

6.5 Increasing Communication Coverage by ACCs

This section compares TSS against conventional ACMPs with respect to the ACC Communication Coverage. ACC Communication Coverage is defined as the communication load of direct ACC-to-ACC connections to the total communication load (communication load over direct ACC-to-ACC connections plus communication load from/to ACCs) in (6.1). To aggregate the results, we sort the design points by ACC Communication Coverage.

\[
ACC\ Comm\ Coverage = \frac{(Comm)_{ACC\rightarrow ACC}}{(Comm)_{ACC\rightarrow ACC} + (Comm)_{ACC\rightarrow from\ SW}}
\]  

(6.1)

Using the analytical model, ACMPerf, we identify pareto optimal design choices as designs of interest. We select six design points for different communication coverage from 0% to 100% with the goal of maximizing relative throughput of TSS over conventional ACMPs. We showed the relative TSS’s improvement over conventional ACMPs in Figure 6.10 for the same set of applications.
in Table 6.2 as communication coverage increases. Different design points are evaluated in this set to investigate the influence of communication coverage.

Figure 6.10: Relative throughput of TSS over ACMPs WRT. communication coverage using ACM-Perf

With increasing communication coverage, TSS increasingly outperforms conventional ACMPs. The only difference of TSS with a conventional ACMP is the way that direct connections (orchestration and mostly, communication overhead that are imposed to shared resources) are handled. To find the impact of overhead reduction on the TSS’s relative throughout imporvement, we calculated the relative overhead of TSS over ACMP when communication coverage increases in Figure 6.11. Figure 6.11 shows the strong correlation between relative overhead reduction and relative throughout imporvement.

However, there are some differences between relative throughout and relative overhead. For instance, in the vision application (Figure 6.10b), when communication coverage increases from 60% to 80%, there is no difference in overhead, but the throughput is significantly increasing. Similarly, in case of MpegII (Figure 6.10d), when communication coverage increases from 60% to 80%, the relative overhead reduces, while there is not that much imporvement in relative throughout. For these design points, due to the high amount of communication and orchestration load masked inside TSS, the processing acceleration determines the system performance and throughput. Therefore, no
CHAPTER 6. EXPERIMENTAL RESULTS: TSS VS. ACMP

![Diagram](image)

Figure 6.11: Relative overhead of TSS over ACMPs WRT. communication coverage

matter how much the relative overhead is changing when communication coverage increases, the processing acceleration dominates the other system performance contributors and decides about the system performance and throughput.

6.6 Summary

This chapter has focused on the evaluation of our proposed TSS against conventional ACMPs. As conventional ACMPs are built for sparse integration of ACCs, this chapter first ran eight streaming applications with moderate number of ACCs and constant (about 50%) computation coverage by ACCs to avoid scalability limitations in ACMP architectures. We observed that TSS can outperform ACMPs up to 10x in performance improvement and up to 22x in energy saving due to internal realization of direct ACC-to-ACC connections (the communication and orchestration load of connections).

To gain insight of performance impacts of the architecture templates with varying number of ACCs and scalability limitations, we investigated five more complex applications. With increasing ACCs and ACC computation coverage from 20% to 100%, we observed that TSS can improve throughout up to 1.6x and 130x and reduces energy dissipation up to 3.4x and 209x as compared
Measuring the communication and orchestration load of ACCs in TSS and ACMPs, we observed the correlation between relative throughput improvement and relative architectural overhead reduction in TSS. With more number of direct ACC-to-ACC connections and higher volume of communication load on the connections that are internally realized inside TSS, TSS can achieve more ACCs’ benefits as compared to conventional ACMPs.
Chapter 7

Investigating Network-on-Chip (NoC) for Connecting ACCs

This chapter investigates into Network on Chip (NOC) modeling and analysis. NoCs are a promising interconnect strategy to enhance our TSS architecture template. We envision an enhanced template, TSS Plus, in which an NoC provides more flexible configurable communication across ACCs. To easily tune the NoC parameters based on given application(s) and provide early evaluation of TSS plus based on the applications, this chapter proposes a Transaction-Level Model (TLM) of NoC that gains at least 10x speedup at the cost of 10% - 20% accuracy loss on average as compared to an accurate Bus-Functional Model (BFM) of NoC.

The future work of this dissertation is defined as the integration of the proposed TLM of NoC to the VPs generated (in SCE) for TSS and measure how TSS plus extracts more of the ACCs’ benefits as compared to TSS.

7.1 Evaluation Approaches for Network on Chip (NoC)

With increasing number of ACCs inside the TSS, a lot of pressure is put on the interconnection fabric to carry communication flows among various ACCs efficiently. To power and performance efficiently connect ACCs inside TSS, reusable interconnect architectures are required that provide scalable bandwidth and parallelism [90].

One of the promising alternatives is Networks on Chip (NoCs) as the authors in [17], [14] already have exploited NoC to efficiently connect ACCs. NoCs avoid the need for dedicated wires/paths for each individual communication, and connect ACCs through an on-chip network.
Several advantages compared to dedicated wires/paths include delivering high-bandwidth, low-latency and low-power communication over a flexible and modular medium [77] [21].

However, different NoC design parameters such as topology, communication mechanism, routing method and switching mode impact a multi-dimensional trade-offs space between latency, throughput, communication load, energy consumption, and silicon area that all effect the efficiency of ACCs. Therefore, early evaluation of NoC is in high demand [69]. In general, NoC evaluation can be categorized in 3 groups: emulation/simulation frameworks, static analysis, and abstract modeling.

**Emulation/Simulation Framework:** Many NoC simulators and emulators have been developed. The emulation platform proposed by Dally et al. as a flexible emulation environment implemented on FPGA based on a complete mixed HW-SW NoC emulation framework, Xmulator [70] as an event-driven simulator and Booksim [53] as a cycle-driven simulator are a few instances of tools in this category. All instances impose high implementation cost, maintenance difficulty and long emulation/simulation time.

Some work [47] [118] aims to reduce the emulation/simulation time by changing the kernel scheduler, simulation/evaluation semantics by adding local clocks/schedulers. Nevertheless, their improvements are case-specific, for instance [47] gains more as the size of NoC gets larger.

**Static Analysis:** Static analysis such as [98] and [75] rapidly yields timing parameters such as router service time and packet arrival time. These methods have low accuracy as they abstract away dynamic behaviors influencing NoC performance and bandwidth.

**Abstract Modeling:** Abstract modeling might be placed in between two above categories. It abstracts away some implementation details (such as bit-level communication details) and takes into account only the events occurring per transmission of coarser data granularity. The goal is to accelerate the NoC evaluation while maintaining some accuracy. The architectural model in [49] is one example. It models the HERMES [69] router architecture as a bus and all cores/routers connected to it as individual modules. With keeping track of all routers’ active flows in different FIFOs and prioritizing their requests based on the pre-defined priorities, all the competitions over the shared resources are captured.

Therefore, system level modeling is preferably selected to relieve time to market pressures and the expense of NoC simulation/emulation tools with providing faster architecture exploration, performance evaluation, and functional validation [88].
CHAPTER 7. INVESTIGATING NETWORK-ON-CHIP (NOC) FOR CONNECTING ACCS

7.2 Modeling and Analysis of SLDL-captured NoC Abstractions

This section identifies different NoC abstraction levels according to the visibility of implementation details that is translated as accuracy, and communication granularity that is translated as speed. To this end, this section first defines accuracy impact factors at each level, including contention and arbitration points over shared resources (routers and router modules) within an NoC. A most abstract model treats the whole NoC as one black box, only revealing input and output traffic. Conversely, the most detailed level exposes how individual flits are handled at the micro-architecture level. Abstract modeling of an NoC poses the question of abstraction levels (the amount of detail to be retained in the model). Ultimately, this poses a trade-off between simulation speed and accuracy as visualized in Figure 7.1.

![Figure 7.1: Speed accuracy trade-off.](image)

7.2.1 NoC Abstraction Models

Conceptually, many abstraction levels are possible that may range from an extremely coarse grain model the treats the whole NoC as one black box, to a very fine-grained model that exposes micro architectural implementation details of all the NoC elements. Abstracting NoC can occur with different levels of details; from low covering details such as observing the whole NoC as a communication box to considering all micro-architectural implementation details of all NoC elements.

Although [63] discusses about different abstraction levels, their precise definition and modeling abstraction rules are not clearly presented. Defining abstraction levels helps designers to select communication features to model a given desired speed/accuracy. When comparing abstraction levels, we can consider the following aspects:
CHAPTER 7. INVESTIGATING NETWORK-ON-CHIP (NOC) FOR CONNECTING ACCS

Granularity of data: defines the smallest unit of data transferred through the NoC.

Visibility: defines the level of implementation details of NoC communication observable in the model.

Arbitration points: lists shared resources for which contention is dynamically resolved.

Timing accuracy: outlines the resulting estimation accuracy; meaning that at which level of accuracy, an NoC model can estimate the timing behavior of a real NoC.

Given the characteristics above, we propose five abstraction levels. Table 7.1 summarizes the models, and Figure 7.2 illustrates the 3 most abstract models.

Network-Level Model: models the whole NoC as a black box and only exposes the local ports. This model abstracts away everything inside NoC including traffic paths and contentions over shared resources. The model estimates network latency based on statistical information like average/worst case network latency per pre-defined size of traffic and the amount of traffic transferred through the network.

Router-Level Model: realizes NoC as a set of routers connected to each other via physical channels. In this model, routers are modeled as black boxes which receive packets as input and sends output packets over a physical link to the next router. This model estimates the NoC performance/latency based on the number and size of packets as well as the length of path taken by each packet. It dynamically resolves contention on physical links.

Transaction-Level Model: more details over the router-level model by modeling router internal modules, including input/output ports, cross-bar, routing management, virtual channel (VC) allocation and flow control management.

Figure 7.2: NoC modeling granularity.
In this model, at the end of any transaction, the contentions (and arbitration events to resolve them) which change the system status are collected and the system status is updated. Based on [36], a transaction is defined as injecting the header flit (first part of a packet) by initiator and receiving the last flit of the packet by the receiver.

**Pin-Level Model:** implements all the internal wires/pins per router modules and updates the system status after any individual contention (and arbitration to resolve that) happening per transmission of each bit of the transaction.

**Micro-Architectural-Level Model:** implements the Pin-Accurate model and all of it’s router operations at gate level for final validation. This model is practically an RTL model, very close to the final implementation, most accurate but also the slowest.

From the network-level model to micro-architectural level model, communication and implementation details are added to the model, increasing accuracy at the cost of simulation speed. Table 7.1 summarizes the abstraction levels.

<table>
<thead>
<tr>
<th>Model</th>
<th>Visibility Granularity</th>
<th>Arbitration Point</th>
<th>Time Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network-Accurate</td>
<td>Traffic</td>
<td>-</td>
<td>Loosely Time Estimated</td>
</tr>
<tr>
<td>Router-Accurate</td>
<td>Channel, Packet, Routers</td>
<td>Approximate Time Estimated</td>
<td></td>
</tr>
<tr>
<td>Transaction-Accurate</td>
<td>Channel, Flit, Router Modules</td>
<td>Cycle Estimated</td>
<td></td>
</tr>
<tr>
<td>Pin-Accurate</td>
<td>Wire, Bit</td>
<td>Router Modules</td>
<td>Cycle</td>
</tr>
<tr>
<td>Micro-Arch</td>
<td>Wire, Bit</td>
<td>Router Modules</td>
<td>Cycle</td>
</tr>
</tbody>
</table>

As we aim to propose a model that is abstract and accurate enough, we pick a transaction accurate model, that is the middle class of abstraction defined above. To validate the accuracy of the model, we also implement a pin accurate model of NoC that is as accurate as RTL implementation. We first, proposes the model of router used in both models, and walks through a packet transmission. Then, it discusses different arbitration points taken into account for the proposed models, and at the end, it presents the details of the models.

### 7.3 Proposed NoC Models

#### 7.3.1 Router Architecture

Our router models are based on the HERMES router architecture [69] with slight changes. Figure 7.3 outlines the router’s internal structure with 4 important functional units: Input/Output
CHAPTER 7. INVESTIGATING NETWORK-ON-CHIP (NOC) FOR CONNECTING ACCS

Ports, Routing Management Unit (RMU), Flow Control Unit (FCU) and crossbar unit. RMU includes Routing Logic Unit (RLU) and central table to record the status of virtual channels (VCs) of the output ports. FCU for promulgating free ports to the neighboring output ports. Crossbar unit forwards the packets to the next router determined by the RMU.

Each router is connected to 4 neighboring routers through the 4 input/output ports. One local port connects the router to the local IP core. Each input port has a configurable (8bits in Hermes, 32bits in our model) size of VC-buffers to record the received data/control flits. A flit is the smallest part of a packet. There is no buffer in the output ports as the buffer in the next router’s input port is used. For this, a credit-based flow control mechanism is employed to notify the sending side about the available space on the receiving side. This way, flit is only sent if there is space on the receiving side. The RLU (part of RMU) computes which output port to sent a packet to. The VC allocation unit selects which VC to use for a given output port. One of our changes over the Hermes architecture is supporting individual RLUs for each input port in order to avoid congestion inside the router. Similar to Hermes, the routing method is XY; approaching the destination always first horizontally, then vertically or vice versa. Routing decision is made per header flit which contains destination information and packet length. Flits are switched using the wormhole method.

![Figure 7.3: The router architecture in the proposed NoC abstract models.](image-url)
CHAPTER 7. INVESTIGATING NETWORK-ON-CHIP (NOC) FOR CONNECTING ACCESS

To explore how routing is done, example a received flit is stored in the VC buffers of that input port. In case of a header flit, it’s destination is forwarded to the RMU to determine the output port. After selecting the output port, the RLU consults RMU table to find an available VC on the output port.

After selecting output port and VC, the input port refers to FCU to check if there is enough room for receiving this flit in the next router. In our credit-based flow control, each VC has a credit and when the VC is used by a flit, its credit is decreased. When the flit leaves the input buffer of the destination router, a credit is sent back to the sending router, increasing the VC’s credit count.

Assuming sufficient sending credit is available, the crossbar sends the flit from the input port to the output port (and subsequently sends a credit upstream). The remaining flits of this packet are then sent one by one consulting the FCU about receive buffer credits to the next router. Upon receiving the tail flit, the RMU de-allocates the output buffer and VC.

During packet transmission within a router, various shared resources are used for which accesses need to be arbitrated. Detecting and resolving/arbitrating the contentions impacts the accuracy.

7.3.2 Arbitration Points

One of the most important aspects impacting accuracy in modeling (and especially our models) is detecting contentions over shared resources and resolving them. Shared resources are FCU, crossbar, and output ports. The way how access requests to these shared resources are collected and arbitrated affects the modeling accuracy. We identify one contention type for each resource and how our proposed models treat them (see also summary in Table 7.2):

Connection Establishment: if a router receives two header flits that target the same output port, their requests contend for the RMU. An arbiter is required to select one of requests. The selected request gets access to the RMU (central table of RMU), then starts connection and sends data.

Request Flow Control Grant: simultaneous flow control requests at the same FCU for different VCs create contention over FCU access. Concurrent requests are feasible, as they have already received the credit to send data. In order to guarantee that at one point of time, only one traverse is allowed to a specific output port, an arbiter is necessary to give flow control grant signal to one of the requests. We define this arbitration point as arbitration for same output accesses.
CHAPTER 7. INVESTIGATING NETWORK-ON-CHIP (NOC) FOR CONNECTING ACSS

Crossbar Access: when more than one VCs at the same input port gets the flow control grant simultaneously, there is contention on the crossbar. In our design, to avoid this contention, we define an arbiter for crossbar access from the same input port. This arbiter grants crossbar access to only one of the requests.

<table>
<thead>
<tr>
<th>Arbitration point</th>
<th>location</th>
<th>Resource</th>
<th>Arbitration (BFM)</th>
<th>Arbitration (TLM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection Establishment</td>
<td>RMU</td>
<td>RMU table</td>
<td>FIFO</td>
<td>random</td>
</tr>
<tr>
<td>Request Flow Control Grant</td>
<td>FCU</td>
<td>output port</td>
<td>FIFO</td>
<td>random</td>
</tr>
<tr>
<td>Crossbar Access</td>
<td>IP</td>
<td>crossbar</td>
<td>Round-robin</td>
<td>random</td>
</tr>
</tbody>
</table>

7.3.3 NoC Abstract Models: TLM and BFM

We proposed two abstract model of NoC; Transaction-Accurate Model (TLM) and Pin-Accurate Model (BFM) in the System-level Design Language (SLDL), SpecC. Where the latter is used to evaluate the accuracy of the former, as the former is our target platform for the TSS plus. About the novelty of the proposed TLM compare to the precious work, models the HERMES router architecture as a bus and all cores/routers connected to it as individual modules. With keeping track of all routers’ active flows in different FIFOs and prioritizing their requests based on the pre-defined priorities, all the competitions over shared resources are captured. The drawback of this work compared to the proposed TLM is its evaluation for worst-cases; when all possible contentions over shared resources happen. Similarly, proposes an accurate abstract model for on-chip interconnects. It uses bus protocol specifications to identify a reduced set of timing points. Finding the set of optimal timing points is the drawback of this work compared to the proposed TLM.

Our both models take into account the arbitration points explained in Section 7.3.2 as well as the characteristics of Table 7.1.

However, they differ in the way that requests to shared resources are collected and arbitrated. The BFM gathers resource access requests based on sampling and driving of every single wire at each cycle. Conversely, the TLM gathers the access information at transaction. Consequently, the BFM updates the system status at any cycle, while the TLM updates at transaction boundaries. As the granularity of updating the system status affects the accuracy, TLM is less accurate than BFM.
Both models implement the same arbitration policies. However, as the TLM makes a decision at a coarser granularity, it is more susceptible to the order in which access requests appear. Within the same time quantum, the TLM cannot distinguish between concurrent requests. As the execution order is not specified by the underlying discrete event simulation semantics, the effective arbitration policy for simultaneous (same quantum) becomes random.

Moreover, the BFM is driven by an explicit clock, while the TLM virtually times routers by using the instruction `waitfor`. In some sense the TLM can be considered time driven, while BFM is event-driven. Both models also differ in the number of threads (`sc.module` in SystemC, or behavior in SpecC) used for simulation. The BFM employs active threads for each router module. Conversely, the TLM is mainly channel based (i.e. is call driven) and only uses one behavior (or `sc.module`) for each VC. For instance, assuming 4 VCs per physical link, the BFM has 41 simultaneous threads and the TLM only 20. With the lower number of active threads, the TLM can perform faster (avoiding context switches). Table 7.3 compares BFM and TLM.

<table>
<thead>
<tr>
<th></th>
<th>BFM</th>
<th>TLM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication Implementation</td>
<td>Behavior</td>
<td>Channel</td>
</tr>
<tr>
<td>Arbitration Policy</td>
<td>FIFO &amp; Round-robin</td>
<td>Random</td>
</tr>
<tr>
<td>Timing</td>
<td>Event driven (explicit clock)</td>
<td>Time driven (<code>waitfor</code>)</td>
</tr>
</tbody>
</table>

### 7.4 Experimental Results

This section explores the proposed BFM and validates its accuracy and functionality with respect to the RTL implementation. It then compares the proposed TLM versus the proposed BFM with respect to speed and accuracy. For evaluation of the models, we mainly use hot spot traffic [32]; some nodes in the network receive most of the traffic.

#### 7.4.1 BFM Validation

System performance and throughput are two important metrics for analyzing NoC architectures. Average packet latency is a representative of system performance, and link utilization is a representative of throughput. Packet latency is the packet life-time defined as the difference between its start time label and its end time label.
Link utilization/load is the ratio of link busy time over the whole simulation time. Link busy time is defined as the total time when the link is busy carrying traffic.

In this part and for validation of the proposed BFM, we adopted 40%-hot spot traffic and defined packet size as 10 flits. 40%-hot spot traffic means that 40% of the nodes are the destinations of total traffic injected to the network. Each injector (hot node) injects 100 packets to the network. Figure 7.4 shows the simulation results including link load, average packet latency and simulation time for hotspot traffic injected into the 8*8 mesh. The results are correlated with the results of VC extended HERMES router structure on FPGA [66].

Figure 7.4: Performance validation of the BFM (8x8 mesh, 40%-hot spot).

Figure 7.4a shows link load for different numbers of VCs as the injection rate increases. At small injection rate, the link load increases linearly for all VCs. However, the link load starts to level off from a specific injection rate, around 10%, 15%, 20% and 30% respectively for 1 VC, 2 VCs, 4 VCs and 8 VCs. Based on [66], this point is called saturation point. The saturation degree of network for multiple VCs in our model is higher than what is reported in [66]. The reason lies in the different implementations of the network interfaces. In our work, we define multiple VCs for local
connections as well, which means at the destination node, traffic from different ports can sink into the local PE without being blocked. With the improved mechanism, the network throughput for VCs of 8 can reach 100% under hotspot traffic.

Figure 7.4b shows the average latency as injection rate increases. Up to the saturation point, the average latency is constant for all VCs. With increasing the number of VCs, the average latency drops in a half when VCs goes from 1 to 2. Similar situation for 2 VCs to 4 VCs. However, the average packet latency for VCs of 4 and 8 are similar. The reason is that using 4 VCs has already eliminated most of packet blocking.

Figure 7.4c shows the overall finish/transmission time as the injection rate increases. This time is when all the packets reach their destinations. Since more VCs leads to more traffic overlap on the fly, 8 VCs yields the shortest overall transmission time.

7.4.2 TLM Evaluation

This section evaluates the TLM as compared to the BFM with respect to speedup and accuracy.

**SpeedUp:** depending on the amount of implementation details and number of context switches, the simulation time varies. For comparing the models, simulation speedup is reported. Simulation speedup of the model with higher level of abstraction (H) compared to the model with lower level of abstraction (L) is defined as

\[
\text{Speedup}_{H2L} = \frac{\text{Simu. Time}_L}{\text{Simu. Time}_H}
\]  

As the simulation time strongly depends on the number of context switches, the simulation speed is closely correlated with the network size and traffic intensity. Both network size and traffic intensity affect the the number of behaviors and context switches. Network size is the number of nodes in the network. Network intensity is defined as the number of transactions (number of packets) from sender nodes to the receiver nodes. With larger network or intense traffic simulation time increases.

To evaluate the effects of network size and traffic intensity, 40%-hot spot traffic is simulated with 4 VCs per physical link and 100% injection rate. Figure 7.5a shows the simulation time for increasing network (mesh) size from 2*2 to 8*8. The TLM is 10x to 16x faster than the BFM. With larger networks, TLM achieves higher speedup as a result of abstracting away higher ratio of communication details. Figure 7.5b illustrates the simulation time for network intensity. With increasing the transaction size (from 1 packet to 100 packets), higher speedup (50x compared to 14x)
CHAPTER 7. INVESTIGATING NETWORK-ON-CHIP (NOC) FOR CONNECTING ACCS

is achieved as a result of decreasing the number of transactions (with the same amount of traffic). In the TLM, the less the number of transactions, the less the system status is updated. While in the BFM, at each cycle the system status is updated independent from the size of transaction.

![Graph](image)

(a) Simulation over network size

![Graph](image)

(b) Simulation over network intensity

Figure 7.5: Comparing simulation time of the BFM and the TLM (8*8 mesh, 40%-hot spot).

To measure the accuracy loss, we define accuracy error for each packet. As Equation (7.2) defines, this accuracy error has correlation with the difference between packet latency in the TLM and BFM. Packet latency is the difference between start time label and end time label of the packet.

\[
\text{Error} = \left| \frac{\text{PacketLatency}_L}{\text{PacketLatency}_H} \right| / \text{PacketLatency}_L
\]  

(7.2)

As the TLM differs from the BFM in the effective arbitration policy (due to collection of requests and arbitration among them), measuring the accuracy loss in the TLM requires simulation scenarios with different amount of contentions (requests) over shared resources. The amount of contentions over shared resources is determined by the amount of traffic injected to the network. The more the injection rate, the higher the number of simultaneous requests for the same resources and higher contention and accuracy loss as a result. To demonstrate this, 40%-hot spot traffic is adopted into the TLM and BFM models of 6*6 NoC with 4 VCs per physical links and 100% injection rate. We simulate 100 transactions through the NoC and measure the transmission delay of packets in both models BFM and TLM. To aggregate the results, we report the average error, as well as the cumulative error for 50-percentile and 96-percentile as Figure 7.6a. The 50-percentile (96-percentile) cumulative error indicates the maximal error experienced by 50% (96%) of transactions. As Figure 7.6a represents,
increasing the injection rate, increases the cumulative error probability. Increasing the injection rate from 0.1 to 0.2 increases the average error from 10% to 20%. At 0.1 injection rate 96% of packets observe less than 40% error, while 50% see less then 10% error. Increasing the injection rate to 0.6, makes 50% of packets experience up to 30% error.

Increasing the injection of rate increases the contention over shared resources. One indicator is the congestion rate over physical links. Figure 7.6b shows the cumulative error probability over increasing congestion. All three metrics are strongly related to congestion. And increase until congestion hits 50%. Then, 50% show at most 38% error, while the maximum error measured for 96% of packets reaches 100%. Conversely, at lower congestion rate, e.g. 5%, 96% of packets experience less than 20% error.

Figure 7.6: Cumulative probability of accuracy error.

7.5 Future Work: Integration of the Proposed NoC Model to Connect ACCs inside the TSS Plus

Integration of the proposed TLM NoC into the VPs generated for TSS architecture is defined as the future work for this dissertation. The future steps include: (1) integration of the proposed TLM NoC into SCE and generate the VPs for TSS Plus which is the same as for TSS, but the proposed TLM NoC instead of multi-layer AMBA AHB, (2) Evaluate the TSS Plus in SCE for the same applications in Section 6 and measure how TSS plus can enhance the system performance and throughput as a result of less pressure on the interconnect fabric.
CHAPTER 7. INVESTIGATING NETWORK-ON-CHIP (NOC) FOR CONNECTING ACCS

7.6 Summary

With increasing number of ACCs inside the TSS, a lot of pressure is put on the interconnection fabric to transfer data among various ACCs efficiently. Current MUX-based interconnect that provides dedicated paths for chains lacks flexibility to provide various connections among ACCs more than for which the MUXes are configured. To more flexibly connect ACCs, Network on Chip (NoC) is suggested by previous work. However, NoC design parameters such as topology, communication mechanism, routing method and switching mode impact a multi-dimensional trade-offs space between latency, throughput, communication load, energy consumption, and silicon area that all effect the efficiency of NoC based on the application. Therefore, modeling of NoCs is on high demand for early exploration and evaluation of NoC. In this context, fast and accurate evaluation is important. However, when abstracting NoC models, a trade-off between simulation speed and accuracy exists.

This chapter first identified NoC abstraction levels with respect to data granularity, visibility of internal structures, and modeling of contention points. Then, it proposed two NoC models: a detailed Bus-Functional Model (BFM), and a more abstract one, Transaction Level Model (TLM). The data granularity of bit in BFM and cycle-based modeling of arbitration on all contention points make the BFM as accurate as an RTL-implemented NoC that can be used to estimate the speed and accuracy of the proposed TLM. Larger data granularity of flit in TLM and modeling of arbitration on all contention points when the system status changes (an event occurs) make the proposed TLM more faster about 10X at the cost of cost of 10%-20% accuracy as compared to the proposed BFM.

In the future, we can integrate the proposed TLM NoC to SCE, and then generate VPs with the proposed TLM instead of MUX-based interconnect. More flexibility to directly connecting ACCs inside TSS should provide more energy efficiency.
Chapter 8

Conclusion and Future Work

Current ACMP architectures have a processor-centric view as they were built upon the assumption of sparse integration of ACCs. ACCs are slaves devices requiring many shared resources (communication fabric, DMA, shared memory, and processor for coordination) for their transactions. However, the emergence of new applications with power and performance efficient computing demand calls for integration of more ACCs on chips, which is not effectively supported by processor-centric architectures. Processor-centric architectures suffer from scalability limitations that restrict ACCs’ benefits. With integrating more ACCs, the burden on shared resources increases dramatically even though some ACCs logically communicate directly with each other. To properly address the scalability challenges resulted from ACC communication and pave the way toward more scalable architectures, this dissertation:

- **Analyzed the scalability implication on ACMP performance:** We proposed an analytical model to estimate the load on shared resources as more ACCs are integrated on a chip. We showed that increasing ACCs’ communication load on the communication fabric/DMA and memory along with increasing ACCs’ orchestration load on the processor core overshadow ACCs’ benefits. To quickly explore the whole design space with respect to the number of ACCs and distribution of computation demand on ACCs and find the most efficient design point with maximum benefits, we proposed ACMPerf. ACMPerf is an analytical performance model of ACMPs that estimates the system performance capturing the ACCs’ benefits and ACCs’ load on shared resources. Our model showed ACCs benefits diminish with more ACCs, despite having more direct ACC-to-ACC connections that are supposed not to impose any overhead to shared resources.
CHAPTER 8. CONCLUSION AND FUTURE WORK

- **Identified/formalized semantics for ACC communication:** To lay the foundations for improving the efficiency of ACC integration, we first formalized the semantics of ACC communication: synchronization, data granularity, data marshalling, and data access model. Then, we identified the un-equal view of processor and ACC that turns out as the main source of scalability limitations in conventional ACMP architectures. In conventional ACMPs, ACC communication semantics are realized mostly on processor (data granularity, data marshalling, synchronization) and other shared resources of communication fabric/DMA and memory (data access model). Therefore, more ACCs results in more load on shared resources and less ACCs’ benefits.

- **Proposed Transparent Self-Synchronizing ACCs (TSS):** TSS is a scalable architecture with equal view of processor and ACC to internally realize direct ACC-to-ACC connections and reduce the ACCs’ load on shared resources. TSS gives autonomy to ACCs to self-synchronize and self-orchestrate each other independent of the processor, thereby enabling finest data granularity to reduce the pressure on the shared memory. TSS also exploits a local and reconfigurable interconnect for direct data transfer among ACCs without occupying DMA and communication fabric.

We used automatically generated virtual platforms to evaluate TSS against processor-centric architectures with exploiting the same number of ACCs and mapping of kernels to ACCs. First, we experimented eight streaming applications with balanced computation coverage by ACCs, and we observed 3x performance improvement on average, as well as 9x and 3x load reduction on communication fabric/DMA and processor, respectively. As a result of load reduction on shared resources, TSS reduced energy consumption by 22x as compared to ACMPs.

Then, we experimented five more streaming applications with more complex characteristics and increased computation coverage by ACCs from 0% to 100% to observe how TSS can outperform ACMPs as scalability limitations emerge. The results showed that TSS improves throughput up to 1.6x in 20% ACC computation coverage and up to 130x in 100% ACC computation coverage. This benefits are achieved due to reduced ACC communication load on shared resources by 6.57x and 328x, respectively. TSS outperforms processor-centric ACMP architectures more significantly with more ACC-to-ACC connections and higher communication load on the connections.

The focus of this dissertation was on improving the communication architecture to effect-
CHAPTER 8. CONCLUSION AND FUTURE WORK

To effectively integrate many ACCs on chips, an equal view of ACC and processor and internal realization of direct ACC-to-ACC connections without imposing any overhead to shared resources pave the path toward scalable integration of many ACCs. To implement the equal view between ACC and processor, we distributed orchestration tasks on individual ACC and exploited a multi-layer AMBA AHB among ACCs. Future work includes:

- **Investigate and Optimize Alternative TSS Interconnect Strategies:** This dissertation used a MUX-based interconnect. Other interconnect architectures, such as NoC can be investigated that offer more flexibility in ACC interconnection and relax timing constraints (e.g. through Globally Asynchronous Locally Synchronous (GALS)). The benefits of more complex interconnects and the increased flexibility have to be weighed against an increased area. Research is needed to investigate how much communication flexibility is needed to efficiently realize domains.

- **Domain-specific Architectures based on TSS:** This dissertation has focused on the architecture template as a foundation for platforms with dense ACC coverage. Additional research is needed in how to employ the TSS principles given a concrete domain (i.e. how to design an particular TSS instance). This involves additional research about formalizing the domain concept as well as analyzing applications and clustering them to domains. New research is needed to expand the concept of Design Space Exploration (DSE) to the domain level in order to identify kernels in a domain that are acceleration worthy and benefit the majority of applications in a domain. In that context, communication patters across kernels need to be carefully analyzed as the TSS benefits are maximized with direct ACC-to-ACC communications.
Chapter 9

List of Publications


Bibliography

*City University of New York*, March 2008.


R. Stets, and B. Verghese. Piranha: a scalable architecture based on single-chip multipro- 

[7] B. Barry, C. Brick, F. Connor, D. Donohoe, D. Moloney, R. Richmond, M. O’Riordan, and 
V. Toma. Always-on vision processing unit for mobile applications. *IEEE Micro*, 35(2):56–66, 
Mar 2015.

vision applications using the Pipelined Vision Processor. In *Asilomar Conference on Signals, 

BIBLIOGRAPHY


BIBLIOGRAPHY

2 1080p H. 264/AVC high profile encoder chip. In Proceedings of the 45th annual Design


[63] K. Lu, D. Muller-Gritschneder, and U. Schlichtmann. Accurately timed transaction level
models for virtual prototyping at high abstraction level. In Design, Automation Test in Europe

for high-performance, low-power accelerator-based systems. Computer Architecture Letters


[66] A. Mello, L. Tedesco, N. Calazans, and F. Moraes. Virtual Channels in Networks on Chip:
Implementation and Evaluation on Hermes NoC. In Integrated Circuits and System Design,


Optimization of Heterogeneous Multi-Accelerator SoC. Computer Architecture Letters (CALs),

[69] F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost. HERMES: An Infrastructure for

[70] A. Nayebi, S. Meraji, A. Shamaei, and H. Sarbazi-Azad. XMulator: A Listener-Based
Integrated Simulation Platform for Interconnection Networks. In International Modeling

1997.


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


