Run-Time Reconfiguration of Homogeneous MPSoCs for use in Embedded Systems

XI ZHANG

A THESIS IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
UNIVERSITY OF NEW SOUTH WALES

MARCH 2015
'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project’s design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed .................................................... (Xi Zhang)

Date .................................................
COPYRIGHT STATEMENT

‘I hereby grant the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or part in the University libraries in all forms of media, now or hereafter known, subject to the provisions of the Copyright Act 1968. I retain all proprietary rights, such as patent rights. I also retain the right to use, in future works (such as articles or books), all or part of this thesis or dissertation.

I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstract International (this is applicable to doctoral theses only). I have either used no substantial portions of copyright material in my thesis or I have obtained permission to use copyright material; where permission has not been granted I have applied/will apply for a partial restriction of the digital copy of my thesis or dissertation.’

Signed .................................................... (Xi Zhang)

Date ..........................................................
AUTHENTICITY STATEMENT

‘I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis. No emendation of content has occurred and if there are any minor variations in formatting, they are the result of the conversion to digital format.’

Signed .................................................... (Xi Zhang)
Date ........................................................
ABSTRACT

Multi-processor-system-on-chips (MPSoCs) with different numbers of cores have become increasingly popular in embedded systems. One of the main challenges in MPSoC systems is the exponentially increasing expense of fabricating chips. Instead of fabricating chips for each application separately, the embedded system industry seeks for an MPSoC system that can be efficiently used in a multitude of applications to reduce fabrication cost. In this thesis, we discuss a homogeneous MPSoC platform, and propose a set of novel run-time reconfiguration methods on this platform to execute multiple applications simultaneously and achieve benefits (improve performance, or reduce power consumption and area cost).

The first method (DRMA) allows cores in MPSoCs to be rapidly reconfigured to change data widths. We present a case study with four cores to showcase the flexibility and efficacy of DRMA. Results show that the prototype of DRMA is capable of working as four 32-bit cores, two 64-bit cores, a single 128-bit core, and can achieve up to $4.11 \times$ speed up.

In the second method (ADAPT), we examine hardware/software pipelines, which are useful for streaming applications. ADAPT can quickly detect bottle-
neck stages and add idle cores to bottleneck stages to improve throughput. If there is no idle core, a shuffling of cores across stages will be performed to balance workloads and improve throughput. For a variety of applications, ADAPT takes less than 2 \( \mu \text{s} \) for a run-time adaptation, and achieves up to 2.1\( \times \) speed up, compared to a state-of-the-art method (which is modified and implemented in the same platform for a fair comparison). Results illustrate the applicability of ADAPT for fine-grained run-time management to achieve high throughputs in MPSoC systems.

The third method (E-pipeline) considers the system-level resource utilisation and power consumption when an MPSoC system simultaneously executes multiple pipelines. We show how multiple pipelines with dynamic workload variation can be efficiently executed in such a situation, and discuss how the system can switch cores within a pipeline (intra-elasticity) and across pipelines (inter-elasticity). Compared to the reference design method with clock gating, E-pipeline maintains the same power savings, and reduces core usages by an average of 37.7\%.
Contributions

- A novel mechanism (called \textit{DRMA}) allows cores in the MPSoC system to be combined together for varied data width;

- \textit{DRMA} allows the run-time configuration to be complete cycle by cycle at beginning of an instruction execution with little overhead;

- \textit{DRMA} can take advantage of both instruction level parallelism and data level parallelism;

- A novel run-time method (called \textit{ADAPT}) is developed to maintain/improve the throughput of streaming applications on MPSoC systems under a set of allocated cores;

- \textit{ADAPT} allows fast run-time bottleneck detection in streaming applications;

- \textit{ADAPT} can dynamically add assign cores to tasks, or shuffle cores between tasks in a streaming application with small adaptation overhead;

- A novel method (called \textit{E-pipeline}) is developed to dynamically adapt to workload variations of streaming applications, and minimise the number of cores utilised for multiple applications and power consumption;

- \textit{E-pipeline} enables Intra-Elasticity: A method to switch cores within a single pipeline (from stage to stage) to meet throughput constraints; and,

- \textit{E-pipeline} enables Inter-Elasticity: A method to switch cores from one pipeline to a pool of idle cores which are asleep when the cores are not nec-
ecessary, and a method to obtain cores from the sleep pool to a pipeline when the pipeline needs an additional core to meet the throughput constraint.
Publications


Acknowledgement

I would never have been able to reach this far without the support of the individuals mentioned here. It’s time to recall and acknowledge all of them, and to share the enjoyment and satisfaction I felt during the course of my PhD degree.

First and foremost, I wish to thank my supervisor, Prof. Sri Parameswaran, for his endless support and careful guidance throughout my candidature. Without his guidance and advice about my research, I would not have been able to write this thesis. I greatly appreciate all his support.

I would like to extend my gratitude to the academic giants for reviewing my research throughout my candidature: Dr. Oliver Diesel, Dr. Hui Guo and Dr. Jorgen Peddersen. Their comments and feedbacks always guided my research into the right direction. I would also like to thank the University of New South Wales and the School of Computer Science and Engineering for all the support work throughout my PhD degree.

I would like to thank all the researchers around the world who provided valu-
able feedback on my papers, and all the attendees of my presentations at international conferences for their valuable comments, which always helped me to refine my research. I also thank my thesis examiners for their time reading my thesis, and making suggestions that improve the clarity of my thesis.

My sincere thanks go to Haris Javaid, Muhammad Shafique, Jørgen Peddersen and Jude Angelo Ambrose for his valuable advice and support. I particularly thank Elyse Caitlin WISE for her contribution and cooperation in hardware implementation. My thanks also go to Tuo Li, Roshan Ragel, Swarnalatha Radhakrishnan and Kewal K. Saluja for guiding me and helping me in paper writing. Many thanks to other members in the Embedded Systems Lab, all my colleagues and friends at CSE, UNSW, for providing a helpful and warm environment.

I also take this opportunity to thank my uncle Zhang Bing and my aunt Linlei Gu for their support. Because of them, I am in this good position in my life. I know that thanking is not enough to express how much I owe to my family: my father Zhang Shouping, my mother Chen Dahong and other family members. All their sacrifices and hardship is what developed me and motivated me as a person. I wish I could turn around things and provide them a much better days than they have had.

My humble apologies to anyone whose name I might have not mentioned here, but from the core of my heart I am obliged for your support. To all, whom I have mentioned and whom I forgot to mention, I would like to dedicate this work.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORIGINALITY STATEMENT</td>
<td>iii</td>
</tr>
<tr>
<td>COPYRIGHT STATEMENT</td>
<td>v</td>
</tr>
<tr>
<td>AUTHENTICITY STATEMENT</td>
<td>vii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>ix</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Embedded System</td>
<td>2</td>
</tr>
<tr>
<td>1.1.1 Embedded System Introduction</td>
<td>2</td>
</tr>
<tr>
<td>1.1.2 ASICs</td>
<td>3</td>
</tr>
<tr>
<td>1.1.3 CPUs</td>
<td>3</td>
</tr>
<tr>
<td>1.1.4 FPGAs</td>
<td>4</td>
</tr>
<tr>
<td>1.1.5 ASIPs</td>
<td>5</td>
</tr>
<tr>
<td>1.1.6 MPSoC</td>
<td>6</td>
</tr>
<tr>
<td>1.1.7 Challenges in Embedded Systems</td>
<td>7</td>
</tr>
<tr>
<td>1.2 Run-time Reconfiguration</td>
<td>10</td>
</tr>
<tr>
<td>1.3 Anticipated Future MPSoC Architecture</td>
<td>13</td>
</tr>
<tr>
<td>1.4 Target Applications</td>
<td>15</td>
</tr>
<tr>
<td>1.4.1 Applications with Varying Data widths</td>
<td>15</td>
</tr>
<tr>
<td>1.4.2 Streaming Applications</td>
<td>16</td>
</tr>
<tr>
<td>1.5 Research Aims</td>
<td>18</td>
</tr>
<tr>
<td>1.6 Proposed Methods</td>
<td>20</td>
</tr>
<tr>
<td>1.7 Thesis Contribution</td>
<td>22</td>
</tr>
</tbody>
</table>
# 2 Literature Review

## 2.1 Single-core Processor Review

- **2.1.1 Technology Scaling**

## 2.2 Power Consumption and Frequency

- **2.2.1 Relationship between Power and Frequency**

## 2.3 Instruction-level Parallelism Review

- **2.3.1 Hardware Pipeline Review**
- **2.3.2 Super Scalar Review**
- **2.3.3 VLIW Review**
- **2.3.4 Summary**

## 2.4 MPSoCs

- **2.4.1 From Single-core to Multi-core**
- **2.4.2 Heterogeneous MPSoC**
- **2.4.3 Homogeneous MPSoC**
- **2.4.4 MPSoC Systems with Small Cores**
- **2.4.5 Summary**

## 2.5 Reconfiguration

- **2.5.1 FPGA**
- **2.5.2 Run-time Reconfiguration**
- **2.5.3 Summary**

## 2.6 Optimisation in Streaming Applications

- **2.6.1 Throughput Improvement at Design Time**
- **2.6.2 Adaptation in Initialisation**
- **2.6.3 Run-time Adaptation Work**
- **2.6.4 Low Power Technology in Streaming Application**
- **2.6.5 Summary**

## 2.7 Summary

---

# 3 MPSoC Platform and Methodology

## 3.1 Reconfiguration Methodology

## 3.2 MPSoC Platform Model

- **3.2.1 MPSoC Platform Discussion**

## 3.3 Framework of Reconfiguration

- **3.3.1 Run-time Reconfiguration of Data Width Variation**
- **3.3.2 Streaming Application and Pipeline**
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.6 Experiments</td>
<td>176</td>
</tr>
<tr>
<td>A.6.1 Instruction Level Verification</td>
<td>176</td>
</tr>
<tr>
<td>A.6.2 AES-128 Kernel Algorithm</td>
<td>177</td>
</tr>
<tr>
<td>A.7 Summary</td>
<td>179</td>
</tr>
</tbody>
</table>

**Bibliography**

181
List of Tables

2.1 Technology Improvement ........................................... 27
4.1 Instructions Supported by DRMA ................................. 89
4.2 Hardware Parameters from Synthesis ............................ 89
4.3 The Results of JPEG Encoding ................................. 96
5.1 Results of Case Study 1. ........................................... 121
5.2 Comparison of ADAPT and [95] ............................... 123
6.1 Summary of Results (Including Managers) ......................... 154
A.1 Synthesis Results for DRMA Implementation and PISA ....... 177
# List of Figures

1.1 Fab Equipment Spending from 2010 to 2014, Source: [1] . . . . 11  
1.2 Fab Equipment Spending from 2014 to 2015, Source: [2] . . . 12  
1.3 Examples of Current MPSoC Architectures, Source: [3–5] . . . 13  
1.4 Future MPSoC Architecture and Task Assignment . . . . . . . 14  
1.5 A Typical Streaming Application . . . . . . . . . . . . . . . . . . . 19  
1.6 DRMA Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20  
1.7 ADAPT Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21  
1.8 E-pipeline Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . 22  
2.1 Technique Trend Between 1985 and 2015, Source: [6] . . . . . 27  
2.2 Five Stage Pipeline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30  
2.3 ARM 11 Pipeline, Source: [7] . . . . . . . . . . . . . . . . . . . . . . . . 32  
2.4 ADRES Architecture, Source: [8] . . . . . . . . . . . . . . . . . . . . . 45  
2.5 ERGA Architecture, Source: [9] . . . . . . . . . . . . . . . . . . . . . . 46  
2.6 PipeRench Architecture, Source: [10] . . . . . . . . . . . . . . . . . . 47  
2.7 Bahurupi Architecture, Source: [11] . . . . . . . . . . . . . . . . . . . 48  
2.8 MPSoC with GigaNoC, Source: [12] . . . . . . . . . . . . . . . . . . . . 49  
2.9 Invasive Computing, Source: [13] . . . . . . . . . . . . . . . . . . . . . 51  
3.1 Core Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65  
3.2 System Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66  
3.3 Core Combination . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68  
3.4 Core Assignment Methodology . . . . . . . . . . . . . . . . . . . . . . . . . 70  
3.5 A Pipeline Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70  
3.6 FIFO Communication . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71  
3.7 Shared Memory Communication . . . . . . . . . . . . . . . . . . . . . . . . . 72
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.8</td>
<td>ADAPT Methodology</td>
<td>73</td>
</tr>
<tr>
<td>3.9</td>
<td>E-pipeline Methodology</td>
<td>74</td>
</tr>
<tr>
<td>4.1</td>
<td>Motivational Example</td>
<td>79</td>
</tr>
<tr>
<td>4.2</td>
<td>The Peripherals of a Core</td>
<td>82</td>
</tr>
<tr>
<td>4.3</td>
<td>DRMA for TYPE₁ Instructions (example: or₁₂₈ A, B)</td>
<td>83</td>
</tr>
<tr>
<td>4.4</td>
<td>DRMA for TYPE₂ Instructions (example: add₁₂₈ A, B)</td>
<td>84</td>
</tr>
<tr>
<td>4.5</td>
<td>DRMA for TYPE₃ Instructions (example: srl₁₂₈ A, B)</td>
<td>85</td>
</tr>
<tr>
<td>4.6</td>
<td>DRMA for TYPE₄ Instructions (example: bseq₁₂₈ A, B)</td>
<td>85</td>
</tr>
<tr>
<td>4.7</td>
<td>The Design Flow of DRMA architecture</td>
<td>87</td>
</tr>
<tr>
<td>4.8</td>
<td>The ‘add’ instruction in the CMode</td>
<td>88</td>
</tr>
<tr>
<td>4.9</td>
<td>The Comparison in Performance</td>
<td>92</td>
</tr>
<tr>
<td>4.10</td>
<td>The Comparison of Different Configurations</td>
<td>94</td>
</tr>
<tr>
<td>4.11</td>
<td>The Results for Case 2</td>
<td>95</td>
</tr>
<tr>
<td>4.12</td>
<td>The Configuration in Case 3</td>
<td>95</td>
</tr>
<tr>
<td>5.1</td>
<td>Workload Variations of an H.264 Encoder Pipeline</td>
<td>101</td>
</tr>
<tr>
<td>5.2</td>
<td>Parallelization Model of Pipeline</td>
<td>102</td>
</tr>
<tr>
<td>5.3</td>
<td>Target System Model</td>
<td>104</td>
</tr>
<tr>
<td>5.4</td>
<td>Analysis of Pipeline in Figure 5.2(a)</td>
<td>107</td>
</tr>
<tr>
<td>5.5</td>
<td>An Example of Allocated Shared Memory Layout</td>
<td>114</td>
</tr>
<tr>
<td>5.6</td>
<td>An Example of Worker Program</td>
<td>117</td>
</tr>
<tr>
<td>5.7</td>
<td>ADAPT Under Static Workload</td>
<td>120</td>
</tr>
<tr>
<td>5.8</td>
<td>ADAPT Under Dynamic Workload</td>
<td>122</td>
</tr>
<tr>
<td>5.9</td>
<td>Throughput Under Resource Variation</td>
<td>124</td>
</tr>
<tr>
<td>5.10</td>
<td>Energy Consumption Under Resource Variation</td>
<td>125</td>
</tr>
<tr>
<td>6.1</td>
<td>(a) Pipeline with Four Stages (b) Task Cloning Mode Example</td>
<td>131</td>
</tr>
<tr>
<td>6.2</td>
<td>(a) H.264 Encoder Pipeline (b) Task with Workload Variations</td>
<td>133</td>
</tr>
<tr>
<td>6.3</td>
<td>Number of Necessary Cores Variations of Two Pipelines</td>
<td>134</td>
</tr>
<tr>
<td>6.4</td>
<td>System Overview of E-pipeline</td>
<td>135</td>
</tr>
<tr>
<td>6.5</td>
<td>Manager Program</td>
<td>137</td>
</tr>
<tr>
<td>6.6</td>
<td>System Implementation</td>
<td>143</td>
</tr>
<tr>
<td>6.7</td>
<td>Measured Throughputs in Different Benchmarks</td>
<td>149</td>
</tr>
<tr>
<td>6.8</td>
<td>Measured Power in Different Benchmarks</td>
<td>150</td>
</tr>
<tr>
<td>6.9</td>
<td>Run-time Core Usages (Excluding Managers)</td>
<td>152</td>
</tr>
<tr>
<td>A.1</td>
<td>Implementation Overview, Sourced in [14]</td>
<td>170</td>
</tr>
<tr>
<td>A.2</td>
<td>Data Read in UART, Sourced in [14]</td>
<td>172</td>
</tr>
</tbody>
</table>
### ABSTRACT

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.3 Command Line Sample 1</td>
<td>174</td>
</tr>
<tr>
<td>A.4 Command Line Sample 2</td>
<td>174</td>
</tr>
<tr>
<td>A.5 Command Line Sample 3</td>
<td>175</td>
</tr>
<tr>
<td>A.6 Command Line Sample 4</td>
<td>175</td>
</tr>
<tr>
<td>A.7 Results of DRMA and PISA architectures</td>
<td>179</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In the last decade, computing has increasingly trended towards ubiquitous computing. Ubiquitous computing is now an essential part of people’s lives due to the prevalent deployment of embedded systems [15]. Embedded systems, especially systems with multiple cores, are becoming a main stage of the computer industry.

This thesis explores a series of run-time reconfiguration methods to solve the issue of building a chip that is efficient and reusable for multiple applications. We first review the development of processors, then discuss multi processor architectures and propose three levels of run-time reconfiguration methods, corresponding to different levels of reconfigurability. Each reconfiguration method is discussed with a case study. The first case study (named DRMA) explores data width variation of cores at the instruction level with the run-time reconfiguration method. The second case study describes a method (named ADAPT), which performs task-level reconfiguration to balance workloads of tasks in streaming applications, in order
to improve the throughput of streaming applications. The third case study (named \textit{E-pipeline}) furthers the discussion of \textit{ADAPT} to reconfiguration at application level in a system with multiple streaming applications, in order to improve core utilisation rates and power efficiency of the whole system.

This thesis aims to provide methodologies that can dynamically adapt embedded systems to the feature of applications so that performance or power consumption can be improved when the chip is reused in multiple applications.

This chapter starts with an introduction of embedded systems followed by the presentation of the trends and challenges of specific applications. Lastly, this chapter concludes with the research aims and contributions of the thesis.

1.1 Embedded System

1.1.1 Embedded System Introduction

An embedded system is a hardware/software computing system that is embedded into an electronic device and designed for a range of specific tasks. Common examples of deploying embedded systems include mobile phones, multimedia devices, tablets, wearable devices, vehicle electronics, digital watches/clocks, micro-controllers, etc. According to user requirements of power, performance, cost, etc., embedded systems can be built with ASICs, GPCPs, FPGAs, ASIPs or MPSOCs.
1.1.2 ASICs

Application specific integrated circuits (ASICs) are integrated circuits that are designed for specific functions. ASICs are not flexible. ASIC designs need to be re-designed and re-fabricated when the application is changed, significantly raising Non-Recurring Engineering (NRE) costs \[16\]. For a wide range of application types, designers have to design a diversity of ASIC circuits separately, resulting in increased time-to-design and time-to-market of products.

1.1.3 CPUs

The central processing units (CPUs) are composed of arithmetic logic units, controllers, registers and other electronic components which are designed to execute a set of instructions. CPUs execute assigned instructions through the use of softwares. CPU systems can execute a wide-range of applications since programmers can codify algorithms of applications and compile the code to instructions. Instructions can be rewritable and updatable, hence CPU systems are highly flexible. Moreover, due to the abundant support of a diversity in software and well-developed personal computer design assistance, CPUs facilitates short time-to-design and time-to-market, decrease verification costs, and enable a single chip to be reusable for multiple applications. However, the execution of instructions is much slower than the signal transfers in ASICs, which makes the operation speed of CPUs typically slower than ASICs.
1.1.4 FPGAs

To achieve reconfigurability in hardware, field-programmable gate arrays (FPGAs) are designed to be configured by users, after manufacturing to form functioned circuit topologies. FPGAs can achieve much higher performance than software implementation in the same design and yet still remain configured for further upgrades. An FPGA platform is composed of a large matrix of configurable logic blocks (CLBs) and a reconfigurable interconnect network in the FPGA platform. The CLB can be configured to execute a certain logic function, while the reconfigurable interconnect network allows the blocks to be wired together, like many logic gates that can be inter-wired to perform certain logic functions. CLBs can be utilized from simple logic operations like AND and XOR to complex combinatorial functions. CLBs may also include memory elements (e.g. registers), to build sequential circuits with clock signals. Clock signals are separately managed with other signals, and are normally implemented via dedicated routing networks. Moreover, in modern FPGAs, integrated multipliers and processors are embedded to improve the execution speed of complex computation, thus, FPGAs can be the platform to build CPUs with hardware reconfigurability. Nonetheless, FPGAs include look-up-tables (LUTs) and a large amount of hardware redundancy to enable reconfigurability. As a result, the performance and power efficiency of current FPGAs are still not comparable to pure ASIC designs, especially for devices requiring real-time performance. In addition, the configuration of FPGAs for large scale circuits is time consuming and needs to be re-loaded again when the power for devices is switched off.
1.1.5 ASIPs

Application specific instruction set processors (ASIPs) combine the efficiency of hardware and the flexibility of software together. In ASIPs, the application specific instruction sets include a basic instruction set for general purpose and a set of application specific instructions for the specific application.

Application specific instructions are designed to utilise hardware that are optimised for specific instructions, in order to accelerate the execution of the target application. In many applications, especially multimedia applications, some vital functions are executed repeatedly and occupy a large amount of total execution cycles. Unlike normal instructions that only execute one operation at a time via general purpose functional units (e.g., ALU, shifter and register files), application specific instructions consist of multiple operations and are executed by customised circuits, so that application specific instructions can finish a series of operation in one instruction. ASIPs are widely used in streaming applications, as the main body of many streaming applications are composed of loops of mathematic operations, and they have real-time performance requirements (such as 20 - 30 frames per second). The performance of ASIPs is significantly better than CPUs since specific hardware is utilised, and they are still able to upgrade their softwares. ASIPs are still reusable for other applications by using the basic instruction subset of the total instruction set; however, when executing new applications, ASIPs cannot take advantage of the specific instructions and will perform like normal processors for these applications.

Once ASIPs are fabricated, their instruction sets are fixed. They are specific to
a particular type of applications, and usually cannot achieve high performance in other applications unless they are similar. However, some latest ASIPs are merged with FPGA techniques. In these products (such as Intel’s Xeon+FPGA [17]), the hardware, which executes application-specific instructions, is implemented in a small FPGA, allowing users to define/load the application-specific instructions according to assigned applications at run time. For example, when such ASIPs, named reconfigurable ASIPs, execute the JPEG encoder, specific instructions for the JPEG encoder are loaded into the FPGA to accelerate the execution of the JPEG encoder. When the reconfigurable ASIPs execute the fast Fourier transform (FFT), specific instructions for the FFT are loaded into the FPGA. If a new application, such as JPEG decoder, is assigned to the reconfigurable ASIP, users can define new application-specific instructions for the JPEG decoder and load the new application-specific instructions when the JPEG decoder is executed. Reconfigurable ASIPs can achieve high performance and power efficiency for multiple applications due to the flexibility of FPGAs. Nonetheless, reconfigurable ASIPs still have the disadvantage of FPGAs. The power consumption and area costs of reconfigurable parts are larger than ASICs, which limits the number of gates of FPGAs in each core. Also, the reconfiguration capability is limited by the size of the FPGA.

1.1.6 MPSoc

Technology scaling now enables designers to place billions of transistors on a single chip. Instead of building a large, complex and power consuming core for mul-
multiple applications, constructing multiple/many cores on one chip allows designers to reduce the design complexity and explore task level parallelism. Multiple processor system on chips (MPSoCs) with hundreds or even thousands of cores are anticipated in the future embedded device industry [18, 19]. MPSoCs can achieve high performance by executing multiple tasks in parallel, allow multiple application to be executed in separate cores simultaneously, and turn off unused cores to reduce power consumption. Hence, MPSoCs are naturally suitable for portable platforms executing multiple applications, such as modern smart phones and tablets, as they can assign applications to separate cores without interrupting each other. MPSoCs are developed from single-core systems, and utilise above techniques (i.e., ASICs, CPUs, ASIPs and FPGAs) to build advanced systems in order to meet power, energy and performance requirements of modern embedded systems.

1.1.7 Challenges in Embedded Systems

In embedded systems, there are several challenges:

1. **[General Platform for Multiple Applications]** Modern portable devices, one of the most important markets for embedded systems, are platforms for multiple purposes (e.g., business, office affairs, studies and entertainment). Hence, these platforms should be able to execute multiple tasks simultaneously, and they should be able to execute increasing numbers of future applications. Examples of such platforms and applications are the iPhone
family with applications in Apple store, and the Android family with applications in Google play. A wide range of applications are provided by software programmers and installed in portable platforms. Despite the need for flexibility, embedded systems require application specific configurations. Designing a general platform with application specific configurations is a challenge for embedded system designers.

2. **[Performance and Restricted Power Budget]** Many portable devices are supported by batteries with limited power budgets. Embedded systems need to satisfy the performance requirement with power efficiency, and unutilised components should be able to be turned off to save power. Additionally, power hungry machines require cooling and heavy heat sinks which are not desirable in portable devices.

3. **[Adaptability]** Modern applications with run-time variations (e.g., workload variation) demand embedded systems to adapt to applications at run time. For instance, in a video that contains low amount of motion and then high amount of motion, the motion compensation in high-motion parts of the video requires significantly more computation than that in low-motion video frames [20]. Such computation changes will result in workload variations, and they are unpredictable at design time as they depend on the input and compute differently at run time. It necessitates the capability of runtime adaptation in embedded systems when the low power consumption is required. The embedded system should increase the computation speed to
satisfy the performance requirement when the workload is high, and the embedded system should decrease the computation speed (or sleep some cores) to save power when the workload is small.

4. **[Product Price]** Designing advanced embedded systems requires a company to expend enormous investment for design, verification and fabrication. Such investments contribute a large percentage of the prices of embedded systems on the market. Consumers in the markets are always inclined to buy state-of-the-art products at the cheapest price. Successful companies are those companies that sell products but with better parameters for the same prices as their competitors. Hence, reducing design cost and design investment are challenges for embedded system companies.

5. **[Time to Market]** Embedded system companies upgrade their products periodically to satisfy consumer demands, in order to enlarge the market share. Typically, major mobile phone companies announce their new products with improved parameters (e.g. higher resolution screen) or new functions (e.g. finger print identification) with a six month upgrade cycle. The design and verification efforts are significant in such a short period, requiring the embedded system to be scalable, compatible and reusable.

6. **[Complexity of Design Space Exploration]** Design space exploration grows significantly due to the diverse combinations of components, such as function units, memory hierarchies, communication methods and task mapping algorithms. For example, in terms of communication methods in an MPSoC
system, options include the Kahn Process Network (KPN), Synchronous Data Flow (SDF), Message Passing Interface (MPI), etc. Regarding memory hierarchies, a large numbers of possible configurations of multiple levels of caches can be selected for processor systems. Moreover, the interactions between caches and other memory layers make the problem of design space exploration even more complicated.

1.2 Run-time Reconfiguration

We believe run-time configuration in MPSoC systems is a promising solution to address above challenges. Firstly, MPSoC systems allow one chip to execute multiple applications in parallelism, and the run-time reconfiguration performs adaptation to make the chip optimised for applications. In this way, the chip can be reused in multiple applications with high efficiency. Secondly, run-time reconfiguration can change the hardware configurations on the fly to adapt to dynamic variations of applications, enduing MPSoC systems with the adaptability to run-time variations. Finally, the reconfiguration allow the system to be changed to for future upgraded in the future. It is impossible to discuss the run-time reconfiguration of all these challenges in Section 1.4.2 in a single thesis. Specific issues are discussed in the following sections of this thesis.

One important issue is the fabrication cost as one of the main investments in the semiconductor industry. Figure 1.1 shows the annual fabrication equipment expenditure between 2010 and 2015; billions are invested into fabrication equip-
ment. The trend seems to be continuous, as the fabrication equipment expenditure is expected to increase in 2015 as shown in Figure 1.2 with a 11% annual increase foreseen in the report in [2]. If embedded systems can be reusable and optimised for a wide range of applications/devices, people will avoid repeating the cost of fabricating application specific systems.

![Aunual Fabrication Equipment Spending](10-14)

Figure 1.1: Fab Equipment Spending from 2010 to 2014, Source: [1]

Different applications and devices require different features for the system, such as data width, parallel computing, computing speed, etc. For example, the requirements of microcontrollers in power, performance and data width are different from the requirements of processors in smart phones. Systems with static configurations may not be able to fit the requirements of different applications, and fabricating application specific systems inevitably increases the costs of fabrication. A run-time reconfigurable system that can adapt to features of applications dynamically is a promising solution to balance the cost of fabrication and exhibit flexibility implementation.
Run-time reconfiguration is also useful in run-time situations where design time analysis is not available. Design time analysis, such as design space exploration, is widely used to adapt the chip system to the application. However, design time exploration is not suitable when application parameters varied in the future. For instance, when software is updated, the parameters may vary, and then the original chip may not be suitable for parameters of the new version. Another example is media applications with motion estimation. The motion estimation amount is determined by the difference between the input frame and the reference frame. Such difference cannot be known until the data is fetched. In the cases mentioned above, adapting to these changes at run time by reconfiguration is necessary for the system to work efficiently.

In this thesis, reconfiguration refers to both the hardware reconfiguration that changes the hardware architecture and the software reconfiguration that enables
Introduction

software to adapt to hardware changes. We discuss three run-time reconfiguration methods that are applicable at different levels (instruction level, task level and application level). In the following section, we introduce the applications that are studied in our run-time reconfiguration.

1.3 Anticipated Future MPSoC Architecture

Figure 1.3: Examples of Current MPSoC Architectures, Source: [3–5]
Many processor design companies have announced a variety of MPSoC architectures with many cores. Some of them are TILE-Gx72 with 72 cores [3]; Intel SCC with 64 cores [21]; and ADAPTEVA E64G401 with 64 cores [22] as shown in Figure 1.3. These products contain a large core matrix and allow users to map multiple applications on this matrix. We anticipate that future MPSoC architectures follow this format. Figure 1.4 shows an methodology to assign cores to applications. A set of cores are assigned to an application (the MPEG in Figure 1.3), while another set of cores is assigned to a different application (the AES in Figure 1.3). In this way, users can assign multiple applications into this architecture and allow them to work simultaneously. The rest of cores that are currently not used are set to idle to save power.

Figure 1.4: Future MPSoC Architecture and Task Assignment
1.4 Target Applications

1.4.1 Applications with Varying Data widths

Recent progress in embedded systems have allowed the realisation of more complex, data-intensive applications in areas such as networking data encryption, video streaming and data compression. Although most applications can be executed efficiently on existing 32-bit-processing designs, many applications process large data sizes. Programs with large data sizes (e.g., 128, 196 and 256 bit) are used in data-intensive applications. An example is the address filter. The address filter filter for IPv6 protocol should check 128-bit addresses, while the IPv4 protocol filter only needs to check 32-bit address. In addition, many mathematical algorithms, such as stein’s algorithm (an algorithm to find the greatest common divisor), also require direct computation of large data.

The core with the large data width can process large-sized data faster than the core with the small data width, and can also process small-sized data. However, the price includes the increased area cost for large data operations, and the corresponding growth of power and the data path length. In a case study based on RISC instruction set, the cost of building a 128-bit core with 64-bit multiplier is $3.2 \times$ size, $3.04 \times$ power and $1.14 \times$ clock width, compared to a standard 32-bit processor [23]. It means if this 128-bit core processes applications with 32-bit data size, its power consumption will $3.04 \times$ higher than the 32-bit core, and $1.14 \times$ slower. Hence, the efficient way is dynamically reconfiguring the data width of cores to match the size of data.
1.4.2 Streaming Applications

Applications constructed around some notion of a stream are named streaming applications [24]. Streaming applications (e.g., multimedia applications, signal processing applications and encryption applications) consume a large number of clock cycles on modern customer electronics [25, 26] and are important in handheld systems as such systems become increasingly popular in daily use [27].

1.4.2.1 Multimedia

Multimedia applications are the most common streaming applications. Multimedia applications are the combination of diverse media forms of audio, image, video and animation to provide entertainment / business information for customers. Multimedia applications are also the backbone of other applications such as video games. With the fast growing market for hand-held entertainment in the form of smart mobile phones, multimedia applications are foreseen to be one of the fastest growing areas in portable devices [28].

In embedded devices, multimedia applications are deployed in the following forms:

1. Image Processing: photographs, image encoding/decoding, image pre-/post-processing, etc.

2. Video Processing: video calls/conferencing, video recording, video encoding/decoding, etc.

3. Audio: audio recording, audio encoding/decoding, voice messaging, etc.
Online videos are one of the most important areas in multimedia applications. Online videos have seen a rapid growth over the past few decades, leading to a high level of internet bandwidth. Meanwhile, customer requirements for resolution quality has raised from Quarter Common Interface Format (QCIF, 176 × 144 pixels) to Standard Definition (SD, 720 × 480 pixels) and onto High Definition (HD, 1920 × 1080 pixels). Further improvements to Ultra High Definition TV and Realistic TV, according to MPEG-4 QCIF [29], are expected, due to users’ desire for better quality / experience. Techniques such as motion estimation is utilised to reduce communication workload by compressing data. However, video coding complexity has dramatically increased, which is attributed to compression [30,31]. Typically, H.265 increases the compression efficiency by four times with the price of 10 × additional computational complexity, compared to previous standards [32].

The computational complexity increases difficulty in meeting the real-time constraint of multimedia applications, (e.g., throughput constraint). For instance, many video applications require a minimum throughput of 30 frames/second, and some high-end 3D games require a throughput of over 40 frames/second. The conflict between processing speed and media quality is the main challenge in embedded devices, especially when the power budget is severely restricted.
1.4.2.2 Other Streaming Applications

Since streaming applications are applications that process data as a stream, many applications apart from multimedia applications can also be programmed as streaming applications. In this thesis, the execution mode of streaming applications is defined in Figure 1.5. The input data is sent to the computer system as a sequential data stream, composing of in-order data tokens, denoted by numbered blocks in Figure 1.5. A corresponding output data token is produced and output from the computer system in the sequence similar to the input data stream, and may be consumed by users. Examples can be found in many areas. In multimedia applications, a smart phone can record videos, and upload them to the Internet at the same time. In this case, the data passes through the phone to the internet as a stream. In internet applications, routers keep receiving packages, filtering them and delivering them to clients as a stream. For database programs, a part of the results can be provided to users before the whole database has been searched. Likewise, many signal processing and communication applications can also be executed in the way of streaming applications (e.g., FFT, ADPCM, etc.).

1.5 Research Aims

This thesis discusses run-time reconfiguration in MPSoC platforms. We studied multiple methods of run-time reconfiguration at different levels to allow an MPSoC chip to be used in multiple applications efficiently. The MPSoC system is composed of identical cores, in other words, a homogeneous MPSoC, in order
Introduction

Figure 1.5: A Typical Streaming Application

to reduce the design complexity and strengthen the scalability. The tasks of this thesis are condensed:

1. Design of an MPSoC platform to deploy run-time reconfiguration methods;

2. Design a run-time reconfiguration architecture for varying data widths of cores;

3. Design a run-time reconfiguration method for streaming applications to improve throughput; and,

4. Design a run-time reconfiguration method for multiple streaming applications in an MPSoC system for low power consumption and high core utilization.

19
1.6 Proposed Methods

Based on the MPSoC architecture with many cores, which we anticipate to be prevalent in the future, we proposed a series of run-time reconfiguration methods with target applications at different levels.

The first method, **DRMA**, allows cores in the MPSoC system to be reconfigured to form cores with varied data widths to adapt to applications with different data sizes. Figure 1.6 shows an example of a set of 32-bit cores. Single cores are assigned to applications with 32-bit data; two 32-bit cores are combined together to form an 64-bit core to process applications with 64-bit data; or four 32-bit cores are combined together to form an 128-bit core to process applications with 128-bit data. By such combination, the data width of cores can adapt to the data size of applications.

![Figure 1.6: DRMA Introduction](image)

**MPSoC**

Figure 1.6: DRMA Introduction

In the second method, called **ADAPT**, we focused on streaming applications. Figure 1.7 (a) shows that the **ADAPT** system assigns different sets of cores to
different streaming applications. At the beginning of each streaming application, an initial number of cores in this streaming application are executing tasks of the streaming application, while the rest cores are set to idle. At run-time, idle cores are added to the task that limits the throughput of the streaming application, in order to share the workload and improve the throughput (as shown in Figure 1.7 (b)); or cores can be shifted from one task to the other task of the streaming application when workloads of these tasks change (details will be explained in Section 4). The target of ADAPT is to maximise throughput of streaming applications with the assigned numbers of cores.

![Figure 1.7: ADAPT Introduction](image-url)

In the third method, called *E-pipeline*, we studied the problem of an MP-SoC system with multiple streaming applications running simultaneously under throughput constraints. As shown in Figure 1.8, the system assigns an initial number of cores to each streaming application, and sets a *common idle core pool*. At run time, cores that are redundant in one streaming application (due to workload variations) are put into the common idle core pool, and cores in the *common idle*
core pool are assigned to streaming applications whose throughput constraints are not met, to stimulate their throughput. The purpose of E-pipeline is to minimise the number of cores used for streaming applications under throughput constraints.

1.7 Thesis Contribution

The contribution of the thesis can be summarised as follows:

- A novel mechanism (called DRMA) allows cores in the MPSoC system to be combined together for varied data width;

- DRMA allows the run-time configuration to be complete cycle by cycle at...
beginning of an instruction execution with little overhead;

- **DRMA** can take advantage of both instruction level parallelism and data level parallelism;

- A novel run-time method (called *ADAPT*) is developed to maintain/improve the throughput of streaming applications on MPSoC systems under a set of allocated cores;

- *ADAPT* allows fast run-time bottleneck detection in streaming applications;

- *ADAPT* can dynamically add assign cores to tasks, or shuffle cores between tasks in a streaming application with small adaptation overhead;

- A novel method (called *E-pipeline*) is developed to dynamically adapt to workload variations of streaming applications, and minimise the number of cores utilised for multiple applications and power consumption;

- *E-pipeline* enables Intra-Elasticity: A method to switch cores within a single pipeline (from stage to stage) to meet throughput constraints; and,

- *E-pipeline* enables Inter-Elasticity: A method to switch cores from one pipeline to a pool of idle cores which are asleep when the cores are not necessary, and a method to obtain cores from the sleep pool to a pipeline when the pipeline needs an additional core to meet the throughput constraint.
1.8 Thesis Outline

Chapter 2 gives the literature survey from the single core to MPSoCs. It also covers a range of techniques that are used or related to this thesis. In Chapter 3, the research methodology is introduced, and details of the proposed MPSoC platform are presented. In Chapter 4, 5 and 6, the methodologies and implementations of the three reconfiguration methods are presented. In particular, Chapter 4 introduces the reconfiguration method DRMA. Chapter 5 refers to reconfiguration ADAPT, discussing dynamic task assignment within one application. Chapter 6 discussed the method E-pipeline to improve the system-level power efficiency and core utilisation under throughput constraints. Chapter 7 presents future work, and the Chapter 8 summarises this thesis. In Appendix A, we provide a DRMA prototype which is implemented in an FPGA board with four cores.

1.9 Summary

This chapter starts by discussing the importance of run-time reconfiguration. Two situations (Application with Varying Data Widths and Streaming Applications) where reconfiguration is implemented are discussed. This chapter introduces architectures used in embedded systems and compares their advantages and disadvantages. The challenges in embedded systems are also discussed. Focusing on the situations studied, the aims of this thesis are condensed. Finally, the contributions of this thesis are proposed, followed by the thesis outline.
Chapter 2

Literature Review

In this chapter, the development of processors from single-core to multiple-core is reviewed, followed by a discussion about techniques that are related to this thesis. In embedded systems, single-core systems have reached their limit in both frequency and power. With technology scaling, engineers are able to integrate hundreds or even thousands of cores into a single chip. Multi-core systems, such as MPSoCs, have become the mainstream. MPSoCs normally need to be designed in an application-specific method to explore parallelism and power efficiency. Nonetheless, the exponential growth of the cost of fabrication of chips forces the embedded system industry to find a way for single chips to be reused for multiple applications to reduce the fabrication cost per chip. MPSoCs with run-time reconfigurability is one of the most promising solutions. MPSoCs increase the parallel execution of applications and maintains strong scalability and high power efficiency for multiple applications, while run-time reconfigurability
allows a chip to be fabricated once and then be reconfigured to be application specific at run time to achieve performance and power efficiency. Run-time reconfiguration technologies have been studied intensively; however, there are still some imperfections in the current reconfiguration technologies.

2.1 Single-core Processor Review

2.1.1 Technology Scaling

In this section, we review the development of single-core systems. In 1971, Intel proposed the first commercial microprocessor 4004. There were 2,300 transistors (10μm technology) on a chip with a maximum frequency of 740kHz. In the 1980s, the Intel 80286 processor [33] worked at the frequency of 20MHz with 0.1 million transistors on a chip (1.5μm technology). The Intel Pentium 4 HT 672 [6] can achieve the frequency as high as 3.8 GHz in 90nm technology. Integrating one billion transistors onto a single chip was achieved in 2010 [33]. The manufacturing technology is still scaling up as shown in Table 2.1, and the processor benefits from the fabrication technologies scaling from μm to nm. Technology scaling [34] brings the following benefits:

1. Decreasing gate delays and increasing operating frequencies

2. Increasing transistor densities

3. Reducing energy, power and price to achieve the same performance
Table 2.1: Technology Improvement

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology</td>
<td>10(\mu)m</td>
<td>1.5(\mu)m</td>
<td>600nm</td>
<td>130nm</td>
<td>45nm</td>
<td>10nm</td>
</tr>
</tbody>
</table>

Technology scaling used to follow the famous Moore’s Law: the number of transistors on a chip doubled every 18 to 24 months. This rule governed the semiconductor industry for over half a century. However, improvement of single-core systems met a bottleneck, and Moore’s Law was challenged by power dissipation issues. Power delivery and dissipation was prohibitive after the year 2000 as the power density on the chip grew exponentially [35]. Figure 2.1 shows typical features of technology scaling, frequency and power of processors between the year 1985 and 2015, and the frequency and power consumption have been saturated since 2005, though fabrication technology is still scaling. Since 2005, even though the technology scaling has reached 14nm, most processors maintain the frequency under 4GHz.

![Figure 2.1: Technique Trend Between 1985 and 2015, Source: [6]](image-url)

27
2.2 Power Consumption and Frequency

2.2.1 Relationship between Power and Frequency

In embedded systems, the power consumption of processors is one of the main concerns as many embedded systems are supported by batteries and heat sink design in portable devices is restricted by the size and weight. The power consumption of the processor can be modeled as shown in Equation 2.1:

\[
P = P_{\text{dyn}} + P_{\text{leak}}
\]  

(2.1)

The \( P_{\text{dyn}} \) is the dynamic power, while the \( P_{\text{leak}} \) denotes the leakage power. In modern processors, dynamic power is still the most significant power consumption and can be denoted as shown in Equation 2.2:

\[
P_{\text{dyn}} = C \cdot V^2 \cdot F
\]  

(2.2)

\( C \) is the capacitance being switched per clock cycle; \( V \) is voltage; and \( F \) is the frequency of the component. Since the voltage required for stable operation is determined by the frequency of the circuits [36], if the frequency varies, the power required in the processor may vary significantly as well. For example, if the frequency \( F \) declines, the operation voltage \( V \) can also be reduced, resulting
in an amplified decrease of $V^2 \cdot F$ in Equation 2.2, and therefore finally the total dynamic power can be significantly reduced.

Dynamic voltage and frequency scaling (DVFS) takes advantages of the relationship between frequency and power. It scales the frequency of processors and the corresponding operating voltage, thus the dynamic power consumption can decrease significantly [37]. Clock gating techniques can set a core to idle temporarily by blocking the clock signal [38, 39]. In clock gating, since the frequency $F$ is considered as zero, there is no dynamic power and the system only consumes leakage power. Moreover, power gating technologies [40] not only turn the clock signal off but also turn the whole core off, eliminating both dynamic power and leakage power. However, power gating technologies normally consume much more time/energy overheads than clock gating technologies [41].

2.3 Instruction-level Parallelism Review

The relationship between power consumption and frequency also indicates the potential of exploring parallelism. For example, if designers double the performance of a single core by doubling the frequency, the dynamic power consumption will increase more than $2 \times$ (can be as high as $8 \times$) due to the factor of $V^2 \cdot F$ in Equation 2.2. Nonetheless, if designers double the performance by parallelising tasks into two cores (let us assume that tasks can be perfectly parallelised), the power consumption just doubles. Due to power and frequency walls, rising frequency is not feasible, hence, the exploration of parallelism has become one of the most popular
techniques in the embedded system industry. In the following sections, widely-used technologies exploring instruction-level parallelism in single core processors are discussed (in Section 2.3.1 – 2.3.3); after that, task-level parallelism through multiple cores (i.e. MPSoCs) is discussed.

2.3.1 Hardware Pipeline Review

Process Flow of A Core with Pipeline

![Diagram](image-url)

- **IF**: Fetch an new instruction
- **Dec**: Decode an instruction and fetch data from the register file
- **Exe**: Execute an instruction via functional units (e.g., ALU)
- **MO**: Access the memory (read or write data into the memory)
- **Wait...**: Wait because the memory access may take multiple cycles
- **RO**: Access the register (read or write data into the register)

Figure 2.2: Five Stage Pipeline
Hardware pipeline architectures have been widely used in many processors to increase the parallelism of instructions. In hardware pipelines, the execution of instructions is divided into a set of sequential segments, which are executed by dedicated functional units. Hardware pipelines allow functional units to process segments from successive instructions in parallel, similar to an industrial assembly line [42]. Figure 2.2 depicts a five-stage pipeline example. The execution of instructions is divided into five stages. *Stage IF* (i.e., instruction fetch) fetches the next instructions from the instruction cache; *Stage Dec* (i.e., encoding) decodes the fetched instructions and fetches data from the register file; *Stage Exe* (i.e., execution) utilizes functional units (e.g., ALU, Shifter, multiplier) to execute the instructions; *Stage MO* (i.e., memory operation) writes and reads data from the memory; *Stage RO* (i.e., register operation) writes the result to the register file. Note that, between *MO* and *RO*, the processor may have to wait (Wait... in Figure 2.2), because the memory access may take multiple cycles. In such a hardware pipeline, when the result of instruction $i$ is writing back to the register at the stage *Stage Register Operation*; instruction $i+1$ is executed at the *Stage Memory Operation*; instruction $i+2$ is processed by some functional units, such as ALU; instruction $i+3$ is decoded; and the processor is fetching the instruction $i+4$ from the instruction cache. Thus, the pipeline allows multiple instructions to be parallelled, and the processing speed of hardware pipelines is determined by the longest data path of the stages rather than the longest data path of the instructions [43]. Large numbers of modern processors operate with hardware pipeline architectures to improve their performance [44,45]. In some recent architectures, their pipelines
consist of multiple branches for different types of instructions. The pipeline architecture of processor ARM11 [7] is shown in Figure 2.3. The pipeline selects different branches to process integer/bit operations, multiplication and cache access operations separately, allowing overlap various stages of instructions through different branches.

![Figure 2.3: ARM 11 Pipeline, Source: [7]](image-url)

Hardware pipelines are also used for acceleration of specific instructions by adding extra stages to these instructions. Webb in his paper [46] discussed the method of executing time-consuming instructions, such as floating-point calculations, in extra segments to reduce the datapath of each stage. They implemented dedicated circuits and extra pipeline stages for floating-point calculations. Shigeo et al. in their paper [47] discussed the processor architecture to improve the execution speed for instructions which are used in database operations; they also applied dedicated circuits and extra pipeline stages to reduce the time of accessing
The limitation of pipeline architectures is that the design efforts grow exponentially with the increase in pipeline depth [50]. When the number of stages in a pipeline increases, the complexity of pipeline controller circuits increase dramatically; the chip size increases; and the time overhead of pipeline controlling reduces the final improvement in performance. All these drawbacks hinder the expansion of pipeline lengths.

2.3.2 Super Scalar Review

Super Scalar processors achieve performance improvements by exploiting parallelism at instruction-level, based on pipeline architectures. A super scalar processor contains multiple independent functional units which can be executed separately.

A hardware scheduler in the super scalar processor analyses the dependence between instructions, packs instructions to independent instruction packages, and delivers these instruction packages to different functional units, where instruction packages are processed in parallel [51]. Zhang et al. [52] discussed the method of adapting the instruction-level parallelism to application behaviour, in order to optimize the power-performance ratio.

FabScalar in [53] offered a chain of tools to automatically generate the register-transfer-level (RTL) descriptions of canonical superscalar templates. Large numbers of implementations of canonical pipeline stages are recorded in a canoni-
Literature Review

cal pipeline stage library (CPSL), and then users can further modify canonical pipeline stages. After the user defines the key parameters, an RTL generation tool uses these implementations in CPSL to output the RTL hardware description of the target architecture. Three level verifications including instructions level validation, timing validation, and standard ASIC flows confirmation, are also included in FabScalar.

The amount of instruction-level parallelism (ILP) that can be exploited in super scalar processors also relies on the efficiency of the scheduler. Furthermore, the scheduler has to rearrange the instruction queue if the branch predictors fail. Hence, it depends on the optimization of the software compiler and the programmer. When there is a large amount of instructions that contain data dependency, the scheduler cannot execute them in parallel. Yeager, in his work [54] presented a typical architecture with four instruction queues. The instruction scheduler occupies three stages, where up to four instructions are fetched, decoded, renamed and queued in parallel. Four branch predictors are used, and they can achieve average 87% prediction accuracy for integer benchmarks.

2.3.3 VLIW Review

Very long instruction word (VLIW) processors explore instruction-level parallelism by executing long instructions that are composed of sub-instructions. Compilers view instructions as sub-instructions, and compose sub-instructions that can be executed in parallel together to form long instructions. After being fetched by the VLIW core, those sub-instructions in one VLIW long instruction are then
decoded and executed in separate functional units. Compared to super scalar processors, VLIW processors take full advantage of the compiler to explore the instruction-level parallelism, without the arrangement work of the hardware scheduler.

Aditya et al. in their work [55] described automatic design and synthesis for VLIW processors. The designers specified the functionality, defined the input/output interface, instruction set, data path, etc. at the system level. Their proposed mechanism automatically outputs the RTL level structure of the VLIW processor as well as an estimation of area costs. VLIW architectures are typically used in digital signal processors (DSPs), such as the TMS320C6x family [56].

The performance of compilers plays an important role in the performance of VLIW processors. Colwell et al. [57] showed a typical example of the VLIW compiler algorithm for a VLIW architecture with up to 28 operations in each 1024-bit instruction. The compiler first performed normal optimizations for the source code as per common compiler, such as induction variable simplification. After the common optimization, the VLIW compiler built a flow graph of the program, showing dependencies between instructions and grouped sequential instructions which could be executed in parallel. If there were branch instructions, the compiler applied greedy algorithm in branch prediction and executed the predicted jump in advance. In the case where the prediction was wrong, a predesigned compensation code was executed to undo these operations in the predicted direction. According to different applications, they can reduce 10-60% size of the VLIW code. Rajagopalan et al. [58] showed a optimised VLSI compiler for DSPs.
They analysed the feature of their example processor (Fujitsu Hiperion fixed-point DSP), and exploited irregular instruction-level parallelism in DSP architectures to optimize loops via a preprocessing algorithm, significantly reducing the size of DSP code. Their results showed a nearly double performance compared to the comparison group.

The limitation of VLIW processor is the compiler complexity. To analyze the whole program and parallel instructions properly, VLIW architectures need advanced and large compilers, which consumes a large amount of time to develop such compilers and compile VLIW programs. The compiler lacks techniques to measure the latency variations of hardware units, and this causes synchronization issues. Rau et al. in [59] implemented a hardware controller to detect the latencies and synchronize these operations.

2.3.4 Summary

In this section, we reviewed the history development of processors. The industry gained dramatic benefits from technology scaling. However, the frequency of the core had already met its limit due to the power wall and corresponding problems (thermal dissipation and energy budgets). Parallelism and low power design have become the main trend method to combat these limits [35]. Designers attempted to parallelise instructions to increase the processing speed. Instruction-level parallelism technologies include pipeline architectures, super scalar architectures and VLIW architectures.

Hardware pipeline, super scalar and VLIW architectures allow a single core to
process multiples instructions in parallel [44, 45]. Hardware pipelines are widely used in architectures, but the efficiency of hardware pipelines is affected when there is data independence between sequential instructions [50]. Super scalar cores could achieve instruction-level parallelism via hardware schedulers. For VLIW cores, instruction-level parallelism is explored via their compilers which compose instructions to form long instruction words. For instruction-level parallelism, their complexity significantly increases when the parallel threads rise, which limits the expansion of instruction-level parallelism.

2.4 MPSoCs

2.4.1 From Single-core to Multi-core

There are several issues to consider when building powerful single core processors. Some severe problems are frequency, power and complexity. Designers suffer from the high power consumption due to high frequency in modern chips, and the instruction-level parallelism is limited by the data dependency between instructions. The high design complexity of integration requires more logic units: the speed up is roughly linear to the square root of the increase in complexity (Pollack’s Rule [60]). Multiple cores are not necessarily as powerful as the highest performing single core processor; however, they can achieve high level parallelism at task-level to improve the overall performance at the system level [61].

Compared to the parallelism technologies for single core processor we discussed above, multi-core systems explore task-level parallelism (TLP) rather than
ILP. In task-level parallelism, the parallelism mechanism executes and synchronizes multiple tasks in parallel. Since tasks are exposed to software programmers, they can adroitly design tasks for superior performance. Multi-core systems with TLP achieve nearly linear performance improvement in independent tasks without adding design complexity. Each core in the multi-core system can be configured to different power states for power saving. The system can turn some cores off independently when they are currently not used. Moreover, integrating multiple cores into a chip allows multi cores to share components such as memory cache, and significantly reduce the communication latency and cost.

Many designers of embedded systems have already turned to MPSoC systems. Million or even billions of transistors can be fabricated in a chip so that we can build hundreds and even thousand of cores inside. System-on-chip design reduces the latency of communication between cores and peripherals. Many processor companies have issued MPSoC systems, such as Intel SCC, IBM Cell, ADAPTEVA EPIPHANY, NVIDIA’s Tesla, etc.

Intel SCC [21] is a system with 48 x86 cores on a single die area. Every x86 core is attached with a 16 KB Level 1 instruction cache/data cache, while all 48 cores share a 256 KB Level 2 cache [62]. Cores are connected by a network which allows a maximum 256 GB/s bandwidth. Each core can run at 800 MHz, and the DVFS manager controls each core separately to save power and energy without affecting other cores [63].

The IBM Cell is composed by one power processor element (PPE) and 8 synergistic processing elements (SPEs). The PPE manages the tasks and resources
of the system, while SPEs take on the work of computation. The communication between cores is through element interconnect bus (EIB) [64].

ADAPTEVA’s new EPIPHANY processor family E64G401 contains 64 high performance RISC cores with 2 MB shared memory. The communication is through 102 GB/s network-on-chip (NoC) system (1.875ns per-hop latency). The operating power consumption of the chip is less than 2 Watt [22].

NVIDIA’s Tesla is implemented in a hierarchical architecture. A Tesla core contains 16 streaming multiprocessors (MPs), and each MP contains 8 streaming processors (SPs). The SPs in a SM are connected to a 16-Kbyte shared memory by a interconnection network. The communication between different SMs are through another interconnection network [65].

2.4.2 Heterogeneous MPSoC

Heterogeneous MPSoCs contain multiple cores with different configurations and architectures. Modern portable devices, such as smart phones, are a good illustration of heterogeneous MPSoCs. The smart phones which we use daily may consist of eight cores: one main processor for the OS and general purposes; one ASIP core processing protocols and other control functions; one DSP core, responsible for voice encoding and decoding; an audio ASIP core; a camera ASIP core; and even a video core for new video-call functions [66]. As the usage area of each chip remains the same, most cores are optimized for certain functions. Such heterogeneous MPSoCs achieve high performance and low power consumption, as each core is specifically optimised for one function. However, due to the dif-
ference of cores, heterogeneous MPSoC systems are restricted in their flexibility and scalability. For example, tasks cannot be migrated from one core to another, as their instruction sets are different [67]. If one core is destroyed, its work cannot be taken by another core. Furthermore, heterogeneous MPSoCs increase the complexity in design space exploration, as the amounts of possible task combinations grow exponentially with the increase of cores.

### 2.4.3 Homogeneous MPSoC

Homogeneous MPSoCs are composed of replicated versions of identical core units [68]. They provide better fault tolerance, scalability and flexibility than heterogeneous MPSoCs. Sergio Saponara and Luca Fanucci [68] implemented a heterogeneous MPSoC and a homogeneous MPSoC in an H.264/MPEG AVC video codec and a low-distortion digital audio amplifier application. Their results showed that the heterogeneous architecture gained better power efficiency and smaller areas while the homogeneous architecture enabled higher flexibility and easier scalability. However, Jalier et al. [69] proposed a reconfigurable homogeneous MPSoC architecture and compared it to two heterogeneous architectures in the application of a mobile 4G LTE modem. Their reconfigurable homogeneous MPSoC architecture achieved 3% better performance and 18% better power efficiency than current heterogeneous architectures, as the homogeneous units reduced the complexity and communication overhead of the communication network. Therefore, homogeneous MPSoCs are preferred when designers explore flexibility and scalability.
2.4.4 MPSoC Systems with Small Cores

Under the certain amount of transistors, MPSoC systems with small cores are better than MPSoCs with large cores. For example, in a single die of 1 billion transistors, we can build 500 cores with 2 million transistors in each core, build 100 cores with 10 million transistors in each, or build a heterogeneous architecture with multiple type of cores with differing numbers of transistors. Shekhar Borkar in his work [70] showed a comparison of different multi-core systems with the same number of total transistors in the same technology. Based on the same amount of total transistors, the performance of a core with a small amount of transistors was less than a core with a large amount of transistors, the total performance of multi-core system with small cores was better than the multi-core system with large cores. Flexibility and power efficiency can also be achieved, as the performance of a task can be knobbed by changing the number of cores working on it, or switching the task from powerful cores to small cores. Huang Wei et al. studied the relationship between core size and chip thermal distribution based on air cooling [71]. Their experiments showed that small cores were generally cooler than large cores in the same power density, and show the advantage of MPSoCs with many cores.

2.4.5 Summary

MPSoCs are increasingly popular in embedded systems. They achieve parallelism at a high level as task-level and improve the overall performance at the system
level [61]. Increasingly numbers of multi-core architectures have been presented in the market [22, 62, 64, 65]. Multi-core architectures can be classified into two – heterogeneous and homogeneous. Heterogeneous multi-core systems consist of ASIPs to achieve high performance and low power consumption in a specific application area [66], while homogeneous multi-core systems are better at scalability and flexibility [69].

2.5 Reconfiguration

As discussed in Section 1.2, reconfigurability is desirable. Making application specific chips exponentially increases fabrication cost. The investment of building a new fabrication line for 14 nm technology costs billions of dollar. There is also expensing hardware testing, verification, expensive external equipment (such as wafer probes, oscilloscopes and analyzers), and mask costs. Moreover, static architectures cannot be revised in the future. For example, when widely used methods are replaced by newer method, hardware companies have to fabricate new chips, optimizing for new methods. The run-time reconfiguration capability that allows one chip to be adapted to multiple applications is a promising solution to address fabrication costs. It does not need to be fabricated separately for different chips. Engineers can fabricate one chip and reconfigure this chip at run time for different usage. Hence, it can also adapt to updating to new methods and run-time workload variations. In the following sections, reconfiguration techniques are discussed.
2.5.1 FPGA

Soft cores built on Field-Programmable Gate Array (FPGA) are programmable, reversible and customizable. Modern FPGA boards contain specific mechanism to build soft cores. For example, Altera and Xilinx allow processing components such as multipliers, dividers, memories and floating-point units to be instantiated in their FPGA boards in advance, which allows designers to generate soft cores efficiently [72]. Modern FPGA boards also offer systematic tools such as soft-core libraries and compilers to support processor designers to utilize the FPGA board.

Hammami et al. in their work [73] discussed the methodology of designing many-core systems in Xilinx FPGA boards and offered a case study of a system with 672 processor elements (PEs). Lukovic et al. in their work [74] proposed a design flow to generate softcore architectures by Xilinx Embedded Development Kit (EDK) [75,76] on Xilinx Virtex-II Pro board. Some FPGA boards also enable the run-time reconfigurability in FPGA soft core systems [77,78]. Clemente et al. proposed an low-reconfiguration latency architecture on FPGA based processor system with 55% overhead reduction in experiments. Gohringer et al. proposed Runtime adaptive multi-processor system-on-chip (RAMPSoC). Every core in RAMPSoC contained one or more accelerators on the FPGA. Accelerators can be optimized at run time to extend the instruction set for specific applications.
2.5.2 Run-time Reconfiguration

In FPGA, reconfiguration can be achieved at the circuit level. However, reconfigurability could be also achieved at the level of cores by changing the behaviour and communication of cores.

Sun Fei et al. [79] in their work proposed a multiprocessor system built by extensible processors. Extensible processors are configurable base processors where custom instructions are implemented to extend their instruction sets. These instructions are used to accelerate specific operations that are executed frequently. By reconfiguring the extensible part, the processor can be optimized and evolved according to changing requirements. Chen and Mitra in their work [80] exploited the imbalance in execution time and custom instructions requirement among processors. They proposed an architecture with a shared reconfigurable fabric to improve the extensible cores in MPSoCs. One challenge of extensible processors is their design complexity since the custom instructions have complex interdependencies with task assignment and scheduling. Therefore, the estimation of execution time was important in extensible processors. In addition, the extensible amount was limited to the space of extension in processors. Bouwens et al. proposed the Architecture for Dynamically Reconfigurable Embedded Systems (ADRES) in work [8]. The architecture of ADRES is shown in Figure 2.4. ADRESs employed a VLIW processor to fulfil ILP and reconfigurable matrixes to fulfil DLP, so that ADRESs could gain benefits from both parts. The reconfigurable matrixes, which were composed of functional units and register files, were utilized for accelerating parallel independent tasks. The data flows through matri-
ces were through multiplexors, and the configuration of multiplexors was stored in the configuration RAM, which could be fetched in a few cycles. The VLIW executed the non-parallel part of the code; however, the VLIW compiler still explored instruction-level parallelism (ILP) as per normal VLIW processor. Nevertheless, manual intervention was needed at design time to achieve best performance and the reconfigurable parallelism was limited to the sizes of reconfigurable arrays. Mei et al. in their work [81] provided a case study in multimedia benchmarks in C program, and their experimental results showed that ADRES could achieve roughly 3 - 5 × speedups.
Ansaloni et al. in their work [9] presented an expression-grained reconfigurable array (ERGA) architecture. The architecture of ERGA is shown in Figure 2.5. The array could be configured to form a better equalization of critical paths over different elements and to a flexible design space. To some extent, it was similar to ADRES. However, it focused on the performance of multi stage computations, while ADRES explored strong instruction-level parallelism capabilities.

Goldstein et al. proposed the PipeRench architecture in their work [10]. As shown in Figure 2.6, the architecture consisted of a series of simple processing elements (PEs), including ALUs and register files. Different functions could be
implemented to form hardware pipelines by reconfiguring the PEs. This architecture was used to achieve high throughput in static parallelism with the overhead of reconfiguration time.

![PipeRench Architecture](source)

**Figure 2.6: PipeRench Architecture, Source: [10]**

Kim et al. in their paper [82] proposed a multi-core architecture named a composable lightweight processor. This architecture was composed of a number of small and narrow-issue cores that could be integrated together to form powerful cores in on-the-fly reconfiguration, so that the number of cores and their sizes were adjusted at run time according to application requirements. The operating system could perform management according to the number of threads working in parallel and their criticality, while the hardware adjusted the number of cores per thread in an automated fashion.

Pricopi et al. discussed the architecture Bahurupi in their work [11]. Bahurupi
targeted instruction-level and thread-level parallelism (TLP). It was composed of out-of-order superscalar cores. The architecture of Bahurupi is shown in Figure 2.7. It could efficiently merge 2-4 simple 2-way out-of-order cores to reach or even surpass the performance of more complex and power-hungry 4-way or 8-way out-of-order core, or used these 2-way cores separately to explore task-level parallelism.

Khubaib et al proposed the Morphcore architecture in their paper [83]. The Morphcore architecture was based on a large out-of-order core; however the in-order multi-thread capability was added as an extra working mode. Morphcores architecture could work in out-of-order core mode to explore ILP or in-order multi-thread to explore TLP. The switch between instruction-level parallelism and
task-level parallelism did not need migration of instructions or data in Morphcore, which led to minimal cost in latency and power.

Porrmann et al. in their work [12] introduced a homogeneous MPSoC platform with strong scalability and fault tolerance. They offered a GigaNoC – a hierarchical scalable NoC that can integrate varying number of cores into one cluster, shown in Figure 2.8. The interconnection of cores enabled dynamic reconfiguration when faults happened. The non-functional resources of the core would be replaced by the corresponding functional resources borrowed from the nearest neighbor core.

![Figure 2.8: MPSoC with GigaNoC, Source: [12]](image)

Run-time reconfiguration can also be achieved in heterogeneous MPSoCs through run-time task mapping or dynamically adaptive NoCs. Gohringer et al. in their
work [84] designed a runtime adaptive star-wheels NoC on heterogeneous MPSoCs. According to the tasks being executed, the NoC network can be configured at run time to offer a proper bandwidth for each task. Singh et al. applied runtime task mapping in heterogeneous MPSoC systems. They computed multiple design points with different performance/energy ratios, through design space exploration. At run time, the system dynamically selected the proper design point and performed the configuration of the design point.

Teich et al. introduced the invasive computing [13] paradigm for future parallel computing systems. According to workload variations of applications, invasive computers regulated resource assignments to adapt to workload variations. For example, the application with a heavy workload got more cores while the application with a light workload got less cores. The core assignment variations were managed by invasive computers at run time, based on dynamic workload situations. The architecture of invasive computing is shown in Figure 2.9. CPUs are standard RISC cores; the i-Cores are used for invasive computing; and there are tightly-coupled processor arrays (TCPAs) for loop operations.

### 2.5.3 Summary

The chip that can be reconfigured at run time to adapt to multiple applications is highly valued. FPGA boards have been widely utilized to build reconfigurable MPSoC systems [75, 76]. However, the configuration time of FPGA was at the level of seconds, and the performance and power efficiency were still much less than ASIC. Various types of reconfiguration techniques based on MPSoCs have
been studied as alternative solutions [9–13, 82]; and they can achieve significant speed-up by reconfiguration. Nonetheless, as far as we know, there is still a lack of techniques that can be used to reconfigure to vary data widths of cores.
2.6 Optimisation in Streaming Applications

Throughput, which represents how much data can be processed during a time unit, is a widely-used measurement standard in streaming applications. Many researchers focus on methods to improve throughput of streaming applications or reduce power and resource consumption under certain throughput constraints.

2.6.1 Throughput Improvement at Design Time

In this section, we discuss methods at design time to improve throughput of streaming applications.

Javaid et al. proposed a fast design time exploration method [85] for streaming applications. Their method was based on processor pipelines, and aimed to balance the workload of each core in the pipeline at design time. When workloads of different cores were balanced, the throughput achieved the highest point. In particular, they applied ASIPs to balance workloads and FIFOs as communication buffers in their pipelines.

Zhai et al. in their work [86] worked on Synchronous Data Flow (SDF) to find a just enough parallelism in a pipeline. Just enough parallelism meant that all the available cores were fully utilized, hence the throughput was maximized in current available cores. In SDF, tasks could be replicated and parallelised, and the number of replications was defined as the unfold factor. They proposed a method to determine a proper unfolding factor for each task [86], and the unfolding factor guaranteed that the workload of each core was balanced and the throughput of
each stage matched each other.

Gordon et al. in their work [87] proposed a method with the support of StreamIt compiler. They applied a unified analysis to exploit varying types of parallelism (including TLP, ILP and pipelines) based on the StreamIt [88], which was a program language designed for streaming application programming. They applied their method in a 16-core MPSoC architecture. Experiments results showed that their method could obtain a speedup of 11.2 times of the baseline core.

Design time optimisation can achieve high throughput with low-power consumption and area cost. Design time optimisation normally considers the worst case scenario so that the required throughput can always be achieved. Thus, these methodologies suffer from over-design and are incapable of adaptation to resource/workload variations at run time. Also design time optimisation assigns tasks to cores based on the hardware resources at design time. However, hardware resources may vary at run time. For instance, the devices might be updated, or some resources were unavailable (e.g., damaged) at run time. Hence, design time methods might not be optimal, or they could not achieve the estimated throughput.

### 2.6.2 Adaptation in Initialisation

Mapping-time methods [89–91] can execute reconfiguration at the start of an application’s execution by looking at the available cores and mapping tasks accordingly. Their methods assume that workload information is known in advance and lack adaptability during application execution.

Hölzenspies et al. proposed a reconfiguration work [89] for MPSoCs. Their al-
algorithm allocated hardware resources to a streaming application before the streaming application began. In their architectures, they assumed their cores were connected by an NoC network. They applied a spatial mapping algorithm. When the mapping work started, the target application was first analyzed based on the Kahn Process Network (KPN). The mapping algorithm computed a mapping for the application, and then attempted to map tasks to available low-power resources to minimize the power consumption. After that, communication paths were analyzed to find the minimum communication. The next step was estimating whether the implementation violated the requirements of current running applications. If the implementation satisfied the requirements of current running applications, this implementation was executed, or the algorithm returned a step backward to search for an alternative solution.

Lee et al. presented a dynamic scheduling method [90] based on the StreamIt program language [92] and the IBM Cell BE platform. They aim to maximise throughput and resources variation. The scheduling methodology consists of two steps. The first step was called compile time analysis. The compiler analyzed the synchronous dataflow (SDF) of the application which was specified in StreamIt. In StreamIt, the SDF demonstrated the graphical architecture of the streaming application which showed the data tokens produced and consumed by tasks (named actors in StreamIt). The compiler analyzed the SDF and generated retargetable codes for all the actors in this application, which could be scaled up or down to adapt to varying numbers of cores. At the final stage of compile time analysis, the memory requirement, workloads and other schedule information were computed.
for run-time adaptation. When the system starts a new program, a reconfiguration algorithm was applied to exploit currently available resources, calculate the workload of each core, unroll stateless tasks and compute an optimized pipeline schedule to maximise throughput under current resource constraints. In addition, if the number of cores was insufficient in this MPSoC system, the scheduler would re-distribute tasks to find an alternative pipeline structure. If such a solution was not found, the scheduler would signal the failure. This mechanism enhanced the reliability of the system. According to their experimental results, their approach achieved $3 \times$ performance improvement.

Barker and Chatha in their work [91] proposed another schedule work based on the SDF generated by StreamIt compiler. Compared to the work in [90], they focused on the applications without backward data path, which meant the SDF was acyclic; hence they limited the amount of information at compile time. Most StreamIt benchmarks including signal processing and multimedia programs were acyclic. For cyclic applications, they may be refined to be acyclic by merging tasks in the cycle together as a single task. At the start of the program, a lightweight scheduler detected all available cores, and assigned tasks to available cores based on a workload-balance algorithm. According to their experimental results, their scheduler could achieve solutions within 5% of the optimal schedule.

2.6.3 Run-time Adaptation Work

The run-time balancing methods [93–95] used a fixed number of cores to a streaming application, and moved/duplicated tasks among those cores to improve through-
The main limitation of these methods is that they take considerable amount of time to perform run-time adaptation.

Jahn et al. in their work [93, 94] proposed a run-time remapping method called *Pipelet* to adapt to workload variations. *Pipelet* was a self-organizing run-time remapping methodology for pipelines in MPSoCs with a large class of cores. They focused on pipelines which were acyclic and malleable. The malleability means small stages of the pipeline could be merged into a bigger stage. Merging cores in the task remapping work helped the system balance workloads between cores and reduced communication overhead. They took a distributed architecture, as each core calculated its own workload and compared it with neighbouring cores locally to find the local bottleneck. When the bottleneck was found, the core called a help function named ‘Core Guard’ to work out a possible remapping plan with neighbouring cores. The remapping plan was estimated at run time to decide whether such a remapping would increase the throughput. If the estimation showed that the remapping increases the throughput, the remapping plan would be executed; otherwise, the remapping work would be denied. They evaluated their work with Intel’s Single-Chip Cloud Computer (SCC) which consisted of 48 x86 (P54C) cores. Their results showed that they could achieve 31.2% throughput improvement compared to state-of-the-art task mapping algorithms without a remapping mechanism.

Choi et al. [95] proposed a run-time bottleneck detection and adaptation method on general purpose multi-core architectures. They focused on run-time workload variations in streaming applications. Compared to works in [93, 94], Choi et al.
performed task duplications at software thread level without really assigning them to different cores. After duplication, they assumed that the operating system was equipped with a scheduler that attempted to distribute threads to cores efficiently and properly. In task duplication, they divided the pipelines into stages, and each stage contained one master thread and a number of slave threads. The task of the master thread was activating/inactivating slave threads to control the number of slave threads, while the slave thread performed the task of this stage on a data token. In their method, the slave thread realized data parallelism under the control of the master thread. At run time, the master thread estimated the workloads and regulated the number of slave threads to balance workloads between stages. Their results showed that their method can achieve $4.5\times$ speed-up compared to the multi-threaded baseline.

The run-time methods mentioned above [93–95] consume a large amount of time to perform run-time adaptation. For instance, work in [93] took at least 0.18 milliseconds on an Intel SCC, while [95] took at least several milliseconds to detect bottlenecks. Using 800 MHz running frequency of Intel SCC, the overhead of 0.18 milliseconds indicated that resource/workload variation can only be detected every 144,000 clock cycles. This was not suitable for modern streaming applications with rapidly varying workloads [96].

### 2.6.4 Low Power Technology in Streaming Application

Power consumption is a major concern in many portable devices as they are supported by batteries and the weight of their heat sinks are limited. When through-
put constraints are met, low-power techniques are applied to reduce power consumption in these devices. Researchers studied power saving technologies under throughput constraints for streaming applications.

Javaid et al. in [97] proposed a run-time low-power method based on clock/power gating methods. Power states were utilised to denote clock gating or power gating technologies with different power-saving factors and time/energy overheads. The prediction algorithm (MAMAPBH) was applied in this work to predict the best possible workload for coming data and worked out the most suitable power states to reduce power consumption and maintain a certain level of throughput. Their method saved up to 40% power consumption in streaming applications.

Alimonda et al. proposed a feedback control method with DVFS for streaming applications [98]. In their work, they applied a feedback-control method to estimate the workload of each stage. First, they applied FIFO as the buffer for the communication between stages; the utilization of FIFO was measured as the input of feedback control. According to their theory, the utilization of FIFO indicated workload situations of stages. Under the throughput constraint, they first worked out a standard FIFO utilization, and their feedback control algorithm regulated the frequency and voltage to adjust the utilization rate of each FIFO to approach the standard FIFO utilization. Their results indicated that the method achieved 20% to 50% power reduction.

Jung et al. proposed a power management method in [99] based on the supervised learning algorithm. A power manager predicted the workload based on the supervised learning algorithm, utilized the predicted workload to compute opti-
Literature Review

mal voltage-frequency settings, and assigned voltage-frequency settings to cores. Utilizing the supervised learning method reduced the time overhead of the power manager and increased the prediction rate. Experimental results indicated their work could reduce up to 20% of power consumption.

Ge et al. proposed a run-time frame task mapping work [100]. They designed an agent that measured the workload and the temperature of each core, and the agent also exchanged tasks between neighbouring cores. The goal of such task migration is to distribute high power tasks to different cores based on their power budgets and also ensure that the heat dissipation of each core remains under constraints. Their results showed that their method reduced the thermal gradients by 66.23%, hotspots by 66.79%, and improved performance by 40.21%.

2.6.5 Summary

Streaming applications are increasingly useful in embedded systems, and widely executed as pipelines in MPSoCs for high throughput. Many studies were performed to achieve the maximized throughput for available hardware resources, while studies to meet throughput constraints with minimum power consumption are also highly valued. Traditional design time methods, such as work in [85–87], cannot achieve the best throughputs when the resource and workload vary. Researches in [89–91] proposed techniques that allow the system to map pipelines to available resources at run time. In this way, they can adapt to resource variations; however, they still have to consider the worst-case scenario of workloads, because they do not measure run-time workload variations. Work in [93–95] proposed dy-
namical adaptation methods for run-time workload variations. They applied run-time workload measurement techniques in their systems, and detected the bottleneck stage in a pipeline. Studies [89, 90] remapped the bottleneck task to another core for workload balance, in order to improve the throughput; while another work [91] applied task duplications to the bottleneck stage. Nonetheless, their methods [93–95] are still time-consuming, which is not suitable for modern embedded applications with fast workload variations. Other studies [97, 98, 100, 101] applied dynamic low-power technologies in pipelines with static workload assignments. The drawback of previous studies [97, 98, 100, 101] was that they always occupied a fixed number of cores, even though these cores were not fully utilised (e.g., these cores might be set to idle, but could not be utilised by other applications). As far as we know, there is still a lack of methods that can perform run-time reconfiguration to adapt to fast workload variations in modern streaming applications.

2.7 Summary

In this chapter, we started with the survey of technology scaling in single-core processors. Improvement in the frequency had already approached physical limits, forcing the embedded system industry to explore the potential of parallelism. We described techniques that are widely utilized for single-core processors and compared them with MPSoCs. MPSoCs ranging from tens to hundreds cores can explore task-level parallelism efficiently and reduce design efforts. Between
heterogeneous architectures and homogeneous architectures, we focused on homo-
geneous architectures due to their scalability and flexibility. Particularly, we
reviewed hardware/software pipelines, which were widely used in streaming ap-
plications for high throughput. In pipelines, workload balancing is vital for high
throughput. Designing-time methods typically consider the worst-case scenario,
which leads to over-design in resources and power. Methods which map tasks
to cores at the beginning of the pipelines can adapt to real resource situations.
Nonetheless, they still cannot adapt to run-time workload variations. Recent run-
time remapping techniques and task duplication techniques can measure workload
distribution at run time, and reassign tasks to cores to adapt to workload variations.
Nonetheless, most of current run-time reconfiguration techniques suffer from the
long time overhead of adaptation operations, hence they are incapable for applica-
tions with fast workload variations. Finally, we reviewed methods of low-power
techniques implemented in streaming applications with static task assignments.
They can achieve low power consumption, but they sacrifice the utilization rate of
cores.
Chapter 3

MPSoC Platform and Methodology

3.1 Reconfiguration Methodology

In this thesis, we aim to achieve reconfigurability during run time at both hardware and software levels. At the hardware level of reconfiguration, we allow the MPSoC system to change the hardware connection between cores. We allow the system to reconfigure during every instruction by embedding reconfiguration codes into instructions. We apply the hardware level of reconfiguration to adapt the instructions to differing data sizes. If a 32-bit core processes an instruction with 64-bit data, based on the reconfiguration code in this instruction, the core will be reconfigured to be combined with its adjacent core and form a 64-bit core. If the next instruction is with 32-bit data, the core will be reconfigured to disconnect its adjacent core and execute this instruction with 32-bit data.

At the software level of reconfiguration, we allow task assignments of cores to
be changed at run time. For example, the system can assign extra cores to accelerate video applications, or de-assign cores from video applications to save power. The system may remove cores from non-critical tasks (e.g., videos) and assign these cores to vital tasks (e.g., the auto-driving task or the life sustain task) to improve the reliability of vital tasks. With the task reassignment, the communication of cores should also be reconfigured correspondingly.

3.2 MPSoC Platform Model

To enable the hardware/software level reconfiguration, we build an MPSoC platform with many 32-bit cores to realise our run-time reconfiguration methods. We select homogeneous processors to achieve maximum flexibility. Each core is a 32-bit core that contains its own instruction cache, data cache, and local memory where instructions and data of assigned tasks are stored, as shown in Figure 3.1. A set of I/O ports are utilised for hardware level reconfiguration. I/O ports of a core are wired with I/O ports of its adjacent cores, and the connection can be enabled or disabled by reconfiguration codes. By enabling the connection, I/O ports allow cores to be synchronised and exchange data with its neighbours based on the reconfiguration code contained in the instruction. In addition, FIFOs are also used for communication between adjacent cores. FIFOs allow cores to send data to its neighbours sequentially with low access delay (typically, one clock cycle).

Figure 3.2 shows the overview of the proposed MPSoC platform. Cores are connected with their neighbours by FIFOs and I/O ports. Shared memory is
utilised for software level reconfiguration. A task assignment table is built in shared memory. According to the core’s *id*, each core can find its task assignment at run time, and their task assignments can be changed at run time by rewriting the task assignment in the task assignment table. Shared memory is also used for data communication between cores that are not directly connected by FIFOs or I/O ports. When a core is assigned to a task, the core fetches the program and data from main memory and stores them into its own local memory.

### 3.2.1 MPSoC Platform Discussion

We choose the 32-bit core as it is one of the most common data widths in embedded systems. The reason of choosing I/O ports for hardware level reconfiguration is their asynchronism. It enables signals to be sent to adjacent cores without the need to wait for clock signals, so that the I/O ports can be used to connect at instruction level. This enables the computation and communication to be performed...
in the same clock cycle as long as the data path is shorter than one clock cycle.

Shared memory is used for software level reconfiguration in our run-time reconfiguration methodology. Shared memory can be accessed by all cores; a Mutex mechanism is applied in shared memory to avoid data race when multiple cores attempt to access the same address simultaneously. In this thesis, we utilise shared memory, although some advanced communication methods such as the NoC network can be examined in future.

3.3 Framework of Reconfiguration

3.3.1 Run-time Reconfiguration of Data Width Variation

In this section, we introduce the methodology for hardware reconfiguration with applications containing varying data widths.
3.3.1.1 Necessity of Run-time Data Width Variation

Cores with large data widths (e.g., 128-bit or 256-bit) can process large-sized data faster than cores with standard data widths (e.g., 32-bit or 64-bit), while cores with standard data widths can process 32-bit or 64-bit data with low cost and minimal power consumption. For an MPSoC platform, the number of applications that are executed in parallel and their data sizes may not be predictable. Hence, designer may not be able to decide whether to use a standard core or a wide core. In our system, the data width of cores is variable via run-time reconfiguration, which allows MPSoC systems to adapt to data sizes of applications.

3.3.1.2 Core Combination

To achieve data width variation, we allow cores to be combined to utilise their data widths as one core. In Figure 3.3 (a), if the MPSoC platform executes 32-bit instructions (instructions processing 32-bit data), each core will execute an instruction as a standard core with disabled I/O ports. In Figure 3.3 (b), if the MPSoC platform executes 64-bit instructions, two cores are dynamically combined to form a 64-bit core (adding their data widths together - 32+32 bit) by enabling the I/O ports in-between. Similarly, if the MPSoC platform executes 128-bit instructions, as Figure 3.3 (c), four cores are dynamically combined to form a 128-bit core.
### 3.3.2 Streaming Application and Pipeline

#### 3.3.2.1 Core Assignment for Streaming Applications

Figure 3.4 shows the core assignment methodology when the MPSoC platform executes multiple streaming applications. When a streaming application $A$ is executed on this platform, a set of cores are assigned to this application, and cores in this set execute different tasks of the application in parallel. At run time, if a new application $B$ is going to be executed in this platform, another set of cores are assigned to the new application. Similarly, more applications can be mapped...
onto separate sets of cores, and executed in parallel. Cores that are not utilised currently can be set to idle or turned off, in order to save power and energy. In this way, we allow multiple applications to be executed simultaneously, and explore task-level parallelism within each application.

### 3.3.2.2 Hardware/Software Pipeline

Hardware/software pipelines are deployed in each set of cores to execute streaming applications in order to achieve high throughput. Streaming applications may need to meet given throughput constraints. For example, multimedia applications are required to output a certain number of frames per second, while signal processing applications are required to process a certain amount of data per second.

In hardware/software pipelines, a streaming application is modeled as a series of sequential stages, and we assume that the data is input in sequence. For example, an H.264 encoder sample, shown in Figure 3.5, is composed of six tasks: Color Conversion ($CC$); Motion Estimation ($ME$); Intra-prediction or Motion Compensation ($IPoMC$); Transform, Quantization and Entropy Coding ($TQE$); Inverse Transform and Quantization ($ITQ$); and Writing Back ($WB$). Each stage is a loop of iterations that performs a task of the pipeline. In addition, the $WB$ stage will generate a reference frame in main memory for motion estimation, hence, there is no direct writing back path in the pipeline.

Hardware/software pipelines allow stages in the pipeline to be parallelised, so that the amount of data being processed at a constant time interval increases. Cores are assigned to execute different stages of this streaming application, and
A number of cores are assigned to a streaming application.

Figure 3.4: Core Assignment Methodology

The communication between stages can be through FIFOs or shared memory.

H.264 Encoder Example

Figure 3.5: A Pipeline Example

Figure 3.6 shows the communication through FIFOs. One core is assigned to execute each stage, and FIFOs are used to store data between cores/stages. The advantage of FIFOs is the access speed, as FIFOs typically take only one cycle to send/receive a datum, but a shortcoming of FIFO is that FIFOs are static for one-to-one communication, which is not very flexible in run-time reconfiguration.

Figure 3.7 shows another pipeline methodology of the application. This methodology allows multiple cores to be assigned to one stage, and shared memory is utili-
lized for data communication. Areas of shared memory are utilised as buffers for communication between different stages. The communication by shared memory is not limited by adjacent cores and the communication protocol can be defined by users, hence, pipelines with shared memory can achieve high flexibility. The disadvantage of the shared memory is that the access speed is slower than FIFOs.

Hardware/software pipelines can significantly improve throughput by sharing workload of stages with multiple cores, but the efficiency of pipelines depends on workloads of stages. The throughput of the pipeline is restricted by the stage with the heaviest workload, named as the bottleneck stage. For example, if one stage can output 20 frames per second and the bottleneck stage can output 5 frames per second, the overall throughput of the pipeline will be limited to 5 frames per second, because other stages have to wait the progress of the bottleneck stage. To improve the throughput of the hardware/software pipeline, designers should improve the throughput capability of the bottlenecks stage.
3.3.3 Run-time Reconfiguration for Hardware/software Pipelines

Hardware/software pipelines based on shared memory allow the system to assign multiple cores to the bottleneck stage to share the workload and improve the throughput (details of the mechanism of allowing multiple cores to work in the same stage are discussed in Section 5 and 6). Due to workload variations in stages, the throughput may vary and the initialised task assignments may not match run-time workload conditions. In our software level of reconfiguration methods (in Section 5 and 6), the system dynamically increases/decreases the numbers of cores at different stages by task reassignments. According to different situations, different methodologies are selected.
3.3.3.1 Methodology of Throughput Maximum

(a) shuffle a core from the non-critical stage to the bottleneck stage

(b) Add an idle core to the bottleneck stage

Figure 3.8: ADAPT Methodology

When maximising throughput of a streaming application, the run-time method ADAPT is chosen to maximise the throughput of the pipeline with a given number of cores. The MPSoC system can dynamically detect the bottleneck stage. For other stages, there may be redundant cores that are not fully utilised because the throughput is limited by the bottleneck stage. Redundant cores mean that the system can remove redundant cores from a stage without affecting the throughput. Figure 3.8 shows the reconfiguration methodology of ADAPT. The system estimates whether there are redundant cores in other stages. If there are redundant cores, these redundant cores are shuffled from their original stages (named
non-critical stages) to the bottleneck stage to improve the throughput. If there is no redundant core, the system assigns an idle core to the bottleneck stage. Therefore, \textit{ADAPT} maximises the throughput of the pipeline with a given number of cores by adding idle cores or shuffling under-utilised cores to the bottleneck stage.

### 3.3.3.2 Methodology of Core Minimum

![Figure 3.9: E-pipeline Methodology](image)

When power and resources are constrained, the run-time method \textit{E-pipeline}
is utilised to minimise the total number of cores that are working in the MPSoC system with multiple pipelines. *E-pipeline* measures whether the number of cores in a pipeline is the necessary number of cores for this stage. The necessary number of cores for a stage is the minimum number of cores for this stage to meet the throughput constraint. Figure 3.9 shows the methodology of *E-pipeline*. *E-pipeline* removes redundant cores from pipelines where the throughput constraint is met, and puts them into an idle core pool. Cores in the idle core pool are set to sleep mode to save power, and they can be reused in future reconfiguration when assigned to pipelines where the throughput constraint is not met. In this way, *E-pipeline* optimises the core assignment in the MPSoC system and minimises the total number of cores to meet throughput constraints.

### 3.4 Summary

In this chapter, we discuss the methodology of our run-time reconfiguration methods. Run-time reconfiguration methods are composed of hardware level and software level of reconfiguration. Based on their requirements, a proposed MPSoC system is discussed. The methodologies of hardware level reconfiguration and software reconfiguration are discussed separately. In the following chapters, details of run-time reconfiguration methods are explained.
Data Width Variation for MPSoCs

In this section, we will discuss a run-time method for varying data widths, named Dynamic Reconfigurable Multicore Architecture (DRMA). This method is used in MPSoC systems that are reusable for multiple applications with different data widths. It allows dynamical reconfiguration for multiple purposes, providing instruction level parallelism (ILP) and thread level parallelism (TLP), and enables pipelining of cores. Compared to state-of-the-art reconfigurable MPSoC architectures such as Voltron [103], TFlex [104], CoreFusion [105] and Bahurupi [106], which aggregate cores to perform multi-issue executions (for example: to mix two 2-way cores to one 4-way core), DRMA integrates multiple cores at runtime to support varying data widths (e.g., it integrates 2 cores to create a 64-bit core). DRMA allows reconfiguration at real time. The system necessitated additional hardware, though the overhead is small.

We demonstrate the DRMA method with four 32-bit cores using the PISA in-
Data Width Variation for MPSoCs

struction set. A pair of 32-bit PISA cores can be combined together to create a 64-bit core (not all instructions are 64-bits wide but a fair number are 64-bits). Four cores can be combined together to make a 128-bit core. A detailed comparison to other architectures is also performed. The kernels of few small programs are scheduled and executed on this architecture and the results are compared against single 128-bit and single 32-bit cores. Note that the primary objective of this exercise is the flexibility. We show the advantage of flexibility by varying the data width for differing applications and examples of task level parallelism. There is a cost involved time and area, but these can be amortized for particular applications. In application families, such as encryption, this feature can be exploited for large data width operands and DRMA can provide speed improvements.

4.1 Motivational Example

4.1.1 Varying Bit

With the developments in 3D graphics technology and physical simulation systems, applications with large data size (such as 64, 128 and 256 bits) will become necessity. Additionally, there is a plethora of 32-bit applications which will continue to prevail. Modern block-wise encryption applications for security involve large amounts of instructions that can be naturally operated as fewer instructions with large data width cores [107] [108]. It is these widely varying bit width operations, and the cost of making new masks that motivate this work.
4.1.2 Motivation Example

For simplicity, let us assume that we have three code segments as in Figure 4.1(a), (b) and (c), each with a set of four independent operations. Data width in Figure 4.1(a) and (b) are fixed to 32-bits and 128-bits operations respectively, whereas the set of operations in (c) varies from 128 bit to 32 bit between operations.

Let us schedule the given code segments on a 32-bit x 4 DRMA architecture.
that could be dynamically configured to behave like 4 x 32-bit independent cores ($\mu P_{32}$ in Figure 4.1) or a 128-bit core ($\mu P_{128}$ in Figure 4.1) or a combination thereof in-between.

While the code segment in Figure 4.1(a) can be efficiently scheduled on the 4 x 32-bit independent cores (DRMA configured in separate mode) as shown in Figure 4.1(d), the segment in Figure 4.1(b) can be scheduled on the 1 x 128-bit core (DRMA configured in complete combined mode) as shown in Figure 4.1(e).

The code segment in Figure 4.1(c) has four operations with data widths 128, 32, 64 and 32 bits respectively and they can be efficiently scheduled on cores with the same data width as shown in Figure 4.1(f) where operations 2, 3 and 4 are scheduled together. It is noteworthy that the given core behaves both as a 128-bit core and later as 2 x 32-bit and 1 x 64-bit cores, and it is only possible due to the dynamic reconfigurability in DRMA.

The details of reconfigurability, data sharing and synchronization of DRMA are explained in later sections.

### 4.2 Architecture

The method of DRMA is presented by four 32-bit cores. Both the core and the instruction set architectures of DRMA are presented in this section followed by a description of how the DRMA would function and then how the cores are synchronized when they work on larger data widths.
4.2.1 Core Architecture

Each core in the DRMA architecture, is a pipelined RISC type core. Each core contains separate data and instruction memories and can be treated as a standardized module. Additionally, they are driven by a central clock.

In combination modes, we use processor element (PE) to name the cores used in the DRMA architecture. The features of DRMA claimed earlier (such as operating like a 4 x 32bit cores or a single 128bit core as in Figure 4.1) are achieved by programming cores (named PE in Figure 4.1) into one of their four different modes: (1) Separate Mode (SMode); (2) Initial Mode (IMode); (3) Terminal Mode (TMode); and (4) Combined Mode (CMode).

In the PISA instruction set, the instructions are 64-bit. The first two bits are used to indicate the modes which are used. Bits "00" for SMode, bits "01" for IMode, bits "10" for TMode and bits "11" for CMode are used. PEs work in SMode so that task level parallelism can be achieved. IMode enables ILP. Together IMode, CMode and TMode enables word-size integration.

Figure 4.2 depicts the peripherals that could support the four modes of operations. The peripherals (and therefore the corresponding ports) shown are enabled and disabled by the modes set through instructions. The I/O ports named DataIO and the CtrIO ports are implemented to communicate computation-relevant data and the control signal separately in word-size varying mode. In both word-size varying mode and ILP mode, the load/store instruction will be synchronised for multi-clock action. The I/O block named SyncIO is deployed to handle synchronisation issues. The synchronisation protocol will be discussed in Section 4.2.4.
4.2.2 Instruction Set Architecture

Since the dynamic reconfigurability of the DRMA architecture is achieved by instructions, the ISA of each PE plays a vital role in the realization of word-size variation. The functioning of the PEs could be categorized into four, based on the type of instructions they execute. In this section, four most common instruction types are analysed as typical examples: [TYPE\(_1\)] instructions that involve either load/store or bitwise operations (such as \(or_{128} A, B\)); [TYPE\(_2\)] instructions that involve carry in/out (such as \(add_{128} A, B\)); [TYPE\(_3\)] instructions that involve multiple bit transfers between PEs (such as \(srl_{128} A, B\)); and [TYPE\(_4\)] instructions that need communication among PEs for decision making and sharing (such as branch/jump \(beq_{128} A, B\)).
4.2.3 Functioning of DRMA

Let us assume that the MPSoC has four PEs and they are connected together to form a larger data width (four 32bit cores are combined together to form a single 128bit core). Then, each small PE will fetch identical instructions from their respective memories, fetch data from the same address from their respective data memories or register files separately. Depending upon the instruction type some data will be exchanged (for example: shift, carry taken, etc.) during the execution and the final result in each core will be modified as they process parts of one piece of large-word-size data. From an external standpoint, they work together as a larger word core.

![Diagram of DRMA for TYPE1 Instructions](image)

Figure 4.3: DRMA for TYPE1 Instructions (example: or_{128} A, B)

Figure 4.3 depicts how a TYPE1 instruction (such as or_{128} A, B) would behave. The data that is to be loaded to registers A and B (that are of larger data width) will be divided into four smaller word sizes each: A into (A_1, A_2, A_3 and A_4) and B into (B_1, B_2, B_3 and B_4). Given that there is NO need for data communication in TYPE1 instruction to perform the computation, the DataIO and CtrlIO are disabled. SyncIO will be enabled for multiple cycle instructions (such as load).
only and will not be utilized for single cycle instructions (such as or). In addition, as shown in Figure 4.3 while the far left PE works in TMode and the far right PE works in IMode to define the boundary of this combination group, the other two PEs will work in CMode.

![Figure 4.4: DRMA for TYPE2 Instructions (example: add_{128} A, B)](image)

Figure 4.4 depicts how a TYPE2 instruction (such as add_{128} A, B) would behave. The only difference in TYPE2 instructions compared to TYPE1 is the carry bit each smaller data width instruction produces. Since, such carries have to be integrated in the computations (they have to be sent by the right PE and received by the left PE), a single bit wire from the DataIO port is enabled as shown in Figure 4.4. The rest of the functioning of TYPE2 is similar to that of TYPE1 instructions. The transfer of carry signal increases only a small amount of clock width.

Figure 4.5 depicts how a TYPE3 instruction (such as srl_{128} A, B) would behave. Here, since the instruction is a logical right shift, the shifted data has to be sent across PEs and this is achieved by enabling the DataIO port in the right direction. The right most PE here works in IMode and the left most works in TMode. The rest of the functioning is similar to the previously explained modes.
Data Width Variation for MPSoCs

Figure 4.5: DRMA for TYPE3 Instructions (example: \texttt{sr}_{128} A, B)

Figure 4.6 depicts how a TYPE4 instruction (such as \texttt{beq}_{128} A, B) would function. Similar to TYPE3 instruction, the final decision will be made based on the computation of all the PEs. Partial conditional results from the right PEs will be sent to the left PEs and the leftmost PE (the PE in \texttt{TMode}, PE4 in Figure 4.6) will make the decision for the whole instruction. Now the decision is sent back from PE4 to the rest of the PEs through \texttt{CtrlIO} port. The rest of the functioning of TYPE4 instructions are similar to that of the previous types.

Figure 4.6: DRMA for TYPE4 Instructions (example: \texttt{beq}_{128} A, B)
4.2.4 PE Synchronisation

For multiple cycle instructions, such as Load/Store, the system has to be synchronized. This is particularly important in instructions like memory instructions because it may take varying cycles that the designer may not be aware of. For example, when the instruction involves memory loading, the core will stall until it receives the signal ‘ACK’ (acknowledgement) from the data memory. A PE combined together with others has to wait for its own ‘ACK’ signal and other ‘ACK’ signals attached to the other PEs, because the core system may lose synchronization if any one finishes its loading operation and fetches the next operation while others still wait for the right data from their own memory. In DRMA, the synchronization block will store the ‘ACK’ signal once it is received, and then transfer the information from the right PE to the left. The far left PE of this group will send a control signal to finish the stall of all PEs in one group, only when each PE has received its own ‘ACK’ signal.

4.3 Design Methodology and Implementation

The design flow of the DRMA architecture is shown in Figure 4.7. Firstly, a base ISA is selected for our PEs (note that our PEs are fully fledged cores) and an ASIP design tool is used to build the cores that are capable of behaving as processing elements for DRMA. We used ASIPmeister [109], an ASIP design tool for building each PE.

As shown in the design flow, the instructions that support DRMA will be
copied four times, one for each mode, and the instructions’ micro-operations will be altered. The micro-operations will not only provide the operations of the instruction (as they typically do) but also indicate the connections among cores.
For example, the core of the micro-operations for ‘addCMode’ instruction (add instruction in *CMode*) is shown in Figure 4.8(a) and the corresponding logic is shown in Figure 4.8(b). For the *CMode* ‘add’ instruction, the *carry in* will be received from the right PE and the *carry out* should be passed on to the left PE as shown in Figure 4.8(b).

```vhdl
// Instruction: addCMode
// Stage: EXE
wire cln;
wire cOut;
wire result[31:0];
cln = carry_PE2.read();
<result,cOut> = ADDER32.adc(src1,src2,cln);
nul = carry_PE4.write(cOut);

// Forward result
```

![Figure 4.8: The ‘add’ instruction in the *CMode*](image)

The output from the ASIPmeister tool is the VHDL files for synthesis and simulation. Then the integration of PEs to build DRMA is done manually which is both synthesized and simulated using commercial chip design tools as shown in Figure 4.7.

We implement DRMA with PISA instruction set [110] (only integer instructions) according to the design flow explained, and we had four PEs in our system. PISA is a 32-bit architecture and therefore the largest data width we could achieve with DRMA is 128 bits. Each PE in the architecture had six pipeline stages and their own data and instruction memory, as explained earlier in Section 4.2.
The DRMA architecture is clocked at the fastest possible frequency to allow the longest operation to be finished within this period. To enable the change in clock periods, we needed a multiplexor which can change differing clock signals.

<table>
<thead>
<tr>
<th>Register Type</th>
<th>(lb, lbu, lh, lhu, lw, dlw, sb, sh, sw, dsw)</th>
<th>(dsz, jr, jalr, add, addu, sub, subu, mfhi, mthi, mflo)</th>
<th>(mtlo, and, or, xor, nor, slv, srlv, srav, slt, sltu)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate Type</td>
<td>(lb, lbu, lh, lhu, lw, dlw, lw1, lwr, sb, sh)</td>
<td>(sw, dsw, dsz, swl, swr, beq, bne, bleg, bgtz, bltz)</td>
<td>(bgez, j, jal, sll, srl, sra)</td>
</tr>
</tbody>
</table>

Table 4.1: Instructions Supported by DRMA

<table>
<thead>
<tr>
<th></th>
<th>DRMA</th>
<th>(\mu P) 128bit</th>
<th>(\mu P) 32bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combinational area (# of cells)</td>
<td>157863</td>
<td>123818</td>
<td>36315</td>
</tr>
<tr>
<td>Noncombinational area (# of cells)</td>
<td>136000</td>
<td>102531</td>
<td>33627</td>
</tr>
<tr>
<td>Total (# of cells)</td>
<td>293863</td>
<td>226349</td>
<td>69942</td>
</tr>
<tr>
<td>Ratio</td>
<td>4.2(\times)</td>
<td>3.2(\times)</td>
<td>1(\times)</td>
</tr>
</tbody>
</table>

Global Operation Voltage = 1 v

<table>
<thead>
<tr>
<th></th>
<th>DRMA</th>
<th>(\mu P) 128bit</th>
<th>(\mu P) 32bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average dynamic power(mW)</td>
<td>113.4</td>
<td>86.7</td>
<td>28.6</td>
</tr>
<tr>
<td>Average leakage power(mW)</td>
<td>1.3</td>
<td>1.0</td>
<td>0.3</td>
</tr>
<tr>
<td>Average Total Power(mW)</td>
<td>114.7</td>
<td>87.7</td>
<td>28.9</td>
</tr>
<tr>
<td>Ratio</td>
<td>3.97(\times)</td>
<td>3.04(\times)</td>
<td>1(\times)</td>
</tr>
</tbody>
</table>

Requested Arrival Time = 1 ns

<table>
<thead>
<tr>
<th></th>
<th>DRMA</th>
<th>(\mu P) 128bit</th>
<th>(\mu P) 32bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock Width (ns)</td>
<td>1.69</td>
<td>1.47</td>
<td>1.29</td>
</tr>
<tr>
<td>Ratio</td>
<td>1.31(\times)</td>
<td>1.14(\times)</td>
<td>1(\times)</td>
</tr>
</tbody>
</table>

Table 4.2: Hardware Parameters from Synthesis

Table 4.1 lists the instructions that are available for DRMA type of operations (the instructions that work in all four modes). Not all instructions used by the PEs are available in the DRMA architecture. We did not support instructions such as multiply and divide (‘mult’, ‘multu’, ‘div’, and ‘divu’) due to their complex nature.
when they are needed to be combined. In addition, we did not handle instructions with immediate operands (‘addi’, ‘addiu’, ‘andi’, ‘ori’, ‘xori’, ‘slti’, ‘sltiu’, and ‘lui’). We left both these categories as our future work. At present long word instructions such as 128 bit multiply is performed on one single PE, and a software function is instantiated.

4.4 Experiments and Results

4.4.1 Synthesis

We evaluated the hardware parameters of DRMA (with reconfigurable four cores) and compared it against non-configurable cores (μP-32bit and μP-128bit). We synthesized the RTL and analyzed it using Design Compiler from Synopsys. μP-32bit is the original PISA core (32-bit), and μP-128bit is a core which supports 128-bit data width with PISA instructions, with ALU and memory interfaces modified to support 128-bit data widths. The multiplier in the 128-bit implementation is very large with a large delay (not pipelined). In order to make a fair evaluation, we removed the multiplier to calculate the critical path. The results are summarized in Table 4.2. Column 1 lists the properties analyzed and column 2 presents the numbers for DRMA. Column 3 and column 4 present details for 128-bit and 32-bit single cores, respectively. As shown in the table, the cost of DRMA platform with four PEs is roughly four times the cost of the original 32-bit core in area and power consumption, which indicates that the additional cost to create DRMA is low (no more than 5%).
4.4.2 Case Studies

We took three different case studies to show the efficacy of DRMA: i) Three applications (AES, CRC32 and IPv6) are chosen to test the word variation capability of DRMA and is compared against $\mu$P-128bit and $\mu$P-32bit; ii) All three applications are executed on the DRMA architecture in differing modes to see the efficacy of DRMA; and iii) A pipelined JPEG application is demonstrated to show a pipeline architecture being supported by DRMA.

4.4.2.1 Case 1

We examined three applications in Case 1. The characteristics of the applications are: i) A highly parallel encryption ‘kernel’ of AES (AES 128 bit encryption after round keys generation); ii) CRC32 (32 bit Cyclic Redundancy Check) application; iii) An IPv6 packet filter program that matches 128 bit IPv6 addresses against a series of rules. The applications are experimented on the three different architectures. We simulated the RTL models of the three architectures (as explained in Section 4.3) using ModelSim.

Figure 4.9 shows the number of clock cycles incurred and the corresponding execution time for the three architectures we considered. For AES, the DRMA architecture working in data width variation mode (set PEs to $SMode$, $CMode$, and $TMode$, respectively) outperforms both of its counterparts by $3.99 \times$ and $2.46 \times$
Data Width Variation for MPSoCs

Figure 4.9: The Comparison in Performance

in clock cycles. The increased speed is because DRMA can process 128-bit data directly while, \( \mu P\text{-}32\text{bit} \) spends more than four times the number of instructions to process all 32-bit parts of 128-bit data and then encrypt them. DRMA can also process 32-bit data in parallel while \( \mu P\text{-}128\text{bit} \) can only process one instruction regardless of the data width. In CRC32, since it is a pure 32-bit application, the architectures with large bit width (such as \( \mu P\text{-}128\text{bit} \)) will not gain any improvement in performance. DRMA is distinct in its small area and energy consumption as \( \mu P\text{-}32\text{bit} \), because we can use one core in \textit{SMode} and then turn off redundant PEs or dedicate them to other tasks. For IPv6 packet filter, which is a 128-bit application, the DRMA architecture is as good as \( \mu P\text{-}128\text{bit} \) and outperforms \( \mu P\text{-}32\text{bit} \).
4.4.2.2 Case 2

We use the above three applications to evaluate the total efficiency of DRMA in executing a series of applications. It is not uncommon to see such three applications executing together in a standard system, where data is transported, encrypted and checked for errors. In this section, we mix the three applications into a single application: i) IPv6 filter, if the rule is valid then fetch 128 bit data and store; ii) AES encryption: The data is encrypted as 128-bit blocks (we modified the AES code to take advantage of 128 bit encryption); iii) CRC32: Create checksum for the 128-bit data; and iv) Store and record: The system will store the encoded data and CRC32 value for back-up or for transmission, as well as, record the order information. We can execute these in three different ways. Firstly, we ignore the dependency between applications and allow each core to work in parallel (set them to SMode) (see Figure 4.10(a). AES is occupying roughly three-fourth of the total instructions, hence, becoming the bottleneck in common MPSoC. This architecture achieves a speed up of 1.52× compared to a single PE which executes the same application. In the second sub-case, DRMA is configured into a fixed pipeline architecture, named by implementing FIFO to connect adjacent PEs as shown in Figure 4.10(b), where the output of one core is fed to the next one. A 1.20× speed up is achieved, once again constrained by the large delay of AES (other cores were idling). In the third case as shown in Figure 4.10(c), DRMA can be reconfigured dynamically through its instructions into different configurations:
working in data width variation mode for a 128-bit application in IPv6 filter and AES encryption; working in a pipelined fashion in CRC and working separately in data store, the speed up compared to $\mu$P-32bit is $4.7 \times$, mainly because DRMA is controlled to dynamically optimize its architecture for different applications. Figure 4.11 depicts the clock cycles taken for the three configurations compared to $\mu$P-32bit as baseline.

Figure 4.10: The Comparison of Different Configurations
Data Width Variation for MPSoCs

4.4.2.3 Case 3

Pipeline architectures are suited for data streaming applications in MPSoCs. We implemented the JPEG encoding application in the DRMA architecture like Figure 4.12 with each core executing one stage. Compared to \( \mu P-32\text{bit} \), the speed up is \( 4.11 \times \). The result is shown in Table 4.3.

![Performance of Different Modes](image)

Figure 4.11: The Results for Case 2

![The Configuration in Case 3](image)

Figure 4.12: The Configuration in Case 3
4.4.3 Discussion

The flexibility of DRMA is shown through three case studies: Case 1 shows benefits of word width variation and ILP; Case 2 shows the ability of the DRMA to be configured in differing ways; and Case 3 shows an application where a configuration of pipeline is suited.

The flexibility afforded by DRMA results in a certain level of efficiency. While the examples were small (due to the lack of a suitable compiler), the careful selection of configuration for a particular application(s) can be quite efficient. It is currently not possible to simulate DRMA in a modified system level simulator such as the multi-core version of SimpleScalar simulator \[110\], therefore all simulations were carried out at the RTL level. Hence, DRMA cannot be directly compared to other architectures. Since DRMA can be configured to be a core from a 32-bit to a 128-bit core and can exploit both ILP and TLP, to build a compiler for this architecture, it would involve multiple compiler technologies for each configurations and optimization. We have began to attempt it and do not underestimate the challenge.

The clock cycle should be used a comparative guide rather than as a rule. There are many optimizations possible and have not been implemented. ASIP-meister only offers a single cycle multiplier and divider. It means that large bit
width multiply instruction or divide instruction would increase the clock period. Currently the applications used do not use 128-bit multiplication, thus do not affect our test. Since multiply instruction is uncommon, we can theoretically use a 32-bit one and run it multiple times to get a 128-bit result.

4.5 Summary

In this chapter, we presented a dynamically reconfigurable method, named DRMA, for MPSoCs targeting flexibility and reconfiguration. The main features of DRMA are: i) The ability to vary data width; ii) Configurations are controlled by the executing instruction (not a separate instruction), thus, eliminating the configuration time; iii) A simple and modular architecture with only 5% overhead in area. Due to the reconfigurability, which is based on instructions, DRMA is efficient and flexible. We used three case studies with different data widths to demonstrate the efficacy of DRMA.
In this chapter, we discussed a run-time adaptation methodology for efficient use of MPSoC systems with many cores in executing hardware/software pipelines. Such MPSoC systems (named on-chip many-core systems in the following sections) are becoming increasingly prevalent in the embedded system industry. Much like a single FPGA design which pervades the industry, the next generation of on-chip many-core systems can be programmed in-situ and used in a wide variety of applications. The same many-core system might be used in a car as well as in a set top box, where some cores execute one application, while another set of cores execute another application.

The paradigm of hardware/software pipelines [111] allows an application to be arranged as a pipeline of tasks, which typically results in high throughput.
Examples of pipelined applications include audio, video, imaging and networking applications [112]. In hardware/software pipelines, throughput (e.g., frames per second in video processing applications) is constrained by the bottleneck stage (i.e., the one with the heaviest workload) of the pipeline [95]. Thus, balancing pipeline stages is vital for both high throughput and efficient resource utilization. When resources (such as free cores) and/or application workload vary, quick on-the-fly balancing of pipeline stages is required.

5.1 Motivational Example and Research Challenges

Figure 5.1 shows a typical H.264 encoder pipeline with six stages at the top. Figure 5.1 also shows the workload distribution among different stages and their variations across different iterations (each iteration represents processing of a macroblock from the input video frame). It is noted that the workload varies at run-time for the motion estimation and transformation stages (graph on left in Figure 5.1). More importantly, at some instants, the motion estimation stage is the bottleneck while at other instants, the transformation stage becomes the bottleneck (two graphs on right in Figure 5.1). Moreover, the variation in workload from one iteration to another is significant that requires fast and fine-grained adaptation of pipelines.

In summary, workload variations are quick and significant, and occur simultaneously in multiple pipeline stages. To efficiently adapt a pipeline under such a dynamic workload in a many-core system with a given number of allocated cores,
three research problems need to be addressed:

- Quick detection of bottleneck stages during execution;
- Rapid addition of cores to relieve the bottleneck stage after its detection; and,
- Rapid shuffling of cores among pipeline stages when there are no idle cores and/or number of allocated cores vary.

A run-time methodology named ADAPT which solves the above research problems is discussed in the following sections.

## 5.2 Pipelines and Parallelization

We target applications that are represented as a pipeline of tasks. Figure 5.2(a) shows an example of a hardware/software pipeline with 3 stages. Each circle represents a task, which processes input data blocks from previous stage’s task(s)
and produces output data blocks for the next stage’s task(s). The processing of one data block by a stage is referred to as an iteration, which is marked inside a task. For example, while stage 3 is in its $i+1$ iteration, stage 2 is in its $i+2$ iteration, effectively pipelining the execution of tasks for high throughput. Further, end of one iteration of the last stage means that the whole pipeline has completed one iteration. We assume one task per stage where each task is complex enough to be solely executed by a core, in order for us to perform rapid run-time adaptation. Otherwise, application-level balancing techniques [92] could be used to merge/split tasks beforehand.

![Parallelization Model of Pipeline](image)

**Figure 5.2**: Parallelization Model of Pipeline.

In this work, we use *task clone* for run-time parallelization of a stage to improve its throughput under increased workload. For example, stage 2 is the bottleneck (its latency is highest amongst all the stages, which is marked around the task) in Figure 5.2(a), then its task is cloned to execute two iterations ($i+2$ and $i+3$) in parallel. In other words, stage 2 will process data blocks $i+2$ and $i+3$ in two distinct invocations of its task, which we refer to as *in-parallel iterations*. This would theoretically (assuming no overheads for *task clone*) reduce the average latency of stage 2 by half, improving the throughput by two times. Figure 5.2(b) shows a
later scenario where the workload of the tasks has changed. Now, stage 3 is the bottleneck, and there is no need to execute two iterations of stage 2. Hence, in-parallel iterations of stage 2 are reduced, while in-parallel iterations of stage 3 are increased as part of run-time adaptation.

Note that the tasks are computation bound rather communication/memory bound. This is typical of pipelines because each task processes only one data block in an iteration, and the size of data block is relatively small compared to the entire input (for example, one macroblock of $16 \times 16$ in an HD frame of $1920 \times 1080$ for a video processing application). If the tasks are communication/memory bound, then adding more in-parallel iterations may not improve the throughput. In this case, one needs to use a different parallelization model (for example, task splitting), and memory architecture and optimizations (for example, distributed memories and data layout) which is out of the scope of this work.

A stage ‘expands’ when iterations are added to it, while it ‘shrinks’ when iterations are removed from it. Note that, in case of multiple iterations, each iteration of a stage is executed on a separate core. Further, there is a single software-managed buffer between the stages which can accommodate the output of one iteration [96]. Hence, when multiple iterations of a stage are executing, they complete out-of-order but commit their outputs in-order to the buffer for correct working of the pipeline (details are in Section 5.5.2).
Software Pipeline and Throughput Maximisation

5.3 System Model

Figure 5.3 shows a homogeneous many-core architecture which is the target of ADAPT. Every core (denoted by a rectangle with $C$) consists of a processor with private instruction and data caches, and a local memory (for its instructions and local data). The architecture also has a shared memory which is used for software-managed buffers for communication. A set of cores is assigned to execute a pipeline. In such a set, one core is chosen to manage the pipeline under dynamic workload, named pipeline manager. Other cores are named pipeline workers, and are either active or idle. In addition, a portion of shared memory is allocated for the application which contains buffers for communication between its tasks. The pipeline manager tracks workload/resource variations, and expands (by assigning more workers, and thus increasing in-parallel iterations) or shrinks (by reducing assigned workers, and thus decreasing in-parallel iterations) pipeline stages at runtime.

Figure 5.3 shows two run-time scenarios of the system. In Figure 5.3(a), dif-
ferent number of workers (cores) are allocated to applications App1 and App2. These applications execute independently and are managed by their own managers. There are unallocated cores which are not assigned to any application yet. In Figure 5.3(b), a new incoming application App3 is mapped to that set of unallocated cores. Meanwhile, some idle workers of App1 and App2 have been activated (through their assignment to stages) by their respective pipeline managers. Note that the allocation of a set of workers for an application is assumed in this work by a system manager (external to pipeline manager), and the focus of ADAPT is to efficiently manage the assignment of allocated workers to stages. However, ADAPT is capable of run-time adaptation if the number of allocated workers is changed during execution of an application. Unless specified otherwise, the term manager refers to pipeline manager in the rest of the chapter.

5.4 ADAPT Methodology

A pipeline application with $S$ stages, represented as $\{1, 2, \ldots, S\}$, is given with one software-managed buffer in between the stages. Thus, a total of $B = S - 1$ buffers, represented as $\{1, 2, \ldots, B\}$, are used. A number of pipeline workers, $N = N^a + N^i$, are allocated for the application which are divided into active ($N^a$) and idle ($N^i$) workers. The number of workers assigned to a stage $s$ is denoted by $N_s$ (initially one worker per stage), and thus $N^a = \sum_s N_s$. Further, the maximum number of workers assigned to a stage is denoted as $N_{max} = \max_{s} \{N_s\}$. The aim of ADAPT is to manage the assignment of allocated workers to stages un-
der dynamic workload in order to maximize the throughput of the pipeline. The following subsections explain the concept of ADAPT while its implementation details (assignment of workers, measurement of latencies and buffer delays, etc.) are presented in Section 5.5.

5.4.1 Analysis of Pipelines

We define the following terms to analyze a pipeline:

- **Task latency** is the computation and communication time of one iteration of a task, excluding communication delays. Function $TL(s)$ returns the latency of stage $s$ task.

- **Net latency** is the average computation and communication time per iteration of a stage, excluding the delays incurred in reading/writing from/to buffers. More workers in a stage means lower net latency, although each worker’s individual latency is $TL(s)$. The bottleneck stage has the highest net latency. Function $NL(s)$ returns the net latency of stage $s$.

- **Iteration latency** is the time between two consecutive iterations of a stage. The shorter the iteration latency, the higher the throughput. Function $IL(s)$ returns the iteration latency of stage $s$.

- **Buffer delay** is the waiting time that is incurred during a buffer’s access by its producer and consumer stages. A buffer could either incur a read delay when its empty or a write delay when its full. Function $RD(b)$ returns the read delay of buffer $b$ or zero if there is a write delay. Likewise, function
$WD(b)$ returns the write delay of buffer $b$ or zero if there is a read delay.

Figure 5.4: Analysis of Pipeline in Figure 5.2(a).

Figure 5.2(a) shows a pipeline with task latencies annotated around circles. Assume that two workers ($N_s = 2$, for execution of in-parallel iterations) are assigned to stage 2. Figure 5.4 illustrates the execution of the pipeline (assuming zero overhead for parallelization and synchronization during communication, for the sake of simplicity in this example) where the stages are stacked vertically. For each stage, the rectangles represent its iterations or read/write buffer delays. First, the pipeline should be analyzed at the granularity of $N_{max}$ iterations (in this example, 2 iterations), to allow at least one commit from each worker and avoid any transient effects. Second,

$$NL(s) = \frac{TL(s)}{N_s} \tag{5.1}$$

For instance, $NL(2) = 4000/2 = 2000$, and thus stage 2 is the bottleneck stage. Third,
\[ IL(s) = \frac{TL(s)}{N_s} + \frac{RD(s) - 1 + WD(s)}{N_s \cdot N_{\text{max}}} + \frac{OH}{N_s} \]  

where \( OH \) denotes overhead of parallelization and synchronization during communication. The reason for division of buffer delays by \( N_{\text{max}} \) is that \( RD(b) \) and \( WD(b) \) return accumulated delay of buffer \( b \) over \( N_{\text{max}} \) iterations. For instance, during the interval marked in Figure 5.4, \( IL(1) = 1000/1 + [2000+0]/1 \cdot 2 = 2000 \) (overhead is assumed zero). Likewise, during the same interval, \( IL(2) = 4000/2 + [0+0]/2 \cdot 2 = 2000 \). Fourth, in steady state, the iteration latencies of all the stages synchronize with the iteration latency of the bottleneck stage (which is equal to its net latency when overhead is zero).

Based upon the above observations, the following two observations are important for rapid run-time adaptation, and hold for a steady state pipeline:

1. All buffers before the bottleneck stage incur write (or zero) delays while all buffers after the bottleneck stage incur read (or zero) delays. In case of a balanced pipeline, all buffers incur zero delays.

2. The amount of delay determines the criticality of a stage. The most noncritical (fastest) stage has the highest delay. For instance, during the marked interval in Figure 5.4, stage 1 is the fastest (delay = 2000), followed by stage 3 (delay = 1000).

For the sake of explanation at conceptual level, the following subsections assume the availability of iteration latencies and buffer delays. Their actual measurement
Software Pipeline and Throughput Maximisation

at run-time is detailed in Section 5.5.

5.4.2 Pipeline Manager

The pipeline manager is responsible for assignment of allocated workers to stages under dynamic workload. Algorithm 1 shows the main steps of the manager (lines 3 – 8). The manager finds the bottleneck stage (line 3) and a number of noncritical stages (stages that are faster than the bottleneck stage; line 4). Then, if possible, the manager expands and shrinks the stages to improve throughput (line 5). Note that when a pipeline is balanced, there will be no bottleneck stage as well as no noncritical stages, and thus the assignment of workers will not change.

**Algorithm 1**: Pipeline Manager

```
1 f = S;  // initialize loop’s execution frequency
2 for every f iterations of stage S do
3    bottleneck = GetBottleneckStage();
4    noncritical = GetNonCriticalStages();
5    if ExpandShrinkStages(bottleneck, noncritical) then
6      f = max(3, N_max);
7    else
8      f = N_max;
9  end
```

At the end (lines 6 – 8), the manager adjusts the frequency of its execution loop, that is, how often run-time adaptation should be performed. This is important as the manager should ignore the transient effects after changing the assignment of workers, and should reasonably wait for the steady state before the next run-time adaptation. Initially, the frequency is set to the number of stages $S$ (line
1) so that the pipeline enters steady state before the manager performs the first run-time adaptation. Later, when a worker is added to a stage, then 3 iterations of the whole pipeline (or the last stage) may be required to enter steady state in the worst case (first for availability of its input data from producer stage, second for its own iteration, and third for consumption of its output data by consumer stage). Secondly, recall from Section 5.4.1 that a pipeline should be analyzed at the granularity of $N_{max}$ iterations to avoid transient effects. Therefore, the execution frequency of manager loop should be the maximum of 3 and $N_{max}$ when the assignment of workers changes. Otherwise, the frequency should be set to $N_{max}$. For instance, if $N_{max} = 5$ for stage 2 in Figure 5.2, then waiting for 5 iterations of the last stage by the manager means that all the 5 workers in stage 2 would have committed, and every other worker in the pipeline would also have committed at least once. Thus, at least one execution of all the workers is taken into consideration by the manager for the next run-time adaptation.

The detection of bottleneck stage is detailed in Algorithm 2, which heavily depends on the first observation of Section 5.4.1. A read delay on a buffer means that its producer is slower than its consumer, and thus the producer stage of the first buffer with read delay is the bottleneck stage (lines 2 – 6). A write delay on a buffer means its consumer is slower than its producer. Thus, if there are no buffers with read delays and there is at least one buffer with write delay, then the last stage is the bottleneck stage (lines 7 – 8). Finally, when there are zero delays for all the buffers, then there is no bottleneck stage as the pipeline is balanced (line 10). It is possible to have multiple bottleneck stages in the pipeline, in which
Algorithm 2: GetBottleneckStage

1. `writedelay = false;`
2. `for every buffer b from 1 to B do`
3. `if WD(b) > 0 then`
4. `writedelay = true;`
5. `if RD(b) > 0 then`
6. `return b; // producer stage`
7. `end`
8. `if writedelay == true then`
9. `return S; // last stage`
10. `else`
11. `return 0; // balanced pipeline, no bottleneck stage`
12. `end`

Algorithm 3: GetNonCriticalStages

1. `noncritical = {}; // empty list`
2. `sortedbuffers = sort buffers using delay in descending order;`
3. `for every buffer b in sortedbuffers do`
4. `if WD(b) > 0 then`
5. `noncritical ← b if N_b > 1; // producer stage`
6. `if RD(b) > 0 then`
7. `noncritical ← b+1 if N_{b+1} > 1; // consumer stage`
8. `end`
9. `return noncritical;`

case this algorithm returns at least one of them (stage before the first buffer with read delay, or the last stage when there are buffers with only write delays). Other bottleneck stages will be returned in subsequent calls given that those stages are still bottleneck stages and the currently returned stage is no longer a bottleneck stage.
Algorithm 3 shows the detection of noncritical stages. The aim of this step is to find stages that are faster than the bottleneck stage so that they could be shrunk, if required. Here, ADAPT heavily uses the second observation of Section 5.4.1, that is, the stage connected to the buffer with the highest delay will be the fastest stage.

At first (line 2), the algorithm sorts the buffers using their delays in descending order, and then traverses over them starting from the buffer with the highest delay. During the traversal, the producer stage of every buffer with write delay or the consumer stage of every buffer with read delay is added to the list of noncritical stages, if those stages have more than one worker (lines 4 – 7; $N_b$ and $N_{b+1}$ are the number of workers in producer and consumer stages of buffer $b$ respectively). This is to ensure that the noncritical stages could be shrunk later. Note that the list of noncritical stages will inherently be ordered such that the first stage is the fastest one.

The final step of the manager is to expand and shrink the stages based upon the detected bottleneck stage and available noncritical stages. Algorithm 4 checks the availability of idle workers (changes due to variations in workload and/or resources), and assigns one to the bottleneck stage (line 1). In case there are no idle workers, then a shuffle of worker from one of the noncritical stages to bottleneck stage is required. It is important to avoid throughput degradation and oscillation when a shuffle is performed. For instance, when net latencies of bottleneck stage and a noncritical stage are close, then a shuffle may result in the noncritical stage being the new bottleneck stage and the bottleneck stage being the new noncriti-
Algorithm 4: ExpandShrinkStages

1 if bottleneck > 0 and \( N_i > 0 \) then
2 Add a worker to bottleneck stage;
3 return true;
4 else
5 for every stage s in noncritical do
6 \[ \text{est} = \left( IL(s) - \frac{RD(s-1) + WD(s)}{N_s \cdot N_{\text{max}}} \right) \cdot \frac{N_s}{N_s - 1}; \]
7 if est < IL(bottleneck) then
8 Shuffle a worker from stage s to bottleneck;
9 return true;
10 end
11 return false;

The precise measurement/estimation of the effects of a shuffle is complex due to the dynamism of the pipeline. Thus, we use a quick estimation at line 6. The first factor estimates the noncritical stage’s new net latency including overhead using Equation 5.2. The second factor scales the estimate assuming one less worker in the noncritical stage. If the estimated latency is higher than the iteration latency of bottleneck stage, then the throughput may degrade in the worst case. Thus, a shuffle is only performed when the estimated latency is less than the bottleneck’s iteration latency. For example, in Figure 5.2(a), if TL(3) increases to 3000 (due to workload variation), then stage 3 will be bottleneck with IL(3) = 3000. Further, WD(1) = 4000, WD(2) = 4000, and \( N_{\text{max}} = N_2 = 2 \). A shuffle from stage 2 (non-
critical) to 3 (bottleneck) will not be done because \( est = 4000 \) which is higher than \( IL(3) \). Intuitively, a shuffle from stage 2 to 3 means that stage 2 will become the new bottleneck with \( IL(2) = 4000 \), which is worse than the current bottleneck of \( IL(3) = 3000 \). Starting from the fastest (most noncritical) stage, the algorithm repeats the above process until a beneficial shuffle is found (lines 5 – 9). Otherwise, the assignment of workers remains unchanged.

### 5.5 ADAPT Implementation

This section details the implementation of ADAPT’s concepts assuming \( N \) workers (cores) with enough portion of shared memory are allocated to a pipeline with \( S \) stages and \( B \) buffers. Further, a global timer is assumed to be present in the system.

#### 5.5.1 Memory Layout

![Figure 5.5: An Example of Allocated Shared Memory Layout](image)

Figure 5.5: An Example of Allocated Shared Memory Layout
Figure 5.5 shows typical layout of the allocated shared memory. The *workers assignment table* is an array where each entry is the stage to which a worker is currently assigned. For instance, worker 2 is assigned to stage 4. A value of zero means the worker is idle, while a value of -1 means that the core is not allocated to this application. Therefore, during application execution, *system manager* can change the number of allocated cores while *pipeline manager* can change the assignment of workers to stages by updating the *workers assignment table*. Note that locks are used to avoid race conditions during table updates. The rest of the allocated shared memory contains $B$ software-managed buffers.

### 5.5.2 Buffer Management and Instrumentation

Each software-managed buffer contains fixed-size memory (known apriori, and passed to the pipeline manager) for the data block that will be written to it. For the sake of simplicity, read and write pointers, empty and full signals, and locks to avoid race conditions during buffer access by multiple readers/writers are not shown. A counter, named *delay*, is used to track the buffer’s delay. If a worker reads from an empty buffer, then the delay is decremented by $D$, which is the latency of accessing the buffer (equal to shared memory access latency). Likewise, if a worker writes to a full buffer, then the delay is incremented by $D$. Hence, in steady state, a positive value indicates write delay while a negative value indicates read delay. The functions $RD(b)$ and $WD(b)$ from Section 5.4.1 use this delay.

The third portion of a buffer $b$ stores the number and timestamp of last $N_{max} + 1$ iterations that have been committed by its producer stage’s workers. This serves
two purposes: (1) After a worker completes its iteration $i$, it commits the output to buffer only if the last committed iteration is $i - 1$. Otherwise, the worker waits for previous iterations to commit, which ensures that the iterations are committed in-order although they complete out-of-order. (2) The iteration latency of producer stage $s$ during last $N_{max} + 1$ iterations can be calculated by Equation 5.3.

$$IL(s) = \frac{CT(i - 1) - CT(i - 1 - N_{max})}{N_{max}} \quad (5.3)$$

where $CT$ returns the commit time of iteration $i$. Note that computation of iteration latency using above Equation 5.3 inherently includes task latency, buffer delays and overhead, and thus is equivalent to Equation 5.2.

### 5.5.3 Worker Program

Figure 5.6 shows the code structure of a worker’s program. Every worker contains the complete application code (including all the tasks) in its local memory, and executes only the task of the stage that is assigned to it. First, a worker fetches the stage that is assigned to it from the workers assignment table using `GetStage()` function. Then, it reads the number of the last committed iteration ($it$) and data block ($in$) from the input buffer of the assigned stage. During the read, if the buffer is empty, then read delay is logged as explained in Section 5.5.2.

Once the input data block is read, the worker executes the task of the assigned stage. Finally, the output data block ($out$) is committed to the output buffer. As
function workerprogram() {
    while(1) {
        s = GetStage();
        read(s, it, in);
        switch (s) {
            case 1:
                task1(); break;
            case 2:
                task2(); break;
            ...
            default:
                break;
        }
        write(s+1, it, out);
    }
}

Figure 5.6: An Example of Worker Program.

explained in Section 5.5.2, the commit is only done if all the previous iterations of
the assigned stage have committed (last committed iteration of output buffer ==
it – 1). Further, during the commit, write delay is logged if the buffer is full. It
could be seen that assigning a worker to a stage (to increase in-parallel iterations)
is very quick as the worker needs to access only the workers assignment table.

5.5.4 Manager Program

The manager program implements Algorithm 1 which repeatedly executes its con-
trol loop. In every execution of the loop, the manager accesses buffer delays to
detect bottleneck stage and noncritical stages, and commit times to calculate iter-
ation latencies of the stages as explained in Section 5.5.2. It is important to note that the manager only needs buffer delays and commit times to decide how to expand/shrink the stages, which enables ADAPT to perform rapid (fine-grained) run-time adaptations. Once the manager decides to expand/shrink stages, it updates the workers assignment table. The workers assignment table along with complete application code in each worker enables quick assignment/shuffling of the workers to increase/reduce in-parallel iterations of the stages at run-time. Note that unallocated workers (workers with a value of -1 in workers assignment table) are not used when stages are expanded/shrunk. Thus, resource variations (changes in allocation of workers) are taken into account by the manager during every run-time adaptation. Further, the manager does not use an operating system, and thus avoids the typical delay of operating system based run-time adaptive methods [95].

5.6 Experiment

We implemented ADAPT methodology in an on-chip many-core system with 48 Tensilica’s Xtensa LX4 [113] cores. Tensilica’s cycle-accurate multiprocessor simulation platform, XTensa Modeling Protocol (XTMP), is used to measure performance and energy consumption. Each core is configured at 1 GHz frequency and 45nm technology, with 1 KB instruction cache, 1 KB data cache and 1 MB local memory (for instructions and local data). The input/output buffers of pipeline stages are mapped to a 25 MB global shared memory. The shared memory is di-
vided such that each application occupies a pre-allocated section. For the benchmarks used in experiments, the shared memory per application was less than 4 KB, and thus 25 MB is sufficient for a system with 30+ applications executing at the same time. In our setup, the average delays for accessing L1 cache, local memory and shared memory are 1, 3 and 10 cycles respectively. Further, a global timer in the system is used by all the cores for timestamping of iterations.

We evaluate ADAPT using three case studies. In case study 1, we examine the efficacy of ADAPT under static workload, that is, the workload does not change from one iteration to another. In case study 2, we compare ADAPT to the state-of-the-art [95] under dynamic workload, that is, workload changes from one iteration to another based upon the input data. In case study 3, we examine the efficacy of ADAPT (in terms of throughput and energy consumption) under run-time resource variations. In each case study, we map three pipelines to the 48-core system. In case studies 1 and 2, we allocate 16 cores per pipeline, where one core acts as pipeline manager while the other 15 cores act as workers. In case study 3, for each pipeline, we change the number of allocated workers from 6 to 15 at run-time. Note that at the start of each pipeline, every stage is assigned to one worker with the rest of the allocated workers set to idle, which are assigned to the stages by the manager at run-time.

### 5.6.1 Case Study 1

In this case study, we used *MJPEG* (6 stages, 150 iterations), *ADPCM* (8 stages, 128 iterations) and *Bitonic sort* (9 stages, 750 iterations) applications as pipelines.
Figure 5.7: ADAPT Under Static Workload.

The assignment of workers to stages at run-time and the resulting throughput are depicted in Figure 5.7 for the first few iterations of the pipelines. Note that we show only the interesting stages, in particular stages that are expanded/shrunk. Labels A to E show that the throughput increases when the corresponding stages are expanded. After a while, the throughput stabilizes as all the workers have been assigned to stages. Table 5.1 shows the total execution time, energy consumption (including cores and caches), active on-chip area (from average number of active workers) and average time for one run-time adaptation. The time per adaptation varies from 624 ns to 628 ns.
Table 5.1: Results of Case Study 1.

<table>
<thead>
<tr>
<th></th>
<th>M_JPEG</th>
<th>ADPCM</th>
<th>Bitonicsort</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of allocated workers</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Active area [\text{mm}^2]</td>
<td>3.12</td>
<td>3.12</td>
<td>3.12</td>
</tr>
<tr>
<td>Execution time [ms]</td>
<td>60</td>
<td>58</td>
<td>76</td>
</tr>
<tr>
<td>Energy consumption [mJ]</td>
<td>32.16</td>
<td>31.10</td>
<td>40.73</td>
</tr>
<tr>
<td>Time per adaptation [ns]</td>
<td>624</td>
<td>628</td>
<td>624</td>
</tr>
</tbody>
</table>

### 5.6.2 Case Study 2

In this case study, we used \textit{MPEG\_DEC} (6 stages, 600 iterations), \textit{FFT} (6 stages, 500 iterations) and \textit{H.264enc} (6 stages, 500 iterations) applications as pipelines. The number of stages with dynamic workload in \textit{MPEG\_DEC}, \textit{FFT} and \textit{H.264enc} is 2, 3 and 2 respectively. We adopt the work in [95] for comparison as its parallelization model (i.e. \textit{task clone}) is similar to ADAPT, in contrast to [93,94] where task re-mapping is used. We adopted the \textit{work-ratio based method} from [95] and use the pipeline manager to perform \textit{task clone}. We implemented their method in the same 48-core system without an operating system for a fair comparison. Note that their method waits for all the tasks of a stage to complete (for synchronization) before starting the next round of iterations, which unnecessarily reduces throughput.

The assignment of workers to stages at run-time and the resulting throughput are depicted in Figure 5.8 for the first few iterations of the pipelines. Table 5.2 compares ADAPT with [95] for total execution time and time for one adaptation. ADAPT outperforms [95] by $1.15\times$, $2.1\times$ and $1.31\times$ for \textit{MPEG\_DEC}, \textit{FFT} and \textit{H.264enc} respectively. Further, time for one run-time adaptation in ADAPT is
Figure 5.8: ADAPT Under Dynamic Workload.
only 12% to 37% of the adaptation time of [95]. Note that the time per adaptation for ADAPT varies because Algorithm 4 needs to find a suitable noncritical stage for shuffling of a worker.

Table 5.2: Comparison of ADAPT and [95].

<table>
<thead>
<tr>
<th></th>
<th>Execution Time [ms]</th>
<th>Time per adaptation [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPEG_DEC</td>
<td>104</td>
<td>624 – 1816</td>
</tr>
<tr>
<td></td>
<td>120</td>
<td>4909</td>
</tr>
<tr>
<td>FFT</td>
<td>156</td>
<td>625 – 1814</td>
</tr>
<tr>
<td></td>
<td>327</td>
<td>4912</td>
</tr>
<tr>
<td>H.264enc</td>
<td>187</td>
<td>624 - 1816</td>
</tr>
<tr>
<td></td>
<td>245</td>
<td>4911</td>
</tr>
</tbody>
</table>

### 5.6.3 Case Study 3

This case study demonstrates the adaptation to run-time resource variations and corresponding changes in throughput and energy consumption. Each pipeline is initially allocated to 6 workers, and is executed multiple times. Every time a pipeline is repeated, the number of workers allocated to it is varied to 10, 12, 15 and back to 12 by changing the *workers assignment table* (to simulate resource variations). The pipeline manager becomes aware of such resource variations and considers them during run-time adaptation as described in Section 5.5.4. Figures 5.9 and 5.10 show the normalized throughput and energy consumption (calculated using active workers) when resources change. It is observed that increasing the number of workers allocated to a pipeline may increase or decrease its energy consumption, even though the performance improves. Thus, this case study also
Figure 5.9: Throughput Under Resource Variation

Normalized Throughput Improvement

6 Workers
10 Workers
12 Workers
15 Workers

ADAPT
[95]
ADAPT
[95]
ADAPT
[95]
124

6 Workers
10 Workers
12 Workers
15 Workers
12 Workers
Figure 5.10: Energy Consumption Under Resource Variation
hints that a designer should perform design space exploration. For example, if a designer wants to achieve a normalized throughput of 2 while keeping normalized energy consumption below 0.9 for $H.264_{enc}$ pipeline, then he/she should allocate 10 workers to be managed by ADAPT at run-time. Note that, in all pipelines, ADAPT gets better performance and lower energy consumption than [95].

5.6.4 Overhead of Shared Memory Access

Recall that the workers assignment table and all the buffers between pipeline stages with their instrumentation are located in shared memory. The throughput reported earlier in all the three case studies already includes the overhead from contention encountered during shared memory access. Here, we further quantify that overhead by recording the time when any worker waits due to another worker in accessing shared memory (that is, the time spent in acquiring locks). For each worker, we observed an overhead of 168,229 ns on average during complete execution of pipelines, which translates to a loss of 0.8% in total throughput. Thus, contention during shared memory access was not the bottleneck in our experiments.

5.7 Summary

In this chapter, we presented ADAPT, a quick run-time adaptive methodology for high throughput execution of hardware/software pipelines under dynamic workload and resource variations. ADAPT is designed to be implemented in homo-
Software Pipeline and Throughput Maximisation

geneous on-chip many-core systems with shared memory. For a given pipeline, ADAPT adds workers (cores) to its bottleneck stage (when there are idle workers) or shuffles workers across stages (when all the workers are active) to maximally improve throughput. Our experiments with pipelines under static and dynamic workloads, and resource variations illustrate that: (1) ADAPT improves throughput significantly without thrashing; and, (2) run-time adaptation is fast (as fast as 624 ns) compared to state-of-the-art methods.
In this chapter, we expand our run-time adaptation methodology mentioned in previous chapter to system level optimization with multiple streaming applications for low-power consumption. On-chip many-core systems are anticipated to be a large part of the embedded device industry with hundreds and even thousands of cores on a chip [114, 115]. In order to use such a system in embedded applications, the run-time adaptation is necessary which allows the homogeneous many-core system to achieve power and performance efficiencies by dynamically adapting the system to specific applications.

The hardware/software pipeline divides a stream application into sequential stages and assigns a number of cores to different stages [93, 97]. The streaming applications are implemented as hardware/software pipelines when high throughputs are necessary. Traditionally, such pipelines in many-core systems are designed in such a way that the throughput constraints (e.g., frames per second) are met even
in the worst case of workloads. For streaming applications with workload variations, such as H.264 [97], worst-case designs may overestimate workloads by more than an order of magnitude for the average case. In order to adapt workload variations, the work proposed in [93] studied run-time task mapping, and the work in discussed in [95] studied run-time task duplication. However, their works focused on maximizing throughput rather than minimizing both resource usage and power consumption under the throughput constraint. Also their management overheads are typically at the level of hundreds of milliseconds, which are over $10 \times$ the workload variation intervals in many modern streaming applications. Hence their works are not suitable for fast adaptation. Works in [97, 98] applied run-time low power techniques to reduce the dynamic power when the run-time workloads are much lower than the worst case. However, their works still allocate as many cores as necessary for the worst case designs.

In this chapter, we consider the scenario where multiple pipelines with workload variations are executing on the same on-chip many-core system. Since their peak times of workload may not happen at the same time, we studied the system to reuse cores between pipelines, thus, reducing the need for resources and reducing power. We called it Elastic computing.

Elastic computing technologies can adapt to workload variations and reuse cores between applications. We propose $E$-pipeline, a fine-grained elastic computing methodology for streaming applications in on-chip many-core systems. Inside a pipeline, $E$-pipeline applies $intra$-$elasticity$ which adapts core assignments to the workload variations so that the throughput constraint is met. Between separate
Elastic Pipeline

pipelines which are executed in parallel, *E-pipeline* applies *inter-elasticity* which allows the unnecessary cores in one pipeline to be dynamically given up for use by other pipelines or applications running on the same chip. In addition, *E-pipeline* applies low-power technologies (clock gating) to necessary cores to reduce power consumption when throughput constraints of all pipelines are met.

### 6.1 Task Cloning Mode in E-pipeline

![Diagram of pipeline with four stages and task cloning example](image)

Figure 6.1: (a) Pipeline with Four Stages (b) *Task Cloning Mode* Example

Fig. 6.1(a) describes a hardware/software pipeline with four stages (*S*₁, *S*₂, *S*₃, and *S*₄). The input data of each stage is packed as data tokens. The throughput is measured by frames per second and each frame is composed of a certain amount of data tokens. Each stage works on an input data token and then produces a processed data token, which is the input data token of the next stage. The communication between stages is through buffers. An iteration of one stage includes fetching a data token, performing the task of this stage on this data token...
and sending the processed data token to the buffer. The workload of a stage is measured as the average clock cycles consumed per iteration.

We focused on a particular program mode of hardware/software pipelines, named task cloning mode, which can be applied to various streaming applications (we applied it to 14 benchmarks ranging from media applications such as MPEG encoder, and communication applications to signal processing applications, such as FFT). In task cloning mode, the execution for an iteration of the stage only depends on its input data token and does not depend on other data tokens or other stages. Hence, the task cloning mode allows a task of a stage to be cloned several times, and each task clone works on different data tokens independently. Fig. 6.1(b) shows a task cloning mode example for the pipeline in (a). There are various numbers of task clones in each stage. The throughput of a stage is the total throughput of task clones in this stage. The stage with the heaviest workload (the bottleneck stage) determines the effective throughput of the pipeline (within brackets of Fig. 6.1(b)). Based on the discussion in work [20,98], we can increase the number of clones at the bottleneck stage to escalate the effective throughput until this stage is no longer the bottleneck. Similarly, we can decrease the number of task clones in other stages without affecting the effective throughput until the stage becomes the bottleneck stage. Details of discussion are explained in [20,98].

In this chapter, it is noteworthy that one core executes one task clone for the sake of simplicity thus reducing overheads. Intra-elasticity and inter-elasticity are employed by changing the numbers of cores/clones which are assigned to the stages of pipelines.
6.1.1 Motivational Example

In this section, we examine a hardware/software pipeline example of H.264 encoder with a throughput constraint of 30 frame/s. The H.264 encoder shown in Figure 6.2 (a) is composed of six tasks: task CC for color conversion; task ME for motion estimation; task IPoMC for intra-prediction or motion compensation; task TQE for transform, quantization and entropy coding; task ITQ for inverse transform and quantization; and task WB for writing back. The workloads of task ME and task TQE depend on the input data (similarity between the current input frame and the reference frame).

![H.264 Encoder Example](image)

Figure 6.2: (a) H.264 Encoder Pipeline (b) Task with Workload Variations

The workload distributions of Task ME and TQE (represented by execution time for a data token) of the H.264 example is shown in Figure 6.2 (b), where the execution time of ME and TQE varies significantly. Note that the worst case of the
workload for ME is not the worst case for TQE: When the workload of ME is high, the workload of TQE can be low, and vice versa. It necessitates the intra-elasticity to change core assignments between different worst cases of the pipeline.

![Number of Necessary Cores Variations of Two Pipelines](image)

With run-time workload variations, the number of cores which are necessary for the H.264 pipeline to meet the 30 frame/s constraint also varies with time. There are two pipelines of the H264 encoder with different input data (H.264_1 and H.264_2). Figure 6.3 shows the necessary run-time numbers of cores needed by H.264_1 and H.264_2. It is worth noting that, in Figure 6.2 (c), the total number of cores used in the two pipelines varies (the third curve from the top) and never exceeds 26 (the second dotted line from the top). Each pipeline will occupy 15 cores at their peak points, however, they do not occupy 15 cores at the same time. If the pipelines are designed based on worst case designs, they will always occupy 30 cores in total (each pipeline will occupy 15 cores). Similarly, if we execute two or three differing applications, then we would be able to execute on a lesser number of cores, compared to the design of allocating the worst case number of
cores for each application. Thus *E-pipelines* allow switching for cores between pipelines, reducing the necessity for allocating a maximum number of cores.

### 6.2 E-pipeline Overview

![E-pipeline Diagram](image)

Figure 6.4: System Overview of E-pipeline
In E-pipeline, the on-chip many-core system assigned a core named ‘manager’ for each hardware/software pipeline to be executed. Fig. 6.4 shows the overview of the system with three pipelines (Pipeline A, Pipeline B and Pipeline C). The managers perform the run-time adaptation, while the workers are assigned to separate pipelines to execute specific tasks (workers for different stages are separated by vertical lines within the pipelines in Fig. 6.4). The communication between cores in a pipeline is based on the shared memory. There is a common sleep core pool where unnecessary cores - depicted by S - are set to sleep (i.e., clock gating). Fig. 6.4 (a) and (b) shows two scenarios before and after the adaptation, which are demonstrated by directed dotted arrows. For Pipeline A, an adaptation of intra-elasticity is performed as a core is switched from one stage to the other stage. For Pipeline B and C, inter-elasticity is performed through sleep core pool. A sleep core is assigned to Pipeline B, while a core in Pipeline C is set to sleep and added to the sleep core pool, indicated by arrows. Note that, in this chapter, we assume that there is a sufficient number of cores in the system. If there are not enough cores, it may indicate that the system can only guarantee the throughput of important pipelines. Such a priority based pipeline system is not studied here, but is a simple extension of this work.
6.3 Methodology

6.3.1 E-pipeline Methodology Overview

Figure 6.5 (a) shows the program executed on the manager of a pipeline. The input is a pipeline with $S$ stages. The manager initializes the pipeline with one core assigned to each stage. At run time, the manager monitors the execution information and performs an adaptation. In the adaptation, it first reads the monitor information, then finds the bottleneck stage and the non-critical stage (the bottleneck stage restricts the throughput and the non-critical stage contains one or more cores that are not necessary), performs the elasticity management (adaptation), and finally sends task assignments to cores (task assignments consist of choosing cores and sending task assignment information of cores if the cores are used for adaptation; if a core is to be sent to sleep, a sleep signal is sent to it). The execution of the worker is explained in Section 6.3.2. After the adaptation, the manager monitors the pipeline again. A new adaptation is performed after a set number of iterations.

![Diagram of Manager Program]

Figure 6.5: Manager Program
6.3.2 Worker Program

Algorithm 5: Execution of A Worker

```
while the pipeline termination condition is false do
    pipeline id = FetchPipelineid();
    stage id = FetchStageid();
    FetchDT(pipeline id, stage id);
    ExecuteStage(pipeline id, stage id);
    SendDT(pipeline id, stage id);
    CollectandSendTimeInformation();
end
```

Algorithm 5 presents the execution loop of a worker. In every iteration of the execution loop, the core first checks the pipeline it is assigned to by fetching `pipeline id`, and then checks the stage of the pipeline it is assigned to by fetching `stage id`. The `pipeline id` and the `stage id` are the task assignment information sent from the manager. Based on the `pipeline id` and the `stage id`, the core fetches one input data token `[FetchDT()]`, executes one iteration of the assigned stage `[ExecuteStage()]`, sends one output token using function `[SendDT()]` to the output buffer and executes function `CollectandSendTimeInformation()`. Function `CollectandSendTimeInformation()` collects time information for adaptation and stores time information in memory. After `CollectandSendTimeInformation()`, the core checks to see whether the manager has updated the `pipeline id` and the `stage id` or whether it has asked the core to go to sleep. If the manager has changed the status, the core executes the new task or goes to sleep, else it continues with the old task.
6.3.3 Adaptation Methodology

The adaptation method of manager is shown in Algorithm 6. The new adaptation is triggered when \( n \) iterations are complete after the last adaptation (note that we have determined \( n \) experimentally). The input of the adaptation is an array of \( t \) (\( t_1 \) to \( t_n \)) and an array of *unutilized time* of each stage. Both arrays are the time information collected by workers (details of time information collection method are explained in Section 6.4.3). The \( t_1 \) to \( t_n \) denote the finishing times of each of the last \( n \) data tokens. The average throughput of the previous \( n-1 \) iterations is calculated by:

\[
\text{Throughput} = \frac{t_n - t_1}{n - 1}
\]  

(6.1)

The *unutilized time* is the average locking time per worker in a stage. The locking time is caused by *write/read locking mechanism*, which is employed for synchronization. For example, the reading operation on the input data is locked when the input buffer is empty, while the writing operation on output data is locked when the output buffer is full. The locking time of a stage (the time locked by *write/read locking mechanism*) reflects the workload difference between this stage and the *bottleneck stage* [20].

The adaptation is composed of *ThroughputMeasurement()*, *FindBottleneckNon-criticalStage()* (see below), and *ElasticityManagement()* (see below). Finally, the manager resets the time information, and then restarts another adaptation after \( n \)
Algorithm 6: Adaptation Methodology

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>while $t_u &gt; 0$ do</td>
</tr>
<tr>
<td>2</td>
<td>throughput = ThroughputMeasurement(t);</td>
</tr>
<tr>
<td>3</td>
<td>FindBottleneckNoncriticalStage();</td>
</tr>
<tr>
<td>4</td>
<td>ElasticityManagement();</td>
</tr>
<tr>
<td>5</td>
<td>Reset time information;</td>
</tr>
<tr>
<td>6</td>
<td>end</td>
</tr>
</tbody>
</table>

6.3.3.1 Find Bottleneck Stage and Non-critical Stage

Algorithm 7 shows the algorithm of FindBottleneckNoncriticalStage(). As mentioned in Section 6.3.3, the unutilized time of a stage (the average locking time of write/read locking mechanism per worker in a stage) reflects the average under-utilization extent of a worker in this stage. The stage with the least under-utilization is considered as the bottleneck stage ($S_{bottleneck}$). For other stages, we can calculate the normalized number of cores that are utilized ($N_{normalized}$), and then estimate the throughput after removing one core in this stage. If the result is still greater than the constraint, this stage is named the non-critical stage ($S_{noncritical}$). When there are multiple bottleneck stages and non-critical stages, the last one of them is identified as $S_{bottleneck}$ or $S_{noncritical}$. When there is no non-critical stage or bottleneck stage, the value of $S_{noncritical}$ or $S_{bottleneck}$ remains 0.
6.3.3.2 Elasticity Management

Algorithm 8 shows the algorithm of ElasticityManagement(). According to the comparison between the measured throughput (throughput) and the throughput constraint, different strategies are chosen. When the throughput is smaller than the constraint, the strategy, based on the discussion in Section 6.1, is to increase the throughput by increasing the number of workers in the bottleneck stage. The first option is to reassign a worker from the non-critical stage to the bottleneck stage (intra-elasticity). If there is no non-critical stage, the manager awakes a sleep core from sleep core pool and assigns this core to the bottleneck stage (inter-elasticity). When the throughput is greater than the constraint, the strategy, based on the discussion in Section 6.1, is to decrease the number of workers in the non-critical stage and set the removed core to sleep (inter-elasticity). If there is no
non-critical stage, no action is performed.

---

**Algorithm 8: Elasticity Management**

```plaintext
if throughput < constraint then
    if \( S_{\text{noncritical}} = 0 \) then
        \( C_i = \text{AwakeACore(sleep core pool)}; \)
        // inter-elasticity
        Core-to-Stage(\( C_i, S_{\text{bottleneck}} \));
        Exit;
    end
    \( C_i = \text{SelectCore}(S_{\text{noncritical}}); \) // intra-elasticity
    Core-to-Stage(\( C_i, S_{\text{bottleneck}} \));
end

if throughput > constraint then
    if \( S_{\text{noncritical}} = 0 \) then
        Exit;
    end
    \( C_i = \text{SelectCore}(S_{\text{noncritical}}); \)
    SleepCore(\( C_i \)); // inter-elasticity
end
```

---

### 6.4 Implementation

This section details the implementation of the *E-pipeline*. The platform setting is presented, followed by the implementation details of memory layout, throughput/unutilized time measurement and core-to-stage assignments.
6.4.1 Platform Setting

Fig. 6.6 shows the overview of the implementation. The basic prototype of Elas tic Pipeline is built with 48 Tensilica’s XTensa LX4 [113] cores and two memories. Each core is working in the frequency of 1 GHz and tailed with 1KB instruction cache, 1KB data cache, 1MB local memory (for use of instructions as well as local data storage). A 256MB main memory is used for general purposes, such as storing input data, experiment results and run-time measured power. A shared memory of 16MB is used for the management and communication of hardware/software pipelines. The management and communication of a pipeline in our benchmark library require no more than 8KB, hence, the size of 16MB is sufficient to run multiple pipelines in parallel. Finally, there is a global timer which is
used for cores to get time stamps.

### 6.4.2 Memory Layout

The shared memory are divided into blocks (labeled by $A$, $B$, etc. in Fig. 6.6) for the management and communication of pipelines. Block $A$ is the *core-to-stage assignment table*. Each core has a *core id* ($C_1$ to $C_{48}$) which corresponds to an address in the *core-to-stage assignment table*. Each address stores a two-byte data. The upper byte denotes the *pipeline id* while the lower byte denotes the *stage id* (if the core is sleeping, the number is set to 0). Block $B$ is the *sleep core table*. When a core is set to sleep, the *manager* writes its *core id* into this table. When this core is assigned to a pipeline, the *manager* removes the *core id* of a core from the *sleep core table*.

The remaining part of the memory is divided into blocks for multiple pipelines ($Pipeline_1$, $Pipeline_2$, etc.). Each block (e.g. $Pipeline_1$) is composed of sub-blocks (as shown in Fig. 6.6): Block $C$ is an *assigned core table*, where the *manager* records *core ids* of workers in the pipeline. The sub-blocks from $B_1$ to $B_n$ are the $n$ buffers that are used for communication between stages, and $Mutex_1$ to $Mutex_n$ are for the mutual exclusion (Mutex) locking mechanism for these buffers. Each Mutex contains *read and write pointers, empty and full signals, locks to avoid race conditions during buffer access by multiple readers/writers* and *data sequence signals*. The *data sequence signal* stores the sequential number of the last data token stored in the buffer. The new data token (e.g. data token $i$) can be sent to the buffer only when its previous data token (e.g. $i-1$) is already recorded by the
data sequence signal. Finally the sub-blocks $D$ and $E$ store the array of $t$ ($t_1$ to $t_n$) and the array of unutilized time of each stage, which are used by adaptation. The empty and full signals of each buffer are also used to detect the write/read locking.

6.4.3 Throughput and Unutilized Time Measurement

When the final stage of a pipeline completes a data token, the final stage creates a time stamp and stores it in the shared memory in sequence as $t_1$ to $t_n$. The time stamps $t_1$ to $t_n$ are fetched by the manager for throughput measurement (illustrated in Equation 6.1) when $n$ time stamps are available. After adaptation, the time stamps are cleared by the manager so that new time stamps can fill this area. The unutilized time of each stage is stored in data structures in the area labeled as $E$ in Fig. 6.6. Each data structure is a software accumulator, to which all workers of a stage send their unutilized time. When the empty and full signal of a buffer indicates buffer empty, a core which is going to read data from the buffer is locked by the write/read locking mechanism, and the core creates a time stamp to record the beginning of the locking time. The core keeps checking the empty and full signal of this buffer periodically until the lock is released, and then creates another time stamp to record the end of the locking time. The difference between the two time stamps is the unutilized time for this core. When the empty and full signal of a stage is full, the unutilized time is similarly recorded. The unutilized time is accumulated in the software accumulator of the stage. The manager gets the average unutilized time of a core in this stage through dividing the sum of the accumulator by the number of cores assigned to the stage.
6.4.4 Core-to-stage Assignment Implementation

The pointers of different blocks are known to cores (for example, the addresses of A, B, etc. in Fig. 6.6 are known) when a manager performs Algorithm 8. The function SelectCore(i) returns a core id of a core from the stage i by checking assigned core table. The function AwakeACore(S) wakes a core from the sleep core pool and returns the core id of the core by checking sleep core table. The returned core id is used in the function Core-to-Stage(core id, stage id). Function Core-to-Stage(core id, stage id) changes the task assignment of the core by writing stage id to the corresponding address of core id in the core-to-stage assignment table. At the start of one iteration, the worker fetches the pipeline id and stage id from the core-to-stage assignment table, and jumps to the new stage. The time overhead of the worker to switch from one stage to the other is 231 ns (measured during experiments described below).

6.5 Results

We first executed one benchmark at a time to verify intra-elasticity of the system, and then executed several benchmarks in parallel to verify inter-elasticity of the system. The results of E-pipeline are compared to reference designs (minimum number of cores needed to satisfy throughput in the worst case) with clock gating as used in [97]. In E-pipeline, when a core is added to the sleep core pool, it is set to sleep (applying the same clock gating technique as in [97]). Note that the work in [97] used a large number of cores so that worst cases could be handled,
and the cores were switched off, when the worst case scenario was not presented. The *E-pipeline*, on the other hand, allowed run-time task reassignments from one stage to another or from one pipeline to another. We used identical methods to find *bottleneck stages*, *non-critical stages* and measure throughputs in both techniques so that we can make a fair comparison.

Seven benchmarks with workload variations were used. These were H.264 encoder (H264), MPEG decoder (MPEGdec), MPEG encoder (MPEGenc), finite impulse response filters (FIR), fast fourier transform (FFT), Compress Algorithm (Compress), and insert-sort algorithm (Insertsort). For media applications, the throughput constraint was set to 30 frames per second; while for other applications, the throughput constraint is set to the maximum throughput that can be achieved in a hardware/software pipeline with 15 *workers* in the reference design.

### 6.5.1 Experiment Setting

We built our *E-pipeline* using Tensilica’s cycle accurate multiprocessor simulation tool – XTensa Modeling Protocol (XTMP) [116]. The average memory delays of accessing caches, local memories, the shared memory and the main memory were set to 1 cycle, 3 cycles, 8 cycles and 64 cycles. We assumed that the shared memory was non-blocking for multiple accesses. The clock gating technique was applied to both *E-pipeline* and the reference designs for fairness of comparison. Based on the LX₄ parameters given by XTMP (in 45nm technology), the switching overhead for clock gating was set to 1 clock cycle, and the dynamic power/leakage power was set to 41.23/4.25mW per core. We assumed
that when a core was working, it consumed both dynamic power and leakage power (45.48mW); while when a core was set to sleep, it consumed only leakage power (4.25mW). The power consumption was calculated by the number of working cores multiplied by 45.48mW and the number of sleeping cores multiplied by 4.25mW. We evaluated the average power consumption throughout the lifetime of executing benchmarks. In addition, the manager was considered to be a working core and its power consumption was included in the measurement.

The adaptation started every \( n \) iterations. Differing values for \( n \) were explored. We changed it from 4, 6, 10, 12 to 15, and measure the average throughputs of all benchmarks with different \( n \). The average normalized throughputs with different \( n \) were 1.044, 0.967, 1.016, 0.975 and 1.029 (the normalized throughput constraint is 1). There was no clear pattern, due to the varying benchmarks and the differing loads. However, for the sake of experimentation we used \( n=10 \), which provided the closest throughput compared to the throughput constraint, in experiments for which the results are provided below.

### 6.5.2 Experiment Results

Fig. 6.8 and 6.7 shows the execution of running single benchmarks. Fig. 6.8 compares the normalized average throughputs between **E-pipeline** and the reference design with clock gating (we set the normalised throughput constraint as 1) while Fig. 6.7 shows the normalized average power consumptions of **E-pipelines** and the reference design (we set the normalized power consumption 1 when all cores of the reference designs are always working without the manager). The average
Figure 6.7: Measured Throughputs in Different Benchmarks
Figure 6.8: Measured Power in Different Benchmarks
Elastic Pipeline

throughput of $E$-pipeline (normalized 1.013) is slightly less than the reference designs (normalized 1.016). This is due to the switching overhead for reassigning cores to tasks in $E$-pipeline (typically 231 ns in experiments). However, the $E$-pipeline achieves better power saving (0.70) than the reference design (0.72) because it allows cores to be reassigned from one stage to another. For example, the $E$-pipeline can reassign one core from Stage A to Stage B; while the reference designs have to awake another core in Stage B, and then set one core in Stage A to sleep, consuming extra leakage power. The higher the leakage power is, the better power saving the $E$-pipeline can achieve. The management time overheads vary from 678 to 1912ns. The 678ns is the time overhead when the manager does not find a non-critical stage and the adaptation function exits (line 9-10 in Algorithm 8). The 1912ns is the maximum time overhead, including adaptation overheads and Mutex locking delays. This time only includes intra-elasticity adaptation.

Fig. 6.9 shows the results of running benchmark combinations in parallel. Fig. 6.9 (a) is the execution of the combination of FIR and FFT; (b) is the combination of H.264 and MPEGdec; (c) is the combination of MPEGenc and MPEGdec; (d) is the combination of H.264, MPEGdec and MPEGenc; (e) is the combination of H.264, MPEGdec and FFT; and (f) is the combination of Insertsort, Compress and MPEGenc. The curves at the bottom (labeled as $A$ in Fig. 6.9) are the runtime numbers of cores used for each pipeline in $E$-pipeline. The dotted curves at the middle (labeled as $B$ in Fig. 6.9) are the total number of cores used in $E$-pipeline. The first curves from the top (labeled as $C$ in Fig. 6.9) are the numbers of cores used for the reference designs with clock gating, and the second curves
from the top (labeled as D in Fig. 6.9) are the maximum numbers of cores used in E-pipeline. Note that, the numbers of cores in Fig. 6.9 do not count managers, as the number of managers do not change at run time. From Fig. 6.9, we can see that resources are saved since the worst case workloads of different pipelines normally do not occur at the same time. For reference designs with clock gating,
unnecessary cores in a pipeline can only be set to sleep rather than share with other pipelines. Thus, the maximum number of cores being used by pipelines is reduced in *E-pipeline* by sharing cores between pipelines. The management time overheads vary from 694ns to 2352ns. The maximum overhead (2352ns) includes both inter- and intra-elasticity adaptation. Compared to works in traditional methodology with adaptive cores which take hundreds of milliseconds in management time overheads [93, 95, 117, 118], the management time overheads of *E-pipeline* are significantly small.

Table 6.1 summarizes system-level results of different benchmark combinations in Fig. 6.9 (i.e. (a) shows results of the benchmark combination in Fig. 6.9 (a), and (b) shows results of the benchmark combination in Fig. 6.9 (b), etc.). In Table 6.1, *E-pipe* denotes *E-pipeline* and *Ref.D* denotes the reference design. The results include the total number of cores used in the system at run-time (*Number of Cores* - the range and the average number), the average power of the system (*Power*) and the energy consumption of the system (*Energy*). Note that, the managers are considered in system-level results. From the table, we can see that *E-pipeline* can achieve nearly the same power saving factor as the reference designs with clock gating. However, the *E-pipeline* can save an average 37.7% of resources compared to reference designs (measured by the average number of cores used in *E-pipeline* and the number of cores used in the reference design). As the number of pipelines being executed increases, the number of cores saved continues to increase, allowing additional applications to be executed in the many-core system.
Table 6.1: Summary of Results (Including Managers)

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>Power (mW)</th>
<th>Energy (mJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E-pipe</td>
<td>Ref.D</td>
<td>E-Pipe</td>
</tr>
<tr>
<td>Rang. Ave.</td>
<td></td>
<td>E-Pipe</td>
</tr>
<tr>
<td>(a) 16 ∼ 29</td>
<td>18</td>
<td>33</td>
</tr>
<tr>
<td>(b) 17 ∼ 28</td>
<td>20</td>
<td>31</td>
</tr>
<tr>
<td>(c) 17 ∼ 27</td>
<td>20</td>
<td>31</td>
</tr>
<tr>
<td>(d) 22 ∼ 40</td>
<td>28</td>
<td>46</td>
</tr>
<tr>
<td>(e) 23 ∼ 38</td>
<td>26</td>
<td>46</td>
</tr>
<tr>
<td>(f) 21 ∼ 44</td>
<td>36</td>
<td>48</td>
</tr>
</tbody>
</table>

6.6 Summary

The chapter describes *E-pipeline*, an elastic computing method for power and resource efficiencies in on-chip many-core systems. Based on task cloning mode, *E-pipeline* explores intra-elasticity and inter-elasticity for multiple hardware/software pipelines operating in parallel on a chip. The manager cores monitor the execution of each pipeline, change task assignments of workers to adapt to workload variations and meet throughput constraints, reuse cores between pipelines by the use of a sleep core pool and set unnecessary cores to sleep. In experiments on a platform of 48 cores, the results show that *E-pipeline* can meet the throughput constraints for pipelines with fine-grained workload variations, achieve the same power efficiency as the reference designs with clock gating, and save 37.7% in the use of cores for a variety of benchmarks, with roughly 2µs adaptation overhead.
Chapter 7

Future Work

7.1 Compiler

A sophisticated compiler is important for the future development of our run-time reconfiguration methods. On the DRMA platform, designers currently write assembly code as shown in Chapter 4. However, if there is a compiler that can explore parallelism, composing assembly codes and compiling assembly code; the time and effort for programming applications in DRMA will decrease significantly. As future work, we plan to build a compiler for DRMA, which is similar to the VLIW compiler that is used in VLIW architectures [119]. The VLIW compiler explores instruction-level parallelism, and composes sub-instructions that can be executed in parallel together to form long instruction words. The DRMA compiler will explore instruction-level parallelism and data width reconfiguration. An adroit compiler can apply these modes flexibly to achieve better performance. An
algorithm example is shown in Algorithm 9 for an MPSoC system with 32-bit cores. Firstly, the compiler analyses the sizes of operands of the programs, decides the proper data width for each operation, and then compiles the program into instructions, arranging from 32-bit instructions to large-bit instructions. Secondly, the compiler parallelises instructions and maps them into cores. Thus, the DRMA compiler can check the data size of the program and parallelise instructions with different data widths.

**Algorithm 9: DRMA Compiler**

```plaintext
for each Operation from 1 to \( t \) do
    Data Width Compilation;
    if The operation \( y \) is 32-bit operation then
        Compile the operation \( y \) into 32-bit instructions;
        Assign one core to execute instruction \( y \);
    end
    if The operation \( y \) is 64-bit operation then
        Compile the operation \( y \) into 64-bit instructions;
        Assign two cores to execute instruction \( y \);
    end
    if The operation \( y \) is 128-bit operation then
        Compile the operation \( y \) into 128-bit instructions;
        Assign four cores to execute instruction \( y \);
    end
end
```

### 7.2 Multi-constraint Management

In Chapter 5, the target of the run-time reconfiguration method ADAPT is to maximise the throughput under a constrained number of cores. In Chapter 6, E-
pipeline focuses on minimising the number of cores used at run time and to allow cores to be shared in multiple pipelines. It is also possible to mix both methods together and build a system with multiple constraints. For example, when multiple pipelines are executed in parallel. Some of them can maximise throughput with a fixed number of cores; while some of others can minimise the number of cores for throughput constraints. In addition, other constraints, such as energy constraints or thermal dissipation constraints can be applied in pipelines. Multi-constraint management is a problem worth researching, since the users’ requirements for different applications would differ markedly.

7.3 Run-time Reconfigurability for Reliability

Module redundancy techniques, such as double-module-redundancy (DMR) and triple-module-redundancy (TMR) [120], are widely used to improve system reliability. Nidhi Aggarwal et al. in [121] discussed a module redundancy methodology utilising cores in MPSoC systems as redundant modules to improve the reliability of MPSoC systems. Run-time reconfiguration methods may improve the flexibility of module-redundancy techniques in MPSoC systems. According to the reliability requirement of the task, users can dynamically decide the number of redundant modules. For example, in DRMA, we can combine four 32-bit cores to form a 128-bit core, or combine two 32-bit cores to form a 64-bit core. Similarly, in the future, we may combine four cores to form a core with 4 redundant modules to provide high reliability, or combine two cores to form a core with 2 redundant
modules to provide standard reliability, based on the reliability requirements of given applications. In addition, applying reliability-aware methodology [122] in ADAPT/E-pipeline may allow the system to dynamically identify the soft error vulnerability of a pipeline stage [123]. Based on the error vulnerability of the stage, the system can deploy sufficient redundant modules in the stage carefully to achieve cost-efficient reliability enhancement.

7.4 Other Challenges of MPSoC

There are further challenges to be addressed. Traditional programming languages are based on sequential processing which is not naturally working in parallelism [124]. The program model should be scalable and easy to be parallelised, so that it can be executed by differing numbers of cores. There are modern programming languages that are designed for parallelism in MPSoC systems. Thies et al. in their work [125] proposed StreamIt — a programing language that is suitable to explore parallelism in streaming applications. Stone et al. in [126] discussed OpenCL, a program mode used for task and data parallelism in heterogeneous MPSoCs. Designing a programing language that can map applications to cores is also useful for our run-time reconfiguration method. In addition, there are already a large number of current programs written in C, Java, etc. Migrating them to new program models would consume a large amount of time and effort. Developing a methodology to convert these programs into the format for MPSoC systems is also a promising area to study.
Future Work

Shared memory is widely used for communication between cores in MPSoC systems. With the growth of numbers of cores on a chip, memory congestion is increasing. Gabriel et al. [127] built a many-core system framework with MPI. In their experiments, throughput was improved significantly by adroitly selecting task mapping. Briere et al. discussed a system with optical NoC network for streaming applications [128]; and their optical NoC network gained $1.35$ to $3.9 \times$ speed up compared to the baseline MPSoCs. Zvika Guz et al. in [129] discussed the mathematical model of performance in many-core systems when the number of cores increased. They found that the latency of accessing shared caches was the critical part when the number of cores approached 1000. Message passing interface with NoC networks is a potential solution to solve communication issues [21, 22]. Kim et al. in their work [130] proposed a new pipeline NoC router. This router minimized the communication latency by reducing the critical pipeline path through look-ahead routing based on information from its neighbourhood. Beigne et al. in work [131] proposed a low-latency NoC model. Cores on the chip were modeled as synchronous islands, and globally asynchronous locally synchronous (GALS) techniques were implemented to reduce communication latency, as the synchronization of data transfer was only performed for routers on the data path. It is also possible to implement NoC systems in our ADAPT and E-pipeline to reduce the workload of the shared memory.
7.5 Summary

In this section, possible further development of run-time reconfiguration techniques are discussed. The framework of the DRMA compiler is considered as a combination of a task scheduler, a wide bit width compiler and an instruction scheduler. The compiler is designed to explore task-level parallelism, instruction-level parallelism and data width reconfiguration. The run-time adaptation of streaming applications is discussed as multiple constraints can be added to match users’ demand.
Conclusion

In this thesis, we have discussed the run-time reconfiguration method in MPSoCs. Due to their parallel nature and power efficiency, MPSoC systems are increasingly popular in embedded systems. The platform of our run-time reconfiguration method was a homogeneous system with many cores (e.g. 48 cores). Each core was composed of a core with its own instruction cache and data cache, a local memory and a register file used for recording reconfiguration information. Shared memories and FIFOs were used for communication between cores.

Based on the homogeneous MPSoC platform, the thesis studied a series of run-time adaptation methods. The run-time reconfiguration is a promising method to address the issue of massive fabrication expense, since run-time reconfiguration allows a single chip to be application specific to multiple applications at run time. The purpose of making a chip application specific is to improve the performance of the target applications, reduce the power consumption or raise core utilisation
rates. Without run-time reconfiguration, the chip may lose flexibility if it is fabricated for certain applications; it cannot adapt to other types of applications or dynamic workload variations.

Three run-time reconfiguration methods have been discussed with separate case studies. In the first study, the run-time reconfiguration method *DRMA* was developed to solve the issue of varying data widths. In the second, we studied streaming applications, including multimedia applications, network applications, signal processing applications, etc., which were executed as hardware/software pipelines in the MPSoC platform. To solve the issue of the run-time workload variation, a run-time reconfiguration method *ADAPT* was designed to adapt to workload variation and to maximise throughput with the assigned number of cores. The system level optimisation was then studied in the third case study via a run-time reconfiguration method *E-pipeline*, which assigns idle cores to critical applications and sets redundant cores to idle. *E-pipeline* allows cores to be shared between multiple pipelines running in parallel, in order to improve the core utilisation rates and power efficiency.

Chapter 4 explained the issue of varying data widths. With the ongoing development of software, large data (e.g. 128-bit data) is increasingly useful; however, building cores with large data widths is not economical, since their large data widths are wasted in computation of normal-size data (e.g. 32-bit data). Hence, it was necessary to develop a method allowing cores to adapt to different sizes of data. In Chapter 4, DRMA was introduced as a run-time reconfiguration method that could combine cores with normal data widths together to form cores with
Conclusion

large data widths. The DRMA was applied to cores with 32-bit data width. Two 32-bit cores can be combined to form a 64-bit core, while four cores can be combined to form a 128-bit core. The combination could be performed at run time, and the combination command was embedded into the instructions of 32-bit cores. In the combination, each 32-bit core executed a sequential 32-bit section of the whole large-size data, and the data transfer between different 32-bit sections was through I/O ports that were wired to the adjacent cores. The processor design tool ASIPMeister was utilized to realise the methodology of DRMA, and the output of ASIPMeister was VHDL code of the DRMA processor element. The architecture of the DRMA processor element was synthesized in Synopsys synthesis tool Design Compiler. After synthesis, a platform of a four-core DRMA system was simulated in Modelsim and three experiments were conducted. Experimental results showed that DRMA had the ability to vary word sizes with small configuration time and a simple and modular architecture with only 5% overhead in area.

In Chapter 5, the streaming application was discussed for hardware/software pipelines. Such pipelines allow an application to be arranged as a pipeline of stages, and typically resulted in high throughput. In pipelines, balancing pipeline stages was vital for both high throughput and efficient resource utilization. When resources (such as free cores) and/or application workload vary, quick on-the-fly balancing of pipeline stages is required. In Chapter 5, the run-time reconfiguration method ADAPT was proposed. To enable run-time high-speed adaptation for rapid workload variation, we targeted pipelines where data was packed as data to-
kens and the processing of one data token by a stage was referred to as an iteration. Further, we assumed one task per stage where each task was complex enough to be solely executed by a core, in order for us to perform rapid run-time adaptation. In this type of pipelines, task duplication could be used at run time to improve its throughput under increased workload. The MPSoC system can execute multiple pipelines in parallelism. When a pipeline was assigned to the system, a set of cores was assigned to execute the pipeline. A core was assigned as the *manager*, the rest of the cores were utilized as *pipeline workers*. The run-time reconfiguration was performed by the *manager*, and the algorithm includes bottleneck detection, adding new cores and core shuffle. For a given software pipeline, ADAPT added *workers* to its bottleneck stage (when there were idle workers) or shuffles *workers* across stages (when all the workers are active) to maximally improve throughput. The prototype system with 48 Tensilica’s Xtensa LX4 cores was built by Xtensa MPSoC cycle accurate simulation tool *XTMP*, and the communication between cores were through shared memory. Experiments with pipelines under static and dynamic workloads, and resource variations showed that ADAPT improved throughput significantly; run-time adaptation was faster than state-of-the-art methods.

Chapter 6 extended the work of Chapter 5 to the system level optimisation. For streaming applications under throughput constraint, traditional designs were normally based on worst case design and overestimate workloads. We considered the scenario where an MPSoC system executed multiple pipelines in parallel, and the workloads of the pipelines varied at run time. Acknowledging that their
workloads may not achieve their peak at the same time, it is possible to perform run-time adaptation to share cores among pipelines so that the total demand of necessary resources and the corresponding power of the system is reduced. The method is named the *E-pipeline*, and includes intra-elasticity and inter-elasticity. At the application level, inter-elasticity was applied to dynamically set unnecessary cores to idle, or reassign them to another pipeline that needs extra cores to meet the throughput constraint. Within a pipeline, the intra-elasticity was applied to adapt core assignments to workload variations in order to meet the throughput constraint. Low-power techniques were utilized in idle cores to reduce the power consumption of the system. The architecture of *E-pipeline* allows the idle cores to be stripped from the pipeline and grouped as a common sleep core pool.

For the manager core, a core number minimisation algorithm was applied under a throughput constraint. The manager monitors the throughput and unutilized time at run time, and then the bottleneck stage and the non-critical stage were detected by the manager. After detection, the manager performed intra-elasticity to meet the throughput and then inter-elasticity to reduce the number of cores to the minimum necessary number under the throughput constraint. The last step of adaptation was to put unnecessary cores to the common sleep core pool and apply low-power techniques on them. Experiments showed that *E-pipeline* could perform fine-grained adaptation to workload variations in a variety of benchmarks with roughly 2μs adaptation overhead, meeting their throughput constraints and saving 37.7% in the use of cores.
Implementation of DRMA architecture on FPGA

The chapter describes a hardware MPSoC implementation on an FPGA board to realise the run-time reconfiguration method DRMA proposed in Chapter 5, called DRMA implementation. This work is a cooperation work with Elyse Caitlin Wise [14]. The DRMA implementation consists of four cores (called DRMA processing elements), and can vary data width from 32-bit to 128-bit. The implementation details are discussed, and experimental results illustrate the efficiency of DRMA.

A.1 ARGUS Platform

The DRMA architecture is implemented on a state-of-the-art FPGA board – Xilinx Virtex VI FPGA board [132] via the ARGUS platform. ARGUS Platform is an
FPGA development platform for rapid deployment and execution of FPGA-based processor systems. By utilising the ARGUS platform, the time used for prototyping hardware designs significantly decreases, compared to traditional implementation approaches. The front-end of ARGUS is a graphic user interface with three functions: a module generator, a module connector and an assembly compiler. The module generator enables developers to load components described by hardware description languages (i.e., VHDL) and uses these components as modules. In the module connector, developers can select a number of predefined component modules and design the inter-connection of the component modules. The assembly compiler compiles assembly code to machine code, and allows developers to define their own assembly code for their processors. The back-end of ARGUS is aligned to the Xilinx Softcore design tool chain, which assists developers to implement their own cores and load programs to their cores. By utilising the ARGUS and Xilinx Design tool Chain, developers can generate, map, load, route and execute their own processor systems effectively on the target Xilinx FPGA board.

In ARGUS, we firstly loaded the VHDL code of DRMA cores generated by ASIPMeister (refer to Section 4.3) into ARGUS as component modules. These component modules were connected when we built MPSoC systems with four DRMA cores. Such module connection was performed via a user-friendly drag-and-drop interface, and then ARGUS could automatically generate the VHDL design codes for connection, which reduces the design effort and makes the topology visible.
A.2 Implementation

The Virtex VI ML-605 evaluation board [133] was chosen as the physical prototyping platform for implementation. The Virtex VI includes 37,680 slices, 14.976 Kb Block RAM and 61.6 MB configuration memory. The system level overview of the four-core DRMA implementation is shown in Figure A.1. The system is composed of a core network and a reading network that allows the system to retrieve data results. The core network is the main body of the DRMA implementation, composing of DRMA system with four inter-connected DRMA cores, instruction/data memories (IMEM and DMEM in Figure A.1) and bridges between cores and memories. The reading network is implemented to select data for evaluation of this implementation.

A.2.1 AXI/ASI Bridge

The ARGUS platform generates a bridge structure (named the AXI/ASI bridge in Figure A.1), which is used as a communication solution for components with different bus protocols. The AXI/ASI [134] bridge connects each core with its own IMEM and DMEM, and also connects the memory and cores to the reading network (as shown in Figure A.1), so that the users can monitor the data.

A.2.2 Memory

Each core in DRMA is provided with separate 10-kilobyte instruction memory and 10-kilobyte data memory, which are synthesised by partitioning from the
Figure A.1: Implementation Overview, Sourced in [14]

Block RAM. The Block RAM is much faster than distributed RAM or ROM as it is based on the incorporation of a look-up table (LUT). This scheme helps the acceleration of memory operations which are deemed as the most time-consuming
instructions in the DRMA instruction set.

A.2.3 Reading Network

In the Virtex VI ML-605 evaluation board, it does not offer a mechanism to access the data in memories directly at run time, which hinders users from monitoring data in memories at run time and evaluating the behaviour of the DRMA implementation. In order to monitor run-time data changes, the reading network is implemented on the board to fetch data from the memory and the processor elements. The key component of the reading network is a PISA processor as the controller of the reading network. The controller executes a program to systematically read data from memories and sends them to the UART port. Firstly, the controller sends a series of requests to the direct memory access (DMA) through AXI bus, as shown in Figure A.1. The DMA retrieves data in the data memories of cores and then stores them in a FIFO structure (i.e., the controller data FIFO in Figure A.1). Secondly, the controller accesses the FIFO to fetch data and then formats the data. Finally, the formatted data is sent to the UART port. The UART port is connected to a server computer where users can access the UART to read data. The reading network can access multiple data memories and send data to the server. Figure A.2 shows a sample of data read from the UART port.
A.2.4 Performance Evaluation

To evaluate the performance of the DRMA implementation, an event register (the event in Figure A.1) was attached to each DRMA core. A time stamp instruction is used to trigger the core to store its cycle count into the event register. The PISA controller of the reading network in Figure A.1 can then read and display these time stamps to know the execution time of benchmarks.

A.3 Assembly Support for DRMA

An assembly tool is created to assist the program work of DRMA. The tool allows users to enter PISA-like assembly codes, and the tool compiles assembly codes to
DRMA instructions and loads instructions to corresponding cores automatically.

### A.3.1 Assembly Syntax of DRMA

The DRMA assembly language was defined in ARGUS based on the PISA assembly language. The instruction syntax and semantic rules are similar to normal assembly codes, while extra command parameters are added to specify the data width reconfiguration.

In this assembly syntax, each line is composed of four instructions, and each instruction is executed by one of the four processor elements. Hence, programmers command the four processor elements at the same time. The delimiter `::` is used to separate commands for different processing elements. Figure A.3 shows an example of a set of 32-bit instructions of `add`, `sub`, `slt` and `sll`, working in separate processing elements (PEs). The top line in Figure A.3 is the command syntax. The first processor element executes an add operation with operands from registers r1 and r2, and the result is stored in r3. Similarly, the second processor element executes a sub instruction, while the third processor element executes a right shift by 5 bits, and the fourth processor element executes a left shift by 12 bits.

The run-time reconfiguration work of combining multiple processor elements to form a core with large data widths can be achieved by the prefix `#N`. The number N indicates the number of cores that are combined, ranging from 1 to 4 since there are four processor elements in this DRMA system. Figure A.4 shows a syntax with two instructions and each instruction combines two PEs together. Since each PE
Add r1, r2, r3 :: Sub r3, r2, r4 :: Srl r2; r2, 0x04 :: Sll r9, r3, 0x12

Figure A.3: Command Line Sample 1

is a 32-bit core, #2 means two 32-bit cores (PEs) are combined together to form a 64-bit core to execute a 64-bit instruction. In Figure A.4, the first instruction combines two PEs together to execute a 64 bit add, whilst the second instruction executes a 64-bit sub. It is worth noting that 32-bit instructions can be executed parallel with 64-bit instructions. Figure A.5 shows that the first two processor elements are connected to execute 64-bit instructions and the third and fourth cores execute 32-bit instructions separately.

Figure A.4: Command Line Sample 2

Figure A.6 shows a 128-bit slt (set less than) instruction. The assembly tool automatically set processor elements into separate modes to complete the reconfiguration. In this instruction, the first processor element is set by the assembly tool to IMode; the two processor elements in the middle are set to CMode and the
final processor element is set to $TMode$ (details of these models are described in Section 4.2.1).

### A.4 Experiments and Results

This section presents the evaluation and experimental results of the DRMA implementation on the Xilinx Virtex VI FPGA board. A summary of synthesis results on the FPGA board is proposed, and an AES encryption kernel is used as a benchmark to evaluate the performance. The results are compared to a 4-way common PISA implementation without run-time data width reconfiguration.


A.5  Hardware Synthesis

The 4-way DRMA implementation and a 4-way PISA implementation are deployed and compared. Table A.1 presents a summary of the placement and timing results of the generated FPGA implementations. The reading network with the PISA controller is not included in the result as the reading network is implemented only for the purpose of inspecting data results and makes no contribution to the functionality of the DRMA system. Results in Table A.1 indicate that the DRMA architecture costs an extra 2% area overhead compared to the corresponding PISA architecture with the same number of processor elements, close to but smaller than the estimation results of Synopsys Design Compiler (referring to Table 4.2). The maximum frequency (12.5 MHz) is much slower than the result in Synopsys Design Compiler (referring to Table 4.2), which is mainly due to the performance gap between FPGAs and ASICs. As a prototype implementation, the flexibility of the DRMA implementation and the functionality verification are of greater concern than the implementation type, and the FPGA implementation is suitable for research purposes.

A.6  Experiments

A.6.1  Instruction Level Verification

An instruction level test was performed in each DRMA processor element to check the functionality of the instructions. The instruction of memory opera-
Table A.1: Synthesis Results for DRMA Implementation and PISA

<table>
<thead>
<tr>
<th></th>
<th>DRMA</th>
<th>4-core PISA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock Frequency</td>
<td>12.503 MHz</td>
<td>12.504 MHz</td>
</tr>
<tr>
<td>Maximum Path Delay</td>
<td>79.978 ns</td>
<td>79.972 ns</td>
</tr>
<tr>
<td>Occupied Slices</td>
<td>18641</td>
<td>17712</td>
</tr>
<tr>
<td>Slice Registers</td>
<td>24459</td>
<td>24414</td>
</tr>
<tr>
<td>Total Slice LUTs</td>
<td>53083</td>
<td>52655</td>
</tr>
<tr>
<td>LUTs Used as Logic</td>
<td>49838</td>
<td>49401</td>
</tr>
<tr>
<td>LUTs Used as Memory</td>
<td>3093</td>
<td>3093</td>
</tr>
</tbody>
</table>

The operation was first verified by checking whether the stored data matched the original data and whether the address was correct. After verifying the memory operation instructions, other instructions were tested four times with different data and the results were stored in the data memory. The results in the data memory were compared with the estimated results to verify the correctness of instructions. Finally, each instruction was tested with a combinations of five/six other instructions, and the results were also compared with the estimated results. All of the instructions of the DRMA architecture were verified throughout the instruction level test.

A.6.2 AES-128 Kernel Algorithm

An AES-128 kernel algorithm was selected as the suitable benchmark to evaluate the improvement due to varying data widths. AES-128 kernel algorithm is an encryption algorithm [135]. It includes the encoding of a 128-bit input value, matrix computation, bit operations, etc. In this algorithm, the focus was on the operation of encryption as the main body of the AES encryption (the round key
is assumed to be known), and the feature of this AES benchmark is that it is composed of both 128-bit operations and 32-bit operations, which can test the run-time reconfiguration capability of the DRMA architecture. The AES benchmark is composed of 10 rounds where the following four steps are performed:

1. Add Round Key: A logical XOR is performed on each input block and a block of the round key, the results of which are stored in a matrix, named state matrix.

2. Sub Bytes: Each byte of the state matrix is utilised as an index of a substitution matrix. The value that is pointed out by the index replaces the original value in the state matrix. This operation offers the non-linearity.

3. Shift Rows: The rows of state matrix are shifted according to a pre-defined rule.

4. Mix Columns: Each column of the state matrix is multiplied via another matrix.

The AES kernel was coded into assembly languages and loaded into the DRMA implementation via ARGUS. The result of the DRMA implementation was compared to the 4-way PISA implementation. The DRMA implementation execute the AES-128 benchmark by all four processing elements with run-time adaptation, while the 4-way PISA implementation executed four AES-128 benchmarks in parallel. Figure A.7 shows the comparison of results. The DRMA implementation achieved $1.24 \times$ speedup compared to the 4-way PISA implementation. The
DRMA implementation can flexibly reconfigure the system to execute four 32-bit operations in parallel or form one 128-bit core to execute one 128-bit operation; but the PISA implementation can only utilise the parallelism. The 128-bit instructions can transfer data to adjacent cores in one clock cycle (by shift instructions), avoiding the extra operations of storing/loading data in the memory. This feature is especially beneficial for architectures where the memory accesses are slow.

![Figure A.7: Results of DRMA and PISA architectures](image)

**A.7 Summary**

In this section, the DMRA implementation on the FPGA board Xilinx Virtex VI is discussed. The DRMA processor elements are loaded into the FPGA board and connected with memories through the development tool ARGUS. An AES-128 kernel was used as a benchmark to test DRMA, and results show that DRMA can achieve $1.24 \times$ speedup compared to the 4-way PISA implementation.
Bibliography


[22] ADAPTEVA, “Epiphany-iv (e64g401) datasheet,” ADAPTEVA Document. (Cited on pages 14, 39, 42, and 159.)


processor system-on-chip (mpsoc),” in DATE, 2008, March. (Cited on pages 53, 59, and 60.)


[135] NIST, Announcing the ADVANCED ENCRYPTION STANDARD (AES). National Institute of Standards and Technology (NIST). (Cited on page 177.)