PERFORMANCE MODELING OF ADAPTIVE MULTI-CORE ARCHITECTURE

VENKATARAMANI VANCHINATHAN

NATIONAL UNIVERSITY OF SINGAPORE

2015
PERFORMANCE MODELING OF ADAPTIVE MULTI-CORE ARCHITECTURE

VENKATARAMANI VANCHINATHAN
B.Comp(Hons), NUS, 2012

A THESIS SUBMITTED FOR THE DEGREE OF MASTER OF COMPUTING

SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE
2015
Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety.

I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

______________________________________________________________

Venkataramani Vanchinathan
March 16, 2015
Acknowledgment

Foremost, I would like to express my sincere gratitude to my dissertation advisor Prof. Tulika Mitra for the continuous support of my study and research, for her patience, motivation, enthusiasm, and immense knowledge. Her way of looking at problems and passion have always fascinated me. She provided me an opportunity to work on different projects and gain knowledge on multiple fields of computer architecture. I hope to become a researcher like her someday.

I would like to next thank Huawei Technologies Co., Ltd for generously funding my research. Specifically, I would like to thank Ge Zhiguo for providing constructive feedback from time to time.

I thank my fellow labmates Malai and Mihai who never refrained from answering my questions. I was able to learn a lot by working with them. I would also like to thank Alok, LeeKee, GuanHua, TanCheng, Yao Yuan and Henry for tolerating me and making me believe in myself during my lows.

Last but not the least, I would like to thank my Amma, Appa, Akkas and Paatis for making me the way I’m today and providing complete freedom. Words are insufficient to show my gratitude.

March 16, 2015
# Contents

1 Introduction ............................................. 1
   1.1 Thesis contribution .................................. 5
   1.2 Thesis outline ...................................... 6

2 Related Work ........................................... 7
   2.1 Static heterogeneous multi-core ....................... 7
   2.2 Dynamic heterogeneous / Adaptive multi-core ........ 8
   2.3 Full system simulators ................................ 9
   2.4 Performance modeling .................................. 11
   2.5 Online Scheduling .................................... 12

3 Bahurupi Architecture ................................. 15
   3.1 Sentinel Instruction .................................. 17
   3.2 Bahurupi Execution Example ........................... 18
   3.3 Summary ............................................... 20

4 Full System simulation of Adaptive multi-core ........ 21
   4.1 Bahurupi architecture implementation using gem5 Full System Simulator .... 23
      4.1.1 Sentinel Instruction ............................... 24
      4.1.2 GRF Unit transactions .............................. 29
      4.1.3 Branch mis-prediction ............................. 29
      4.1.4 Recovering from Memory Hazards ................. 30
4.1.5 Library function calls ........................................... 30
4.1.6 Shared components ........................................... 32
4.2 Evaluation of Bahurupi architecture on gem5 ............... 35
4.2.1 Setup .......................................................... 35
4.2.2 Benchmarks .................................................... 36
4.2.3 Performance Analysis ......................................... 36
4.3 Discussions and Summary ........................................ 42

5 Performance modeling of Adaptive Architectures ............. 45
5.1 Performance Modeling ............................................. 50
5.1.1 $CPI_{\text{steady}}$ estimation .................................. 51
5.1.2 CPI stack model .............................................. 53
5.1.3 Inter-coalition misses ........................................ 54
5.2 Experimental Evaluation .......................................... 56
5.2.1 Experimental Setup ......................................... 57
5.2.2 Training and Test Benchmarks .............................. 57
5.2.3 Intra-core validation ........................................ 58
5.2.4 Inter-core validation ........................................ 59
5.3 Application of Performance Modeling .......................... 63
5.3.1 Problem Definition .......................................... 63
5.3.2 Online Scheduling algorithm ............................... 63
5.3.3 Experimental Setup ......................................... 64
5.3.4 Experimental Evaluation .................................... 67
5.4 Summary .......................................................... 68

6 Conclusions and Future work ...................................... 69
6.1 Conclusions ...................................................... 69
6.2 Future Work ...................................................... 70
Abstract

Static heterogeneous multi-cores have emerged as a promising alternative to homogeneous multi-core systems in the power constrained embedded domain. However, they lack flexibility in adjusting to dynamic requirements in applications. Dynamic heterogeneous multi-cores (also known as adaptive multi-cores) have been proposed as an alternative because they can dynamically form a coalition of multiple simple cores to increase instruction level parallelism as per application needs. In this thesis, we implement a prototype of a realistic adaptive multi-core architecture on gem5, a micro-architectural simulator supporting commercial Instruction Set Architecture (ISA). We evaluate this prototype and find that adaptive multi-cores can perform better than traditional out-of-order cores by adapting to workload needs. One of the key challenges in exploiting adaptive multi-cores is to make effective use of the performance flexibility. We propose a software-based modeling technique for predicting the performance on any target coalition given the execution profile of the application on the current coalition. Experimental evaluations suggest that this model is accurate and can be used in making well-informed decisions during runtime in a performance-aware scheduler.
Keywords: Adaptive multi-core, Full system simulator, Performance modeling, Scheduling, Regression, Heterogeneous multi-core
List of Figures

1.1 Evolution of single-core performance [19] . . . . . . . . . . . 2
1.2 Simplified multi-core processor block diagram . . . . . . . . . . . 2
1.3 Speedup of multi-cores according to Amdahl’s law [46] . . . 3
1.4 Simplified block diagram of Samsung Exynos Octa 5 SoC . . . 4
1.5 Simplified block diagram of an adaptive multi-Core system . . 5

3.1 Bahurupi architecture: additional resources [38] . . . . . . . 16
3.2 Schema of Sentinel Instruction . . . . . . . . . . . . . . . . . . . 18
3.3 Bahurupi Execution Example: The figure on the left shows the CFG of the application trying to run in Bahurupi fashion. The figure on the right shows the distributed execution of the application on a 2-core coalition [38] . . . . . . . . . . . 19

4.1 Bahurupi multi-core system design . . . . . . . . . . . . . . . 23
4.2 ARM co-processor workflow . . . . . . . . . . . . . . . . . . . . 25
4.3 ARM Co-processor instruction format . . . . . . . . . . . . . . 25
4.4 Percentage of basic blocks within certain threshold of Live-in 26
4.5 Percentage of basic blocks within certain threshold of Live-out 26
4.6 Basic Block Information Table Design . . . . . . . . . . . . . . 27
4.7 Basic Block Information Table System Design and Workflow 29
4.8 Basic Block sequence showing LAS conflict . . . . . . . . . . . 30
4.9 Handling library space function calls . . . . . . . . . . . . . . . 31
4.10 Shared branch predictor work flow. (a) The instruction address of the next basic block is sent in parallel to the branch predictor unit. (b) The predicted address is updated in GPC before releasing the GPC. (c) The outcome of the branch instruction is updated in the Branch Prediction Unit once the result is ready.

4.11 Bahurupi speedup normalized to 2-way out-of-order core

4.12 Bahurupi speedup normalized to 2-way out-of-order core

4.13 L1 I miss rate of a coalesced 2-core Bahurupi multi-core system with respect to 4-way out-of-order processor

4.14 L1 D miss rate of a coalesced 2-core Bahurupi multi-core system with respect to 4-way out-of-order processor

4.15 Branch mis-prediction rate of a coalesced 2-core Bahurupi multi-core system with respect to 4-way out-of-order processor

4.16 Number of Live In/Live Out per Basic Block executed

4.17 Register dependency within a Basic Block with starting instruction address 0x88b4

5.1 Illustrative example showing the schedule on homogeneous multi-cores

5.2 Illustrative example showing the schedule on adaptive multi-cores based on First Come First Serve policy

5.3 Illustrative example showing the schedule on adaptive multi-cores based on a different scheduling policy

5.4 Periodic CPI measurement of mcf benchmark

5.5 Phase behavior of benchmarks

5.6 Performance estimation work flow

5.7 Fitting CPI for a sub-set of training benchmarks

5.8 Estimated CPI using CPI model for a sub-set of test benchmarks
5.9 Predicted CPI from 2-core Coalition to 1-core Coalition and
vice-versa for a sub-set of training benchmarks . . . . . . . . 61
5.10 Predicted CPI from 2-core Coalition to 1-core Coalition and
vice-versa for a sub-set of test benchmarks . . . . . . . . . . 62
5.11 Speedup comparison on different systems using online schedul-
ing with respect to homogeneous S1 configuration . . . . . . 68
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Taxonomy of modern full system simulators</td>
<td>11</td>
</tr>
<tr>
<td>4.1</td>
<td>Parameters for baseline cores</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>Benchmarks used in our simulation</td>
<td>36</td>
</tr>
<tr>
<td>5.1</td>
<td>Cache size during different coalitions</td>
<td>57</td>
</tr>
<tr>
<td>5.2</td>
<td>Training and Test Benchmarks</td>
<td>57</td>
</tr>
<tr>
<td>5.3</td>
<td>Benchmarks used in Online Scheduling</td>
<td>67</td>
</tr>
</tbody>
</table>
List of Algorithms

1. Online scheduler for ideal Adaptive multi-core . . . . . . . . 65
2. Algorithm for task migration for ideal Adaptive multi-core . 66
Chapter 1

Introduction

The last few decades have seen unprecedented advances in computing systems. They have come a long way from the large and bulky mechanical computers that often occupied a whole room to modern wearable computing devices in terms of performance, battery life and space requirements.

The initial research in computing systems dealt with increasing the Instruction Level Parallelism in a single-threaded application (ILP). This made computer architects to come up with complex single-core processors that can simultaneously execute as many instructions as possible. Figure 1.1 shows the evolution of performance in single-core processors over time. The very last complex single-core processor was developed between 2002 – 2006, where micro-architectures with out-of-order execution, dynamic speculation and Simultaneous multithreading (SMT) capabilities were designed in order to meet the requirements of the contemporary applications. However, with increasing complexity of single-core processors, there was a corresponding increase in power and energy consumptions. Due to limits in power and thermal contraints, architects moved towards multi-cores. One of the primary phenomena fueling this rise has been the Moore’s Law [42] that predicts a hundred percent increase in the number of transistors per unit area every eighteen months. Also supporting this theory was Robert
Dennard [12] who proposed that the power requirements would reduce alongside the reduction in chip size and area. Thus the performance per watt would approximately follow the trend as observed by Moore. These two observations have enabled architects to propose designs comprising of many simple (homogeneous/symmetric) and power-efficient cores on the same die as shown in Figure 1.2.

Figure 1.1: Evolution of single-core performance [19]

Figure 1.2: Simplified multi-core processor block diagram

**Homogeneous multi-core:** The emergence of homogeneous multi-core systems have made it possible for application developers to parallelize their applications. Complete parallelization of an application might not
always be possible due to its sequential fraction. As per Amdahl’s law [2], the speed-up of an application is bottlenecked by the sequential portion in the application. Even when with 99% parallel code, the maximum speedup is only 100 as shown in Figure 1.3.

Hence, in order to accelerate the sequential and parallel portion of an application, a system should be able to extract high ILP and Thread Level Parallelism (TLP) respectively.

![Figure 1.3: Speedup of multi-cores according to Amdahl’s law](image)

**Static Heterogeneous multi-core:** In order to benefit emerging workloads consisting of different requirements, a system should consist of heterogeneous processing units with different power-performance characteristics. Heterogeneity in a system can be achieved by incorporating functionally different devices such as CPU, GPU, DSP, etc. (functional heterogeneity) or similar devices with different processing capabilities such as ARM Cortex A15 CPU, ARM Cortex A7 CPU, etc. (performance heterogeneity) on the same die.

Heterogeneous devices with performance heterogeneity have been widely used in modern commercial platforms such as the ARM big.LITTLE [10].
System on Chip (SoC) consisting of power-efficient in-order cores and high performance out-of-order cores. Wearable Processing Unit (WPU) from Ineda is yet another example of performance heterogeneous device that consists of cores with varying power-performance characteristics. Specifically, Exynos 5 Octa SoC used in Samsung Galaxy S4 is built upon ARM big.LITTLE architecture integrating four power-efficient Cortex A7 cores and four high-performance Cortex A15 cores as shown in Figure 1.4. In such systems, the sequential portion of an application can run in the high-performance core while the parallel portion can be accelerated by launching multiple threads on the power-efficient cores.

![Cortex-A7 Quad](Cortex-A7 Quad) ![Cortex-A15 Quad](Cortex-A15 Quad) ![PowerVR SGX544 GPU](PowerVR SGX544 GPU)

Figure 1.4: Simplified block diagram of Samsung Exynos Octa 5 SoC

**Dynamic Heterogeneous multi-core:** The types of cores in static heterogeneous multi-core platforms currently available in the market are fixed. This makes these architectures inflexible. However, if the performance capabilities can be dynamically changed during run time, the architecture would be able to easily adapt to new workloads. This class of multi-core processors that can dynamically adapt according to workload demands are known as Adaptive Multi-core systems. Figure 1.5 shows one such system in which simple cores can be coalesced to provide better performance as per application demands. Applications requiring high ILP can be made to run on coalesced simple cores while those needing high TLP can
be launched as independent threads on multiple cores. Though adaptive multi-core architectures seem to be a better alternative when compared to static heterogeneous multi-core architectures, there has not been many works which build and evaluate such systems. Hence, it would be necessary to build a prototype of these systems on commercially used Instruction Set Architecture (ISA). This would enable us to verify and evaluate the efficiency of such architectures and identify potential drawbacks. As a first step, we would like to evaluate the performance of an adaptive multi-core system accurately.

![Simplified block diagram of an adaptive multi-core system](image)

Figure 1.5: Simplified block diagram of an adaptive multi-Core system

### 1.1 Thesis contribution

In this thesis, our contributions revolve around evaluating the effectiveness of an adaptive multi-core architecture and building a performance model which can be used to determine the degree of coalition during runtime. To the best of our knowledge, this is the first work that builds an adaptive multi-core system based on a commercially used ISA on a system-level simulator. Specifically, we make the following contributions:

**Full System Simulation of Adaptive Multi-Core:** We build a full
system simulator for simple 2-way super-scalar out-of-order cores which can dynamically be coalesced to improve ILP as per application needs. We choose a current state of the art adaptive multi-core architecture called Bahurupi \cite{38} as our implementation choice since it provides a simple yet elegant core coalition logic through hardware software cooperation. Our system is built on gem5 \cite{6}, an ARM based simulator which mimics real hardware with high accuracy. We identify the overhead involved and hardware components that need to be shared across cores in coalition. We also propose optimizations that can be applied to improve the performance of such an architecture. We also evaluate how emerging applications can make use of an adaptive multi-core architecture to perform better than traditional out-of-order cores.

**Performance model for Adaptive Multi-Core:** We propose a performance model that can determine the number of cores required to be coalesced based on application demands. We also show how our performance model can be used to determine coalition decisions during run-time.

### 1.2 Thesis outline

This thesis is organized as follows. Chapter 2 gives an overview of existing adaptive multi-core architectures, full system simulators and performance models. Chapter 3 describes one such architecture which we would be implementing, Chapter 4 describes how an adaptive multi-core full system simulator has been built on top of a state-of-the-art simulator. Chapter 5 describes a performance model that can determine the number of cores that need to be coalesced during run-time for an ideal adaptive multi-core architecture. Chapter 6 concludes the report with future research directions.
Chapter 2

Related Work

In this chapter, we briefly discuss some of the related works on static and dynamic heterogeneous multi-core architectures. We also explain the reason behind choosing the dynamic heterogeneous architecture for implementation. A literature review on full systems simulators and performance modeling for multi-cores is presented in the last part of this chapter.

2.1 Static heterogeneous multi-core

The first static heterogeneous architecture was proposed by Rakesh Kumar et al. in [27] and [29]. A heterogeneous multi-core system was built by connecting cores with different size and performance parameters. These works showed that a heterogeneous multi-core system can achieve low power by dynamically migrating cores during runtime. [27] also gave some insights on energy benefits by using a heterogeneous system. [43] used a heterogeneous system to accelerate sequential portion of a multi-threaded application. The current trend in static heterogeneous architectures is to include cores with different micro-architectures on the same die. Some of the commercially available static heterogeneous multi-core systems include ARM big.LITTLE [16], nVidia Kal-El [36] and Wearable Processing Unit(WPU) [44] from Ineda.
2.2 Dynamic heterogeneous / Adaptive multi-core

Considerable previous works have been done to propose an architecture that can dynamically adapt itself during runtime. Previously proposed adaptive multi-core architectures include TFlex [25], Voltron [51], Federation [45], Core Fusion [21] and Bahurupi [38]. These architectures, except for [38], either require heavy modification in the hardware or use completely different Instruction Set Architecture (ISA). Hence, they require aggressive optimizations while compiling and also require the software model to be completely re-engineered.

Core Fusion can fuse homogeneous out-of-order cores and presents a detailed description of required changes. It proposes a hardware containing reconfigurable and distributed front-end and a fused back-end. It uses a complex remote waking mechanism without additional register ports, wake-up buses, bypass-paths or instruction queue ports. This scheme uses complex hardware and hence is able to extract high performance.

TFlex is based on TRIPS multi-core processor and uses the Explicit Data Graph Execution (EDGE) ISA. It minimizes modifications in hardware but uses completely different ISA. This scheme also employs a sophisticated next-block-predictor to extract ILP. This scheme uses a completely new ISA requesting the software model to be re-engineered. It also needs aggressive compiler support.

Federation combines simple scalar in-order cores to behave like an out-of-order core. This is achieved by adding additional logic to the internal pipeline stages. This proposal has limited capabilities since the in-order cores require complex mechanisms to mimic out-of-order execution.

Voltron consists of homogeneous multi-cores. In this architecture, the multi-cores are dynamically adapted for different kinds of workloads. This
processor behaves as a Very Long Instruction Word (VLIW) processor in coupled mode. Voltron uses a complex compiler to extract parallelism from a serial program by partitioning it into small threads, scheduling instructions to cores and directing communication between cores. However, obtaining parallel instructions from the serial code due to VLIW nature of the processor can be tedious.

Bahurupi uses a simple yet elegant method to coalesce simple homogeneous out-of-order cores during runtime. It uses a distributed execution model to extract ILP across basic blocks through hardware-software cooperation. It uses a special instruction called sentinel instruction to specify the dependencies across basic blocks. Bahurupi also proposes minimal changes in hardware and pipeline stages. These features have made us to choose Bahurupi as our implementation choice.

2.3 Full system simulators

Prototyping a system on real hardware is expensive and hence computer architects have heavily relied upon simulators to evaluate their designs. SimpleScalar [11] was one of the most successful simulators due its accuracy, performance and usability.

A full system simulation always had the problem of long execution time due to limited hardware performance. SimOS [41] was the first successful full-system simulator to provide fast simulation by using techniques like dynamic translation of binary and direct-execution.

The increase in hardware performance over the years were exploited in simulators by implementing an accurate and detailed full system model. Another full system simulator Flexus [49] used rigorous statistical sampling techniques for achieving high accurate and low execution time. In this simulator, specific regions of an application can be run, thereby reducing
the overall simulation time.

GEMS \[34\] simulator was designed for modeling memory systems accurately. It uses Ruby model for providing a high-level of flexibility in configuring and modifying caches, interconnections and coherence protocols. A custom coherence protocol in GEMS can be implemented using a special language called SLICC.

M5 \[7\] was initially designed to simulate networked systems. It provides a highly configurable simulation framework for multiple ISAs and diverse CPU models.

MARSS \[37\] and PTLsim \[50\] integrated closely with virtual machines to provide a fast and accurate simulation of x86 based machines.

OVPsim \[20\] and Simics \[33\] are fast functional simulators that can run numerous operating systems. It achieves this by modeling the necessary components. In order to obtain higher accuracy, extra simulation technology needs to be added to the existing simulator.

Since numerous simulators are currently available, we need to take different parameters like ISA support, flexibility, accuracy, licensing and project support into consideration before deciding the simulator. Table 2.1 summarizes a taxonomy of different simulators available today on the parameters stated above.

For building and evaluating an adaptive multi-core system, we need a micro-architectural simulator that is accurate, provides infrastructure to assemble hardware components, code base for modifying the pipeline stages and support commercial ISAs. We use gem5 \[6\], a simulator obtained by integrating M5 and GEMS, as our implementation choice due to the following reasons:

- it supports commercial ISAs like x86 and ARM
- it provides a code base for modifying pipeline stages
- it can boot several operating systems like Android and Ubuntu
<table>
<thead>
<tr>
<th>Simulator</th>
<th>Maintained</th>
<th>ISA(s) Supported</th>
<th>License</th>
<th>Accuracy</th>
<th>Guest OS(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexus</td>
<td>No</td>
<td>SPARC, x86 (Requires Simics)</td>
<td>BSD</td>
<td>Cycle-Accurate</td>
<td>Linux, Solaris</td>
</tr>
<tr>
<td>gem5</td>
<td>Yes</td>
<td>Alpha, ARM, MIPS, POWER, SPARC, x86</td>
<td>BSD</td>
<td>Cycle-Accurate</td>
<td>Android, FreeBSD, Linux, Solaris</td>
</tr>
<tr>
<td>GEMS</td>
<td>No</td>
<td>SPARC (Requires Simics)</td>
<td>GPL</td>
<td>Timing</td>
<td>Solaris</td>
</tr>
<tr>
<td>MARSS</td>
<td>Yes</td>
<td>x86 (Requires QEMU)</td>
<td>GPL</td>
<td>Cycle-Accurate</td>
<td>Linux</td>
</tr>
<tr>
<td>OVPsim</td>
<td>Yes</td>
<td>ARM, MIPS, x86</td>
<td>Dual</td>
<td>Functional</td>
<td>Android, Linux, and others</td>
</tr>
<tr>
<td>PTLsim</td>
<td>No</td>
<td>x86 (Requires Xen and KVM/QEMU)</td>
<td>GPL</td>
<td>Cycle-Accurate</td>
<td>Linux</td>
</tr>
<tr>
<td>Simics</td>
<td>Yes</td>
<td>Alpha, ARM, M68k, MIPS, POWER, SPARC, x86</td>
<td>Closed</td>
<td>Functional</td>
<td>FreeBSD, Linux, NetBSD, Solaris, Windows, and others</td>
</tr>
</tbody>
</table>

Table 2.1: Taxonomy of modern full system simulators [40]

- it is currently maintained, supported and widely used in academia
- it provides an infrastructure to configuring different hardware components like caches

### 2.4 Performance modeling

Analytical performance models for different processor types have been proposed earlier [10] [14] [24]. The two main approaches that are generally used in the previous works for building a performance model, can be classified as mechanistic modeling and empirical modeling.

The mechanistic models for performance are built using insights of the target processor architecture. In [23] [24], a simple interval based mechanistic model was proposed for out-of-order processors with a sustained background performance level punctuated by transient miss events. The models presented in the above two works were further improved in [15] by weighing the dispatch width in detail. The mechanistic performance model presented in [10] for in-order processors proposed penalty models for non-unit instruction execution latencies, inter-instruction dependen-
cies, cache/TLB misses and branch mis-predictions. These models were then used to compute the actual performance model.

Empirical models treat the performance model as a black box and deduce them using statistical / regression techniques. [22] employs a simple non-linear regression technique to obtain performance model based on miss events values from a set of training benchmarks while [30] uses spline-based regression technique for inferring the performance and power model across all core configurations.

[14] uses a hybrid mechanistic-empirical model for computing the Cycles Per Instruction (CPI) stack using real hardware counter values with few simplistic assumptions. [39] builds an accurate power-performance model using a compile-time approach for obtaining structural hazards and inter-instruction dependency, and a hybrid mechanistic-empirical model for modeling mis events. We adapt [39] to efficiently model performance for every coalition configuration.

### 2.5 Online Scheduling

Scheduling applications on a flexible multi-core system is different when compared to a classic job-scheduling problem. This is because decisions on when simple cores need to be combined to form a virtual powerful core need to be made apart from allocating tasks efficiently.

The overall objective any scheduler is to maximize system performance by making effective use of available resources. Specifically, in flexible heterogeneous multi-core systems, the scheduling decisions should be based on the heterogeneity of a system, the different workloads and the varying requirements of tasks during runtime. Moreover, the overhead involved in taking decisions should be minimal.

One of the early methods for scheduling multi-threaded tasks on static
heterogeneous multi-core systems [3] uses a dynamic policy for allocating threads. This dynamic policy is shown to be able to extract different levels of thread level parallelism when compared to traditional homogeneous multi-cores.

[17] proposes algorithms for scheduling sequential applications on static and dynamic scheduling environments. However, the proposed algorithms are based on TFlex adaptive architecture [25]. Moreover, the complexity of the proposed algorithm is \( O(nm^2) \), where \( n \) is the number of applications and \( m \) is the number of cores in the system.

[35] proposes a scheduling algorithm on a realistic adaptive multi-core system. However, it assumes that the execution time of benchmarks are known during runtime. Moreover, the online scheduling algorithm suggested in this work assumes that the workloads have uniform requirements and moldable in nature, i.e., once they are scheduled, they cannot be preempted.

In this thesis, the applications are modeled as preemptive *malleable* tasks. Malleable tasks are those that may be simultaneously processed by a group of cores. The processor speedup of malleable tasks are dependent on the number of cores allocated to the task. These tasks are allowed to preempt and change the number of cores it is executing on. Other works using malleable tasks include [5] for simulating molecular dynamics, [13] for Cholesky factorization, [8] for operational oceanography and [9] for berth and quay allocation.

To the best of our knowledge, ours is the first work that does not rely on profile information on execution time and proposes an online scheduling algorithm for malleable workloads on dynamic heterogeneous multi-core systems.
Chapter 3

Bahurupi Architecture

An application’s performance can be improved by exploiting ILP, TLP and Data Level Parallelism (DLP). The sequential part of an application can be accelerated by executing on a powerful core while the parallel portion of the application can be accelerated by launching multiple threads on different cores. An adaptive multi-core system can dynamically adapt and exploit both ILP and TLP. The TLP in an application can be exploited by launching multiple threads on simple cores. However, the ILP increase is achieved by building a powerful virtual core by coalescing simple cores. The logic used to coalesce simple cores to form a complex powerful core varies from one adaptive architecture to another. In this chapter, we briefly describe how Bahurupi multi-core architecture builds a complex powerful core out of simple cores.

A Bahurupi multi-core architecture is fabricated as a shared memory clustered architecture. It coalesces simple cores using a simple yet elegant hardware-software cooperative design with minimal change in hardware. It uses a distributed execution model across simple cores to obtain a performance equivalent to a complex core. The unit of work performed by a core is at a Basic Block granularity level consisting of a group of instructions with single entry and exit. A Bahurupi multi-core system tries to
distribute basic blocks on different cores in coalition. These basic blocks are then executed simultaneously to attain higher performance.

A schematic of Bahurupi architecture is shown in Figure 3.1. As seen in this figure, the system consists of two clusters each consisting of four simple 2-way superscalar out-of-order processors. Each cluster is limited to four cores since ILP is fairly limited beyond 8-way superscalarity. Two or more cores in the cluster are coalesced to form a virtual 4/6/8-way core. Each core in the system has a private L1 cache. The L2 cache is shared across all clusters. It is important to note that each cluster can have at most one coalition at a time. A high ILP application can run on a coalition of four cores, medium ILP application can run on coalition of two cores while a low ILP application can run on a single core thereby making complete use of performance flexibility.

![Figure 3.1: Bahurupi architecture: additional resources](image)

When cores are coalesced, they are connected to a binding coalition logic, comprising of a coalition bus, a Ticket/Serving Unit, a Global Program Counter (GPC) and a Global Register File and renaming logic (GRF Unit). The GPC is responsible for providing the address of the next in-
struction to be fetched. The GRF is used for supplying correct register values across cores in coalition. The coalition bus is used to pass the liveness information / register values to the GRF Unit and to other cores in coalition. The Ticket/System mechanism ensures that the instructions are fetched and committed in program order. Both these counters start with value zero. Whenever an instruction address from the GPC is obtained, it is given a ticket number and the ticket value is incremented. All instructions belonging to this basic block are assigned with this value. An instruction is committed if its ticket value matches the currently serving value and is the head instruction in the ROB of that core. The serving value is incremented when all instruction in the basic block are committed.

3.1 Sentinel Instruction

Bahurupi architecture needs to resolve memory and register dependencies for ensuring correctness due to distributed execution of basic blocks. Moreover, a core also needs to know the boundary of a basic block to determine when to start fetching instructions from the next basic block. Additionally, a core also needs to know whether the basic block ends with a branch instruction in order to find the next basic block to be executed.

The register dependencies across basic blocks are resolved through static liveness analysis of an application. A register which is used before it is modified inside a basic block is called as a live-in register. The register value which is updated within a basic block and is used inside another basic block is known as a live-out register. Similarly, the number of instructions within a basic block and whether the basic block ends with a branch instruction are obtained through static analysis of the application binary.

A special instruction called sentinel instruction is added to every basic block in Bahurupi architecture to supply the liveness information and
boundaries of basic blocks. This instruction supplies the list of registers values used and modified by a basic block. It also provides the number of instructions in a basic block and whether this basic block ends with a branch instruction. For every basic block in an application, the sentinel instruction corresponding to the basic block is first fetched and decoded, (to identify the live-in and live-out registers) before executing the actual instructions within the basic block.

Figure 3.2 depicts the schema of a sentinel instruction used in Bahurupi architecture. The first field in this instruction is used to specify the OPCODE of the sentinel instruction. BB_SIZE specifies the size of the basic block. This information is required to determine the start address of the next basic block. BB_TYPE is used to state whether the basic block ends with a branch instruction. LO_0 through LO_2 specifies the list of live-out registers while LI_0 through LI_2 specifies the live-in registers used in this basic block.

![Figure 3.2: Schema of Sentinel Instruction](image)

### 3.2 Bahurupi Execution Example

Figure 3.3 gives an example of Bahurupi execution on a 2-core coalition. The figure on the left denotes the Control Flow Graph (CFG) of an application. The two cores in coalition try to obtain a lock on GPC. Core 0 succeeds, locks the GPC and obtains the address of the sentinel instruction corresponding to basic block B0. It also obtains the current ticket number with value 0. As seen in this figure, Core 0 tries to fetch, decode and rename global registers (SF) according to the fetched sentinel instruction. Meanwhile Core 1 needs to wait since the GPC is still locked.
Once SF stage of B0 is over, Core 0 releases the lock on GPC while instructions in B0 go through the regular pipeline stages in the core (i.e., fetch, decode, rename, execute and commit stages - RF). Core 1 obtains the lock on GPC and gets a ticket value of 1 upon release by Core 0. Core 1 proceeds to fetch and execute B1 in a similar manner.

It is important to note that a core can obtain a lock on GPC only after all instructions in currently fetched basic block are renamed. This prevents Core 0 from obtaining a lock on the GPC as soon as Core 1 releases it. This condition is imposed in order to prevent global renaming from interfering with local renaming.

From this figure, B1 completes execution before B0. However, Core 1 cannot commit instructions in B1 since all instructions in B0 have not been committed. After B0 is completely committed, B1 starts to commit. A similar scenario happens to basic blocks B3 and B4 on Core 0 and Core 1 respectively. In this application, Bahurupi based execution can outperform a 4-way out-of-order processor since it fetches B3 and B4 in parallel. However, a traditional 4-way core has to wait until all the instructions in B3 are executed.
3.3 Summary

In summary, Bahurupi execution model performs an in-order fetch, out-of-order execution and in-order commit. The in-order fetch and commit is required for ensuring correctness of the program though it can introduce idle cycles. However, it has the potential to outperform a 4-way core by exploiting loop level parallelism.
In this chapter, the implementation of a prototype adaptive multi-core architecture using gem5, a state-of-the-art full system simulator is described.

In an adaptive multi-core system, Thread Level Parallelism (TLP) is generally extracted by launching multiple threads of an application on simple cores. Hence, changes in the existing system are not required for achieving high TLP. On the other hand, in order to exploit Instruction Level Parallelism (ILP) more than that can be achieved on a single core, the adaptive multi-core system needs to dynamically build a powerful virtual core out of simple cores [28]. The logic that is used to create a powerful virtual core out of simple cores varies from one adaptive architecture to another.

Bahurupi [38] multi-core architecture was chose for this implementation, since it uses a simple yet elegant approach for adapting to application needs with minimal effort in hardware.

An earlier evaluation of Bahurupi architecture in existing work [38], was performed on SimpleScalar, a cycle accurate simulator based on Portable Instruction Set Architecture (PISA) Instruction Set Architecture (ISA).
However, this evaluation is not complete since PISA ISA is not used on commercially platforms. Moreover, SimpleScalar being a cycle-accurate simulator, does not model full system accurately. Therefore, we implement the Bahurupi adaptive multi-core architecture on the gem5 micro-architectural full system simulator, in order to extensively assess its effectiveness. This implementation is based on the ARM ISA, since it is widely used in the embedded domain.

gem5 (an integration of GEMS and M5 simulators) is a configurable, modular and event-driven computer system simulator with multiple CPU/ISA support and a detailed and flexible memory system. This simulator provides an infrastructure to assemble hardware components in the system (using interconnects) and a code base for modifying the pipeline stages. Moreover, this simulator can also boot operating systems like Android and Ubuntu.

The inter-connections in the gem5 simulator are modeled using python, while the functionality for each of the hardware components and pipeline stages are implemented using C++. In this simulator, interactions between python and C++ codes are made possible by converting python functions to C++ functions using swig parser. Features like component level reconfiguration and modifiable code base for the pipeline stages, make gem5 an ideal choice for implementing an adaptive multi-core system prototype.

The rest of the chapter is organized as follows: Section 4.1 describes the full system implementation of Bahurupi multi-core architecture and Section 4.2 compares the performance of running applications on the virtual powerful core (formed using Bahurupi architecture) with traditional out-of-order cores.
4.1 Bahurupi architecture implementation using gem5 Full System Simulator

The Bahurupi multi-core architecture system implemented in this work consists of two 2-way out-of-order cores as the base cores. These cores can be coalesced during run time to form a powerful virtual core. Firstly, additional hardware components required in Bahurupi architecture were identified. These components, which included Global Program Counter (GPC) Unit, System Ticket Serving Unit, Global Register File Unit (GRF Unit) as stated in Chapter 3, were added and connected to the system using python. The layout of the Bahurupi multi-core system implemented using gem5, has been shown in Figure 4.1.

![Bahurupi multi-core system design](image)

Figure 4.1: Bahurupi multi-core system design

Next, the functionality for each of the new components and pipeline stages were added by modifying the base C++ code. The following subsections describe the process of incorporating each of the features of Bahurupi architecture into our system.
4.1.1 Sentinel Instruction

The key aspect of Bahurupi architecture is the sentinel instruction that supplies information regarding the live-in/live-out registers, basic block size and whether the current basic block ends with a branch instruction. There are two key challenges in adding a new instruction to an existing commercial ISA. The first challenge is whether the instruction size in a commercial ISA is sufficient for encoding basic block information. In case of currently used ARMv7 ISA, an instruction is 32 bits long. This size is much lesser than the currently proposed Bahurupi architecture, which assumes that the size of a sentinel instruction as 64 bits. In the newer ARMv8 ISA, the instructions are 64 bit long. The second challenge is whether the commercial ISAs provide extensions for new instructions. In case of ARM ISA, new instructions can be added to existing ISA only if they can be encoded as a co-processor instruction.

A co-processor is an external chip connected to ARM data and control buses. Whenever an ARM processor is unable to execute an instruction, it sends a broadcast on the co-processor bus. If a co-processor identifies that this instruction belongs to it, it will reply back to this request. The ARM processor sends this instruction through the co-processor bus when the co-processor is ready to accept a new instruction. A schematic of this approach used in ARM as shown in Figure 4.2.

In an ARM co-processor instruction, certain bits are reserved. The format of an ARM co-processor instruction is shown in Figure 4.3. From this format, we find that bits (31-28) denote conditional field, (27-24) is always set as 1110, (11-8) is used to specify co-processor number and (4) is always set as 1. Hence, there are only 19 remaining bits.

Therefore, it is not possible to encode a sentinel instruction as a co-processor instruction due to the following reasons:

- Figures 4.4 and 4.5 show that we have approximately 5 live-in and 4
ARM Coprocessors
ARM Core
Coprocessors
Coprocessor
FPU
COPROCESSOR BUS

- External chip connected to ARM data and control buses
- Can be used for implementing sentinel instructions

Figure 4.2: ARM co-processor workflow

Figure 4.3: ARM Co-processor instruction format

live-out registers in most of the basic blocks. The 19 bits left in a co-
processor instruction is not sufficient to hold the live-in and live-out
registers.

- The overhead introduced due to handshake between ARM core and
cooprocessor and communication of values between ARM core and
cooprocessor due to sentinel instruction is large

The above reasons necessitate us to come up with an alternate scheme for
supplying basic block information to cores in coalition. We still make use
of ARM co-processor instructions to direct the core for starting and ending
coalitions. This is achieved by using one unique co-processor number for starting coalition and another unique co-processor number for ending coalition. However, a small fast access memory called Basic Block Information Table (BBIT) is proposed for storing the basic block information.

Figure 4.4: Percentage of basic blocks within certain threshold of Live-in

Figure 4.5: Percentage of basic blocks within certain threshold of Live-out
Basic Block Information Table

In this thesis, a small fast access memory called Basic Block Information Table (BBIT) is used to store the basic block information. We work on the principle that a unique basic block identifier (BBID) would enable us in fast retrieval of basic block information from this table. Figure 4.6 shows the design of BBIT. From this figure, we observe that information like live-in/live-out, basic block size, whether the last instruction is a branch/library function call can be cached and indexed in the BBIT using lower bits of the basic block id. We use a new register called Global Basic Block ID (GBBID) for storing the BBID corresponding to the basic block starting from instruction address equal to Global Program Counter (GPC).

Encoding Basic Block information: The basic block information is similar to sentinel instruction and can be found using static analysis of the executable binary. After obtaining the execution binary of the program, a unique id is assigned to every basic block in the binary. The taken basic block id is computed using the taken instruction address if the basic block ends with a branch instruction. The non-taken id of an application basic block is \((id + 1)\) and hence it is not stored in the BBIT entry. The basic
block information per BBID is then added to the data section of the binary executable as a 20 bit BBID followed by the liveness information.

**Obtaining Basic Block information:** A master core is the one that initiates the first coalition. When the master core requests for starting a coalition, information corresponding to the first few basic blocks are fetched from the data section into the BBIT. Until the master core ends coalition, the liveness information is obtained (by accessing the BBIT) by the core which is successful in obtaining a lock on the GPC.

Figure 4.7 shows the system design and workflow for obtaining information from the BBIT. The Bahurupi adaptive multi-core system consists of two levels of memory to store the basic block information (BBIT and BBIT L2$). As seen in this figure, only one of the cores in coalition is successful in locking the GPC at a time (Marked as 1). Once the GPC is locked, its value is used to fetch the next instruction, while the lower bits of GBBID value is used to index into the BBIT and obtain the basic block information (Marked as 2). This basic block information obtained from the BBIT is then sent to the requested core for (i) renaming global and local registers and (ii) predicting the next instruction address (GPC) and the next BBID (GBBID) (Marked as 3). If the BBID entry is not found in the BBIT, the liveness information is searched and obtained from the lower levels of memory (Marked as 4) and sent to the requesting core (Marked as 5).

In order to reduce the overhead in obtaining the liveness information, we fetch basic block information in parallel with an instruction cache miss in the L1-I$ and L2$. Whenever, an instruction cache miss occurs, the basic block information corresponding to the next few basic blocks are pre-fetched from BBIT L2$ or main memory. Since different basic blocks generally belong to different cache lines, this pre-fetching ensures that the basic block information is available in most of the cases. The benchmarks
used in this thesis, never have unavailable information in the BBIT.

Figure 4.7: Basic Block Information Table System Design and Workflow

4.1.2 GRF Unit transactions

Whenever a new basic block is fetched, the live-in and live-out registers indices are obtained from the BBIT. The live-in register values are copied from global register file to local register file. In case of live-out registers, renaming happens in the global register file. When a following basic block dependent on the live-out register is fetched, it needs to wait until the global register file is ready. The global register values are broadcasted to all the cores waiting on it once it is ready.

4.1.3 Branch mis-prediction

The ticket values need to be updated correctly when a branch is mis-predicted. The global ticket value is broadcasted to the other cores in the coalition. All basic blocks having a ticket value greater than the broadcast value are squashed. In addition, the live-in and live-out register values are reverted correctly in the local and global register files.
4.1.4 Recovering from Memory Hazards

Consider two basic blocks as shown in Figure 4.8. B0 contains a store to address 0xabcd while B1 has a load from address 0xabcd. It is possible for this load instruction to execute before the store instruction has executed. This kind of violation across load and store instructions is called Load after Store violation (LAS). The load instruction causing LAS violation can appear at any position within the basic block. In such cases, all instructions starting from the load instruction need to be squashed. However, we need to revert ticket value, live-in/live-out registers in the global and local register file appropriately. Additionally, the entire basic block containing the load instruction can completely be squashed due to branch mis-prediction. Therefore, we wait until all the instructions before the load are committed and later start re-fetching instructions starting from load in case of a LAS violation.

![Figure 4.8: Basic Block sequence showing LAS conflict](image)

4.1.5 Library function calls

In all applications we can have calls to library functions. Examples include functions from math and string library in C programming. In such cases, our implementation needs to be able to execute correctly. We can either have a function call to library functions or return from a library function
to a function defined in an application. In both these cases, the local and global registers need to copied correctly. We use a conservative approach in each of these cases as follows:

**Application basic blocks initiating Library function**

We wait until all the instructions in the application basic blocks are committed before library function fetch happens. Once all instructions in the application basic block are committed, we copy the global register file values to the local register file.

**Library function returning to application basic blocks**

We wait until all the instructions in the library function are committed before application basic block instruction fetch happens. Once all instructions in the library function are committed, we copy the local register file values to the global register file.

A summary of the above approaches can be found in Figure 4.9.

It is important to note that it is possible to obtain the source code of library functions and hence the liveness information of library functions. However, this is beyond the scope of this thesis.

Figure 4.9: Handling library space function calls
4.1.6 Shared components

One of the other challenges in implementing Bahurupi execution model is identifying components that need to be shared across cores in coalition.

Core specific registers

ARM ISA allows predicate instructions. These instructions help in converting control dependency to data dependency. These instructions use special registers called Current Program Status Registers (CPSR) and Saved Program Status Registers (SPSR). Apart from this, ARM uses banked registers when instructions execute in the supervisor mode. The registers are unique to a core and can be used across different basic blocks. In Bahurupi execution model, every core in coalition needs to be aware of these register value changes. Hence, these registers are shared across core in coalition.

Branch Predictor

Since different basic blocks execute across different cores in coalition, it might not be possible to learn correctly from mis-predictions. Hence we share the branch predictor across coalition cores.

At a given point, only one core can use the branch predictor to predicted the next instruction address. This is because the branch prediction unit is only used by the core locking the GPC. As seen in Figure 4.10, the instruction address of the next basic block to be fetched is sent to the core that obtains a lock on the GPC and the shared branch prediction unit in parallel. Once the prediction result is ready, the GPC value is updated. The branch prediction unit is updated once the branch instruction completes execution. This update can happen very rarely by two cores in the same time. However, we can easily increase the number of write channels to the branch predictor to tackle this problem.

Therefore, our decision in sharing the branch prediction unit is accept-
able since there will be no contention in using it.

Figure 4.10: Shared branch predictor work flow. (a) The instruction address of the next basic block is sent in parallel to the branch predictor unit. (b) The predicted address is updated in GPC before releasing the GPC. (c) The outcome of the branch instruction is updated in the Branch Prediction Unit once the result is ready.

Cache

An application constantly reads and writes to memory locations residing in the data cache (L1-D$). In Bahurupi execution, it is highly likely that two basic blocks executing on two different coalition cores will read and modify values residing in the same memory location. Traditionally, a bus based coherence protocol has been employed to maintain consistent values across different data caches in the system. However, in Bahurupi architecture, this will introduce too many coherent misses since there is extensive data sharing across cores in coalition. Hence the L1-D$ needs to be shared across all coalition cores. Similarly, L1-I$ and L2$ also need to be shared across all cores in coalition.

Load Store Queue

In traditional out-of-order processors, instructions can be rearranged during execution. The register dependency across instructions that are rearranged during execution is preserved due to in-order fetch and renaming of instructions. However, out-of-order execution of memory instructions accessing the same location can lead to wrong results when compared to sequential execution of instructions on a traditional in-order processor. Hence in an
out-of-order processor, a check needs to be performed before executing a memory instruction. To make this check faster, a Load Store Queue (LSQ) containing the list of in-flight memory instructions is employed.

Out of the four possible combinations of memory instruction occurrences (load/store), rearranged execution of older Load after younger Load, older Store after younger Store and older Store after younger Load instruction do not cause any correctness problem due to in-order commit of instructions. Correctness problems arise only when an older store instruction executes later than a younger load instruction. In this case, the younger load instruction would have read a wrong value. Hence, before executing a store instruction, the LSQ is checked to find if any younger load instruction reading from the same memory location has completed execution. Whenever a younger load instruction reading from the same memory location has completed execution, all instruction starting from the younger load instruction are squashed.

In Bahurupi execution model, a younger load instruction can be scheduled on a different core when compared to the older store instruction. Hence, the check performed by a traditional out-of-order core is not sufficient. Broadcasting the store instruction address to all coalition cores is very expensive since it involves a huge overhead. Hence, we need to share the LSQ across coalesced cores to recover from out-of-order execution of older store after younger load instruction. Additionally, we will also be able to perform data forwarding by sharing the LSQ.

Every instruction fetched by a processor is assigned with a unique identifier called sequence number. Older instructions have a smaller sequence number when compared to a younger instructions. The memory violation check in LSQ requires ordering of instructions and hence sequence number of instruction is used. In order to enable sharing of LSQ, we need to ensure that the sequence number of instructions are maintained correctly. Hence,
we share the global sequence number counter across all coalition cores. Whenever, an application basic block is fetched, the global sequence number is incremented with the size of the current basic block. Thus, ordering of instruction by identifier is maintained by sharing the global sequence number counter and updating it appropriately.

In summary, all of the above mentioned components are shared across cores in coalition mode and private to the cores in normal mode.

4.2 Evaluation of Bahurupi architecture on gem5

We compare the performance of benchmarks on a 2-core coalition with a traditional 2-way out-of-order core and 4-way out-of-order processor. Ideally a coalesced 2-core should have a performance close to a 4-way processor.

4.2.1 Setup

Table 4.1 summarizes the configuration of the baseline 2-way and 4-way out-of-core used in our evaluation. Our Bahurupi multi-core system consists of two 2-way baseline cores. The branch predictor, L1-I$, L1-D$, L2$ and LSQ are shared across cores in coalition.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>2-way</th>
<th>4-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROB Size</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>IQ Size</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>LSQ Size</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>Int ALU</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Int MULT</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>FP ALU</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>FP MULT</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>L1-D$</td>
<td>32Kb</td>
<td>64Kb</td>
</tr>
<tr>
<td>L1-I$</td>
<td>32Kb</td>
<td>64Kb</td>
</tr>
<tr>
<td>Unified L2$</td>
<td>1Mb</td>
<td>2Mb</td>
</tr>
</tbody>
</table>

Table 4.1: Parameters for baseline cores
4.2.2 Benchmarks

We use single-threaded benchmarks from MiBench [18] and SPEC2006 [1] suites as shown in Table 4.2 in our evaluation since the parallel portion of a multi-thread benchmarks are not dependent on core coalition. These benchmarks are compiled using arm cross compiler and O2 optimization.

<table>
<thead>
<tr>
<th>Benchmark Type</th>
<th>Name</th>
<th>Suite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security</td>
<td>Twofish, RC6, AES</td>
<td>MiBench</td>
</tr>
<tr>
<td>Consumer</td>
<td>Matrix multiplication,</td>
<td>MiBench</td>
</tr>
<tr>
<td></td>
<td>Integer square root,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bitcount</td>
<td></td>
</tr>
<tr>
<td>General</td>
<td>Bubble sort, Geometric</td>
<td>Embedded</td>
</tr>
<tr>
<td></td>
<td>mean</td>
<td></td>
</tr>
<tr>
<td>Network</td>
<td>Susan smoothing</td>
<td>MiBench</td>
</tr>
<tr>
<td>Automotive</td>
<td>Dijkstra, Patricia</td>
<td>MiBench</td>
</tr>
<tr>
<td>Physics</td>
<td>Libquantum</td>
<td>SPEC2006</td>
</tr>
<tr>
<td>Game AI</td>
<td>sjeng</td>
<td>SPEC2006</td>
</tr>
</tbody>
</table>

Table 4.2: Benchmarks used in our simulation

4.2.3 Performance Analysis

The execution cycles of applications stated in Table 4.2 are obtained by running them on a coalesced 2-core Bahurupi multi-core system, a 2-way out-of-order processor and a 4-way out-of-order processor. Ideally, the performance improvement obtained from a coalesced 2-core Bahurupi multi-core system should be equivalent to a 4-way out-of-order processor. Figures 4.11 and 4.12 show the speedup of a 4-way out-of-order processor and 2-core Bahurupi multi-core with respect to a 2-way out-of-order processor.

From the speedup results, we can classify applications into three types. The first type of applications perform better than a 4-way processor. This is because of the fact that Bahurupi can extract loop level parallelism by looking far ahead in the future. Examples of such applications include RC6 encoder/decoder, susan smoothing, AES decoder, bitcount, twofish...
encoder/decoder and embedded geometric mean. Some applications like patricia, bubble sort and AES encoder perform similar to a 4-way processor (ideal case). The speedup of these applications are shown in Figure 4.11. The third type of applications as shown in Figure 4.12 perform slightly worse than a 4-way out-of-order processor. It is imperative to find out the reasons that lead to decrease in performance for these applications.

![Speedup Results](image)

Figure 4.11: Bahurupi speedup normalized to 2-way out-of-order core

Degradation in performance of applications arise due to miss events (like L1-I$, L1-D$ misses and branch mis-predictions) and application characteristics (register dependency and types of instruction in the application). Hence, in order to find out the cause of performance degradation, the miss-events and dependencies for each of the benchmarks from Figure 4.12 are compared between a 4-way out-of-order processor and a 2-core coalesced adaptive multi-core architecture. Ideally, miss-events and structural hazards (arising due to register dependency and instruction type within a basic block) occurring in a coalesced 2-core adaptive multi-core architecture needs to be identical a 4-way out-of-order processor.
The miss rates in the L1-I$ and L1-D$ are computed using Equations 4.1 and 4.2. The miss rate values in a coalesced 2-core coalition is normalized to a 4-way out-of-order processor since the range of miss rates across benchmarks is large. Figures 4.13 and 4.14 show the L1-I and L1-D miss rates occurring in a coalesced 2-core Bahurupi multi-core system normalized to a 4-way out-of-order processor. In these graphs, if the normalized miss_rate value is close to 1, then a coalesced 2-core coalition performs as good as a 4-way out-of-order processor. From these figures, we find that the cache miss rates in the L1 data cache are very similar on a 4-way out-of-order processor and a coalesced 2-core Bahurupi multi-core system. However, the L1-I$ miss rate in case of a 2-core coalition is slightly higher than a 4-way out-of-order processor because Bahurupi architecture tries to fetch instructions at a basic block level granularity. Instructions from different basic blocks can belong to different cache blocks, therefore leading to additional cache misses.
\[ l1i\_miss\_rate = \frac{l1i\_misses}{l1i\_accesses} \] (4.1)

\[ l1d\_miss\_rate = \frac{l1d\_misses}{l1d\_accesses} \] (4.2)

Figure 4.13: L1 I miss rate of a coalesced 2-core Bahurupi multi-core system with respect to 4-way out-of-order processor

Figure 4.14: L1 D miss rate of a coalesced 2-core Bahurupi multi-core system with respect to 4-way out-of-order processor

Figure 4.15 shows the normalized branch mis-prediction rate (computed using Equation 4.3) of a coalesced 2-core Bahurupi multi-core system with respect to a 4-way out-of-order processor. From this figure, we see that the branch mis-prediction rate is very similar between 2-core Coalition and 4-way out-of-order processor since all the normalized values are greater than 1. This is because the branch predictor is shared across all cores in
coalition.

\[
\text{branch\_miss\_rate} = \frac{\text{Number of branch mis-predictions}}{\text{Total number of branches}} \tag{4.3}
\]

In short, the miss events do not cause any performance degradation in case of Bahurupi multi-core architecture. Hence, applications are profiled to identify the program characteristics leading to performance loss. The number of live ins, live outs and instructions are computed for every executed basic block. Consider an application containing basic block \(X\) and \(Y\). \(X\) contains 4 live ins (\(X_{li}\)), 4 live outs (\(X_{lo}\)) and 4 instructions (\(X_{inst}\)). \(Y\) contains 5 live ins (\(Y_{li}\)), 5 live outs (\(Y_{lo}\)) and 5 instructions (\(Y_{inst}\)). Let \(X\) be executed 4 times (\(X_n\)) and \(Y\) be executed 5 times (\(Y_n\)). The average number of live ins per executed basic block is computed as:

\[
\text{Live\_In\_Per\_Exec\_BB} = \frac{X_{li} \times X_n + Y_{li} \times Y_n}{X_n + Y_n} = \frac{4 \times 4 + 5 \times 5}{4 + 5} = 4.56 \tag{4.4}
\]

Similarly, the number of live outs per executed basic block and the
number of instructions per executed basic block are computed. Figure 4.16 shows the number of live ins/outs and instructions per executed basic block. From these figures, we find that the number of live-ins/live-outs in these benchmarks are quite high when compared to the number of instructions present in it. Therefore, there exist heavy register dependency within and between basic blocks in case of applications shown in Figure 4.12. Moreover, the type of instructions in many of the basic blocks are very similar in nature. These two characteristics in the basic blocks make a 4-way out-of-order processor perform better than a coalesced 2-core Bahurupi multi-core system.

For example, Figure 4.17 shows one such basic block from the integer square root benchmark (isqrt). In this basic block, there is a basic block carried dependency since the last instruction branches to the first instruction of the basic block (if taken). Thus, registers 0, 1, 2 and 12 occur as both live in and live out. The first four instructions in this basic block are waiting on three live in registers 0, 1 and 2. The first few instructions produces results which are internally used in the basic block. The last few
instructions dependent on the internal registers produce the live out register values 0, 1 and 2. Hence, delay in executing the first few instruction leads to performance degradation.

The delay in first few instructions of the basic block happens because out the first five instructions, four instructions are of the same type (mov). Two of these instructions can be simultaneously executed on a 4-way processor as we have two integer ALUs. However, these instructions have to be executed one after another in Bahurupi since this particular basic block is scheduled only on one 2-way core with one integer ALU. This added with high register dependency described above leads to performance degradation of this benchmark.

![INSTRUCTION SRC DEST](image)

**Figure 4.17: Register dependency within a Basic Block with starting instruction address 0x88b4**

### 4.3 Discussions and Summary

In this chapter, we implemented a realistic adaptive multi-core system called Bahurupi on gem5, a micro-architectural simulator supporting commercial ARM ISA.

Our implementation on gem5 simulator is better than the previous eval-
uation on SimpleScalar because gem5 is a full-system simulator that in-
cludes support for interrupts, Translation Look aside Buffer (TLB), Page
tables, Memory Management Units (MMU), I/O devices, low-level buses
or interconnects and other functionally relevant system components. More-
over, the gem5 simulator provides a framework for modifying memory hi-
erarchy/system layout and also allows reconfiguration of cache protocols.
This flexibility helps in analyzing different components of our system. Ad-
ditionally, gem5 simulator can model a commercial ISA like ARM or x86.
Thus, our framework can be used for identifying potential bottlenecks and
bugs before fabrication.

This full-system simulation helped us in identifying hardware compo-
nents that need to be shared across cores in coalition, which were not con-
sidered in the previous evaluation. We also proposed a scheme for removing
sentinel instructions with minimal overhead by using an additional memory
as most commercial ISAs do not allow new instructions to be added to the
existing instruction set.

We evaluated benchmarks on the coalesced 2-core Bahurupi multi-core
system that was implemented by comparing their execution times with tra-
ditional 2-way and 4-way out-of-order processors. From this evaluation, we
concluded that a 2-core coalition in Bahurupi adaptive multi-core system,
performs equally or better than a traditional 4-way out-of-order core for
most of the applications.

The two potential scenarios in which Bahurupi execution model is
slightly worse than a traditional out-of-order processor are:

- **Inter-Instruction dependency and Instruction Mix:** Since
  Bahurupi execution model works on the granularity of basic blocks,
  high dependency across basic blocks and similar instruction types in
  the same basic blocks will face performance degradation when com-
  pared to a 4-way processor.
- **LAS violation across Basic Blocks**: LAS violations across basic blocks can appear in Bahurupi architecture resulting in re-fetching and executing of instructions starting from the load. However, in such cases, we found that the average improvement in performance when compared to 2-way processor was 35.8% in case of a 2-core coalition and 63.2% in case of a 4-way processor.

The performance degradation due to both of the above scenarios can be easily reduced by getting help from modern compilers. We can have an additional optimization pass that keeps the architecture in mind before generating the final executable.
Chapter 5

Performance modeling of
Adaptive Architectures

In an adaptive multi-core architecture, the number of cores that are used to run a task can be changed during runtime. The performance gain on increasing the number of cores for a task might not be substantial. Similarly, breaking a coalition when the performance loss is negligible will benefit other sequential / parallel tasks requiring additional cores. Hence, coming up with a scheduling algorithm to determine when and how many cores to coalesce is one of the key problems in an adaptive multi-core architecture.

Illustrative Example: We would like to show a visual illustration of the scheduling problem in an adaptive multi-core architecture. In our example, we have three sequential tasks tracking, texture (from Vision benchmark suite [48] and h264 (from SPEC [1]), which are launched simultaneously at time zero. We try to run this task set on a homogeneous multi-core system consisting of four 2-way out-of-order cores and on adaptive multi-core architecture consisting of four simple 2-way out-of-order cores.

In the case of a homogeneous multi-core system consisting of four 2-way out-of-order cores, each of these applications need to be scheduled on one core as shown in Figure 5.1. From this figure, we find that the makespan
(the time when all applications finish execution) is 2788 million cycles.

![Figure 5.1: Illustrative example showing the schedule on homogeneous multi-cores](image)

In case of adaptive multi-core architecture, we can have many different schedules. For instance, the scheduling policy can favor coalition for tasks that arrive first as shown in Figure 5.2. In our example, *tracking* is the first task that is launched followed by *h264* and *texture*. Hence, *tracking* is allowed to run on 2-core coalition followed by *h264*. In this schedule, *texture* does not get to run on a 2-core coalition since it completes execution before *h264* is complete.

![Figure 5.2: Illustrative example showing the schedule on adaptive multi-cores based on First Come First Serve policy](image)
Figure 5.3 shows another possible schedule. The makespan obtained in this schedule (1768 million cycles) is better than the makespan obtained from first come first serve based schedule (1952 million cycles).

From the illustrative example stated above, we observe that it is challenging to come up with a scheduling policy in the context of an adaptive multi-core architecture because of the following reasons:

- tasks can be allocated variable number of resources over time
- tasks can be migrated to other cores during its execution
- any realistic adaptive multi-core architecture imposes additional scheduling and allocation constrains

Previous work [35] suggests an online scheduler for an adaptive multi-core architecture. This work relies on the fact that the execution time on different coalition configurations is known during runtime of an application. Moreover, it assumes that the performance speedup is uniform throughout the life-time of a task.

We perform a case study on mcf application from SPEC2006 suite for showing varying requirements within the same application. For evaluation purposes, we obtain measurements from an ideal adaptive architecture, i.e. 2-way and 4-way out-of-order processors. In the remaining part of this
chapter, measurements from 2-way and 4-way processors are denoted as 1-core Coalition and 2-core Coalition respectively. The Cycles Per Instruction (CPI) value of this benchmark is measured for every 10 million committed instructions using gem5 simulator as shown in Figure 5.4. From this figure, we observe the following:

- CPI value for the first 50 million instructions and the remaining application are dramatically different
- Performance improvement between 1-core Coalition and 2-core Coalition is 41.16% for the first 50 million instructions and 26.14% for the remaining application

![Figure 5.4: Periodic CPI measurement of mcf benchmark](image)

Our scheduler can take the performance improvement at different phases into consideration for determining the degree of coalition. A phase in an application is defined as a set of continuous program points having similar CPI values. Figure 5.5 shows the phase behavior across other benchmarks from SPEC2000 and SPEC2006 suites.

Obtaining the execution time of applications at different program phases is challenging since the inputs to the benchmarks can change during runtime. Moreover, it is challenging to profile every new workload before they can execute on real hardware. Hence, it will be beneficial to use a predictive technique to obtain performance characteristics and hence take well-informed coalition decisions during runtime. For achieving this, we
need to build a performance model that can accurately predict the performance of any target coalition when running on an arbitrary coalition.

Performance modeling of applications on adaptive multi-core is similar to models used in heterogeneous multi-core systems. Earlier proposals in heterogeneous multi-core system \[4, 26, 32\] uses the fact that memory intensive applications can be mapped on the low performance core while compute intensive applications can be mapped on the high performance core. \[47\] has shown that this scheme can lead to sub-optimal mappings and hence it is necessary to obtain accurate performance values. The Performance Impact Estimation mechanism (PIE) in \[47\] proposes a dynamic
technique that collects profile information on currently running core to estimate the performance on the other cores available. \cite{39} works on some of the shortcomings of PIE and proposes a model for predicting the performance of a target core type when running on a host core type. We adapt some of the proposed techniques from this work to predict the performance of a 2-core coalition when running on 1-core coalition and vice-versa. The counter values used in our models are those provided by commercial ARM based processors like Cortex-A15 processor.

5.1 Performance Modeling

The main objective of performance modeling is to predict the behavior of an application on the target coalition given its profile information on the current coalition. The performance model in our technique revolves around building the CPI stack of a given application. It is challenging to build the CPI stack on any out-of-order processor since different pipeline and external events can overlap.

The total CPI of an application can be computed as the sum of CPI due to application behavior (\(CPI_{\text{steady}}\)) and CPI due to miss events (\(CPI_{\text{miss}}\)). \(CPI_{\text{steady}}\) comprises of cycles spent due to architectural events tightly coupled to pipeline like data dependency among instructions and structural hazards while \(CPI_{\text{miss}}\) comprises of cycles spent on external events like branch mis-predictions and cache miss events.

\[
CPI = CPI_{\text{steady}} + CPI_{\text{miss}}
\]  

The performance estimation framework consists of three steps as shown in Figure 5.6. Firstly, a set of training benchmarks are analyzed off-line using execution profiles obtained from hardware counters for building a CPI model for every coalition configuration. The CPI of an application on
coalition $P$ can be expressed as a function of steady CPI ($CPI_{steady}$), miss events ($miss_X$) and penalty ($latency_X$) involved in each miss event $X$.

$$CPI^P = f^P(CPI_{steady}^P, miss_X^P, latency_X^P)$$ (5.2)

Secondly, we build an inter-coalition branch/cache miss model for every pair of coalition configurations off-line using a set of training benchmarks. This model can estimate the miss events on a target coalition $P'$ given the number of such occurrences on host coalition $P$. This model is built by correlating corresponding events across different coalition configurations.

$$\overline{miss}_X^{P'} = g_{X \rightarrow P'}^P(miss_X^P)$$ (5.3)

Finally, we obtain the miss events during runtime on the current coalition $P$ by profiling hardware counters. We supply these values to the inter-core model to estimate the miss events on a target coalition $P'$. The CPI on the target coalition $P'$ is then obtained by plugging in the estimated miss events in the CPI model of $P'$.

$$CPI_{steady}^{P'} = f^{P'}(CPI_{steady}^{P'}, \overline{miss}_X^{P'}, latency_X^{P'})$$ (5.4)

### 5.1.1 $CPI_{steady}$ estimation

$CPI_{steady}$ of an application can be denoted as a function of structural hazards and inter-instruction dependencies.

$$CPI_{steady} = f(structural\_hazard, \ inter\_instruction\_dependency)$$ (5.5)

Counters computing inter-instruction dependency and structural hazards are not available on real hardware since increased book-keeping of such
counters introduce additional overhead. [14] suggests using $CPI_{steady}^P = 1/D$ where $D$ is the dispatch width of processor $P$. This simplistic assumption can hold true only on processors with completely balanced pipeline where structural hazard do not occur (number of functional units is equal to dispatch width). Moreover, this equation does not take dependency between instructions into consideration.

[39] uses the optimization pass in modern compilers like gcc that takes instruction scheduling into consideration based on the target processor architecture. During such optimizations, the compiler does instruction scheduling based on detailed target processor pipeline and hence it is aware of inter-instruction dependency and structural hazards. In this work, an approximate $CPI_{steady}$ is computed by finding the execution cycles of every basic block and predicting the frequency of every basic block using a set of heuristics provided by gcc. We can adapt such a method to compute $CPI_{steady}$.

Detailed hardware pipeline description used in the optimization pass is not implemented for all coalition configurations. Therefore, it will not be
possible for us to use this compile-time technique. We employ a simple linear regression model to obtain $CPI_{steady}$ using committed instructions and functional unit busy counters as shown below:

$$CPI_{steady} = \frac{\beta_0 \cdot N_{int} + \beta_1 \cdot N_{fp} + \beta_2 \cdot f_{u_busy}}{N}$$

(5.6)

where $N_{int}, N_{fp}$ denote the number of committed integer and floating point instructions, $f_{u_busy}$ denotes the number of times a functional unit was busy when requested and $\beta_0, \beta_1$ and $\beta_2$ denote unknown values fitted by regression.

### 5.1.2 CPI stack model

Our CPI stack model for a given coalition $P$ is based on mechanistic-empirical model suggested in [14]. We estimate the total number of cycles spent by an application on coalition $P$ using the following equation:

$$C = N \cdot CPI_{steady} + miss_{L1I}\cdot c_{L2} + miss_{L1D}\cdot c_{L2} + miss_{br}\cdot (c_{br} + c_{fe}) + d_{miss_{L2}}\cdot \frac{c_{mem}}{MLP}$$

(5.7)

Once the total number of cycles are estimated, the CPI value is computed by dividing the cycles by the total number of instructions $N$.

$$CPI = \frac{C}{N}$$

(5.8)

In equation (5.7), the first term $CPI_{steady}$ captures the inter-instruction dependency and structural hazards. It is computed using the model suggested in Equation (5.6). The idle cycles spent on waiting for registers and functional units is then computed by multiplying $CPI_{steady}$ with total number of committed instructions $N$.

$miss_{L1I}$ in the next term is the number of times an instruction was not
available in the first level of cache. The total cycles spent in waiting for instructions to be fetched from L2 to L1 cache is obtained by multiplying $miss_{L1I}$ with $c_{L2}$, the latency spent in accessing L2. Similarly, the idle cycles due to data misses in L1-D$ is computed by finding the product of the number of L1-D$ misses $miss_{L1D}$, and the access time for L2, $c_{L2}$.

Cycles spent due to branch mis-predictions constitute the next term. The penalty paid for a mis-prediction is dependent on $C_{fe}$ the front-end length of the pipeline and $c_{br}$, the time it takes to find the outcome of a branch instruction (branch resolution time). The branch resolution time varies since a branch instruction may or may not be dependent on another instruction in the pipeline.

The final term in the above equation denotes the number of cycles spent in fetching from the last level of cache. The number of accesses to the last level of data cache in dependent on the number of L2 misses. Since L2 is a unified cache, it can have misses due to instructions and data. We take only data misses from L2$ into consideration since the instruction misses are negligible when compared to data misses. Since out-of-order execution overlaps different data accesses to last level of cache, we used a Memory Level Parallelism correction term denoted as:

$$MLP = \beta_3 \cdot \left( \frac{d_{miss_{L2}}}{N} \right)^{\beta_4}$$

where $\beta_3$ and $\beta_4$ are unknowns obtained using regression.

More information on the intuition behind each of these terms can be found in [10].

5.1.3 Inter-coalition misses

An adaptive multi-core system consists of multiple simple cores of the same type. The simple cores are identical to each other and hence have similar branch predictors and caches.
A coalition of \( n \) cores will have \( n \) times the size of L1-I$ and L1-D$ of a simple core. For simplicity, we assume that an \( n \)-core coalition will use only one of the available L1-I$ and L1-D$ since they are shared across cores in coalition. We also assume that L2$ is a unified instruction and data cache.

For estimating the CPI of coalition \( P' \) given its execution profile on coalition \( P \), we need to predict \( \text{miss}_{L1I}, \text{miss}_{L1D}, \text{miss}_{br}, d\text{miss}_{L2} \) and \( f_{ubas} \) (arising from \( CPI_{steady} \)). The following section describes how each of these values are estimated for coalition \( P' \) given its execution profile on current coalition \( P \). The value of \( \beta \) used in each of the models are obtained from regression fitting.

**Inter-coalition cache and branch miss estimation**

L1-I$ and L1-D$ across different coalitions have the same associativity and size. Hence, we find that L1-I$ and L1-D$ misses can be estimated using a simple linear regression model. Similarly, branch mis-predictions can also be realized using a linear regression model. The following models are used to estimate \( \text{miss}_{L1I}, \text{miss}_{L1D} \) and \( \text{miss}_{br} \) on a target coalition \( P' \) given its execution profile on the current coalition \( P \). In the following equations, \( \beta_5, \beta_6, \beta_7, \beta_8, \beta_9, \beta_{10} \) are unknown values fit by regression.

\[
\text{miss}_{L1I}^{P'} = \beta_5 + \beta_6 \cdot \text{miss}_{L1I} \bigg|_P 
\]

\[
\text{miss}_{L1D}^{P'} = \beta_7 + \beta_8 \cdot \text{miss}_{L1D} \bigg|_P \tag{5.11}
\]

\[
\text{miss}_{br}^{P'} = \beta_9 + \beta_{10} \cdot \text{miss}_{br} \bigg|_P \tag{5.12}
\]

In case of \( \text{miss}_{L2} \), we employ a polynomial regression of order two due to high variability in this counter value across different data sets.
\[
\text{miss}_{L2}^{P} = \beta_{11} + \beta_{12} \cdot \text{miss}_{L2} + \beta_{13} \cdot \text{miss}_{L2}^2 \bigg|_P
\] (5.13)

**Inter-coalition functional units busy estimation**

The number of committed integer and floating point instructions used for computing \(\text{CPI}_{\text{steady}}\) do not require any inter-coalition modeling. Hence it is sufficient if we model \(f_{u_{\text{busy}}}\) events. The number of functional units increases linearly with an increase in coalition. The number of times the functional units are full depends on the types of currently executing instructions and the number of times the functional units were busy on the current coalition configuration. The following model computes \(f_{u_{\text{busy}}}\) events on coalition \(P'\) given the number of instructions belonging to each type (branch/load/store/integer/floating point) and the number of times the functional units were busy on configuration \(P\).

\[
f_{u_{\text{busy}}}^{P'} = \beta_{14} + \beta_{15} \cdot f_{u_{\text{busy}}} + \beta_{16} \cdot N_{\text{int}} + \beta_{17} \cdot N_{\text{fp}} + \beta_{18} \cdot N_{\text{br}} + \beta_{19} \cdot N_{\text{mem}} \bigg|_P
\] (5.14)

In the above equation, \(\beta_{14}, \beta_{15}, \beta_{16}, \beta_{17}, \beta_{18}\) and \(\beta_{19}\) are unknown values fit by regression.

Finally, the CPI value for coalition \(P'\) is obtained by plugging in the estimated miss events for \(P'\) into its CPI model.

### 5.2 Experimental Evaluation

In this section we evaluate the accuracy and robustness of our performance model.
5.2.1 Experimental Setup

Our ideal adaptive multi-core system is built using gem5 simulator. Our system consists of 2 simple 2-way out-of-order cores which can be coalesced during runtime. Ideally, a 2-core coalition behaves like a 4-way out-of-order core. Hence our system models a 2-core coalition as a 4-way core.

The I-Cache and D-Cache in the L1 level are separate while the L2$ is a unified cache shared across all cores in coalition. Table 5.1 summarizes the size of the caches in different scenarios.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>1-core Coalition</th>
<th>2-core Coalition</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1-D$</td>
<td>32Kb</td>
<td>64Kb</td>
</tr>
<tr>
<td>L1-I$</td>
<td>32Kb</td>
<td>64Kb</td>
</tr>
<tr>
<td>Unified L2$</td>
<td>2Mb</td>
<td>2Mb</td>
</tr>
</tbody>
</table>

Table 5.1: Cache size during different coalitions

5.2.2 Training and Test Benchmarks

Our training and test benchmarks are obtained from SPEC2000, SPEC2006 [11] and Vision [48] benchmark suite with reference inputs. These benchmarks are compiled using arm cross compiler provided by gcc with -O2 optimization flag.

Our training benchmark set was obtained after categorizing them as compute-intensive, memory-intensive and intermediate benchmarks. Table 5.2 lists the training and test benchmarks sets used in our evaluation.

<table>
<thead>
<tr>
<th>Training Benchmarks</th>
<th>art, astar.r, bzip2, disparity, equake, h264, mcf, omnetpp, perlbench, sjeng, tracking, twolf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Benchmarks</td>
<td>astar.r, calculix, GemsFDTD, gobmk, hmm, lbm, mser, namd, sift, texture_synthesis, tonto, vortex</td>
</tr>
</tbody>
</table>

Table 5.2: Training and Test Benchmarks
The counter values used in our models are obtained from the gem5 simulator.

The first two billion instructions of every benchmark is run on gem5 after fast-forwarding the first 10 million instructions. The counter values of the benchmarks are obtained by periodically dumping the statistics for every 10 million committed instruction on gem5. Hence, every training benchmark has 200 points which need to be fit using our regression model. In total we have $12 \times 200 = 2400$ entries in the training set. The accuracy of the models developed is tested using a test benchmark set. These benchmarks are also profiled on a 10 million instruction window.

5.2.3 Intra-core validation

We first evaluate the correctness of the proposed CPI stack model.

Model Fitting for training benchmarks

We fit the training benchmark points on the CPI model to obtain the $\beta_i$ values. The fitting error for the training benchmarks on the CPI model is 3.12% and 4.47% for 1-core coalition and 2-core coalition respectively. Figure 5.7 shows the CPI fit for a sub-set of training benchmarks.

Model Validation for test benchmarks

We validate the CPI model obtained using training benchmarks by plugging in the values of various miss events on a completely new set of test benchmarks. Figure 5.8 plots the measured and estimated CPI values for a subset of test benchmarks. We obtain an average prediction error of 3.49% and 6.38% and maximum prediction error of 21.7% and 25.1% for the test benchmarks. The prediction error is slightly higher in 2-core coalition since minute deviations in CPI will increase the percentage of prediction error in a 2-core coalition.
5.2.4 Inter-core validation

We build the inter-core models for various miss events like $miss_{L1}, miss_{L1D}, dmiss_{L2}, miss_{br}$ and $fu_{busy}$ for the training benchmark points. We compute branch, L1-I$, L1-D$, L2$ misses and $fu_{busy}$ occurrences on 1-core coalition when the benchmark is currently running on 2-core coalition and vice-versa. We plug in these values into each of the CPI stack models.

Model Fitting for training benchmarks

The inter-core fitting error is 5.3% when we predict the CPI of 2-core coalition from 1-core coalition with maximum prediction error of 19.76% and 95% of the benchmark points having error less than 12%. The inter-core prediction error is 3.35% when we predict the CPI of 1-core coalition from 2-core coalition with maximum prediction error of 16.4% and 95% of
Figure 5.8: Estimated CPI using CPI model for a sub-set of test benchmarks

the benchmark points having error less than 7%. Figure 5.10 shows the inter-core fitting CPI for a sub-set of test benchmarks.

Model Validation for test benchmarks

The inter-core prediction error is 6.61% when we predict the CPI of 2-core coalition from 1-core coalition with maximum prediction error of 26.9% and 95% of the benchmark points having error less than 13%. The inter-core prediction error is 3.61% when we predict the CPI of 1-core coalition from 2-core coalition with maximum prediction error of 24.16% and 95% of the benchmark points having error less than 9%. Figure 5.10 shows the inter-core predicted CPI for a sub-set of benchmarks.
Figure 5.9: Predicted CPI from 2-core Coalition to 1-core Coalition and vice-versa for a sub-set of training benchmarks
Figure 5.10: Predicted CPI from 2-core Coalition to 1-core Coalition and vice-versa for a sub-set of test benchmarks
5.3 Application of Performance Modeling

The performance models which were developed in Section 5.1 can predict the performance on any target coalition configuration. We can have a threshold performance value using which we can decide whether cores need to be coalesced/uncoalesced/remain in the current configuration during runtime. A performance-aware scheduler can be built using this decision.

5.3.1 Problem Definition

We formally define the dynamic job-scheduling problem as follows: Given \( n \) malleable tasks \([31]\) (number of cores allocated per application can be dynamically changed during runtime through preemption) \( T_0, T_1, ..., T_{n-1} \) and \( m \) physical homogeneous cores \( P_0, P_1, ..., P_{m-1} \), our objective is to allocate and schedule tasks so as to minimize the makespan \( C_{\text{max}} = \max_j \{C_j\} \) where \( C_j \) is the finish time of Task \( T_j \). In our system, tasks can arrive at any time. For simplicity, we assume that at a given time \( n \leq m \).

5.3.2 Online Scheduling algorithm

Our online algorithm is based on the fact that tasks can be preempted and migrated to other core coalitions if their requirements have changed. \( \text{wait\_queue} \) denotes the tasks which could not be scheduled due to non-availability of resources while \( \text{curr\_queue} \) contains the currently executing tasks.

Algorithm 1 presents the framework of our online algorithm. Procedure \text{ONLINE_SCHEDULER_TICK()}\) is called by the operating system at every scheduling tick. In this procedure,

(i) tasks in the \( \text{wait\_queue} \) are scheduled on one simple core if free resources exist (Lines 7-11),

(ii) freed resource counts are updated for completed tasks (Lines 15-16),

63
(iii) currently running tasks are preempted and migrated to appropriate coalition configurations based on their requirements (Lines 17-18).

Our migration policy stated in Algorithm 2[2] considers the changing performance requirements of currently running tasks. The objective of this algorithm is to find the lowest coalition size, in which the predicted speedup is within a predetermined threshold range.

Let $T_j$ be a task that is currently running on a $x$-core coalition and has not yet completed execution. The performance of $T_j$ is predicted for all possible coalition configurations starting from the lowest value (i.e. 1). Let $y$ be a target coalition which can be formed. Let $th_{speedup}$ denote the predetermined threshold value on moving from $x$-core to $y$-core coalition ($ThRESHOLD[x][y]$). Our scheduler preempts tasks and moves it from $x$-core to $y$-core coalition if resources are available and either one of the following condition is met:

- $x > y$ and predicted speedup from $x$-core to $y$-core coalition is lesser than $th_{speedup}$ (Lines 9-14)
- $x < y$ and predicted speedup from $x$-core to $y$-core coalition is greater than or equal to $th_{speedup}$ (Lines 15-21)

5.3.3 Experimental Setup

Our adaptive multi-core system consists of four 2-way out-of-order cores which allows at most one 2-core coalition. We do not allow four cores to be coalesced since the performance improvement is rather limited when we move from 2 to 4-core coalition.

Our threshold value for moving to a 2-core coalition ($th_{speedup}$) is 1.4. i.e., an application is allowed to form a 2-core coalition, if the speedup on a 2-core coalition with respect to 1-core coalition is predicted to be greater than 1.4. Similarly, ($th_{slowdown}$) value is assigned as 0.71. This means that a 2-core coalition can be broken if the speedup of 1-core coalition with
**Algorithm 1** Online scheduler for ideal Adaptive multi-core

1: **procedure** INITADAPTIVEMULTICORE$(m, n)$
2: \hspace{1em} $free_{cores} \leftarrow m$
3: \hspace{1em} $free_{clusters} \leftarrow m/4$
4: \hspace{1em} ASSIGNTHRESHOLD()
5: **end procedure**

6: **procedure** ONLINESCHEDULERTICK()
7: \hspace{1em} while $wait_{queue}.empty() == FALSE$ do
8: \hspace{2em} if $free_{cores} > 0$ then
9: \hspace{3em} next_task $\leftarrow task_{queue}.front()$
10: \hspace{3em} PLACETASK(next_task)
11: \hspace{2em} end if
12: **end while**
13: while $curr_{queue}.empty() == FALSE$ do
14: \hspace{1em} curr_task $\leftarrow curr_{queue}.front()$
15: \hspace{1em} if curr_task is finished then
16: \hspace{2em} UPDATEFREECORES(curr_task, free_cores)
17: \hspace{1em} else
18: \hspace{2em} MIGRATETASK(curr_task)
19: \hspace{2em} end if
20: **end while**
21: **end procedure**

22: **procedure** PLACETASK(task)
23: \hspace{1em} ALLOCATECORES(task, 1)
24: \hspace{1em} task.core_usage $\leftarrow 1$
25: \hspace{1em} task.is_coal $\leftarrow FALSE$
26: \hspace{1em} free_cores $\leftarrow free_{cores} - 1$
27: **end procedure**
Algorithm 2: Algorithm for task migration for ideal Adaptive multi-core

1: procedure MigrateTask(task)
2:    max_cores ← Get_max_cores(task)
3:    curr_usage ← task.core_usage
4:    use_cores ← max_cores + curr_usage
5:    use_cluster ← (use_cores > 1) or task.is_coal
6:    for y ← 1 to use_cores do
7:        pred_speedup ← PredictSpeedUp(curr_usage, y)
8:        th_speedup ← THRESHOLD[curr_usage][y]
9:        if (y > curr_usage) and (pred_speedup ≥ th_speedup)
          and (free_clusters − use_clusters) > 0 then
10:           AllocateCores(task, y)
11:           free_clusters ← free_clusters − use_clusters
12:           task.is_coal ← TRUE
13:           use_cores ← y
14:           break
15:     else if (y < curr_usage) and (pred_speedup < th_speedup) then
16:        AllocateCores(task, y)
17:        if y == 1 then
18:           free_clusters ← free_clusters + 1
19:        end if
20:        use_cores ← y
21:        break
22:    else
23:       use_cores ← curr_usage
24:    end if
25: end for
26: task.core_usage ← use_cores
27: free_cores ← free_cores − use_cores
28: end procedure
respect to 2-core coalition is greater than 0.71.

The benchmarks used in our evaluation are those which require 2-core coalition for at least 25% of their execution time. Table 5.3 lists the benchmarks which satisfy this criteria and hence used in our evaluation.

<table>
<thead>
<tr>
<th>Training Benchmarks</th>
<th>art, bzip2, mcf, h264, tracking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Benchmarks</td>
<td>vortex, GemsFDTD, lbm, sift, texture</td>
</tr>
</tbody>
</table>

Table 5.3: Benchmarks used in Online Scheduling

In our experiments, we allow a maximum of four tasks to exist at a time. We compute the task sets by selecting every possible one/two/three/four combination from Table 5.3 summing to a total of 385 combinations.

5.3.4 Experimental Evaluation

We find the makespan for each of the task sets in the following systems:

- homogeneous/symmetric multi-core system consisting of four 2-way OOO cores ($S_1$)
- homogeneous/symmetric multi-core system consisting of two 4-way OOO cores ($S_2$)
- adaptive multi-core described before using measured performance ($Adaptive_{measured}$)
- adaptive multi-core described before using predicted performance ($Adaptive_{predicted}$)

Figure 5.1 shows the speedup on different systems $S_1$, $S_2$, $Adaptive_{measured}$ and $Adaptive_{predicted}$ with respect to the makespan of $S_1$ for all task sets using the online scheduler suggested in the previous section. From this figure, we find that $S_2$ performs slightly better than adaptive systems when the number of applications present in the system is very low since our online scheduler does not allow an application to run on a 2-core coalition if the speedup with respect to 1-core coalition is less than 1.4. When number of applications running on the system is high, we can have a better speedup.
with respect of configuration $S_2$. We also find that our online scheduling using performance prediction model achieves a speedup as good as an online scheduler using measured values.

![Speedup comparison](image)

**Figure 5.11**: Speedup comparison on different systems using online scheduling with respect to homogeneous S1 configuration

### 5.4 Summary

In this chapter, we proposed an accurate model for predicting the performance behavior of an application on a target coalition when running on a current coalition. This model used a predictive technique involving linear and non-linear regression. From experimental evaluation, we found that the model is accurate in predicting performance for all possible coalitions. As an application, we used this model in a runtime scheduler to dynamically shift applications to different coalition as per their needs and available resources. We observe that the scheduling decision taken using predictive technique is as good as scheduling based on measured performance.
Chapter 6

Conclusions and Future work

6.1 Conclusions

The emerging workloads in today’s scenario require high ILP and TLP. Static heterogeneous architectures using the same ISA, have proved as an alternative to existing homogeneous multi-core architectures. However, their performance characteristics cannot be modified during runtime. Dynamic heterogeneous architectures, also known as adaptive multi-core architectures, have been proposed to adapt to new performance requirements during runtime.

In this thesis, a realistic adaptive multi-core architecture based is implemented and evaluated on a full system simulator with a commercial ISA. This prototype has helped in obtaining detailed modifications which are required in a real system to enable adaptive re-configuration during runtime. This evaluation has also helped in identifying the strengths and weaknesses of adaptive multi-cores with respect to performance.

This thesis also suggested a performance model that can aid in determining the degree of coalition during runtime on an ideal adaptive multi-core system. This model was based on the fact that an application does not have a uniform requirement throughout its lifetime. The suggested perfor-
mance model was later used in a performance-aware scheduler to help in making well-informed task mapping decisions during runtime.

6.2 Future Work

The prototype of an adaptive multi-core architecture proposed in thesis has created a framework for future research in adaptive multi-core computing. This framework will also help in performing architecture specific optimizations.

Modern compilers do not take the adaptiveness of an architecture into consideration when generating binary executables. An optimization pass specific to adaptive multi-cores can be implemented in compilers. This optimization pass will help in making adaptive multi-cores perform better than traditional out-of-order cores in more scenarios.

Finally, a power model similar to performance model suggested in this thesis can be developed to perform energy-aware scheduling during runtime.
Bibliography


[51] Hongtao Zhong, Steven A Lieberman, and Scott A Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In High Performance Computer Architecture,