A DYNAMIC HETEROGENEOUS MULTI-CORE ARCHITECTURE

MIHAI PRICOPI

(B.Eng., M. Eng., UNIVERSITY "GHEORGHE ASACHI" OF IASI)

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

January 2014

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.

Mihai Pricopi

January 29, 2014

Abstract

Computing systems have made an irreversible transition towards parallel architectures with the emergence of multi-cores. The existing trends indicate that multi-cores (and even many-cores) will comprise of collections of simple cores rather than complex cores. However, contemporary applications have highly diverse computation requirements that are hard to satisfy with a set of simple homogeneous cores. While parallel applications can benefit from thread-level parallelism offered by such multi-core solutions, there still exist a large number of applications with substantial amount of sequential code. The sequential programs suffer from limited exploitation of instruction-level parallelism in the simple cores.

In this thesis, we design and evaluate a novel dynamic heterogeneous multi-core architecture, called Bahurupi, that can successfully reconcile the conflicting demands of instruction-level and thread-level parallelism. Bahurupi can accelerate the performance of serial code by dynamically forming coalition of two or more simple cores to offer increased instruction-level parallelism. In particular, Bahurupi can efficiently merge 2-4 simple 2-way out-of-order cores into coalition to reach or even surpass the performance of more complex and power-hungry 4-way or 8-way out-of-order core. For floating-point SPEC benchmarks, on average, dual-core and quad-core Bahurupi improve performance by 23% and 57% compared to 4-way and 8-way cores. While for SPEC integer benchmarks, dual-core and quad-core Bahurupi achieves, on average, 92% and 91% of the performance of 4-way and 8-way cores.

We also introduce a novel reconfigurable L1 data cache architecture for Bahurupi that is able to accommodate the memory demands of a dynamic heterogeneous multi-core architecture with low area and energy overhead. Our design consumes 4.16X less energy per access compared to the alternative multi-ported cache design.

A fundamental challenge in exploiting dynamic heterogeneous multi-cores arises from appropriately scheduling the workload on such a flexible architecture design. We design offline and online schedulers that intelligently reconfigure and allocate the cores to a mix of sequential and parallel applications so as to minimize the overall makespan. Experimental evaluation confirms that dynamic heterogeneous multi-core architectures

can substantially improve the performance compared to homogeneous and static heterogeneous multi-core architectures with average speedups ranging from 17% to 28%. When doing online scheduling on Bahurupi, utilization results also put dynamic heterogeneous multi-core architectures on the first place (76% average utilization).

Any problem in computer science can be solved with another level of indirection.

— David John Wheeler

... except for the problem of too many layers of indirection.

— Kevlin Henney

List of Publications

1. M. Pricopi and T. Mitra. Bahurupi, *Bahurupi: A Polymorphic Heterogeneous Multi-core Architecture*, ACM Transactions on Architecture and Code Optimization, TACO, 8(4):22:122:21, January, 2012.

   Presented at 7th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC) 2012.

2. M. Pricopi and T. Mitra, *Polymorphic Heterogeneous Multi-Core Architecture*, Patent number: PCT/SG2012/000454, 2012.

3. M. Pricopi and T. Mitra, *Task Scheduling on Adaptive Multi-core*, IEEE Transactions on Computers, TC, (PrePrints), 2013.

4. K. Mysur, M. Pricopi, T. Marconi, and T. Mitra, *Implementation of Core Coalition on FPGAs*, In Preceedings of the 21st International Conference on Very Large Scale Integration, VLSI-SoC, pages 198203. IFIP/IEEE, 2013.

5. M. Pricopi, T. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin, *Power-Performance Modeling on Asymmetric Multi-cores*, In International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES, pages 1-10, 2013.

6. T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin., *Hierarchical Power Management for Asymmetric Multi-core in Dark Silicon Era*, In Proceedings of the 50^{th} Annual Design Automation Conference, DAC, pages 174:1–174:9, ACM, 2013.

Acknowledgements

Foremost, I would like to express my deepest gratitude to my advisor, Professor Tulika Mitra for her continuous support during my PhD. Professor Tulika has been an invaluable mentor for me, she always encouraged my research and allowed me to develop as a research scientist and as a person. Most important, her extreme passion for research, her commitment and professional attitude have been a true inspiration for me, inspiration that I will carry with me in my professional life. I owe a great debt of thankfulness to Professor Tulika.

I would like to extend my gratitude to my thesis committee members, Professors Weng Fai Wong and Soo Yuen Jien for their valuable comments and feedback during the different steps of my PhD. Their comments allowed me to greatly improve and clarify my thesis.

Special thanks go to friends from our Embedded Systems Lab, with whom I had the chance to work in different projects over the years: Malai, Kaushik, Chen Liang, Vanchi and Andrei. A heartfelt thanks to the friends that always supported me in my research. With them I shared my ideas and beliefs which many times served as the topic of very long and constructive discussions, especially during the lunch breaks: Bogdan, Cristina, Dumi, Cristi, Narcisa, Marcel and Nicolas.

Deep gratitude and consideration go to Ge Zhiguo and Naxin Zhang with whom I had the chance to collaborate in a constructive environment which helped me to enhance part of my thesis with valuable ideas.

These lines cannot express my feelings for my parents and my brother, Andrei, for their constant and unconditional support. Without you I would not be the person I am today. I owe them my full gratitude and respect for supporting me. In all these years I always felt them very close even though home has been so far away.

Above all, I would like to thank the most important person in my live - my fiancee Delia. She has been a constant source of love, balance, strength and inspiration for me. There have been several times when I felt the lack of motivation and strength in continuing my research. Her constant determination and encouragement made me always keep on going until seeing this work done. Thank you for everything.

Table of Contents

List of Publications **iii**

Acknowledgements **v**

Table of Contents **vii**

List of Figures **xi**

List of Tables **xv**

List of Algorithms **xvii**

1 Introduction **1**

1.1 Thesis contributions 9

1.2 Thesis outline 13

2 Related Work **15**

2.1 Heterogeneous multi-core architectures 15

2.1.1 Static heterogeneous multi-core architectures 15

2.1.2 Dynamic heterogeneous multi-cores 18

2.2 Reconfigurable caches 24

2.3 Task scheduling 27

3 Bahurupi Adaptive Multi-Core **30**

3.1 Bahurupi execution model 31

3.1.1 Sentinel instruction 34

3.1.2 Execution model 37

3.2 Architectural details 41

3.2.1 Live-in register renaming 41

3.2.2 Live-out register renaming 42

3.2.3 Branch misprediction and exceptions 44

3.2.4 Memory hierarchy 45

3.2.5 Memory hazards 45

3.2.6 Reconfiguration overhead 46

3.2.7 Compiler support 47

3.3 Experimental setup 47

3.3.1 Simulator 47

3.3.2 Compiler 48

3.3.3 Benchmarks 50

3.4 Experimental results 50

3.4.1 Overall speedup 50

3.4.2 Energy consumption 53

3.4.3 Load balancing 54

3.4.4 Global register file access 54

3.4.5 Traffic on coalition bus 58

3.4.6 Sentinel instruction overhead 58

3.4.7 Area and delay overhead of coalition logic 59

3.5 Bahurupi FPGA implementation 60

3.5.1 Fabscalar synthesizable out-of-order core 60

3.5.2 Core coalition logic 63

3.5.3 Prototype synthesis and evaluation 67
3.6 Summary 71

4 Reconfigurable Data Cache Architecture 73

4.1 Experimental setup 76
4.2 Limitations of multi-ported shared L1 cache 77
4.2.1 Area and energy overhead 79
4.3 Limitations of single-ported shared L1 cache 79
4.3.1 Simultaneous memory accesses 79
4.3.2 Performance impact 80
4.4 System reconfiguration 82
4.5 Network reconfiguration and address mapping 85
4.5.1 Network routing and reconfiguration examples 89
4.5.2 Bank conflicts 96
4.6 Cache reconfiguration 97
4.6.1 L1 data cache miss rates 97
4.6.2 Area and energy consumption 101
4.6.3 Miss rate improvement and performance analysis 102
4.7 Comparison with multi-ported shared L1 cache 104
4.8 Summary 105

5 Scheduling on Bahurupi Architecture 106

5.1 Optimal schedule on ideal dynamic heterogeneous multi-core 109
5.1.1 Optimal schedule with continuous resources 110
5.1.2 Optimal schedule with discrete resources 113
5.2 Task scheduling on Bahurupi 116
5.2.1 Constraint C1 116
5.2.2 Constraint C2 117

5.2.3 Constraint C3 118
5.2.4 Online schedule for Bahurupi 120
5.3 Quantitative results 123
5.3.1 Workload 123
5.3.2 Multi-core configurations 123
5.3.3 Speedup functions 126
5.3.4 Scheduling on homogeneous and static heterogeneous multi-cores 128
5.3.5 Limit study of dynamic heterogeneous multi-core 128
5.3.6 Realistic performance benefit of dynamic heterogeneous multi-core 131
5.3.7 Reconciling ILP and TLP 132
5.4 Summary 134

6 Conclusions **136**
6.1 Summary of the thesis 136
6.2 Future work 138

References **141**

List of Figures

1.1 Evolution of single-core processor performance [45]. 2

1.2 Speedup of multi-cores according to Amdahl's law [96]. 3

1.3 Single-core, homogeneous and heterogeneous multi-cores performance (Figure from P. Hofstee, IBM Austin). 5

1.4 Examples of homogeneous, static and dynamic heterogeneous architectures. 6

1.5 Bahurupi architecture. 10

2.1 big.LITTLE static heterogeneous multi-core architecture. 17

2.2 TFlex micro-architecture and internal organization [52]. 19

2.3 Federation architecture [93]. 21

2.4 Block diagram of a 4-core Voltron architecture [104]. 21

2.5 Example of a logarithmic interconnection network. 27

3.1 Speedup trends for Ferret kernels and overall speedup with reconfigurable architecture. 31

3.2 Bahurupi architecture. Additional resources required for coalition are highlighted. 32

3.3 The sentinel instruction format. 34

3.4 Percentage of basic blocks with number of live-in and live-out registers below a threshold. 36

3.5 Bahurupi distributed execution model: (a) Control flow graph (CFG) of a program, and (b) Execution of the CFG on 2-core Bahurupi architecture. 39
3.6 Global and local register renaming. 42
3.7 Bahurupi speedup normalized to 2-way core for (a) SPEC and (b) embedded benchmarks. 51
3.8 Bahurupi energy consumption normalized to 2-way core for (a) SPEC and (b) embedded benchmarks. 55
3.9 Load balance on 4-core Bahurupi. 56
3.10 Percentage of destination registers renamed to global register file. 56
3.11 Broadcasts on coalition bus in (a) 2-core Bahurupi and (b) 4-core Bahurupi. 57
3.12 Code size increase due to sentinel instructions. 58
3.13 Fabscalar pipeline with coalition logic. 61
3.14 Two 2-way core coalition. 62
3.15 Register flow across cores - an illustration. 65
3.16 Area utilization (8-way core equivalent in performance to 4-core coalition could not be synthesized). 70
3.17 Area breakup of coalition logic w.r.t baseline core. 70
3.18 Clock frequency for various core configurations. 71

4.1 Example of 4-core coalition with data cache merging. 74
4.2 Miss rate for shared cache vs. coherent caches. 76
4.3 Multi-ported shared cache configurations. 78
4.4 Shared cache area and energy increase for different number of ports (normalized w.r.t to 1 port). 80
4.5 Average number of L1 data cache accesses per cycle for 2-core coalition, 3-core coalition and 4-core coalition. 81

4.6 Performance impact for 2-core and 4-core coalition run on cache configurations with four ports and one port (normalized w.r.t performance of a baseline single core). 82
4.7 Novel L1 data cache architecture for dynamic heterogeneous multi-cores. 83
4.8 System reconfiguration for private and coalition modes in case of a 4-core coalition. 84
4.9 CPU to memory interconnection network architecture for 4-cores and 8 access points. 87
4.10 Address mapping in the interconnection network. 88
4.11 Address mapping examples for different scenarios. 90
4.12 Address mapping examples for simultaneous requests. 92
4.13 Reconfiguration of the number of access points example. 94
4.14 L1 data cache average accesses per cycle per banks for 4-core coalition connected to the log network. 96
4.15 L1 data miss rates improvement relative to a 4KB, 2-way set associative cache. 97
4.16 Cache reconfiguration for coalition mode. 98
4.17 Example of cache reconfiguration. 100
4.18 Miss rate improvement for 2 banks, 4 banks and 8 banks (normalized w.r.t miss rate of 2 banks). 102
4.19 Speedup of 4-core coalition connected to 2, 4 and 8 banks using the log network (normalized w.r.t performance of a baseline single core). 103
4.20 Comparison between our design (8banks) and four-ported cache. 104

5.1 Illustrative example showing the schedule of a mix of sequential and parallel applications on different architectures. 108
5.2 Resource transformation example for $n = 2$. 112

5.3 Piecewise interpolation of speedup function for *mcf* 113
5.4 Rectangle packing for discrete resource problem. 114
5.5 Example of imposing constraint C2. 117
5.6 Example of imposing constraint C3. 119
5.7 Speedup functions for sequential and parallel tasks. 127
5.8 Comparison of dynamic and static multi-cores under off-line schedule. The speedup is w.r.t. the baseline homogeneous S1. 129
5.9 Utilization of different multi-cores in offline schedule. 130
5.10 Comparison of Bahurupi with static multi-cores under online schedule. The speedup is w.r.t. baseline homogeneous S1. 131
5.11 Utilization of different multi-cores in online schedule. 132
5.12 Speedup of sequential applications averaged across all task sets normalized w.r.t. execution on native 2-way core. 133
5.13 Speedup of parallel applications averaged across all task sets normalized w.r.t. execution on one 2-way core. 134

List of Tables

2.1 Comparison between different dynamic heterogenous multi-core architectures. 23

3.1 Parameters for baseline cores. 48
3.2 Workloads used for simulation. 49
3.3 Additional ports within the baseline core. 68
3.4 Additional RAMs/CAMs for coalition per core. 68
3.5 Global resources for coalition. N is # of cores. 68

4.1 Benchmarks description. 77
4.2 Amount of shared cache accessed based on the *Access Point* bitfield width and the cache reconfiguration switches position. 101

5.1 Characteristics of benchmarks used in our study. 124
5.2 Multi-core configurations used in our study. 125
5.3 Configuration parameters for out-of-order cores: issue, commit, dispatch width; reorder buffer (ROB) size; load-store queue (LSQ) size; number of ALU, floating point (FP), and load-store (LSU) units; instruction-data TLB size, L1 instruction-data cache size, and unified L2 cache size. 125

List of Algorithms

1 Malleable Task Scheduler on Bahurupi 115

2 Online scheduler for Bahurupi 122

Chapter 1

Introduction

For the past two decades, the computer architects have transparently accelerated sequential applications through aggressive exploitation of instruction-level parallelism (ILP). Figure 1.1 shows the performance evolution of single-core processors over time where their complexity increased proportionally with the performance. Between 2002 – 2006, the last very complex superscalar micro-architectures with out-of-order execution, dynamic speculation and SMT capabilities were designed in order to meet the requirements of the various applications. Limited power and thermal budgets have reversed this trend; instead of building more complex cores, the architecture community moved irreversibly towards multi-cores. The increase of number of transistors on-chip allowed the computer architects to propose designs comprising of many homogeneous, simple and power-efficient cores on the same die.

Homogeneous multi-cores. The multi-core architectures are ideally suited for exploiting the thread-level parallelism (TLP) existing in the applications. These architectures are also named *homogeneous multi-cores* because all the cores have the same area, performance and power capabilities. But there still exist a large class of applications with substantial sequential code fragment that are difficult, if not impossible, to

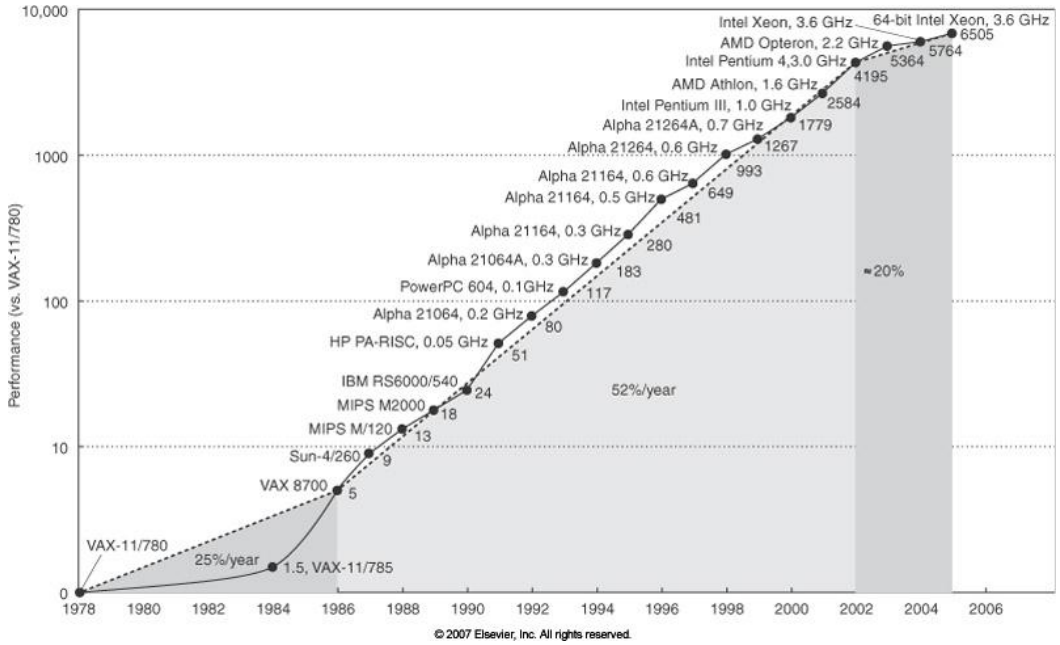


Figure 1.1: Evolution of single-core processor performance [45].

parallelize. Here, we cite *Amdahl's Law* which captures the overall gain obtained by enhancing parts of a computation. The law states that the performance improvement of an entire application obtained by enhancing a fraction f of a computation by speedup factor S is given by the relation:

$$Speedup_{homogeneous}(f,S) = \frac{1}{(1-f)+\frac{f}{S}} \tag{1.1}$$

In case of applications running on multi-cores, the speedup of the parallel portion of the code is governed by the total number of cores, while the overall speedup of the application will be limited by the performance of the serial code. In Figure 1.2 we show the overall speedup of an application with different fractions of parallel code that runs on different number of cores. This plot assumes an ideal multi-core architecture that can support a large number of cores and an application that can be easily parallelised with no communication overhead. Assuming that we can parallelise 95% of the application,

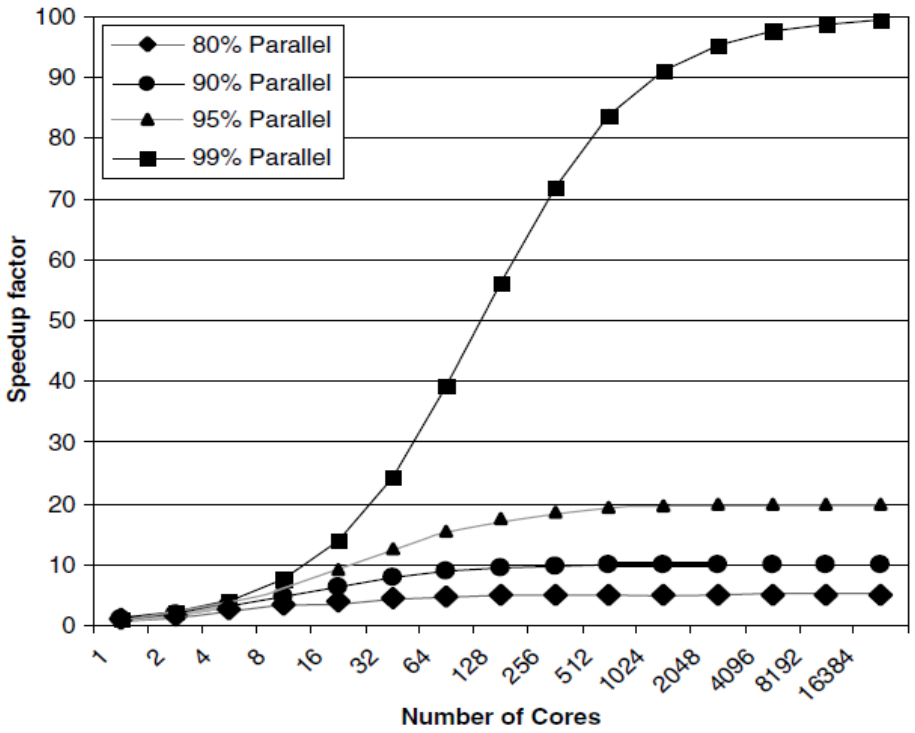


Figure 1.2: Speedup of multi-cores according to Amdahl's law [96].

the overall speedup that can be obtained is limited to just 20% even when running on large number of cores. Moreover, if we can parallelise 99% of the application, then the overall speedup can reach only close to 100% even when running on large number of cores (more than 16384). Of course, real multi-core architectures have a relatively small number of cores (4–16) and real parallel applications are not easily threaded without paying the communication overhead. Consequently, the multi-cores are unable to offer more speedup as they cannot accelerate the sequential fraction of code through instruction-level parallelism (ILP) exploitation. Only complex out-of-order execution engines with high-degree of superscalarity can transparently accelerate sequential code fragments through aggressive exploitation of ILP. Power and thermal limits as well as reliability issues, however, do not permit deployment of such complex cores on the same die.

In Figure 1.3 we show how different processor architectures try to keep up with

the *Moore's Law* (named after Gordon Moore), which states that the number of on-chip transistors doubles approximately every two years. Ideally, the processors' performance should follow the Moore's Law. We can see that single-cores have reached a steady state of performance years ago, while the homogeneous multi-cores will soon reach a steady state of performance; they clearly cannot ride the growth line mostly due to the poor exploitation of ILP.

A related concept to the Moore's Law is the *Dennard's Scaling* (named after Roberd Dennard, who first described this effect in 1974). The key idea of the Dennard's Scaling is that, while transistors get smaller, the power density (the amount of power per unit volume) remains constant. For example, if the transistor's linear size is reduced by two then the power it consumes will be reduced by four – with voltage and current both being reduced by half. While the Moore's Law still remains valid, the Dennard's Scaling has already failed. The ability to reduce the supply voltage and current has reached the reliability limits, making the Dennard's Scaling infeasible. As a result, the dynamic power mostly remains constant while the static power consumption increases due to the increased number of transistors. As a consequence, we can still add more cores on the chip but we cannot switch them all on at the same time due to power and thermal constraints. This phenomenon, known as Dark Silicon [40], demands alternative architectural designs.

The next logical step in order to overcome the limitations of homogeneous multi-cores and continue to follow the Moore's Law trend (Figure 1.3) is to design *heterogeneous* multi-cores. We identify two main categories of *heterogeneous* multi-cores: *static* and *dynamic*. Static heterogeneous multi-cores are comprising of cores that share the same Instruction Set Architecture (ISA) but have different area, power and performance capabilities. On the other side, dynamic heterogeneous multi-cores are built on top of homogeneous multi-cores and are able to create at run-time virtual cores by combining homogeneous simple cores. The virtual cores are more powerful and capable of

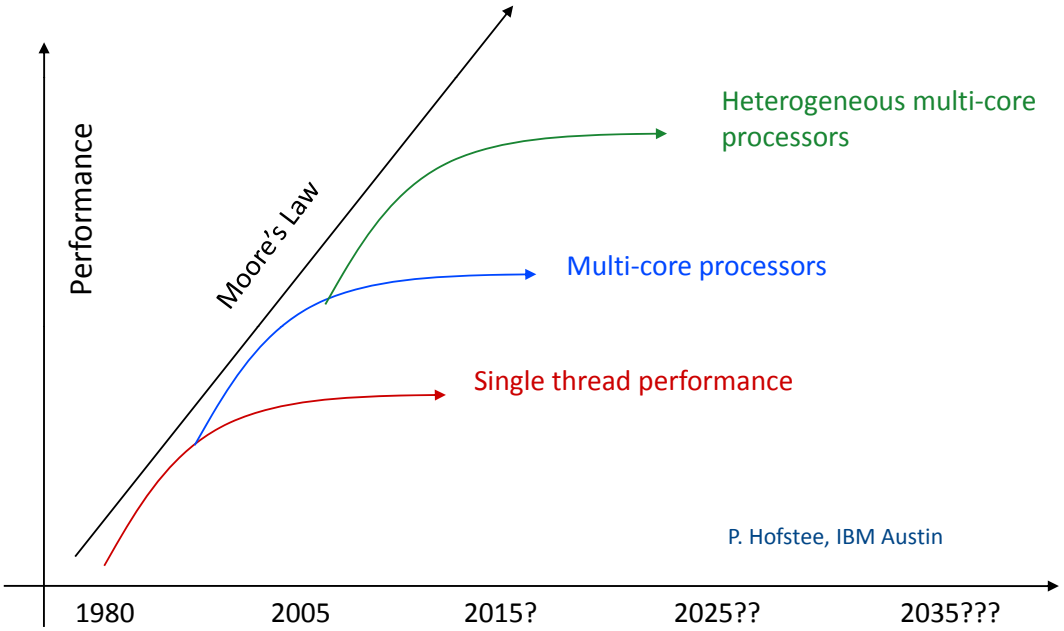


Figure 1.3: Single-core, homogeneous and heterogeneous multi-cores performance (Figure from P. Hofstee, IBM Austin).

accelerating sequential applications.

For illustration, Figure 1.4 depicts four examples of multi-core architectures. We present two types of homogeneous multi-cores: an eight-core architecture where each core is a simple core capable of poor exploitation of ILP and a four-core architecture where each core is a complex superscalar our-of-order core capable of extracting significant amount of ILP. The figure also shows an example of a static heterogeneous architecture with five cores where core C0 is a very complex superscalar, out-of-order core capable of aggressive exploitation of ILP. The other four cores are simple cores. The last example is a eight-core dynamic heterogeneous multi-core architecture where two virtual cores are created. The first virtual core is created by coalescing the cores $\{C1, C2, C3\}$ and the second virtual core is created by coalescing the cores $\{C4, C5\}$.

Another form of heterogeneity (which we do not cover in this thesis) can be found especially in the embedded devices that generally embrace multi-core solutions customized for a particular application domain (i.e., *multi-processor systems-on-chip (MP-*

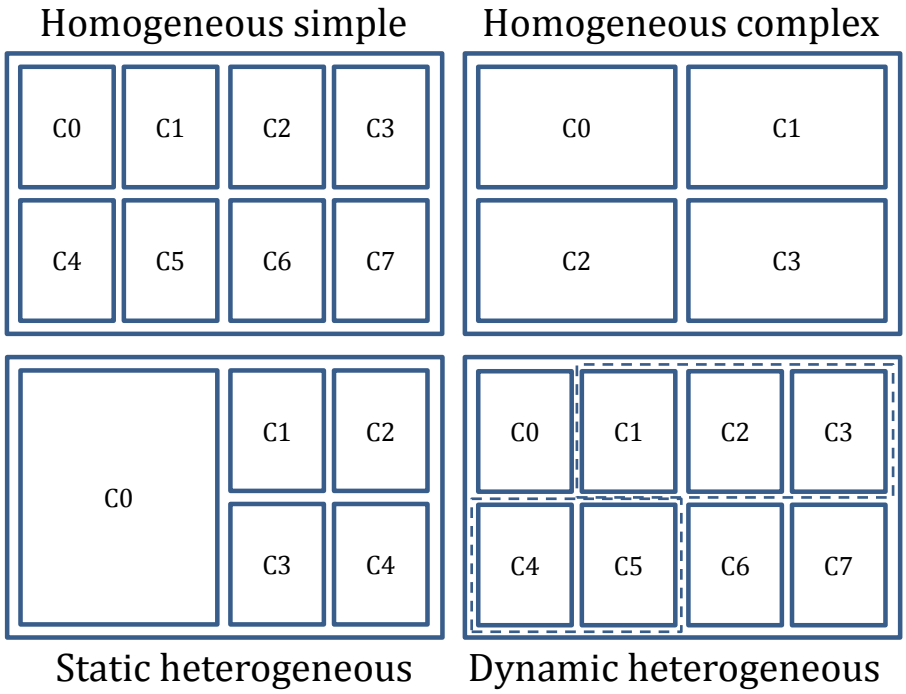


Figure 1.4: Examples of homogeneous, static and dynamic heterogeneous architectures.

SoC)). These architectures can offer significant advantages in terms of performance, power and area. While such customizations are beneficial for specific kernels such as audio, video, image processing, today both desktop and mobile computing platforms need to execute a wide variety of general-purpose applications for which the workload is not known a-priori.

Static heterogeneous multi-cores. Static heterogeneous multi-cores, are already emerging on the market as a promising solution that can achieve power-performance trade-off [83] essential in high performance, energy constrained embedded systems such as tablets, smart-phones, automotive telematics, and others. However, recently, the distinction between mobile embedded architectures and general-purpose computing architectures is rapidly disappearing especially in the consumer electronics space. Today, a smartphone or a tablet is expected to support a very dynamic and diverse landscape of software applications. Moreover, the processor architectures that we find today in the

mobile devices are very similar to the ones found in the general-purpose or server computers. These static heterogeneous multi-core architectures integrate high performance, power hungry complex cores ("big" cores) with moderate performance, power efficient simple cores ("small" cores) on the same chip. The characteristic that distinguishes static heterogeneous multi-cores from heterogenous MPSoC is that the different core types implement the same ISA; that is, the same binary executable can be scheduled to run on either the big or the small core. Examples of commercial static heterogeneous multi-cores include ARM big.LITTLE [41], integrating high performance out-of-order cores with low power in-order cores and NVidia Kal-El [5], consisting of four high performance cores with one low power core. An instance of the former, integrating quad-core ARM Cortex-A15 (big core) and quad-core ARM Cortex-A7 (small core) has already appeared in the Samsung Exynos 5 Octa SoC driving high-end Samsung Galaxy S4 smart-phones. The Cortex-A7 can handle regular low-intensity workloads on a smartphone (e.g., phone call, SMS, music playback) while Cortex-A15 needs to get involved in compute-intensive applications (e.g., gaming, flash heavy website).

Dynamic heterogeneous multi-cores. Even though static heterogeneous multi-cores are clearly positioned to accommodate software diversity (mix of ILP and TLP workload) much better than homogeneous multi-cores with promising results [70, 77, 32, 61, 54, 62], they are still not the ideal solution. As the mix of simple and complex cores has to be frozen during design/fabrication time, a static heterogeneous multi-core lacks the flexibility to adjust itself to the dynamic nature of workload. Any change in the applications requirements would have a big impact on the production costs. The next step forward to support both diverse and dynamic workload is to design dynamic heterogeneous multi-cores that can, at runtime, tailor itself according to the applications. Such adaptive architectures are physically fabricated as a set of simple, homogeneous cores. At runtime, two or more such simple cores can be coalesced together to create

a more complex virtual core. Similarly, the simple cores participating in a complex virtual core, can be disjoined at any point of time. A canonical example is to form coalition of two 2-way out-of-order cores to create a single 4-way ooo core. In other words, we would like to dynamically create static heterogeneous multi-cores through simple reconfiguration.

Speedup benefit of heterogeneity. A simple comparison of the performance potential of the three types of multi-core architectures is presented by Mark Hill and Michael R. Marty in [46] where the classical Amdahl's Law is extended to the new types of multi-core architectures. Traditionally, Amdahl's Law has been applied to *homogeneous* multi-core processors comprising of n cores. The results presented in this work show that heterogeneity clearly can offer performance beyond the Amdahl's Law. Moreover, static heterogeneous multi-cores can offer better performance than homogeneous multi-cores but their performance is still bounded by the lack of resources when exploiting both ILP and TLP. Additionally, this work clearly demonstrates that dynamic heterogeneous multi-core architectures offer speedups that are not limited by the sequential fraction (their speedup curve is an increasing function) of the program as these architectures can dynamically create complex cores that can accelerate the sequential fraction.

The results of this simple analysis leads us to the conclusion that we need to design a dynamic heterogeneous multi-core system that can efficiently coalesce cores at run-time with minimal overhead.

In this thesis we present a dynamic heterogeneous multi-core architecture and its memory system design. We also design offline and online schedulers for dynamic heterogeneous multi-core architectures that clearly show that dynamic heterogeneous multi-core architecture perform the best when compared with homogeneous and static heterogeneous multi-core architectures. In the next sections we describe the main contributions of this thesis and then we present the thesis organization.

1.1 Thesis contributions

The main contributions of this thesis revolve around the design of a dynamic heterogeneous multi-core system — starting with the micro-architecture design of the processor and its execution model, continuing with the design of the first level of data cache necessary for such a system, an FPGA proof-of-concept implementation and ending with a scheduler that is able to smartly allocate mixes of serial and parallel tasks to the cores or coalitions of cores.

Our first contribution is the design of a dynamic heterogeneous multi-core processor called *Bahurupi* (an Indian word meaning a person of many forms and guides, a polymorph), that can be tailored according to the workload by software [74][75]. Bahurupi is fabricated as a homogeneous multi-core system containing multiple identical, simple cores. The main novelty of Bahurupi lies in its ability to morph itself into a heterogenous multi-core architecture at runtime under software directives. Post-fabrication, software can compose together the primitive cores to create a customized multi-core system that best matches the needs of the applications currently executing on the system. Bahurupi successfully re-conciliates the conflicting requirements of applications with explicit thread-level parallelism and single-threaded serial applications with high degree of instruction-level parallelism.

Bahurupi high-level architecture. Figure 1.5 depicts a high-level overview of Bahurupi architecture. Each core is a simple 2-way out-of-order processor. Four such simple cores form a cluster. The example architecture in Figure 1.5 consists of 2 such clusters; but the number of clusters can be easily scaled with technology. In normal mode, this multi-core architecture can efficiently support multi-threaded execution. The simple 2-way cores, however, cannot exploit much ILP from sequential applications. To solve this issue, Bahurupi can form dynamic coalitions of 2-4 cores in a cluster with mini-

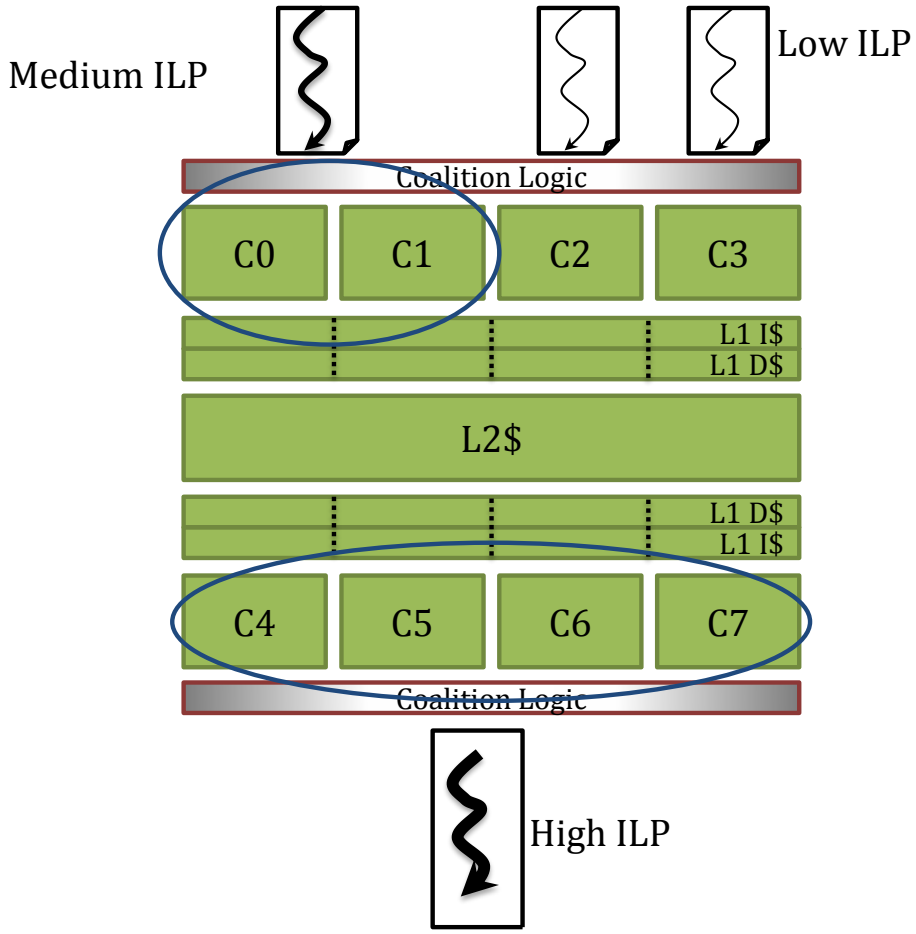


Figure 1.5: Bahurupi architecture.

mal additional hardware such that the merged cores can substantially speed up serial code execution. When 2 cores (4 cores) form a coalition, Bahurupi achieves performance close to that of 4-way (8-way) out-of-order execution engine. Note that Bahurupi allows coalition of *at most 4 cores* as speedup is limited beyond 8-way out-of-order core. The figure shows the architecture running four applications — one high-ILP serial thread running on 2-core coalition, two low-ILP serial threads running on two independent cores and one very high-ILP serial code running on 4-core coalition. In summary, Bahurupi can achieve the performance of complex out-of-order superscalar processor without paying the price of complex hardware and its associated energy inefficiency and reliability issues.

Bahurupi, is a hardware-software cooperative solution that demands minimal changes to both hardware and software. In coalition mode, Bahurupi follows a *distributed execution model* that avoids complex centralized fetch/decode, dependency resolution, and instruction scheduling of previous approaches. It needs support from compiler to identify the basic blocks and their register dependencies. This information is encapsulated in special *sentinel* instructions that precede each basic block. Thus, the dependencies among the basic blocks are explicitly conveyed in the program code. The cores can fetch and execute different basic blocks in parallel, thereby achieving performance speedup. The only additional hardware required is the *coalition logic* shared by the cores forming the coalition. This logic includes a shared global register file to communicate values between basic blocks running on different cores as well as three dedicated registers for synchronization among the cores. A minimal amount of additional resources is added to the internal architecture of the cores to support coalition. The main execution engines of the cores remain completely unchanged.

Bahurupi proof-of-concept implementation. Even though quite promising, the lack of acceptability of dynamic heterogeneous multi-cores, arises from the complexity of the glue logic required to coalesce the simple cores together. No attempt has been made so far to implement any of these architectures in hardware. The open question therefore remains whether dynamic heterogeneous multi-cores are feasible in terms of cycle-time and area overhead. The second contribution of this thesis takes the first step towards answering this question through a concrete, functionally correct implementation of our proposed Bahurupi multi-core architecture in hardware. We design a micro-architecturally accurate hardware synthesizable implementation of Bahurupi [71] architecture and we emulate the architecture in FPGAs. We synthesize a working prototype of the architecture on Xilinx Virtex 6 FPGA platform. Our prototype provides a proof-of-concept implementation that confirms the feasibility of dynamic heteroge-

neous multi-core architecture and establishes the benefit of the architecture compared to homogeneous and static heterogeneous multi-cores.

Bahurupi memory design. In case of dynamic heterogeneous multi-cores, the complex cores that are dynamically coalesced together are able to execute more instructions in parallel when compared with the simple baseline cores. This is similar to the case in which a wider issue superscalar processor with many parallel load/store units is accessing the memory [50, 88, 27, 101, 7]. Consequently, there are more accesses per cycle to the first level of data and instruction caches, imposing special requirements in the design of the L1 cache architecture. In case of instruction accesses, there is no need for special considerations as the fused cores execute the same application. If we assume the absence of self-modifying code, the cores can safely bring in and replicate the same instructions in multiple private caches without any coherency issue.

However, it is more challenging to design the L1 data cache to support multiple accesses per cycle ensuring high bandwidth and low latency as accesses to the L1 data cache have a high degree of sharing — different cores can read and write the data which is part of the same sequential program. Thus, we need a design of the first level of data cache that can be shared across the cores in a coalition. Our third contribution presented in this thesis is a novel reconfigurable L1 data cache architecture for dynamic heterogeneous multi-cores [78] that overcomes the limitations of multi-ported caches, offering high bandwidth and low latency when accessing the shared cache. Our design is also able to switch between two modes of execution: *private mode* (for traditional multi-core) and *coalition mode* (for dynamic heterogeneous multi-core). Additionally, the system can also reconfigure the size and associativity of the shared L1 data cache, which ensures optimal efficiency in terms of energy consumption, cache miss rate and performance.

Scheduling support for Bahurupi. Dynamic heterogeneous multi-cores appear well poised to support diverse and dynamic workloads consisting of a mix of serial (multiprogrammed) and parallel (multithreaded) tasks. As this is a nascent area, existing research primarily focuses on developing appropriate micro-architectural techniques to form coalition of simple cores. In reality, such adaptive architectures need to support both sequential and parallel applications executing concurrently. Existing literature is thus missing a realistic evaluation of the performance potential of adaptive multi-cores.

Our fourth contribution of this thesis takes the first step towards filling up this gap through a concrete performance limit study of our Bahurupi adaptive heterogenous multi-core in a scenario where both parallel and sequential applications coexist in the system [76]. Conducting a limit study of adaptive heterogeneous architectures with realistic workload is a challenging problem. As we are interested in identifying the true performance potential of an adaptive heterogeneous multi-core architectures, we have to employ an optimal scheduler that can intelligently reconfigure and allocate the cores to the applications so as to minimize the *makespan* (the time when all applications complete execution). We implement offline and online schedulers for Bahurupi and compare the results with the makespan obtained on homogeneous and static heterogeneous multi-cores. We continue in the next section with the organization of this thesis.

1.2 Thesis outline

This thesis continues with the presentation of the literature survey in Chapter 2. We introduce and describe in detail the micro-architecture of our dynamic heterogeneous multi-core, Bahurupi, in Chapter 3. Here, we begin by describing the execution model and the sentinel instruction format, then we present the architectural details — the live-in and live-out registers renaming process, the branch misprediction and hazard recov-

ery mechanisms, the compiler support and then we present the experimental evaluation that validates the micro-architectural design. In this chapter, we also include the FPGA prototype implementation of Bahurupi, emphasising on the challenges experienced when implementing such an architecture on FPGA, and the impact on the area and the latency. In Chapter 4 we present the architectural details of the first level of data cache, which is able to accommodate the memory requirements of Bahurupi. Here, we start by analysing the positive and negative aspects of choosing an alternative solution — a multi-ported shared cache. Then, we show the details of our reconfigurable solution which can offer the same benefits of the shared cache but with the lowest power and area overheads.

In Chapter 5 we describe our performance limit study of Bahurupi multi-core architecture by introducing task scheduling on dynamic heterogenous multi-cores. Here, we consider a realistic approach where both serial and parallel tasks coexist in the system. We begin by introducing an optimal static scheduler for ideal dynamic heterogeneous multi-cores with no constraints. Then, we adapt this scheduler for the constraints that Bahurupi imposes. We continue by designing an online scheduler for Bahurupi and we end with a quantitative evaluation where we clearly show that dynamic heterogeneous multi-cores can perform much better than the homogeneous and static heterogeneous multi-cores. The thesis ends with conclusions and future work in Chapter 6.

Chapter 2

Related Work

2.1 Heterogeneous multi-core architectures

As we have shown in Chapter 1 heterogeneous multi-core architectures can be classified into two types: static heterogeneous multi-core architectures and dynamic heterogeneous multi-core architectures. In this section we present the related works on these two types of new architectures. We also comparatively present the differences between other dynamic heterogeneous multi-core architectures and our proposed Bahurupi architecture.

2.1.1 Static heterogeneous multi-core architectures

The static heterogeneous architecture was initially proposed by Rakesh Kumar et al. in [57] and [55]. Heterogeneous multi-cores were built by connecting together cores with different size and performance parameters. It was shown that the strong points of these new architectures are the low power consumption and performance that are achieved by dynamically moving the programs from one core to another in order to reach an optimal tradeoff point. The study in [55] introduces and seeks to gain some insights into the energy benefits available for these new architectures. The particular

opportunity examined is a single application switching among cores to optimize the energy consumption and the performance. This work shows that there can be great advantage to diversity within an on-chip multiprocessor, allowing that architecture to adapt to the workload in ways that a homogeneous multi-cores cannot. A static multi-core heterogeneous architecture can support a range of execution characteristics not possible even in an adaptable single-core processor, even one that employs aggressive power gating. Such an architecture can adapt not only to changing demands in a single application, but also to changing demands between applications, changing priorities or objective functions within a processor or between applications, or even changing operating environments. The work in [57] evaluates a variety of static heterogeneous architectural designs, including processor cores that support Simultaneous Multithreading (SMT). It shows that this approach can provide significant performance advantages for a multiprogrammed workload over homogeneous chip-multiprocessors.

Another static heterogeneous multi-core example was provided by Aater Suleman et al. in [91]. This work targets to overcome the serialization effect of a critical section for multi-threaded application. In some multi-threaded applications, threads do not usually finish their jobs in the same time (unbalanced). Thus, the synchronization mechanisms can make a considerable number of fast jobs threads wait for the slow jobs threads. This work proposes that processors should have a dedicated powerful core that would help accelerate the slow job threads in order to reach their deadline faster.

Asymmetric Cluster Chip Multi-Processing [66] proposed by Tomer Morad et al., is another variation of the initially proposed static heterogeneous multi-cores in which the cores have the same ISA with different areas, but they can have a completely different micro-architecture which translates into different performance capabilities. If we consider the current static heterogeneous multi-core architectures (e.g., ARM big.LITTLE [41] existing on the market), we can see that this work was predicting well the future architectures. Now, the trend is to include on the same die heterogeneous cores with

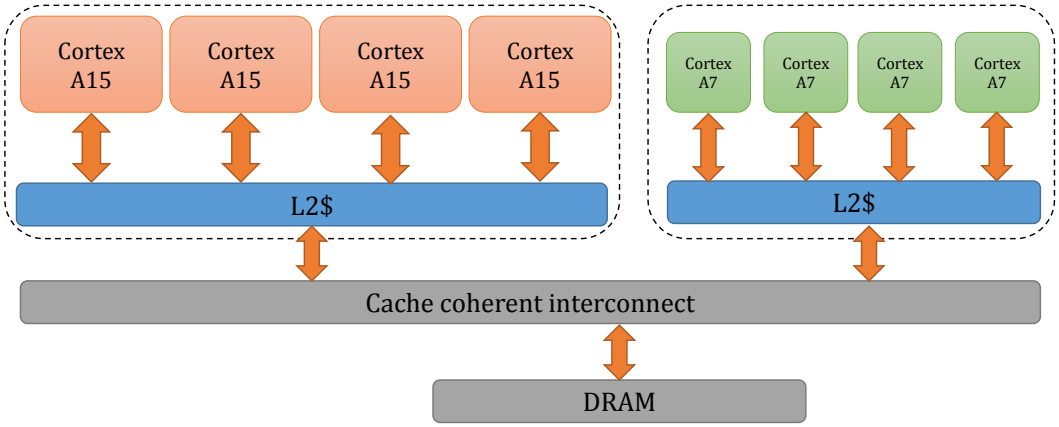


Figure 2.1: big.LITTLE static heterogeneous multi-core architecture.

the same ISA but with very different micro-architectures. In Figure 2.1 we show the architecture of the big.LITTLE static heterogeneous multi-core architecture that exists currently on the market. It consists of high performance Cortex-A15 cluster and power efficient Cortex-A7 cluster. All the cores implement ARM v7A ISA. The Cortex-A15 is complex out-of-order superscalar core that can execute high intensity workloads, while Cortex-A7 is a power efficient in-order core meant for low intensity workloads. While each core has private L1 instruction and data caches, the L2 cache is shared across all the cores within a cluster. The L2 caches across clusters are kept seamlessly coherent via a cache coherent interconnect.

Balakrishnan et al., performed a study in [8] to explore the impact of the static heterogeneity in the emerging multi-core architectures from software developers perspective. The results show that static heterogeneous multi-core architectures introduce a high degree of unpredictability for the software applications. Performance heterogeneity in multi-core systems breaks a long-standing assumption made by multi-threaded application developers. These developers typically assume all computational cores provide equal performance when they write their parallel algorithms and applications. However, this study investigates the impact, if any, of computational heterogeneity on the behavior of multi-threaded applications. This work tries to answer some important

questions, for example, does computational heterogeneity result in unpredictable performance characteristics in a commercial server, which must meet certain performance guarantees? Does the heterogeneity expose a scalability problem in applications that otherwise would not have manifested? Computer architects should consider the implications of multi-core proposals on application behavior, and the developers must design applications that are robust enough to dynamically deal with changing compute power.

2.1.2 Dynamic heterogeneous multi-cores

Considerable previous work has been done in order to propose an architecture that will efficiently adapt multi-cores to the continuously changing demands of the applications. The idea of composing simple cores together to accelerate serial code has been explored before. However, such solutions either require significant additional shared hardware resources (e.g., Rakesh Kumar et al. with *Conjoined-Core Chip Multiprocessing* [56]) and modifications to the internal architecture of the composable cores (e.g., Engin Ipek et al. with *Core Fusion* [48]) or follow a radically different instruction-set architecture that require complete re-engineering of the software model and aggressive compiler optimizations (e.g., Changkyu Kim et al. with *TFlex* [52]).

The Core Fusion architecture [48] can fuse homogeneous cores and presents a detailed hardware solution to support adaptive core fusion. The proposed architecture has a reconfigurable, distributed front-end and instruction cache organization that can leverage individual core's front-end structure to feed an aggressive fused back-end, with minimal over-provisioning of individual front-ends. Comparing to Bahurupi design, Core Fusion uses more complex hardware which brings the most impact on the performance. It also implements a complexity-effective remote wake-up mechanism that allows operand communication across cores without requiring additional register file ports, wake-up buses, bypass paths, or issue queue ports. Comparing to Core Fusion, our design does not use a distributed ROB organization and it uses a much simpler

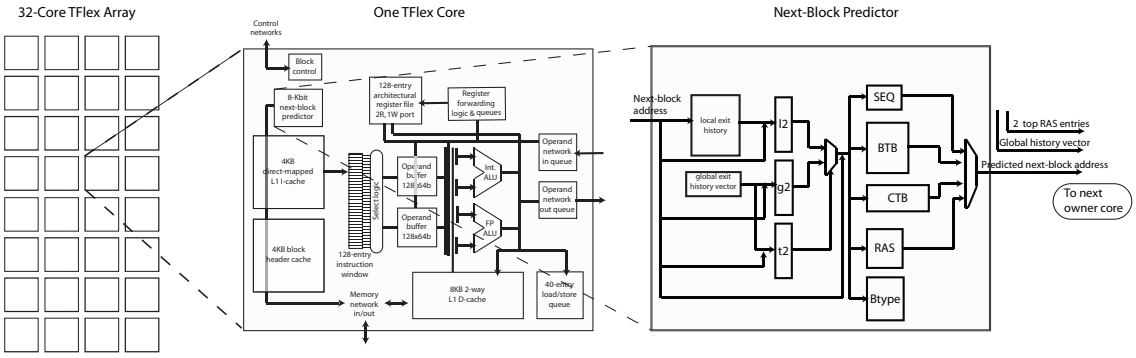


Figure 2.2: TFlex micro-architecture and internal organization [52].

global renaming table which does not have to track the mapping done by the individual cores and also it does not bypass the internal core renaming logic.

TFlex processor [52] uses no physical shared resources among the cores which would make it a very good alternative to our design. Instead, TFlex is dependent on a special distributed micro-architecture called Explicit Data Graph Execution (EDGE) which is configured to implement the composable lightweight processors. Bahurupi, on the other hand, can be implemented on top of conventional CISC and RISC architectures. EDGE ISAs creates programs in which the instructions are grouped in consecutive blocks of instructions. This approach is similar to ours in which we use the liveness information provided by the compiler to create special instructions (sentinel instructions) that will separate the basic blocks and carry the dependency between them. Computing liveness is a basic step done by an ordinary compiler in order to implement optimisation techniques. TFlex EDGE ISA adds complex information and book-keeping to the block structures and it relies on point-to-point communication to exchange information between cores. In Figure 2.2 we show the internal micro-architecture and organization of a 32 core TFlex architecture. The figure also shows the internals of the next block predictor. This predictor predicts the next block of instructions which will be fetched and executed.

David Tarjan et al. proposed *Federation* [93] as an alternative approach. Federation

makes a pair of scalar cores to act as a 2-way out-of-order core by adding additional stages to their internal pipeline (shown in Figure 2.3). The key insight that makes federation work is that it is possible to approximate traditional out-of-order issue with much more efficient hardware structures, replacing CAMs and broadcast networks with simple lookup tables; and that these out-of-order structures can be placed between a pair of scalar cores and use the fetch, decode, register file, cache, and datapath of the scalar cores to achieve an ensemble that is competitive in performance with an out-of-order superscalar core. Federated cores are best suited for workloads that usually need high throughput but sometimes exhibit limited parallelism. Federation provides faster, more energy-efficient cores for the latter case without sacrificing area that would reduce thread capacity for the former case.

Other works approached the idea of dynamic heterogeneous multi-core systems together with code parallelization as in the case of the *Voltron* processor proposed by Hongtao Zhong et al. in [104]. As shown in Figure 2.4, Voltron uses multiple homogeneous cores that can be adapted for single and multi-threaded applications. The cores can operate in coupled mode when they act as a VLIW processor that will help exploit the hybrid forms of parallelism found in the code. Voltron relies on a very complex compiler that is able to exploit parallelism from the serial code, partition the code into small threads, schedule the instruction to the cores and direct the communication between the cores. In this case, the cores are only passing values among themselves on specialised buss. Code parallelization was shown to be a tedious process that can introduce errors as well. Bahurupi does not use a special unit which is orchestrating the instructions to the cores. It only using a lock-unlock mechanism over a shared register which holds the address of the next basic blocks.

Rakesh Kumar et al. give another interpretation of the core coalition in *Conjoined-Core Chip Multiprocessing* [56]. That is, nearby cores can share functional units (e.g., FP units, crossbar ports, instruction caches, data caches etc.).

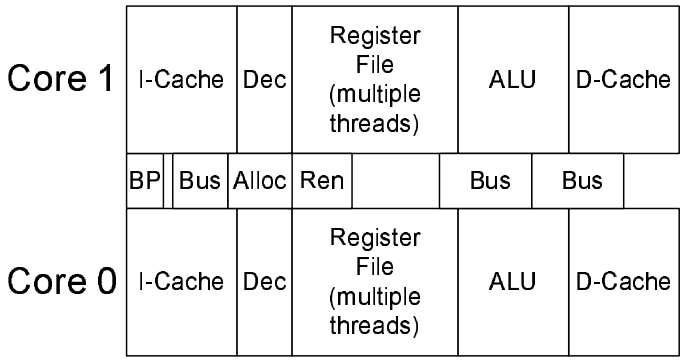


Figure 2.3: Federation architecture [93].

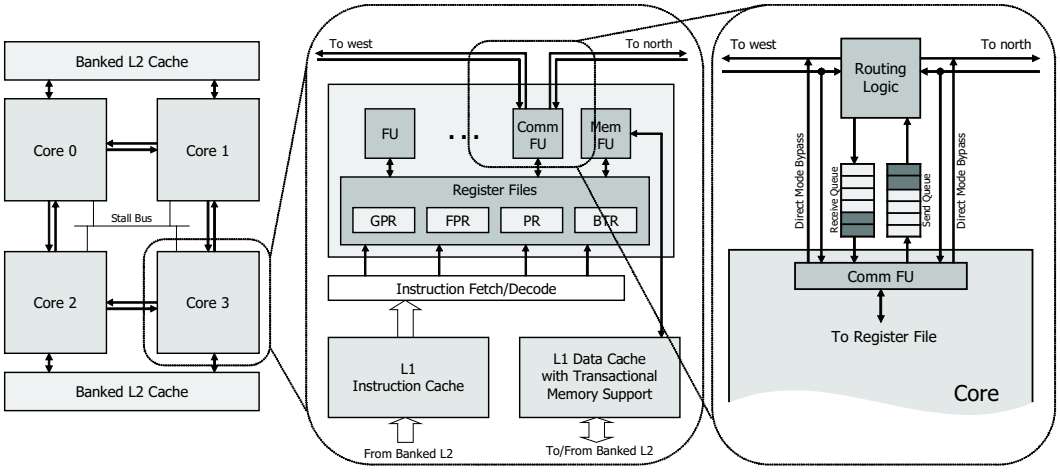


Figure 2.4: Block diagram of a 4-core Voltron architecture [104].

The idea of using special instructions that carry the liveness information between blocks of code with similar format was proposed by Sriram Vajapeyam et al. in [95]. Using trace descriptors, this design allows processors to go beyond basic block limits in program order. This architecture uses a shared ROB, a shared fetch unit and very simple execution units (that have only fetch, issue and execute stages) instead of ordinary cores. Bahurupi, on the other hand, does not need major modifications to the existent commonly used structures. As in our case, the design uses a shared renaming map for the live-out and live-in registers.

Recently, Khubaib et al. have proposed *MorphCore* architecture [51]. Morphcore

is an adaptive core that is created by starting with a traditional high performance out-of-order core and making internal changes to allow it to be transformed into a highly-threaded in-order SMT core when necessary. The main idea stems from the fact that in general, the architecture designers build two types of cores: (a) powerful out-of-order cores capable of exploiting high ILP from the sequential code (e.g., Intel i7, ARM Cortex-A15) but they are power inefficient when exploiting the TLP or (b) low power in-order cores that are very good in exploiting the TLP from parallel applications but they lack the capability to offer good performance to single-threaded applications. Usually big out-of-order cores consume much more power than the small in-order cores; but as they finish the execution of the program much faster, they can offer better energy results or better energy-delay product (EDP). EDP is a metric worth to be considered especially in the field of mobile computing where users are interested in applications that run fast with the lowest energy consumption. MorphCore brings modifications to the internal pipeline of an out-of-order core by adding components that allow fast switching between out-of-order mode and in-order mode. The fetch stage is modified such that it can switch between 8 threaded in-order SMT core and a dual-issue out-of-order core. The decision to switch between execution modes is automatically taken care by the hardware system and not by the operating system. Generally, when the OS spawns more than two tasks (threads), the hardware switches to SMT mode and when the number of threads reduce to less than two, then the hardware switches to out-of-order mode.

Andrew Lukefahr et al. proposed *Composite Cores* architecture [64] that uses a similar concept as MorphCore. It reduces switching overheads by creating heterogeneity within a single-core. The proposed architecture pairs simple and complex pipeline engines together inside a single chip. Essentially, there are two different pipelines connected together on the same CPU die – an out-of-order pipeline and an in-order pipeline. The connectivity between these two allows fast migration of processes from one engine

Related Works	Programming model		Architecture		
			Type of merging and shared resources between the basic cores		
	Special ISA	Special compiler	Core level coalition	Core adaptation	Shared resources between cores
Core Fusion [48]			X		X
TFlex [52]	X	X	X		
Federation [93]				X	
Voltron [104]		X			X
MorphCore [51]				X	
Composite Cores [64]				X	
Bahurupi			X		X

Table 2.1: Comparison between different dynamic heterogenous multi-core architectures.

to another. The two engines share the front-end of the pipeline, the branch predictor and the instruction and data caches. An extra hardware component is added to the system – a reactive PID controller that is in charge of detecting when to migrate from one pipeline to another. The online controller tries to minimize the energy saving by choosing the right core configuration at runtime. The controller integrates a complex performance estimator to decide where the task will be migrated.

The literature presented so far confirms that more research must be dedicated in order to efficiently create dynamic heterogeneous multi-core systems. Bahurupi is an alternative to these proposed designs. It not only reduces the production costs and increases performance, but it also helps reduce the power consumption and ease software

development.

We summarise this section by offering in Table 2.1 a high-level comparison among the existing dynamic heterogeneous multi-core solutions. Architectures like *MorphCore* or *Composite Cores* need core adaptations as they heavily modify the internals of the core in order to easily switch the applications between different types of pipelines. As shown in this table, Core Fusion is closest to Bahurupi architecture. However, the amount of shared resources between the cores and the complexity of the hardware is significantly more in Core Fusion compared to Bahurupi.

2.2 Reconfigurable caches

In this section, we present related work on reconfigurable caches with special consideration for the connectivity between the cores and the first level of caches.

There is a general consensus in the computer architecture community that the L1 instruction and data cache must be private to the core in order to accommodate the frequent accesses to the data or instructions. Moreover, there has been extensive research in the organization and reconfiguration of the lower level of caches starting from private L2 [28, 12, 79] to shared L2 or adding private or shared caches at lower levels (e.g., L3) [90, 53, 47, 82, 81, 80, 49]. However, computing systems like GPUs need high bandwidth and low-latency access to the first level of memory (i.e., tightly coupled scratchpad memories) [72] for which very little research has been done. Similarly, dynamic heterogeneous multi-cores like Bahurupi need to share the first level of cache. When coalescing cores, Bahurupi generates more memory accesses per cycle that puts high pressure on the first level of cache. Traditional bus-based interconnects or crossbars are not able to offer this level of bandwidth and low latency even if advanced techniques are used (e.g., out-of-order completion or multiple outstanding transactions) [3][1]. Similarly, network-on-chips [13][34] are well positioned for ac-

commodating the bandwidth requirements, but their increased latency [18] makes them impractical for being used as an interconnect between the core and first level of cache.

The recently proposed dynamic heterogeneous multi-core designs presented in the previous section focus mostly on the internal micro-architecture, the compiler, the programming model and the execution model. They mostly ignore or make simplifying assumptions regarding the memory hierarchy. A common assumption in all these works is that the first level of data and instruction caches must support reconfigurability. However, none of the designs delve into the details of the reconfigurable cache architecture. Core Fusion [48] uses a reconfigurable instruction cache that can merge each core's private cache into a fused group. In coalition mode, each cache keeps a replica copy of tags, and broadcast is needed as the cache line is distributed across the participating caches. This incurs area and latency overhead. As for data cache, Core Fusion keeps the data cache private, which can have poor performance in coalition mode. Similarly, Federation [93] employs a recongurable L1 instruction and data cache. In coalition mode, all the cache banks behave as a large set-associative shared cache without analysing the overheads introduced by composing the L1 caches. TFlex [52] also makes its L1 caches composable when multiple cores are fused. However, it does not offer any details about the interconnection network between the cores and the L1 caches. Even in the case of MorphCore and Composite Cores, the system still needs reconfigurable caches as the big cores require more throughput to the first level of cache.

Recent proposals on reconfigurable caches seek to make caches dynamically adapt to applications and power requirements [14][40] [89][97][103]. However, they generally focus on overcoming the non uniform cache access (NUCA) effects or increasing or decreasing the available cache size or associativity without considering the required cache access throughput [14][40]. Similarly, interconnection networks required for cache resizing are often simplified or ignored [89][97]. MorphCache proposed by Shekhar Srikantaiah et al. in [89] can dynamically merge and split its L2 and L3 cache

slices. However, it uses a simple segmented bus to connect or isolate adjacent cache slices. A miss on a local L2 (or L3) cache is put on the segmented bus to be delivered to all shared L2 (or L3) caches. This broadcast scheme and the segmented nature of the bus can lead to frequent multi-hop bus transactions that result in additional performance overhead. Molecular Caches proposed by Keshavan Varadarajan et al. in [97] dynamically reconfigures the cache to accommodate diverse application requirements. It varies the size of cache and cache line, associativity and cache replacement policy to achieve power efficiency. However, it does not offer detailed information about the intra tile (group of molecules) interconnection.

Sharing the L1 cache is a good design point for tightly coupled multi-core processors. In such systems, the high availability of TLP can be limited by the memory access latency, especially due to the interconnection network latency between the processing cores and L1 memory units. For this reason, a new Mesh-of-Trees (MoT) implementation of interconnection network was recently proposed, which outperforms the traditional networks in terms of area, bandwidth and latency [10]. Based on this network, Abbas Rahimi et al. proposed in [84] a parametric, fully combinational *logarithmic interconnection network* to support high-performance, single-cycle communication between processors and multi-banked, tightly coupled L1 data cache. In Figure 2.5 we present a logarithmic network that connects four cores with eight memory banks. As we can see there is an unique path from each core to the arbitration layer. The network comprises of two layers: a pure combinational logarithmic routing layer and an arbitration layer. The basic routing elements are switches and the arbiters simply arbitrate in case of multiple accesses going to the same bank of memory.

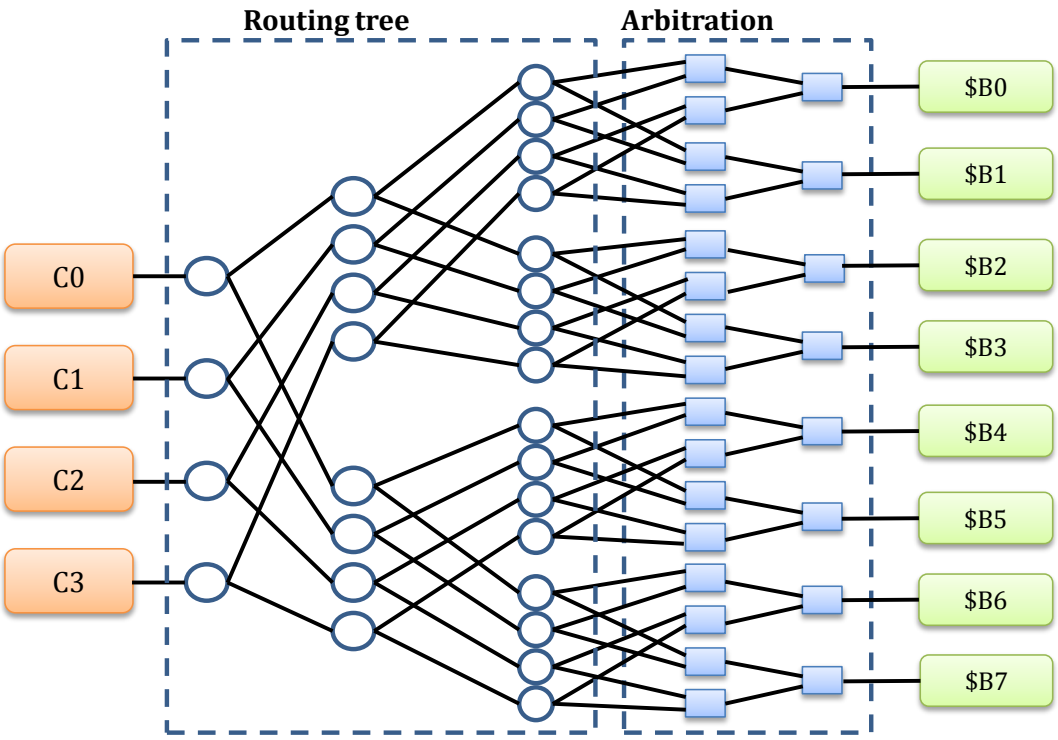


Figure 2.5: Example of a logarithmic interconnection network.

2.3 Task scheduling

In this section, we cover the related work on task scheduling on heterogeneous multi-core systems. Scheduling on such systems is different from the classic scheduling problem on homogeneous multi-cores as now the system must not only smartly allocate the tasks to the cores but also merge the cores if necessary, which turns out to be a challenging task.

An early method to dynamically allocate threads on static heterogeneous multi-core systems is presented in [11] by Michela Becchi and Patrick Crowley. In order to take advantage of a heterogeneous architecture, an appropriate policy to map running tasks to processor cores must be determined. The overall goal of such a strategy must be to maximize the performance of the whole system by accurately exploiting its resources. The control mechanism must take into account the heterogeneity of the system, the

workload, and the varying behavior of the threads over time. Moreover, it must be easily implementable and introduce as little overhead as possible. This work shows that a heterogeneous system adopting a dynamic assignment policy is able to accommodate a variety of degrees of thread-level parallelism more efficiently than a homogeneous multi-core.

Scheduling only sequential applications on dynamic heterogeneous multi-core architectures is studied in [42] by Divya Gulati et al. where different algorithms for static and dynamic scheduling are proposed. However, this work is built on top of the TFlex [52] adaptive architecture that we have seen requires a special ISA (EDGE) configured to support distributed execution of sequential applications. Moreover, this work proposes an optimal static scheduling algorithm with high time complexity, $O(n\ m^2)$, where n is the number of applications and m is the number of cores in the system.

For our scheduler presented in this thesis we first model the applications as independent preemptive *malleable* tasks. Scheduling malleable tasks has recently received significant attention. Malleable tasks are parallel tasks that may be processed simultaneously by a number of cores, where the processor speedup of the task is dependent on the number of allocated cores. Malleable tasks are allowed to preempt and change the number of cores during execution. Scheduling malleable tasks is a promising technique for gaining computational speedup when solving large scheduling problems on parallel and distributed computers [23, 102]. Real applications for malleable tasks have been presented among others in [15] for simulating molecular dynamics, in [35] for Cholesky factorization, in [19] for operational oceanography and in [20] for berth and quay allocation.

The malleable task model was first proposed in [94] by John Turek et al. and later studied in [63, 99] and [67]. Scheduling independent malleable tasks without preemption is proved to be NP-hard [36] and related work on this topic focus on finding sub-optimal solutions. John Turek et al. found a polynomial λ-approximation

algorithm for the malleable tasks problem starting from any λ-approximation algorithm for the 2D bin-packing problem. Following this work, Walter Ludwig presents a two-approximation algorithm in [63] and Gregory Mounie et al. developed a heuristic in [67] with a worst case performance guarantee of $\sqrt{3}$, which was later improved to $\frac{3}{2}$ in [68]. Additionally, scheduling malleable tasks on clusters of multi-cores is proposed by Pierre-Francois Dutot and Denis Trystram in [37] where allocation of tasks to clusters is also considered.

Operating with malleable tasks presents significant challenges for the scheduling system. In our thesis, we also use a variation of the malleable model called the *moldable* model. Moldable tasks are parallel tasks that can be executed using an arbitrary number of cores but they cannot change the core allocation during execution. Similarly, their performance is directly related to the number of allocated cores. Suboptimal solutions for scheduling moldable tasks have been studied in [21], [22] by Jacek Blazewicz et al. and in [29] by Guan-Ing Chen and Ten-Hwang Lai.

Chapter 3

Bahurupi Adaptive Multi-Core

In this chapter we present a dynamic heterogenous multi-core architecture called Bahurupi, that can dynamically create static heterogeneous configurations in order to better accommodate the software requirements.

Motivating Example. As a concrete motivating example, we present here a case study of *Ferret* benchmark from PARSEC suite [16] that performs image similarity search. The application consists of six kernels. We first run the sequential version of the application on one complex core configured as 2-way, 4-way, and 8-way out-of-order execution engines, respectively, using MARSS [73] simulator. We collect execution time for each individual kernel. Next we create 2-core and 4-core homogeneous multi-core systems where each core is a 2-way out-of-order engine. For 2-core system (4-core system), we create 2 threads (4 threads) for each of the kernels except for *load* and *out*, which are hard to parallelize. Now we run this multi-threaded application on multi-core and collect execution time for each kernel. Figure 3.1 shows the speedup trend for each kernel as we increase ILP and TLP normalized w.r.t. 2-way core. *seg* and *rank* can benefit from TLP while *extract* and *vec* benefit from ILP.

For the whole application, the speedup from homogeneous configurations (4-way

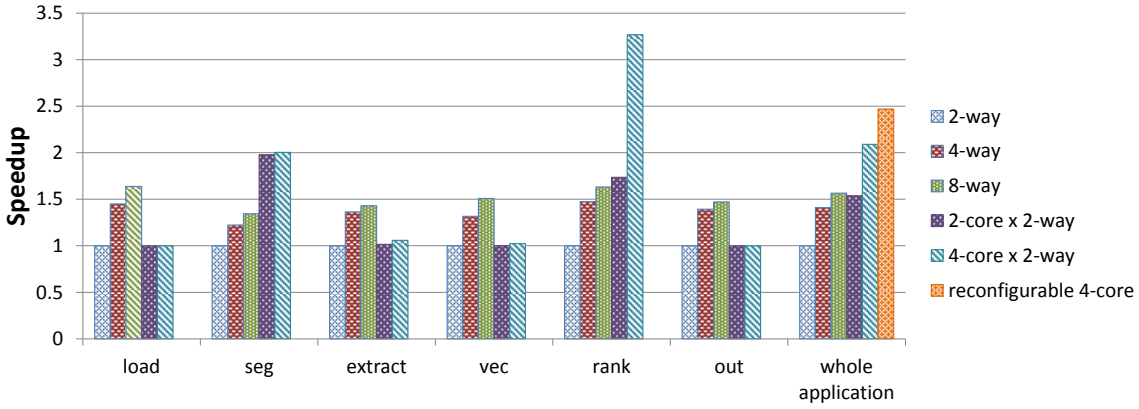


Figure 3.1: Speedup trends for Ferret kernels and overall speedup with reconfigurable architecture.

out-of-order, 8-way out-of-order, 2x2-way core, and 4x2-way core) are shown on the right. None of these configurations can exploit both ILP and TLP. A 4-core dynamically reconfigurable architecture like Bahurupi can run *seg* and *rank* on 4x2-way cores to exploit TLP, while the rest of the kernels can be run on virtual 8-way out-of-order engine by forming coalition of 4 cores. Thus, the 4x2-way reconfigurable architecture improves the speedup by 38% compared to static homogeneous 4x2-way cores. This case study confirms once more that dynamic reconfigurable architectures, such as Bahurupi, that can seamlessly transition between ILP and TLP can provide significant performance boost to applications in comparison to homogeneous multi-cores.

3.1 Bahurupi execution model

Bahurupi reconfigurable multi-core architecture allows cores to form coalitions so as to improve single-thread performance. A *coalition* is defined as a group of cores working together to accelerate the execution of a serial stream of instructions. In normal mode, Bahurupi executes multi-threaded application on a set of homogeneous cores. One simple core might not be powerful enough to exploit the amount of ILP [98] available in some threads. In that scenario, Bahurupi architecture configures a subset of its cores

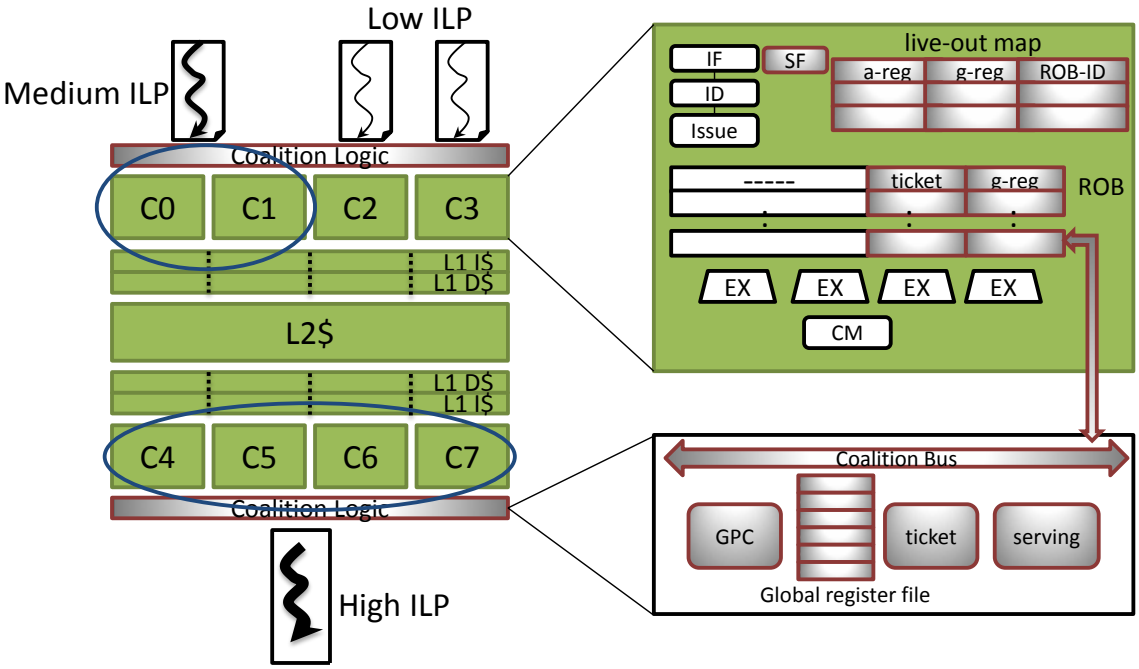


Figure 3.2: Bahurupi architecture. Additional resources required for coalition are highlighted.

to run in coalition mode so that the virtual cores can extract more ILP and implicitly execute the threads faster. The design uses limited amount of additional hardware that is shared among the cores and minimal compiler modifications.

In Figure 3.2 we show the high-level architecture of Bahurupi together with the additional resources required for coalition. Here, we present two four-core clusters sharing the last level of L2 unified cache. In each cluster, Bahurupi allows at most one coalition of cores to be formed. The number of cores found in a coalition can be decided at runtime by the run-time system. In this figure, we show an example of two low ILP threads being spawned on the cores C2 and C3, a medium ILP application being scheduled on a coalition of two cores (C0 and C1) and a high ILP application being scheduled on a coalition of four cores (C4-C7).

When found in coalition, the cores are connected to the *Coalition Logic* which comprises of *Coaltion Bus, Global Register File and Renaming Logic, GPC, Ticket and*

Serving registers. The *Coalition Bus* allows the cores to pass the values of live-in and live-out registers from one core to another or to the *Global Register File*. The *Renaming Logic* takes care of renaming the live-in and live-out registers to allow out-of-order execution of basic blocks between coalesced cores. The *GPC* register contains the address of the next basic block to be fetched by a core found in a coalition. Finally, the *Ticket* and *Serving* registers help to commit the instructions from different basic blocks in program order — when renamed, a basic block is given a *Ticket* value such that all the instructions are labeled with this *Ticket* value. At commit time, the instructions are only allowed to be committed if their *Ticket* value is equal to the *Serving* value. Besides the *Coalition Logic* components, there are few other internal components (e.g., *live-in map, live-out map, ROB-ID*) which will be explained in detail in the following sections.

Bahurupi architecture follows a distributed execution model in coalition mode. The unit of execution for a core is a basic block — a sequence of instructions with single entry and single exit point. A core fetches and executes one basic block of instructions at a time. Our goal is to execute the basic blocks in parallel on the cores that form coalition and thereby achieve speedup for serial code.

Bahurupi execution model is similar to thread pool pattern in parallel computing where a number of threads are created to execute a number of tasks. The number of tasks is usually much more compared to the number of threads. As soon as a thread completes its task, it requests the next ready task until all the tasks have been completed. In Bahurupi architecture the cores correspond to the threads and the basic blocks correspond to the tasks. The cores fetch basic blocks for execution. As soon as a core completes fetch, decode, and rename of all the instructions in a basic block, it attempts to fetch the next available basic block.

To achieve the execution model of Bahurupi, we need to first resolve register and memory dependencies among the basic blocks so as to maintain correctness during parallel execution. Instead of relying on a complex hardware mechanism to detect inter-

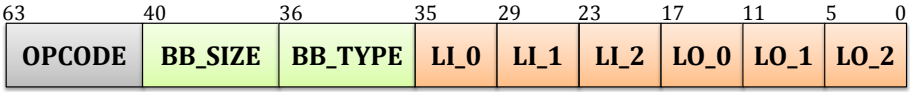


Figure 3.3: The sentinel instruction format.

dependency among the basic blocks, we resort to a hardware-software co-operative solution. Second, we need to ensure that the cores fetch, rename and commit the basic blocks in program order. The execution of the instructions from different basic blocks can be performed out-of-order and the main speedup of Bahurupi comes from this out-of-order parallel execution of instructions from different basic blocks on different cores.

3.1.1 Sentinel instruction

We let the compiler detect the live-in and live-out registers corresponding to each basic block. The term *live-in register* indicates a register that is alive at the entry of a basic block and is actually used inside the basic block. The term *live-out register* stands for a register that is alive at the exit of a basic block and is actually updated inside the basic block. The live-in and live-out registers correspond to inter basic block dependencies. It is possible for the same register to appear as both live-in and live-out register. Now we need to communicate the live-in, live-out information to the hardware architecture. We introduce a new instruction, called *sentinel*, to encode this information. The compiler adds a sentinel instruction in the beginning of each basic block. That is, the compiler splits the program into basic blocks which are delimited by sentinel instructions.

Bahurupi design can be applied to any ISA that can be extended with the sentinel instruction. Figure 3.3 depicts the format of a sentinel instruction. Our design assumes 64-bit instruction format. The BB_SIZE field specifies the length of the basic block which is delimited by this sentinel instruction. We set this field to 4 bits, that is, we can support at most 16 instructions in a basic block. Experiments show that on average the basic blocks do not contain more than 6 instructions. In the rare case when the size

of a basic block exceeds 16 instructions, it is split into two or more basic blocks. The BB_TYPE is a 1-bit field that specifies if the basic block ends with a branch instruction or not.

The next six fields hold the live-in and live-out registers for the basic block. We decide to use three live-in and three live-out registers after evaluating multiple benchmarks. Figure 3.4 depicts the percentage of basic blocks with number of live-in and live-out registers below certain thresholds for some SPEC and embedded benchmarks. For almost all benchmarks, 90% of basic blocks contain less than or equal to 3 live-in and live-out registers. We believe that 3 live-in and 3 live-out slots are enough but one can add two more slots for live-in and live-out respectively as there are enough bits left from the opcode field. If a basic block contains more than 3 live-in or live-out registers, the compiler splits the basic block into two or more sub-blocks so as to satisfy the constraint. The size of a live-in and live-out field is 6 bits for an ISA with 32 integer registers and 32 floating point registers. If an ISA does not offer enough bits to encode the live-in and live-out registers, we can set aside a limited number of registers as global live-in, live-out registers. This is similar in spirit to using specific registers to transfer parameters during subroutine calls. The compiler has to perform register swapping to map all live-in, live-out registers to these specific set of registers. We then only need to encode which subset of registers from this specific set has been used in a particular basic block.

Figure 3.6 shows the sentinel instruction for a basic block with five instructions. It has one live-in register $r5$ and one live-out register $r4$. Notice that $r5$ is defined locally as well by instruction $I2$. Similarly, $r4$ is defined multiple times ($I0$ and $I3$); but the last definition in $I3$ is treated as live-out.

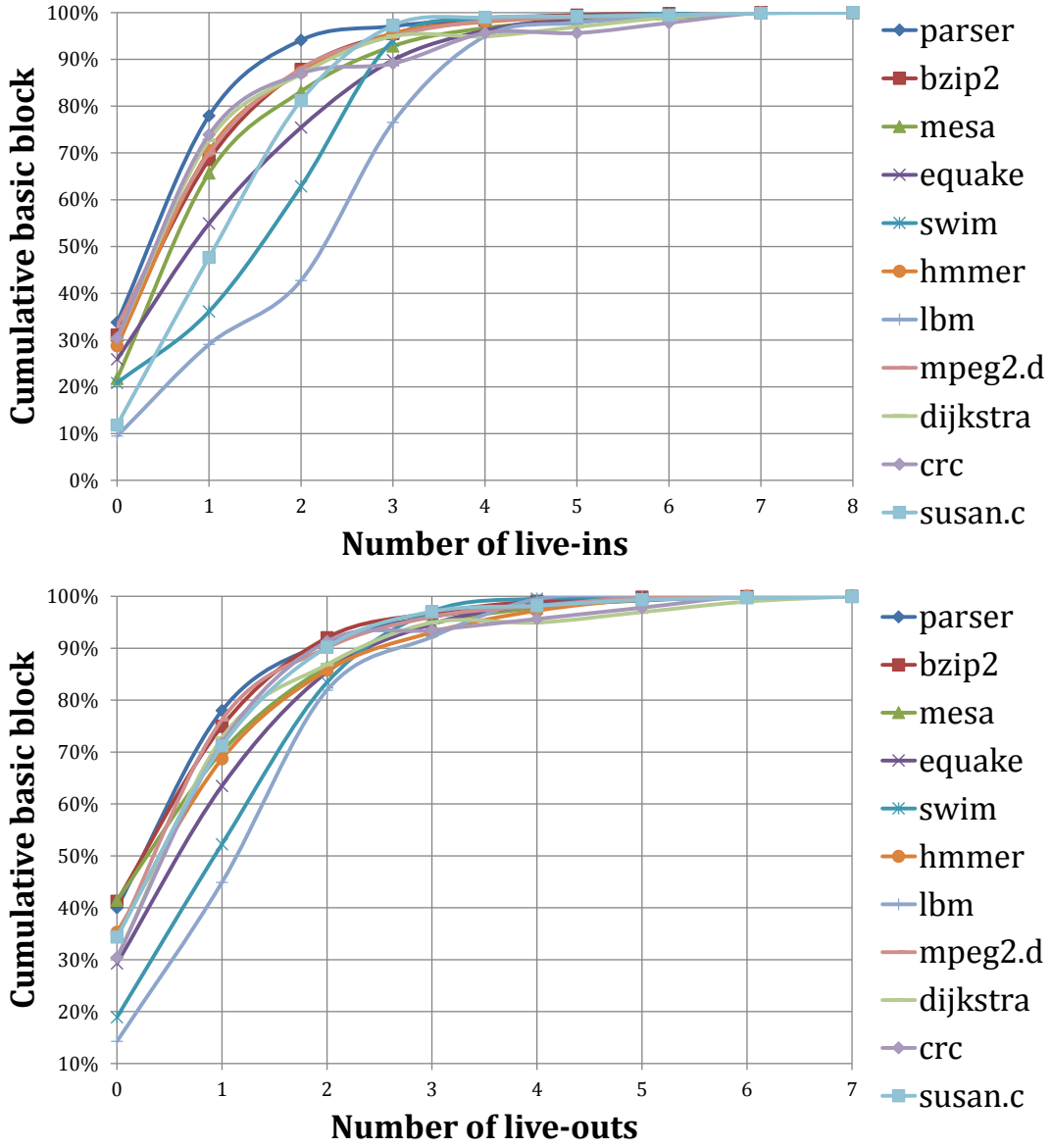


Figure 3.4: Percentage of basic blocks with number of live-in and live-out registers below a threshold.

3.1.2 Execution model

In our architecture, the base processing cores are 2-way out-of-order execution engines. They use register map table to rename architectural registers into physical registers. The registers local to a basic block can remain in the core-local register file. However, the live-in and live-out register values have to be communicated among the cores. So we introduce a global register file that is shared across the cores. This includes a global map table and a global physical register file. The live-in registers have to be read from the global register file, while the live-out registers have to be written into the global register file. Thus live-in and live-out registers have to be renamed using the global map table. The sentinel instructions take care of global register renaming.

Any out-of-order execution core should rename registers in program order as well as commit instructions in program order. As the global (inter basic block) dependencies are encapsulated in the sentinel instructions, we only need to ensure that the sentinel instructions execute and rename global registers in program order. We satisfy this program order requirement by using a global program counter (GPC) that is shared across the cores (see Figure 3.2). The GPC also comes with a locking mechanism such that only one core can access and update the GPC at any point of time [33]. Initially the GPC points to the sentinel instruction in the very first basic block of the program. Once a core fetches the sentinel instruction and completes global register renaming, it updates the GPC to point to the next basic block (i.e., the next sentinel instruction). If the basic block ends with a branch instruction, the GPC will be updated with the predicted target address generated by the branch predictor in the core. Otherwise, the GPC is incremented by the length of the basic block (the length information is encoded in the sentinel instruction). In other words, the GPC always points to a sentinel instruction. After updating, the core releases the lock on the GPC allowing the next core (or the same core) to lock the GPC register. Now the core starts fetching the instructions from

the basic block and then executes them out-of-order whenever their operands are ready (i.e., all the local and global dependencies are resolved).

The in-order commit constraint is to handle speculative execution and precise exception. The in-order commit requirement is handled through a shared ticket lock mechanism. The ticket lock contains two registers: *serving* and *ticket* (see Figure 3.2). Both are initialized to 0. The *ticket* register is used in order to keep track of the order in which the basic blocks are fetched by the cores. When a core locks the GPC, it also reads and increments the current value of *ticket* register. It then tags the reorder buffer (ROB) entries of all the instructions in the basic block with this ticket value. That is, each basic block of instructions is tagged with a unique ticket value and the ticket values are assigned to basic blocks in program order.

The *serving* register dictates which set of instructions are allowed to be committed. At any point in time only one core is permitted to commit instructions. That is the core for which the instructions are ready to be committed and their associated ticket number matches the value held by the *serving* register. The *serving* register is incremented after all the instructions from the basic block are committed. This process ensures that the basic blocks are committed in order.

Example. Figure 3.5 illustrates Bahurupi execution model with an example. The left hand side of the figure shows a simple control flow graph (CFG) corresponding to a program. This CFG contains five basic blocks B0–B4. In the beginning, the global program counter (GPC) points to the sentinel instruction of B0. Let us assume that core 0 manages to get a lock on GPC first. It fetches the sentinel and renames the global registers according to live-in and live-out information. The sentinel also indicates that basic block B0 ends in a branch instruction. Therefore core 0 performs branch prediction, which indicates B1 as the predicted next basic block. So GPC is updated to point to the sentinel of B1 and core 0 releases its lock on GPC. During this period core

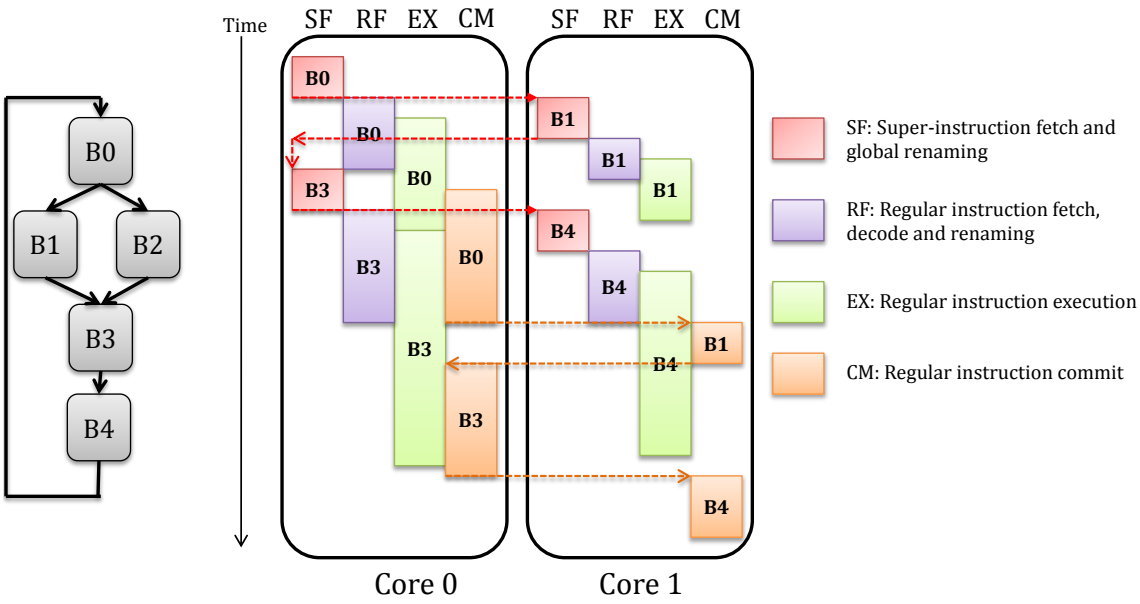


Figure 3.5: Bahurupi distributed execution model: (a) Control flow graph (CFG) of a program, and (b) Execution of the CFG on 2-core Bahurupi architecture.

1 is sitting idle as it cannot obtain a lock on GPC.

Now core 0 starts fetching, decoding, renaming, and executing the regular instructions from basic block B0. Meanwhile, as GPC has been released, core 1 locks GPC and renames global registers corresponding to basic block B1. As B1 does not have any branch instruction in the end, GPC is incremented by the length of B1 and now points to B3. Core 1 also releases the lock on GPC. At this point, both core 0 and core 1 are fetching, renaming, and executing instructions from their corresponding basic blocks.

When core 1 releases the lock, both the cores are still busy fetching instructions from their respective basic blocks. Therefore, none of them attempt to lock the GPC. Only when a core completes fetching all the instructions from its current basic block, it will proceed to get the lock for the next basic block. This is the reason why there is a gap between the completion of execution of sentinel in B1 and the fetching of the sentinel in B3. Next, core 0 completes fetching all the instructions of B0 and locks the GPC for B3. So when core 1 completes fetching all its instructions from basic block B1, it needs to wait for the lock. Even though in this example the basic blocks alternate between the

two cores, it is possible for a core to fetch consecutive basic blocks specially when the other core is handling a large basic block and cannot request lock on GPC.

Now let us focus on the commit. Initially the value of *serving* and *ticket* are both 0. Thus core 0 tags all the instructions of B0 with 0 and *ticket* is incremented to 1. Once the first instruction of B0 completes execution, core 0 can start committing instructions as *serving* value matches the tag of its instructions. Core 1 has tagged all the instructions of B1 with ticket value 1. So core 1 cannot commit in parallel with core 0. Instead, it should wait till core 0 completes all the commit and increments *serving* to 1. This introduces idle cycles in the commit stage of core 1 as shown in Figure 3.5.

In summary, Bahurupi execution model requires (a) in-order fetching of the sentinels so that global register renaming can happen in-order, and (b) in-order commit of the instructions across the cores. The dashed lines in Figure 3.5 highlight the in-order fetch and commit. This can introduce idle cycles but is necessary to maintain correctness of program execution. The fetch, rename, and execute of regular instructions in the cores can proceed in parallel to create the illusion of a single virtual ooo engine.

Bahurupi model of execution can even outperform true out-of-order execution engines. For example, for floating point benchmarks, 2-core Bahurupi architecture performs better than 4-way issue out-of-order processor (see Figure 3.7). This is because Bahurupi can look far ahead in the future. In Figure 3.5, for example, the cores are fetching instructions from basic blocks B3 and B4 in parallel. In the 4-way issue processor, however, the instructions from B3 and B4 have to fetched sequentially. As dependencies between the basic blocks are resolved with the help of live-in, live-out information, Bahurupi can exploit ILP across basic blocks much more easily.

3.2 Architectural details

Bahurupi architecture uses classic register renaming for both local and global register files. As mentioned before, we introduce a *shared global register file* that includes *global register map* and *global physical register file*. The size of the global register map is determined by the number of registers in the processor ISA. The size of the global physical register file, however, depends on the fraction of register accesses that require global register file. In our architecture, we allocate 40 entries for the global physical register file based on empirical evaluation.

3.2.1 Live-in register renaming

When a core fetches a sentinel instruction, it has to handle the live-in and live-out registers. For a live-in register, there can be two scenarios. In the first case, the value corresponding to the global register has already been produced by a previous basic block. The core only needs to copy this value into the local register file. In the second case, the value is not yet ready. The register renaming logic then simply copies the mapping from the global register map to the local register map. That is, the local register map for the live-in register now points to a global register file entry.

Figure 3.6 shows an example of register renaming with a basic block. The basic block has one live-in register (r5) and one live-out register (r4). When the sentinel instruction is fetched, the core accesses the global register map where r5 has been renamed to global physical register GR3. Therefore, in the local register map as well we map r5 to GR3. When the regular instructions are fetched within the basic block, they get renamed as usual using *only* the local register map. Hence, source register r5 in instructions I0, I1, I2 get renamed to GR3. Instruction I2 however redefines register r5. At this point, r5 gets renamed to a local physical register LR9. So the next instruction I3 uses LR9 for source register r5. So the same register r5 initially gets mapped to a

Basic block	Renamed registers	Live-out map
sen 5 0 r5 0 0 r4 0 0		4 GR8 -
I0: addiu r4 r5 -27	I0: addiu LR7 GR3 -27	4 GR8 ROB6
I1: addu r6 r5 r4	I1: addu LR8 GR3 LR7	4 GR8 ROB6
I2: addiu r5 r5 2	I2: addiu LR9 GR3 2	4 GR8 ROB6
I3: addu r4 r5 r4	I3: addu LR10/GR8 LR9 LR7	4 GR8 ROB9
I4: sw r6 (r4)	I4: sw LR8 (LR10)	4 GR8 ROB9

Figure 3.6: Global and local register renaming.

global physical register and then gets mapped to a local physical register. On the other hand, register r6 in instruction I1 is always mapped to local physical register as it does not belong to live-in register list.

3.2.2 Live-out register renaming

The core needs to rename the live-out registers. This process is a bit more involved. First the core requests the global register renaming logic to supply a free global physical register corresponding to each live-out register. This mapping information is maintained in the global register map as well as in a special local table called *live-out map*. The live-out map contains only three entries corresponding to three live-out registers. Each entry is a 3-tuple containing (a) architectural register index, (b) global physical register index, and (c) ROB ID of the instruction that last mapped the corresponding architectural register. Figure 3.6 shows the live-out map of register r4 to free global physical register GR8. The ROB entry is not yet known.

Note that we do not immediately copy the live-out mapping into the local register map. This is because a live-out register can be defined multiple times within a basic

block and only the last write to the register should be communicated to the global register file. For example, in Figure 3.6, live-out register r4 gets defined in both instruction I0 and I3. However, only I3 should write to the global register file.

In a 2-way out-of-order processor, we need to rename 4 source registers and 2 destination registers per cycle. In contrast, sentinel instruction requires renaming 3 source registers and 3 destination registers. However, unlike normal instructions where the hardware needs to identify possible dependencies among the registers being renamed in parallel, we only need to identify 3 free physical registers for sentinel instructions. Thus it is easy to rename 3 registers in one clock cycle for sentinel instruction.

The fetching and renaming of regular instructions in the basic block proceeds as usual. For example, r4 gets renamed to local physical register LR7 in I0 and then to LR10 in I3. Whenever r4 gets renamed, the ROB ID of the corresponding instruction is copied into the live-out map as shown in Figure 3.6. Originally, the ROB ID is ROB6 corresponding to I0 and then it changes to ROB9 corresponding to I3.

Whenever a normal instruction with live-out register gets renamed, we copy the global register file index into the corresponding ROB entry. Later on, if the live-out register is renamed again in the basic block, the global register file index is removed from the previous ROB entry and added to the new ROB entry. For example, initially ROB entry ROB6 for instruction I0 contains global register index GR8 and is later removed when ROB9 for I3 is updated with global register file index GR8.

When an instruction that defines a live-out register completes execution, it writes the value into both the local and global register file. For example, when I3 completes execution, the value of register r4 will be written to both LR10 and GR8. In addition, when a core writes to the global physical register file, it needs to broadcast the information on the coalition bus so that other cores dependent on this register can get the value. Finally, when such an instruction is ready to commit, the value is committed to global register file.

It is possible that a regular instruction that last defines a live-out register value (e.g., I3) completes execution even before all the instruction corresponding to its basic block have been fetched and renamed. In that case, when the instruction executed, it was not guaranteed that it needs to write the value into the global register file. This information is known only when all the instructions in the basic block have been renamed. The process that adds the global register information to the ROB entry at the end of a basic block, also broadcasts the value to global register file and other cores if the value is ready in the ROB entry. For this reason, instructions from a basic block are not allowed to broadcast and commit live-out registers till all the instructions from that basic block have been fetched and renamed.

As mentioned before, we can only allow in-order commit to support speculation. Even though the cores in a coalition can perform fetch, rename, execute, and register writes in parallel, only one core can perform commit per cycle. However, we are *not* restricted to at most 2 instructions commit per cycle. This is because all the instructions with local register destination do not need to commit. Only the instructions with live-out destination registers and memory instructions need to commit to the global register file. So we are only restricted to commit at most 2 instructions with live-out destinations and memory instructions per cycle. Hence when we attempt to emulate 4-way or 8-way processor, the commit stage does not become the bottleneck.

3.2.3 Branch misprediction and exceptions

When a core detects a branch misprediction, it will signal all the cores (including itself) to flush the fetch queues and the ROB entries with *ticket* value greater than the *ticket* value of the mispredicted branch instruction. In other words, all the instruction subsequent to the mispredicted branch are flushed from the pipeline similar to what happens in a normal out-of-order execution. The core will ask any other core locking the GPC to release it. The core with the mispredicted branch will then lock the GPC, restore the

global *ticket* value to one plus the *ticket* value of the mispredicted branch instruction and set the GPC to the correct address. Then it will continue fetching from the new GPC address which now points to the correct sentinel instruction. The same policy is followed to maintain precise exception.

3.2.4 Memory hierarchy

Figure 3.2 depicts an example of a Bahurupi processor with eight 2-way cores where at most four cores can be composed together. To form a coalition, the cores need to share the data and instruction caches. We employ reconfigurable L1 instruction cache and L1 data cache for this purpose [30]. Both L1 instruction and data cache have four banks. In normal mode, each core is allocated a cache bank, which behaves as a direct-mapped cache and the cache mapping is configured accordingly. In coalition mode, all the four cache banks together behave as a large 4-way set-associative shared cache. The combined instruction and data L2 cache is shared across all the cores both in normal mode and coalition mode.

For the experiments presented in this chapter, we adopt the above quick solution as our main focus here is on the micro-architecture and the execution model. However, we do propose a novel memory hierarchy solution for dynamic heterogeneous multi-cores in Chapter 4 that is able to accommodate the memory requirements of Bahurupi with low energy and area consumption.

3.2.5 Memory hazards

As with any out-of-order architecture, our design restricts the store operations to update the memory in the commit stage. Thus we can avoid write-after-write memory hazards. However, we still have the problem of write-after-read and read-after-write memory hazards. A load and store executing on different cores can access the same memory

location. In that case, we have to ensure that they execute in the original program order. If a load instruction is about to read from an address at which a previous store (from another core) has to write, then the load operation may read a wrong value. However, this problem is not unique to Bahurupi architecture. Even in a single-core traditional out-of-order execution engine, a load may execute while the memory address of a previous store instruction is still not ready. Later on, it may turn out that the load and the previous store access the same address and hence the value read by the load operation is incorrect. This is handled through memory disambiguation at commit stage by out-of-order processors. We use the same mechanism in Bahurupi. The mechanism enforces that all load operations should execute two times. First, when their operands are ready (execution stage) and second, when they are ready to commit. When the load is executed the second time, it will check if the value read is different from the value which was obtained at the first attempt. If it is different, then it means that another core has committed previously and it wrote at the same address. All the instructions executed after the load instruction by all the cores (including the current one) are corrupted and the core will have to signal all the cores to flush their internal structures and again prepare the GPC and *ticket* registers for a new fetch.

3.2.6 Reconfiguration overhead

The main advantage of Bahurupi is that it is a reconfigurable heterogeneous architecture. At runtime, it is possible to form coalition of two to four cores if we need to execute high ILP applications. On the other hand, the architecture behaves like a traditional homogeneous multi-core architecture in non-coalition mode. The reconfiguration overhead of Bahurupi is minimal except for the latency to flush the pipeline. A special instruction is used to request a core to join a coalition or leave from the coalition. In coalition mode, the additional coalition logic is simply turned off. When forming or leaving coalition, the L1 cache memories have to be reconfigured to either run in par-

titioned mode (for individual cores) or shared mode (for coalition cores). We assume 100 cycle latency for reconfiguration.

3.2.7 Compiler support

Any optimizing compiler computes the live-in and live-out information for a basic block. We simply use this information. We modify the compiler to insert the sentinel instruction at the beginning of each basic block as well as split a basic block if it exceeds the threshold for either number of instructions or number of live-in/live-out registers.

3.3 Experimental setup

In this section we present an experimental evaluation of Bahurupi architecture. First, we estimate the area and delay overhead due to the coalition logic through synthesis. This is followed by performance and energy evaluation with cycle-accurate architectural simulator. In this chapter we present detailed performance evaluation of individual sequential code to show the effectiveness of the coalition. A study of multiprogrammed workload on Bahurupi will be presented in Chapter 5.

3.3.1 Simulator

We implement a cycle-accurate simulator for Bahurupi architecture by modifying an existing multi-core version [9] of SimpleScalar simulator [6]. The original implementation of the simulator only supports multiprogrammed workload with no sharing between tasks. We implemented the shared coalition between the cores together with the inside modifications per core. Our simulated architecture comprises of 2-way out-of-order SimpleScalar cores that share the same memory hierarchy. We simulate 2-core and 4-core Bahurupi architecture built from two 2-way and four 2-way basic cores, respec-

Parameter	2-way	4-way	8-way
ROB Size	64	128	256
Int ALU	2	4	8
Int MULT	1	2	4
FP ALU	2	4	8
FP MULT	1	2	4
Predictor	2K Comb	4K Comb	8K Comb
L1-D$	256KB	512KB	1MB
L1-I$	256KB	512KB	1MB
L2$	1MB	2MB	4MB

Table 3.1: Parameters for baseline cores.

tively. Our Bahurupi cycle-accurate simulator can directly execute sequential program binary annotated with sentinel instructions generated by our modified gcc compiler. We verify that the output produced by Bahurupi simulator matches with the original program output confirming its functional correctness.

For comparison purposes, we also simulate baseline 2-way, 4-way, and 8-way processor architectures. Table 3.1 summarizes the processor configuration parameters. The parameters generally get doubled as processor complexity increases. We use 1 cycle latency for L1 data and instruction cache, 6 cycles for L2 data and instruction cache and 18 cycles for the main memory. As mentioned before, our synthesis results indicate that both baseline 2-way processor core and Bahurupi architecture with coalition logic can run at the same clock frequency (0.6ns clock period). The 4-way baseline processor clock can be synthesized at only 0.8ns clock period. However, in the simulation, we optimistically assume that all the baseline architectures (2-way, 4-way, and 8-way) can run at the same clock frequency.

3.3.2 Compiler

We use gcc-2.7.2.3 compiler configured for SimpleScalar PISA instruction set. We modify the compiler by adding an optimisation pass in the back-end after the register

Workload	Version	Type	Input arg./cycles
go	SPEC95	INT	50 9 2stone9.in/470M
compress	SPEC95	INT	10000 q 2131/27M
li	SPEC95	INT	2 test.lsp/832M
ijpeg	SPEC95	INT	specmun.ppm quality 40/49M
perl	SPEC95	INT	primes.pl/8M
fpppp	SPEC95	FP	natoms.in/7M
wave5	SPEC95	FP	wave5.in/575M
parser	SPEC2000	INT	2.1.dict first phrase/180M
bzip2	SPEC2000	INT	input.program/1.71B
gzip	SPEC2000	INT	input.compressed/1.58B
mesa	SPEC2000	FP	test meshfile mesa.in/179M
equake	SPEC2000	FP	test inp.in/808M
swim	SPEC2000	FP	test swim.in/150M
mgrid	SPEC2000	FP	test mgrid.in/259M
applu	SPEC2000	FP	test applu.in/528M
hmmer	SPEC2006	INT	num 200 bombesin.hmm/192M
lbm	SPEC2006	FP	10 out 0 1 100_100_130/29M
crc	MiBench	telecomm	small.pcm/28M
dijkstra	MiBench	network	small input.dat/57M
rijndael (encrypt)	MiBench	security	input_samll.asc/36M
rijndael (decrypt)	MiBench	security	output_small.enc/36M
susan (edges)	MiBench	automotive	input_small.pgm/3M
susan (corners)	MiBench	automotive	input_small.pgm/1M
susan (smoothing)	MiBench	automotive	input_small.pgm/50M
MPEG2 (decode)	Mediabench2	media processing	input_base_4CIF.mpg/1.72B
MPEG2 (encode)	Mediabench2	media processing	input_base_4CIF.par/534M

Table 3.2: Workloads used for simulation.

allocation is done. Here, the basic blocks are identified together with the live-in, live-out registers and the sentinel instructions are inserted to support Bahurupi execution model. Moreover, the compiler splits a basic block into two or more blocks when either the size restriction of 16 instructions or 3 live-in, 3 live-out registers restrictions are violated.

3.3.3 Benchmarks

We simulate serial benchmarks from both general-purpose and embedded domain exhibiting a diverse workload. Table 3.2 depicts the summary of the selected workloads. The *cycles* column shows the number of cycles to execute the benchmarks on a 2-way core. As the gcc compiler supported by SimpleScalar PISA instruction set is rather old, we had difficulty in compiling the remaining benchmarks from the suites. Note that this is inherent problem with older version of gcc and has nothing to do with our compiler modifications.

3.4 Experimental results

3.4.1 Overall speedup

Figure 3.7 shows Bahurupi's speedup compared to baseline out-of-order architectures. The speedup is normalized w.r.t. the performance of a 2-way architecture. As expected, baseline 4-way and 8-way architectures perform much better than 2-way architecture due to more aggressive ILP exploitation — an average speedup of 2.6 for 4-way and 3.9 for 8-way in embedded benchmarks. Note that the speedup numbers are quite optimistic for 4-way and 8-way cores as we assume the same clock frequency and pipeline for all the baseline cores. As our synthesis results showed, we would need slower clock or more pipeline stages for 4-way and 8-way cores.

What is interesting here is that Bahurupi matches and sometimes even exceeds the performance of true out-of-order execution engines. On average, for embedded benchmarks, dual-core and quad-core Bahurupi outperforms 4-way and 8-way architectures, respectively. The speedup numbers are even more impressive for floating-point SPEC benchmarks, where, on average, dual-core and quad-core Bahurupi improve performance by 23% and 57% compared to 4-way and 8-way cores. Even for SPEC integer

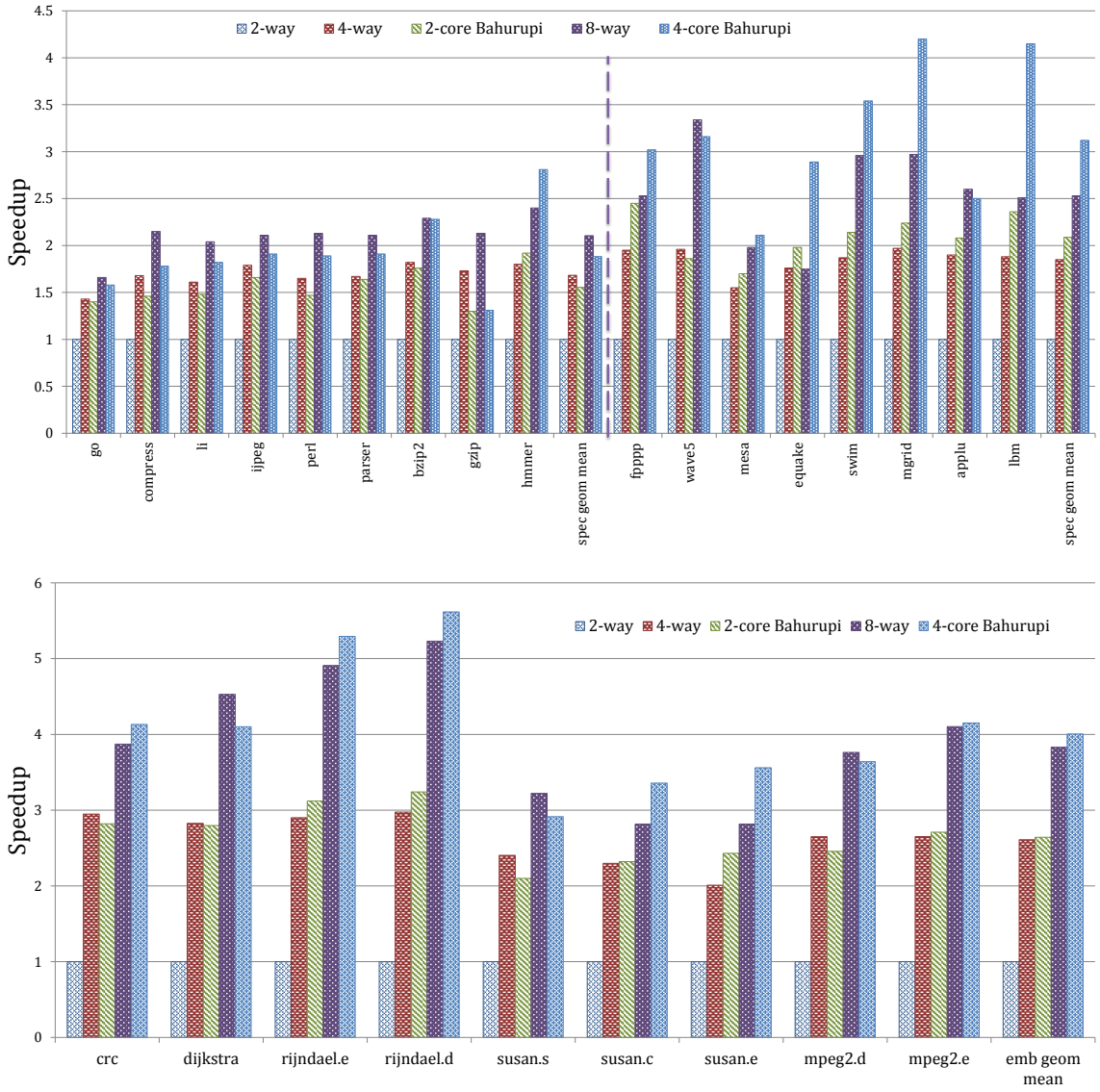


Figure 3.7: Bahurupi speedup normalized to 2-way core for (a) SPEC and (b) embedded benchmarks.

benchmarks, dual-core and quad-core Bahurupi achieves, on average, 92% and 91% of the performance of 4-way and 8-way cores. In summary, for different workloads, Bahurupi can achieve 1.31–5.61 speedup through core coalition compared to baseline 2-way cores used in normal mode.

If we consider the results obtained with Federated cores in [93] we will see that two federated in-order cores can achieve on average 12.9% less performance than the dedicated 2-way out-of-order processor. The closest comparison would be with Bahurupi coalescing two 2-way out-of-order cores. Thus, Bahurupi achieves 8% less performance than the dedicated 4-way core for integer benchmarks and outperforms the dedicated core by 23% in the case of floating-point SPEC benchmarks. Even though Federation is appealing in the context of coalescing two in-order cores, it can introduce significant overhead if we want to reach the performance of even wider processors (e.g., 4-way and 8-way). The work done by Pierre Salverda and Craig Zilles in [86] shows that, under ideal conditions, the steering hardware necessary for obtaining wider cores through coalition is very complex. Additionally, the work shows that in order to obtain the behavior of a dedicated 4-way superscalar processor a large number of in-order cores must be coalesced (12 to 16 cores). Grouping such a large number of cores can dramatically increase the overhead induced by the interconnection logic and issue queues.

Being the closest to our architecture, the results obtained with Core Fusion [48] show an average speedup of 50% for the floating-point SPEC applications and 30% for the integer SPEC applications when using a quad-core fused configuration compared with fine-grain 2-way CMP. In contrast with Core Fusion, quad-core Bahurupi obtains an average speedup of 91% for SPEC integer applications and 210% for SPEC floating-point applications compared to baseline 2-way core.

The reason that 2-core (4-core) Bahurupi can outperform 4-way (8-way) superscalar architecture is because Bahurupi can exploit far-flung ILP. Theoretically, both 4-core

Bahurupi coalition and 8-way baseline normal superscalar core can execute at most 8 instructions per cycle. However, the baseline core is mostly restricted to finding ILP within one or two basic blocks. In contrast, the register dependencies among the basic blocks are identified at compile time and explicitly specified in the binary executable in Bahurupi architecture. This allows the 4 cores in Bahurupi architecture to work on independent instructions from 4 different basic blocks leading to higher ILP. As a concrete example, consider a loop containing only one large basic block with many intra-loop dependencies but no loop carried dependencies. An 8-way superscalar core would fetch one iteration of the loop at a time and would not find many independent instructions to execute due to data dependencies. A 4-core Bahurupi, on the other hand, would fetch and execute 4 independent iterations of the loop in parallel and thus would be able to discover more ILP.

3.4.2 Energy consumption

Next we evaluate the energy consumption of Bahurupi architecture compared to regular out-of-order cores. We use SimWattch [26] suitably modified to compute the energy consumption in the cores. We use the same CACTI [69] tool version (3.0) that is incorporated by the SimWattch to model power consumption of the global register file, global renaming logic, and the broadcast bus and add this energy consumption in the coalition logic to the overall energy consumption of Bahurupi. Figure 3.8 plots energy consumption of Bahurupi normalized w.r.t. the baseline 2-way core. As expected, for SPEC integer benchmarks, 2-core Bahurupi consumes 5% more energy compared to 4-way baseline cores due to slightly increased execution time and the extra energy consumption due to coalition logic and sentinel instructions. But 4-core Bahurupi improves energy consumption by 29% compared to 8-way baseline architecture. For SPEC floating point and embedded benchmarks, on average, 2-core Bahurupi improves the energy consumption by 26% and 11%, respectively, compared to 4-way cores. This is due

to the reduced power per core and the overall improved execution time. Although the transistor technology used by the SimWattch to model the power consumption is quite old, we believe that for smaller transistor features, the energy consumption trend should remain the same.

3.4.3 Load balancing

We do not impose any constraint on the order in which the cores can lock the GPC. Instead, we allow a core to lock the GPC when it is free. To evaluate the impact of this design decision on load balancing, we plot the percentage of instructions committed by each core on a 4-core Bahurupi architecture in Figure 3.9. The figure shows Bahurupi achieves almost perfect load balance among the cores.

3.4.4 Global register file access

Bahurupi achieves considerable speedup when register dependencies are mostly restricted within the basic block, i.e., most register accesses are to the local register file. Figure 3.10 quantizes this characteristics of the benchmarks by plotting the percentage of destination registers that get renamed to global register file compared to the total number of destination registers. On average, for integer SPEC benchmarks 41% of registers are renamed to global register file, whereas only 27% and 24% of registers are renamed to global register file for SPEC floating point and embedded benchmarks. This contributes to better speedup for SPEC floating point and embedded benchmarks in coalition mode compared to SPEC integer benchmarks.

The basic block size has direct impact on the overall speedup of the system. The bigger the basic block size is, the higher the chance for other cores to lock the GPC and fetch the next basic block. This way the cores can fetch and execute different basic blocks in parallel leading to improved performance. With smaller basic block size, the

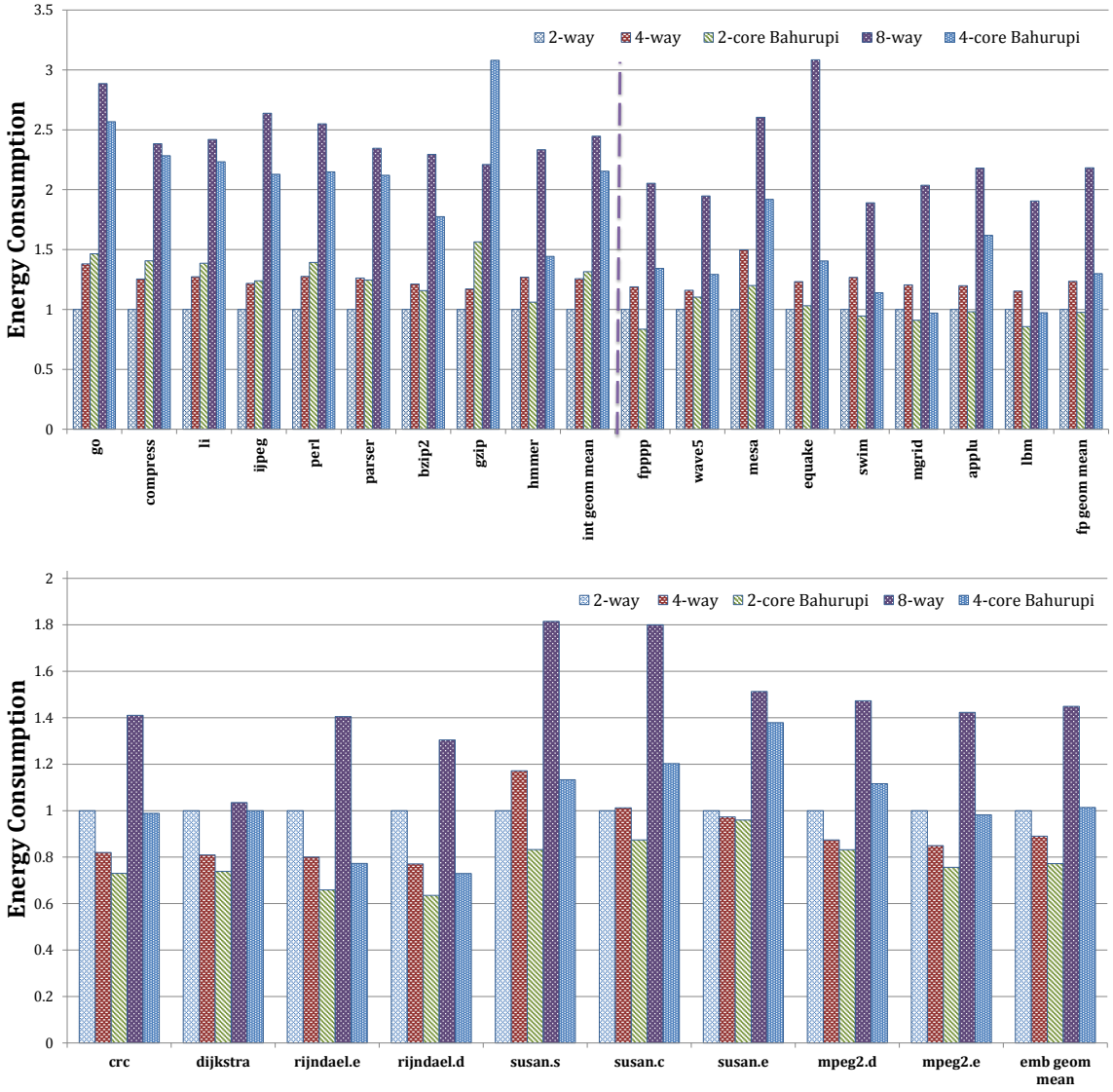


Figure 3.8: Bahurupi energy consumption normalized to 2-way core for (a) SPEC and (b) embedded benchmarks.

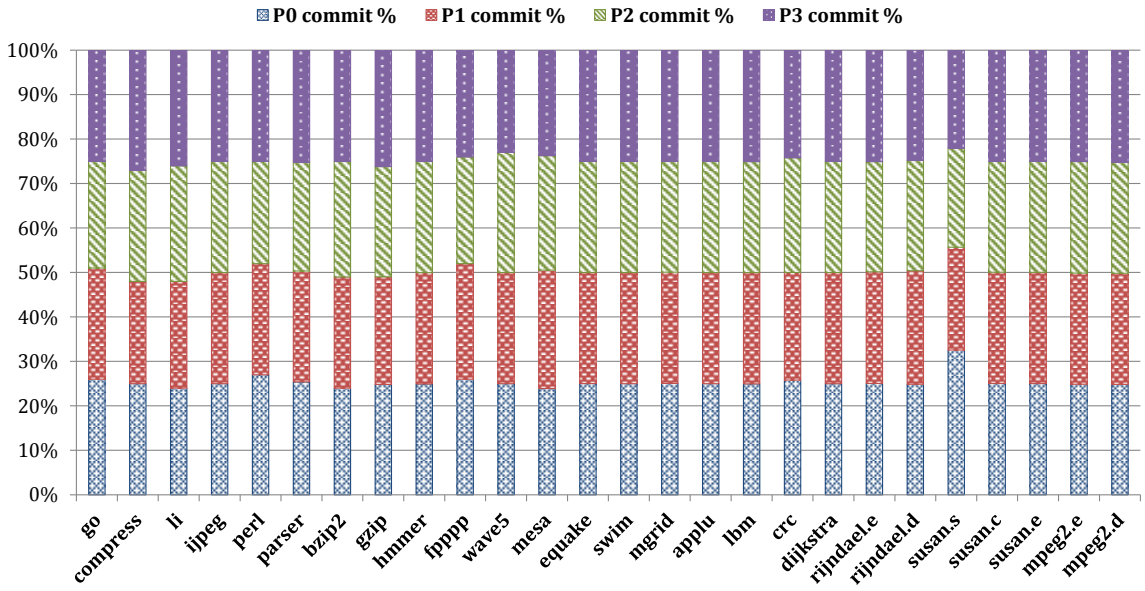


Figure 3.9: Load balance on 4-core Bahurupi.

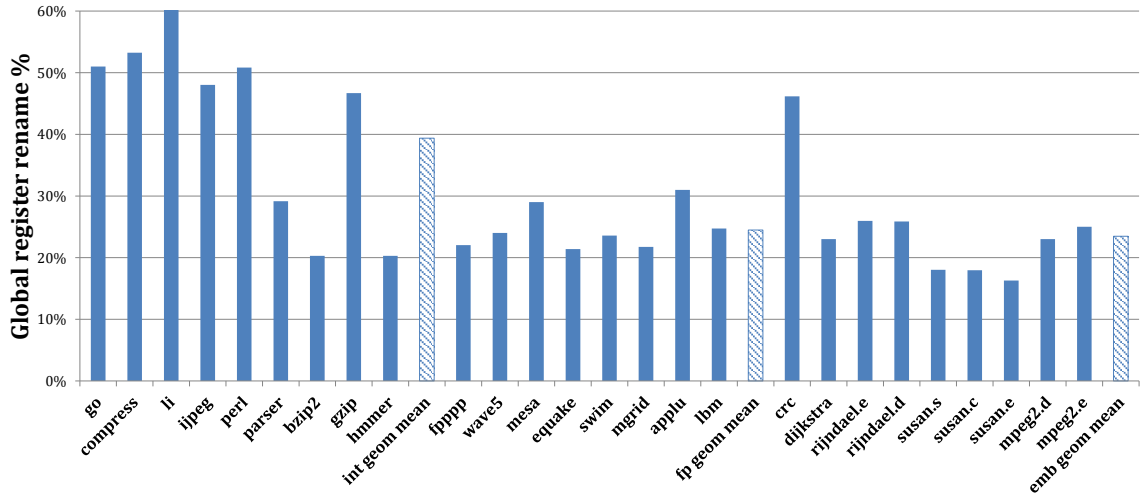


Figure 3.10: Percentage of destination registers renamed to global register file.

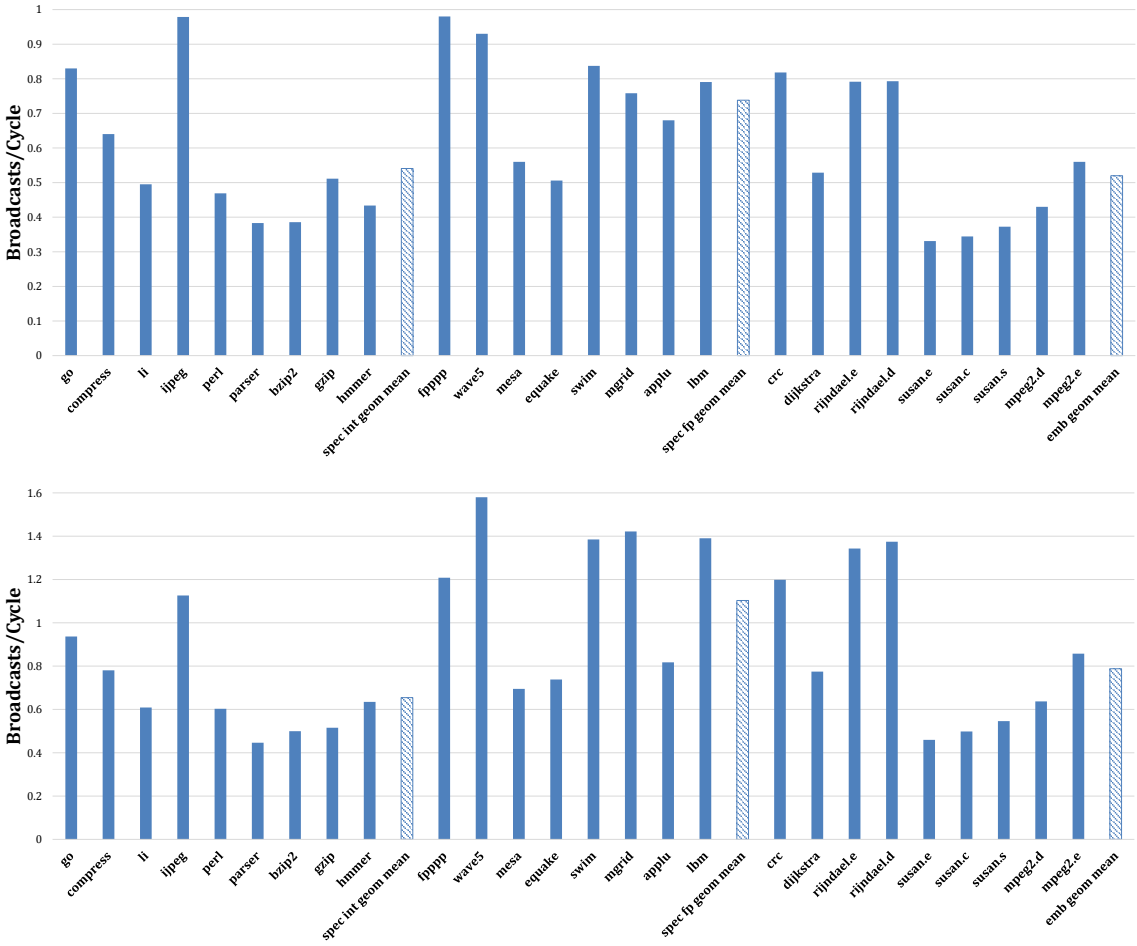


Figure 3.11: Broadcasts on coalition bus in (a) 2-core Bahurupi and (b) 4-core Bahurupi.

same core tends to fetch consecutive basic blocks leading to limited opportunity of parallelism. Basic block size is limited to 16 instructions due to sentinel instruction format. In general, basic block size is much smaller in integer benchmarks compared to floating point benchmarks leading to speedup difference between the two cases. In addition, the amount of ILP and dependency among basic blocks also influence performance. The best case scenario is when an application has big basic blocks, high ILP and low dependency among the basic blocks. Then the application is almost running in parallel on all the cores in Bahurupi architecture.

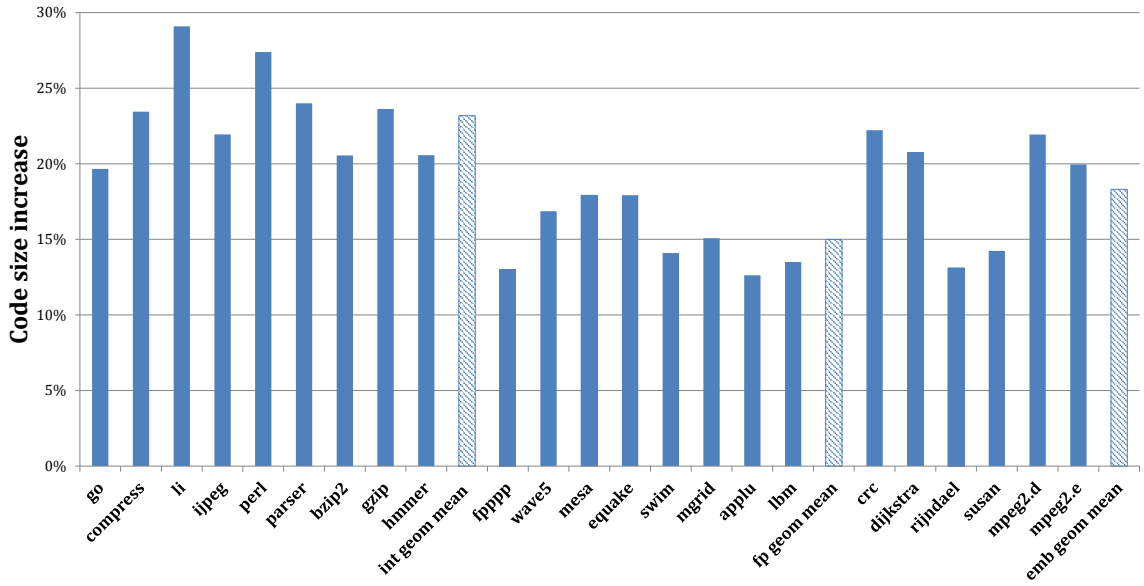


Figure 3.12: Code size increase due to sentinel instructions.

3.4.5 Traffic on coalition bus

If multiple cores attempt to use the coalition bus in parallel to broadcast global register file writes, then we have to serialize the writes leading to a bottleneck. Figure 3.11 shows the average number of broadcasts required on the coalition bus per cycle in 2-core Bahurupi architecture is well below 1.0, while for 4-core Bahurupi, it is below 1.6. This indicates that broadcast due to global register file writes is not a bottleneck. In both cases, floating point applications show higher traffic on the coalition bus as they expose higher ILP.

3.4.6 Sentinel instruction overhead

Bahurupi pays the price for reduced hardware complexity with increased code size due to the addition of sentinel instructions. Figure 3.12 shows that, on average, code size for SPEC integer applications increases by 24% and the floating point ones by 15%. The Mediabench [59] and MiBench applications size increase by 19% on average. Note

that we did not optimize the code. The code size increase can be countered through compiler optimizations such as loop unrolling and superblock formation that increase basic block size.

3.4.7 Area and delay overhead of coalition logic

In order to estimate the area and delay overhead due to the coalition logic, we synthesize a preliminary version of the baseline processor core and the coalition logic. We use a synthesizable version of the Simplescalar processor core [31]. We generate a 2-way and a 4-way Simplescalar core with the same configuration parameters given in Table 3.1. The synthesis is performed using Synopsys 2010 [92] design compiler with FreePDK [39] 45nm technology library.

The main components of the shared resources are represented by the global register file and the corresponding renaming logic that we implement and synthesize. The renaming logic maps 3 live-in and 3 live-out registers per cycle. Note that normal 2-way processor core is expected to map 4 live-in and 2 live-out registers per cycle whereas a normal 4-way processor has to map 8 live-in and 4 live-out registers per cycle. Further, we assume 3 register read ports and 2 register write ports for the global register file. This is because the global register file is read during the renaming of global live-in registers and there are at most 3 live-in registers per sentinel instruction. The global register file is written when an instruction with live-out destination register completes execution. As we have shown earlier, only 24% of the registers are renamed to global register file for embedded benchmarks. Even for a 4-core coalition, this corresponds to $2-\text{way} \times 4\text{cores} \times 0.24 = 2$ global destination registers written per clock cycle.

Synthesis results show that the baseline 2-way Simplescalar core can run at 0.6ns clock period. Note that both the synthesizable version of Simplescalar core we are using [31] as well as the global register file we have designed are not highly optimized. The 4-way baseline Simplescalar core can run at 0.8ns clock period. In contrast, the

global register file can still be synthesized at well below 0.6ns clock period. In other words, the global register file does not contribute to additional delay in the clock period of the baseline 2-way processor core. Thus Bahurupi architecture with coalition logic can easily run at 0.6ns clock period as opposed to 0.8ns clock period for 4-way baseline superscalar processor core.

In terms of area overhead, the coalition logic corresponds to 31.27% of the area of a 2-way Simplescalar core (the area of the core does not include the caches and the TLB) and 23.48% of the area of a 4-way Simplescalar core.

A complete hardware implementation of Bahurupi multi-core architecture will be presented in the next section.

3.5 Bahurupi FPGA implementation

We have claimed in the above sections that Bahurupi needs minimal hardware overhead in order to be implemented on top of existing architectures. In this section, we present the details of implementing the Bahurupi architecture through minimal modification of a real 2-way out-of-order pipeline. A key concern is the latency and the area impact of the glue logic that coalesces the cores together.

3.5.1 Fabscalar synthesizable out-of-order core

We select synthesizable out-of-order superscalar core generated by *Fabscalar* [85] — a parametric micro-architecture generation tool chain for the baseline simple cores. The cores use PISA ISA. FabScalar is a project developed at the Department of Electrical and Computer Engineering in the NC State University under the guidance of Dr. Eric Rotenberg. Fabsclaar works on developing heterogeneous multi-core systems that uses superscalar cores. FabScalar helps researchers to create such systems by offering a Verilog toolset for automatically assemble arbitrary superscalar cores.

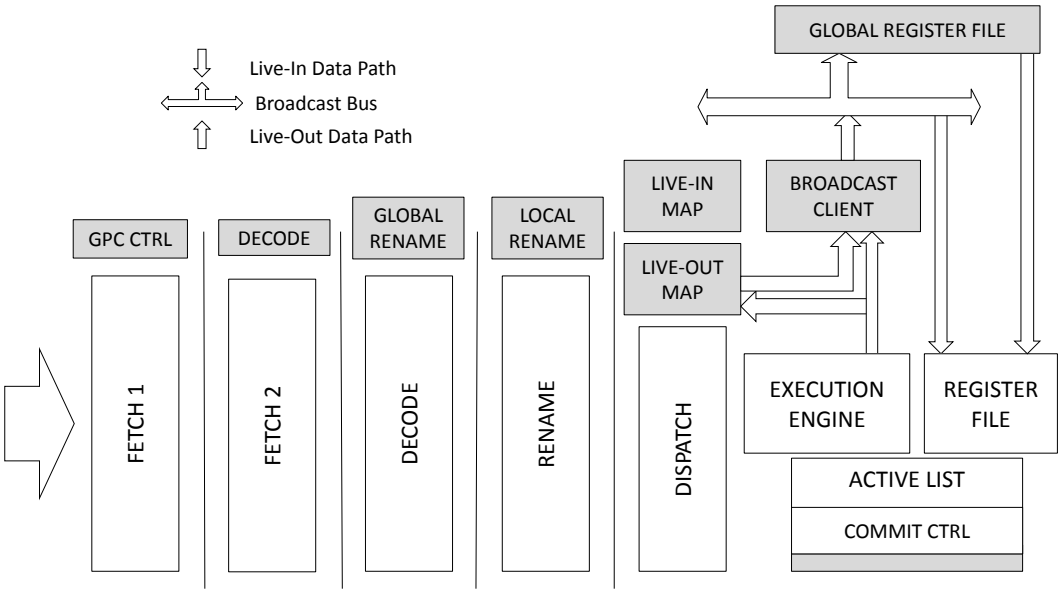


Figure 3.13: Fabscalar pipeline with coalition logic.

We custom generate a 2-way out-of-order core with five in-order frontend pipeline stages and an out-of-order backend. The backend comprises of a centralized issue logic, register-read stage, integer execution units and write-back stage. Two in-order commit stage completes the pipeline. Figure 3.13 depicts the pipeline stages along with the coalition logic that we added on top of it. The non-shaded region is the baseline core.

The critical components for supporting core coalition are the register file implementation and its renaming logic of the baseline core, which we briefly describe here before proceeding to the core coalition implementation. Usually, different real out-of-order superscalar processors use special renaming and register file structures and techniques. Bahurupi high-level design needs to be adapted for these structures. However, we will show in this section that this adaptation process has very little overhead.

The renaming structure of Fabscalar consists of a *Register Map Table (RMT)* and an *Architectural Map Table (AMT)*. The live uncommitted mapping between the architectural registers and the physical registers is maintained in the RMT. In the rename stage, the source registers get the current mapping to the physical registers and the des-

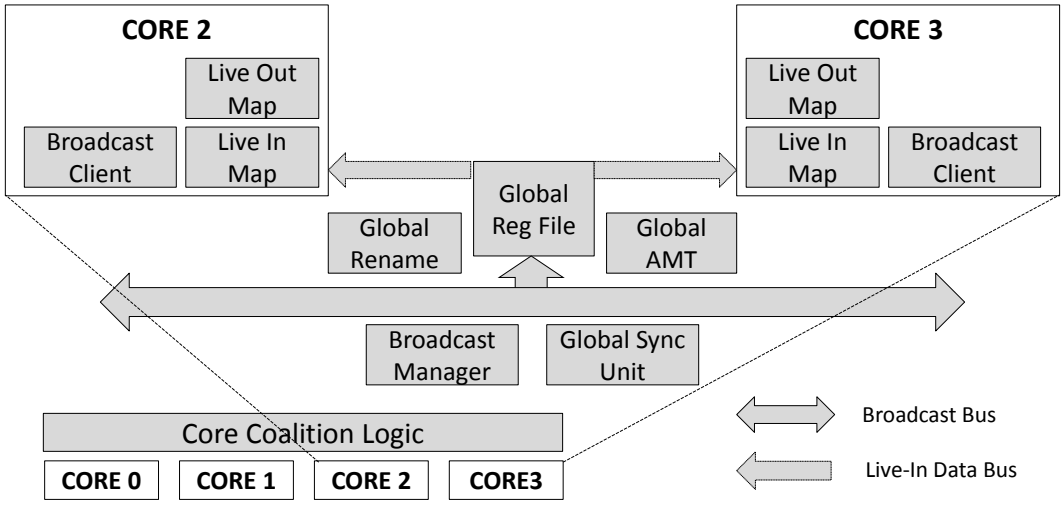


Figure 3.14: Two 2-way core coalition.

tination register is renamed to a free physical register. The new mapping is written on to the RMT for use by the source operands of the following instructions. A dedicated FIFO buffer maintains the list of free registers. Once in-order renaming is complete, instructions are dispatched to the out-of-order execution engine along with an entry in the *Active List*. The Active List maintains the instructions in program order.

As instructions retire in program order from the Active List, the logical-to-physical mapping of the destination registers are committed in the AMT. In case of mis-speculation, the mapping from the RMT are erased and the execution continues with the mapping from the AMT. As an instruction retires, the physical register corresponding to the old mapping in AMT is disassociated from the logical register and is added back to the free list buffer. A busy bit array in the physical register file indicates if the corresponding data is ready for use. A physical register is busy between the rename cycle, where it gets mapped to a destination register, and the write-back cycle for the corresponding instruction when data is ready.

3.5.2 Core coalition logic

The core coalition logic is required so that the cores can cooperate together to create a virtual core with higher degree of superscalarity. Figure 3.14 shows the coalition logic required for a cluster of four 2-way cores. We add three hardware units inside the core, apart from which there is no other significant modification to the existing core. One of key challenge is to make sure that the new hardware units are not in the critical path of existing core.

We will next present step by step how the instructions (sentinel instructions and normal instructions) are processed in the Fabscalar pipeline and what specific hardware structures are used.

Sentinel instruction processing. Each participating core requests for the global PC through the global synchronization unit. We use Least Recently Used (LRU) policy to grant the lock for the global PC. The core that gets the lock proceeds to fetch the sentinel instruction. Assuming that we have a cache hit, it takes 2 clock cycles to fetch and decode the sentinel instruction. We decode it in Fetch-2 stage, one cycle before the actual decode stage so that the lock can be released earlier. If the basic block ends with a branch, we calculate the branch address based on the length of the basic block and index into the branch target buffer and branch prediction unit in the second Fetch stage.

In the third clock cycle, we access global rename module to rename the live-out registers. We also get the existing mapping of the live-in registers. In parallel, the global PC is updated with the address of the next sentinel instruction, that is, a pointer to the next basic block. This will be a speculated address in case of a basic block ending with conditional branch. Otherwise, the next address is calculated by simply adding the length of the basic block to the current global PC. The lock is released at the end of the third clock cycle. The global rename module signals the global synchronization unit to

free the lock.

After the lock is released, the core continues with sentinel instruction processing. The live-in register values reside in the global physical register (GPR) file. To operate on these values, we need to create a local copy. Thus we rename each live-in register within the core to get a corresponding local physical register (LPR). Figure 3.15 provides an illustration of this scenario. From the given assertions, let us say, Sentinel Instruction1 (SI1) for Basic block 1 (BB1) is about to be dispatched. It checks if the GPR corresponding to each live-in (GPR5 and GRP6), is ready. If ready, the data is directly copied into the corresponding LPR (LPR4). However, it finds GPR5 still busy and maintains the GPR-to-LPR mapping for that live-in register in the Live-In Map (GRP5-LPR2).

Similarly, while dispatching the sentinel instruction, the mapping of live-out registers to GPR are stored temporarily in Live-Out Map. In the example, SI_1 stores the live-out register R8 in Live-Out Map (R8-GPR3). Unlike Live-In Map, Live-Out map only consists of 3 entries corresponding to 3 live-out registers. Live-Out Maps are maintained until all the instructions in the basic block have been renamed. We only broadcast the result of the last instruction in the basic block that writes to a particular live-out.

Normal instruction processing. The coalition design is transparent to the processing of the normal instructions. It is visible in only two stages of the pipeline: Live-Out Map module in the dispatch stage and commit control logic in the Active List module. Figure 3.13 shows the seamless integration of coalition logic into the base Fabscalar pipeline.

As instructions in the basic block are dispatched, the destination registers are searched for a match in the Live-Out Map (implemented as a CAM). If matched, the active list entry of the corresponding instruction is updated with the GPR index from the Live-Out Map. In our example, first instruction in BB1, I_{11}, finds register R8 in the Live-Out

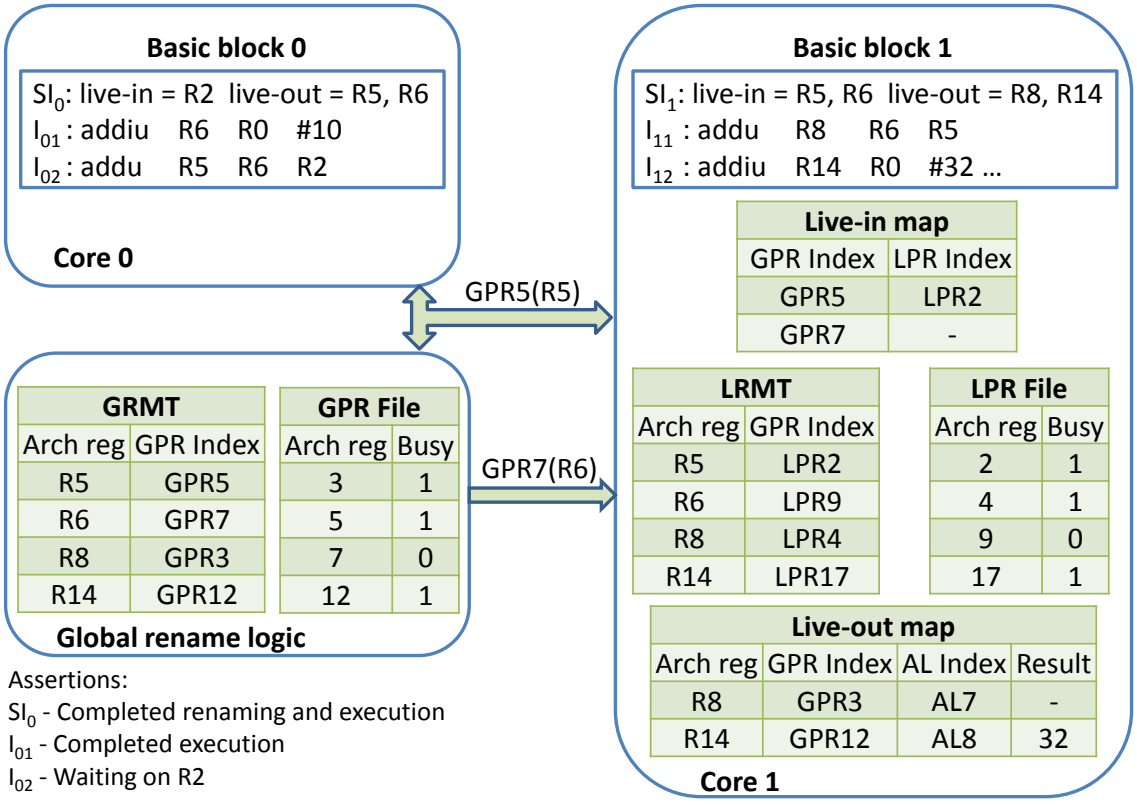


Figure 3.15: Register flow across cores - an illustration.

Map and adds GPR3 into its active list entry (AL7). This is to ensure that when the instruction I_{11} completes execution, it will broadcast the value. Also the Live-Out Map is updated with the active list ID (R8-GRP3-AL7) of the instruction. This happens in parallel to dispatching instructions to the ooo execution engine.

While the basic block is being renamed, if an instruction with live-out register destination completes execution, the result is stored in Live-Out Map. To illustrate, consider that the instruction I_{12} in BB1, completes execution while the following instructions in BB1 are still being renamed. As the system is not in a position to determine if I_{12} is indeed the last instruction in BB1 to write to the live-out register R14, it is illegal to broadcast the result on the *Global Broadcast Bus (GBB)*. Hence, the result of I_{12} is temporarily stored in Live-Out Map (value 32). When the last instruction of a basic block is renamed, we have the final list of instructions that need to broadcast their result onto the GBB. At this point, if some live-out register values are ready in the Live-Out Map, we broadcast them through the Broadcast Client. In our example, I_{12} has completed execution and assuming no other instruction after I_{12} writes to R8, we can now broadcast the result on the GBB. Live-Out register results produced after this point are broadcast as and when they are ready.

In our example, I_{02} would eventually complete and broadcast the result (GPR5) in its write-back stage through Broadcast client. Broadcast client module is the third significant addition inside the core. It is a simple FIFO buffer and has the logic to obtain broadcast bus access. The broadcast bus is monitored by a global module called *Broadcast Manager*. The access is given to one core at a time using LRU policy. Once the client gets the lock, a live-out value is broadcast on GBB.

Live-In Map module passively snoops all the data on the GBB. If an active Live-In Map for the broadcasted tag is found, the data is internally re-broadcasted to wake up the instructions waiting on that live-in operand. I_{11} of BB1 waits in the issue queue for the availability of live-in register (R5). When I_{02} of BB0 broadcasts the result (GPR5),

Live-In Map module of Core 1 (running BB1) finds a match (GPR5-LPR2) and latches the result for internal broadcast. We moved the internal broadcast to the next cycle, which helps reduce the wire length for GBB. The cost of one additional cycle is amortized by the pipelined architecture.

A key aspect to note here is that, though coalition logic is exposed at two stages of normal instruction processing, it does not alter the control path of the normal instructions. Coalition logic does not even appear in the critical path for normal instructions apart from searching for destination registers in the Live-Out Map. As there could be a maximum of only 3 live-out registers per basic block, this CAM search of 3 entries does not impact the clock cycle length. The one cycle access to GBB is not altered by the number of cores in the coalition, as we have split the broadcast into two cycles: global and internal. Increasing number of cores can create contention on the global broadcast bus. However, we have shown in Section 3.4.5 that even for high ILP code, the number of broadcast requests per cycle is below 1.0 for 2-core coalition and below 1.6 for 4-core coalition.

3.5.3 Prototype synthesis and evaluation

We implement our prototype on Xilinx Virtex 6 (XC6VLX240T-1FF1156) [4] platform. In Section 3.4.7, we present a preliminary evaluation of the impact of *only* the global register file and its renaming logic on the clock period. Here we present a full-fledged implementation that brings out the challenges involved.

Resources for core coalition. Table 3.3 shows additional read/write ports required in existing components from the baseline core and Table 3.4 shows block memories required to implement sentinel instruction processing within cores.

The global structures added are independent of the number of cores participating in the coalition. The only exception is the global physical register file for which number of

Component	AMT	RMT	Free List	LPRF	Issue Queue
Write Ports	1	1	1	2	0
Read Ports	1	1	1	0	2

Table 3.3: Additional ports within the baseline core.

Module	RAM/CAM	Component	Bits	R/W Ports
Active List	RAMS	Ticket Number	128*9	2R/2W
		Live-Out Map	128*7	3R/3W
		Sentinel Commit Map	16*90	1R/1W
Broadcast Client		Broadcast FIFO Buffer	16*46	1R/4W
Live-In Map		GPR-to-LPR Mapping	96*7	1R/2W
Live-Out Map		Live-Out-Map	3*46	3R/3W
	CAMS	GPR Search	3*7	2R/3W
		Active List ID Search	3*7	3R/2W

Table 3.4: Additional RAMs/CAMs for coalition per core.

read ports increases linearly with the number of cores in the coalition. The placement of these global resources is a key to achieve good clock frequency. For two 2-way core and four 2-way core coalition, default placement strategies in Xilinx's Place and Route tool with no optimizations enabled could give us the desired clock frequency. Table 3.5 lists the global resources required for core coalition.

Handling multiple ports in FPGAs. A major challenge in synthesizing out-of-order cores in FPGAs is the multi-ported RAMs as most FPGA vendors only support 2-port memories. Many different solutions have been proposed for this problem [58]; but no single technique can fully provide our desired behavior from the RAMs. So we have used three different techniques: *Replication* to provide multiple read ports; *Live-Value-*

Module	Component	RAM in Bits	RAM Ports
Global AMT	Logical-to-GPR Mapping	34*7	3R/3W
Global RMT	Logical-to-GPR Mapping	34*7	3R/3W
Global Free List	Free GPR List Buffer	62*7	3R/3W
GPRF	Global PRF	96*32	NR/1W

Table 3.5: Global resources for coalition. N is # of cores.

Table (LVT) implementation for multiple write ports; and *Virtual Cycle* implementation to mitigate the one cycle delay due to Synchronous block RAMs.

In LVT based design, a RAM is replicated into n number of banks, for n write ports. A table with n write ports called LVT registers the bank number for a particular address where the latest write was sent. Writes to all ports can happen in the same positive edge of a cycle and are available in the following clock cycle. In addition, RAMs in each bank are further replicated into m RAMs, for m read ports. A read coming into an LVT based RAM is sent to all banks and read in parallel in the same positive clock edge. All read data are available in the following clock cycle. The actual output of the read is decided based on the multiplexing logic using the live value obtained from the LVT. LVT itself is implemented as a RAM with n write and m read ports but cannot be realized as a block RAM, instead, uses D-Flip-Flops. As the width of LVT is log(n write ports), the area for the LVT itself should not be a problem. In a real system, this can be further optimized by multi-pumping technique [58].

As the block RAMs are synchronous, data from RAM reads are only available in the following clock cycle. However, our baseline core requires asynchronous read logic in some stages (e.g., data needs to be read in the same cycle in which the address is sent). We use Virtual Cycles to achieve this goal. A global virtual cycle generator unit connects to all the components and alternatively generates virtual stall signal for the entire core apart from the RAMs itself. This single cycle stall helps mitigate the asynchronous read problem. However, this effectively halves the frequency achieved; but as this applies to all the core configurations, we consider this a fair comparison.

Area evaluation. Figure 3.16 presents the area required for different core configurations including the baseline 2-way out-of-order core. We have seen that 2-core (4-core) coalition has similar performance to a native 4-way (8-way) out-of-order core. Our 2-core coalition requires less area than a native 4-way out-of-order core for most

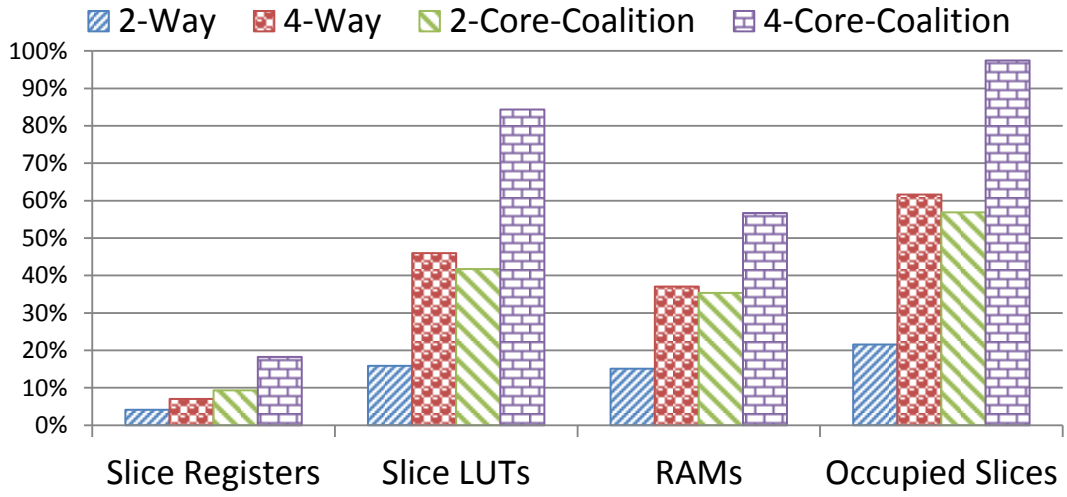


Figure 3.16: Area utilization (8-way core equivalent in performance to 4-core coalition could not be synthesized).

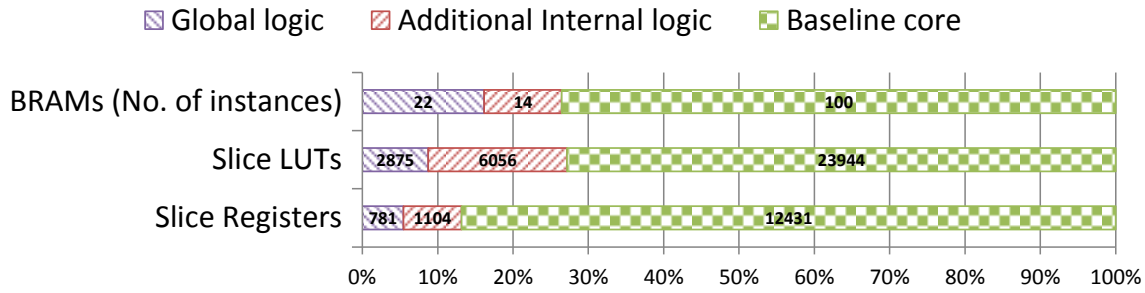


Figure 3.17: Area breakup of coalition logic w.r.t baseline core.

critical resources. We could successfully synthesize and perform place-and-route of a 4-core coalition on Virtex-6 board. However, we could not synthesize an 8-way core on the same board. Thus, Bahurupi has an area advantage over conventional out-of-order cores.

Figure 3.17 shows the break-up of area required for core coalition logic in comparison to a baseline 2-way core. The results show the utilization of the slice registers, LUTs and BRAMs. Core-coalition logic consumes additional 13% of Slice Registers, 26% of BRAMs and 27% of Slice LUTs compared to the the baseline core. Note that in our evaluation, the multi-core is directly connected to the rest of the system without any cache memory hierarchy. Still, the results are encouraging with minimal additional resource requirement.

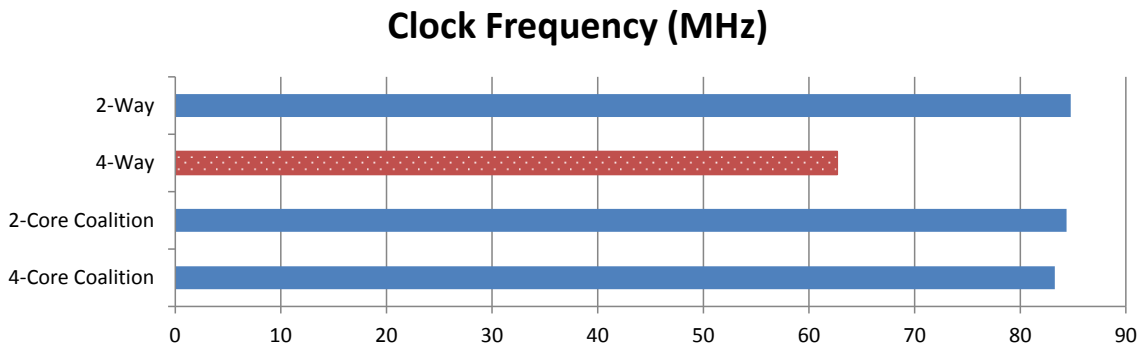


Figure 3.18: Clock frequency for various core configurations.

Clock frequency. The synthesis result concretely supports the simplicity and efficiency of Bahurupi architecture. We observe that core coalition logic has no impact on clock frequency of the baseline core. Figure 3.18 shows that the clock frequency remains almost the same for 2-core (84.4 MHz) and 4-core (83.3 MHz) coalition as baseline 2-way core (84.8 MHz). In contrast, a 4-way core synthesizes to a much lower frequency (62.67 MHz) and an 8-way core could not even be synthesized due to resource limitations (slice LUTs and slice units) on the FPGA platform. The low frequency obtained for the 4-way core is due to the increased delay observed in the critical path which comprises of issue lookup logic, register file and active list.

3.6 Summary

In this chapter we have presented and evaluated our dynamic heterogeneous multi-core architecture, called Bahurupi, that can dynamically adapt itself to support both multi-threaded code with explicit thread-level parallelism as well as sequential code with instruction-level parallelism. Bahurupi can dynamically merge the base 2-way out-of-order execution engines to achieve the performance of 4-way or even 8-way out-of-order processors. Bahurupi is a hardware-software cooperative solution that requires minimal additional hardware resources and compiler support for coalition.

We have also presented a full prototype implementation of Bahurupi in FPGA. We

can successfully create a virtual 4-way (8-way) out-of-order core from two (four) 2-way out-of-order cores. The area of the virtual core is slightly smaller while the clock frequency is signicantly higher compared to the equivalent native core.

Chapter 4

Reconfigurable Data Cache Architecture

In this chapter we present a novel reconfigurable L1 data cache architecture for dynamic heterogeneous multi-cores that overcomes the limitations of multi-ported caches, offering high bandwidth and low latency when accessing the shared cache.

Recently proposed adaptive heterogeneous multi-core designs [48, 52, 93, 104] mostly ignore or make simplifying assumptions regarding the memory hierarchy implemented in such architectures. A common assumption in all these works is that the first level of data and instruction caches must support reconfigurability. Our dynamic heterogeneous multi-core architecture, Bahurupi, also requires the first level of cache to support reconfigurability when the coalition of cores is created. The data and instruction caches are now shared by the cores found in the coalition.

Assuming no self-modifying code, one can afford replication in case of private instruction caches in the dynamic heterogeneous multi-core systems as instruction accesses are read-only. However, it is more challenging to design the L1 data cache that supports multiple accesses per cycle ensuring high bandwidth and low latency. When a virtual core is created by coalescing the cores, this virtual core creates more mem-

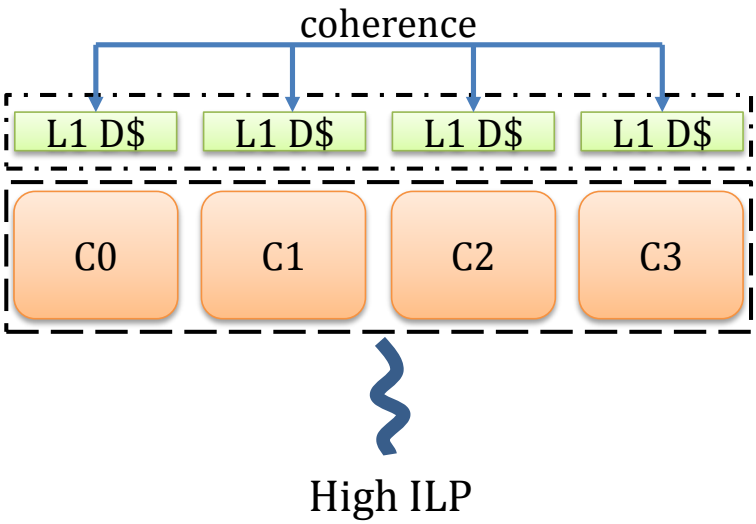


Figure 4.1: Example of 4-core coalition with data cache merging.

ory requests thus stressing the bandwidth to the first level of data cache. Moreover, these accesses to the L1 data cache have a high degree of sharing — different cores can read and write the data which is part of the same sequential program. In Figure 4.1 we show an example of a 4-core coalition executing a program that exposes high amount of ILP. Traditionally, the caches are kept coherent using a coherence bus. We can see that besides coalescing the cores, private data caches merging is also needed.

The traditional coherent data caches will lead to too many coherent misses due to the extensive data sharing among the cores. Figure 4.2 shows the average miss rate across L1 data caches for 2-core coalition and 4-core coalition in case of several sequential SPEC applications when using coherence. For comparison, we also plot the miss rate of the application when the cores found in coalition are sharing a four-ported L1 data cache. In each case, the L1 data cache is a 32KB, 8-way set associative cache. The coherence protocol used for this experiment is MESI. We can clearly see that when using coherence, the L1 data cache miss rate increases to unacceptable levels. Whenever a core is updating a value in its private L1 data cache there will be many cache blocks invalidated in the other caches as the degree of data sharing between basic blocks is

very high. Consequently, this will have a dramatic impact on the performance of the application and on the energy consumption due to excessive access to the lower level of cache.

We identify two possible design choices for the shared L1 data cache. A *multi-ported* L1 data cache that can be shared by all the cores. These ports can be exclusive read ports, write ports, or read/write ports. However, multi-porting the cache means having all the internal memory cells multi-ported, which in turn increases the cache area significantly [87]. Additionally, the access time and power consumption increase dramatically with multiple ports. A second design option to allow multiple accesses to the first level of data cache is to use *independent cache banking*. The cache is split into smaller banks; each bank is mapped to different parts of the address space with independent address and data lines. The advantage is that the small banks are simple, fast to access and they consume low power compared to the big multi-ported cache. On the other hand, it is possible that multiple accesses go to the same bank (bank conflict) in which case, arbitration is needed. Similarly, routing a request to the appropriate cache bank may require additional overhead.

In this chapter we propose a novel reconfigurable L1 data cache architecture for dynamic heterogeneous multi-core architectures that overcomes the limitations of a multi-ported cache, offering high bandwidth and low latency when accessing the shared cache. Our design is also able to switch between two modes of execution: *private mode* (for traditional multi-core) and *coalition mode* (for dynamic heterogeneous multi-core). Additionally, the system can also configure the size and associativity of the overall shared L1 data cache.

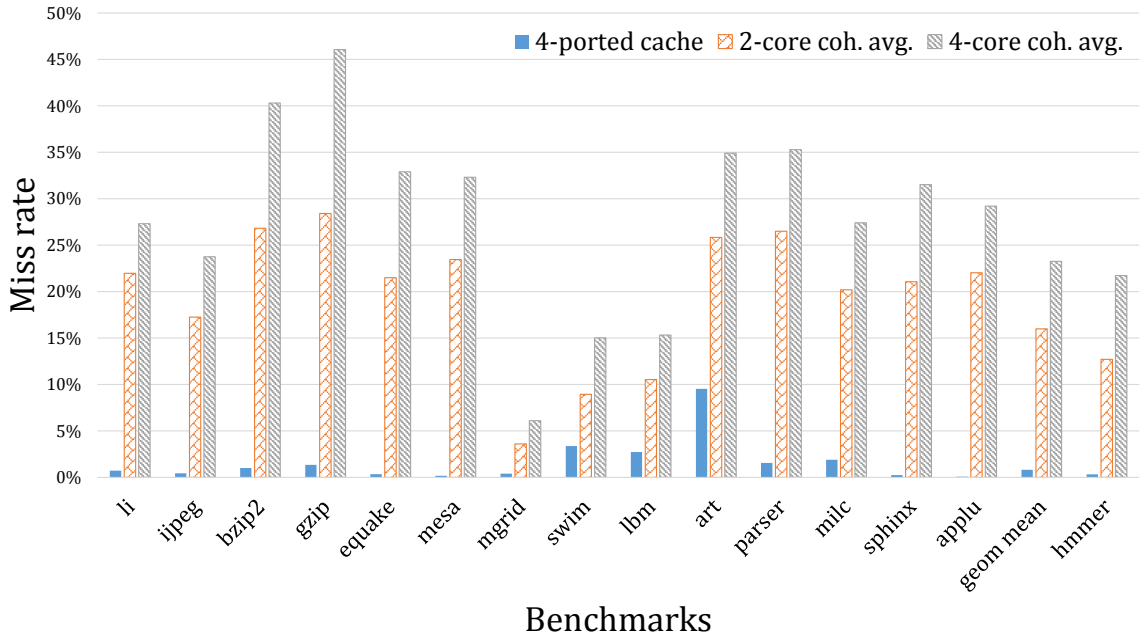


Figure 4.2: Miss rate for shared cache vs. coherent caches.

4.1 Experimental setup

Our evaluation is done in conjunction with Bahurupi adaptive multi-core architecture and we compare our results against a shared multi-ported cache. For evaluating different L1 data cache designs we use the same simulation setup described in Section 3.3.1. In these experiments we configured the cache access latencies to be closer to the ones found in modern CPUs (e.g., ARM Cortex A-15): the L1 cache access latency is configured to 3 cycles for both instruction and data caches, the L2 cache access latency is 19 cycles while the main memory access latency is 140 cycles.

The SPEC95, SPEC2000 and SPEC2006 [2] benchmarks used in all our experiments are described in Table 4.1. For all our experiments the benchmarks were run by fast forwarding over the first 100 million instructions and run for 2 billion instructions.

Name	Benchmark Suite	Description
li	SPEC95	Xlisp interpreter
ijpeg	SPEC95	Image compression/decompression
gzip	SPEC2000	Compression
bzip2	SPEC2000	Compression
parser	SPEC2000	Word Processing
equake	SPEC2000	Seismic Wave Propagation
mesa	SPEC2000	3-D Graphics Library
mgrid	SPEC2000	Multi-grid Solver: 3D Potential Field
swim	SPEC2000	Shallow Water Modeling
applu	SPEC2000	Partial Differential Equations
art	SPEC2000	Image Recognition / Neural Networks
milc	SPEC2006	Physics / Quantum Chromodynamics (QCD)
sphinx	SPEC2006	Speech Recognition
lbm	SPEC2006	Computational Fluid Dynamics
hmmer	SPEC2006	Search Gene Sequence

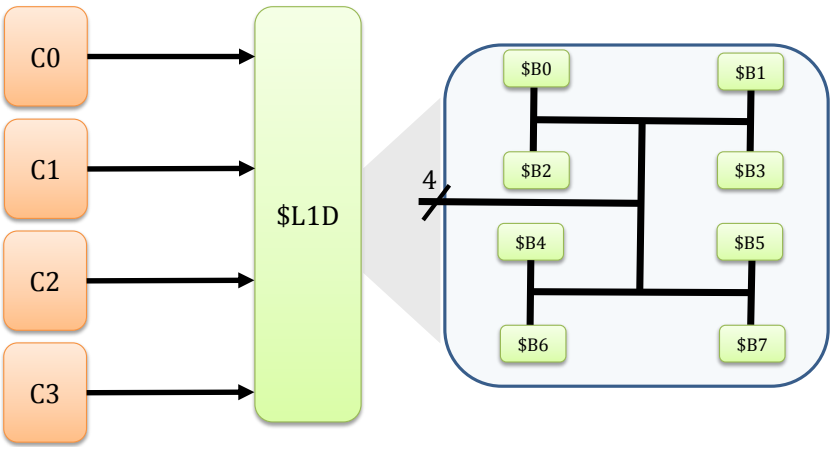
Table 4.1: Benchmarks description.

4.2 Limitations of multi-ported shared L1 cache

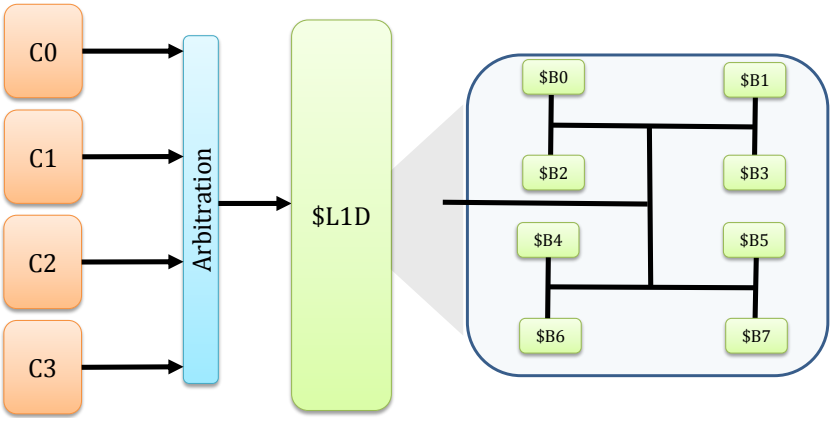
We begin our evaluation by first describing the simple design solution given by a multi-ported cache. We will show the advantages and disadvantages of using such a design.

We constructed two examples of shared cache configurations shown in Figure 4.3: (a) shared cache with 4 read ports and (b) shared cache with one arbiter and one read port. In each configuration the caches have only one write port as Bahurupi only allows one memory store operation per cycle (stores execute in program order). For the configuration with one read port, the arbiter is interposed between the cores and the cache granting only one request out of four in case of bank conflicts.

The internal interconnect of a first level cache is usually in the shape of a h-tree (shown in Figure 4.3) where accesses to the internal banks are guaranteed to be uniform [87]. As a result, this simplifies the pipelining of the cache access increasing the memory access throughput. This interconnect only works for small caches as the lower levels of caches use more complex interconnect to accommodate the NUCA ef-



(a) Shared cache with 4 read ports.



(b) Shared cache with 1 read port.

Figure 4.3: Multi-ported shared cache configurations.

fect [44]. In our evaluation we assume that the L1 data caches contain 8 banks. Along with the network wires, the interconnect also contains various predecoders for selecting the destination bank for a request. Usually each bank has its own tag and data array line decoder.

Modern architectures (e.g., Intel i7, Ivy Bridge) use 32KB, 8-way set associative cache which offers good hit rate. Here, we consider the 32KB, 8-banked, 8-way set associative cache with each bank being 4KB, 8-way set associative as our baseline cache configuration, which we use for various comparisons.

4.2.1 Area and energy overhead

We measure the area occupancy and energy consumption per access of single and multi-ported shared cache configurations using CACTI 6.5 [69] configured for 32nm technology. The results are shown in Figure 4.4 where the values are normalised w.r.t the area and energy of a single ported shared cache. The results show that the area and energy consumption dramatically increase when increasing the number of ports. A dual ported cache occupies 1.9X more area than a single ported cache and consumes 1.5X more energy per access, while a three-ported cache occupies 3.3X more area and consumes 2.4X more energy. Similarly, four-ported cache occupies 6.5X more area and consumes 4.4X more energy.

4.3 Limitations of single-ported shared L1 cache

4.3.1 Simultaneous memory accesses

In case of a core coalition there is high probability of multiple accesses to happen within the same cycle. The virtual core creates now more memory requests per cycle. In Figure 4.5 we present the average number of cache accesses that happen in the same cycle for

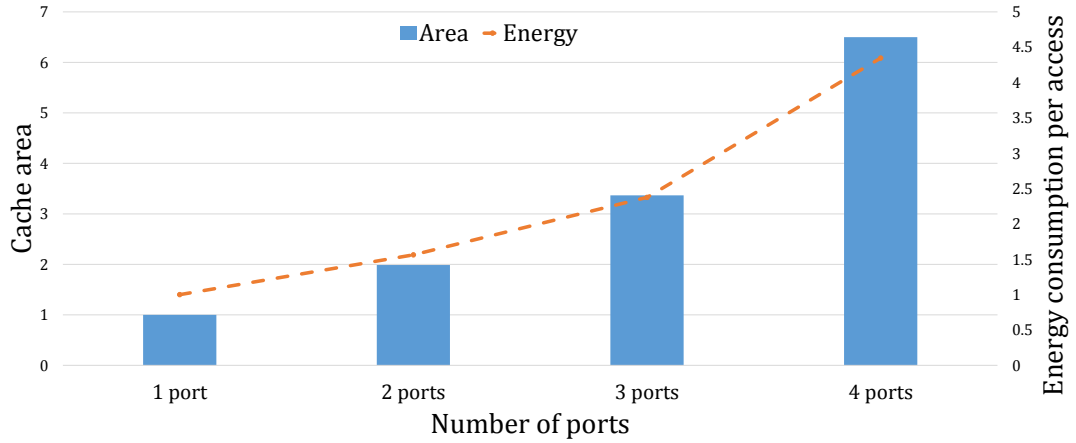


Figure 4.4: Shared cache area and energy increase for different number of ports (normalized w.r.t to 1 port).

2-core, 3-core and 4-core coalition in case of cache configuration with four read ports. On average, for all the core coalition configurations, the number of accesses pe cycle is above one (1.05, 1.16 and 1.30 respectively). The number of accesses per cycle is more than two for 4-core coalition in case of the benchmarks *mgrid* and *lbm*.

4.3.2 Performance impact

Moving to a single ported shared cache offers reduction in both area and energy consumption. In this case, arbitration is needed to handle multiple accesses to the shared cache in the same cycle (see Figure 4.3b), which will impact the overall performance. We implement a simple round-robin priority selection scheme for the requesting inputs. When an arbitration conflict happens, the unsuccessful cores will reinsert the memory operation in the internal issue queue and try to issue it again in the next cycle. Moreover, arbitration introduces an extra stage in the cache access pipeline that increases the cache access latency by 1 cycle.

In Figure 4.6 we show the performance obtained when running coalition of 2-cores and 4-cores on multi-ported cache configurations with four ports and one port. The numbers are normalized w.r.t the performance obtained on a single-core baseline pro-

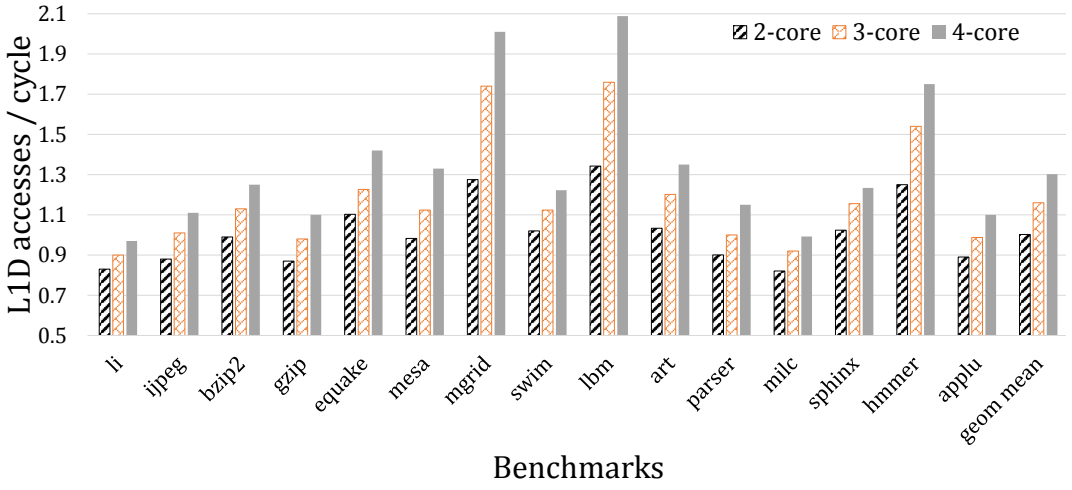


Figure 4.5: Average number of L1 data cache accesses per cycle for 2-core coalition, 3-core coalition and 4-core coalition.

cessor with private cache. We can see how the performance drops due to the arbitration conflicts. On average, the loss in performance for constraining a 2-core coalition to share the single-ported cache is 10%. The loss in performance is even higher (50%) for a 4-core coalition constrained to use only one port.

The performance degradation for *mgrid* and *lbm* benchmarks is the highest as they have the highest number of arbitration contentions. Interestingly, when these two applications run on a 4-core coalition connected to a single-ported cache, the performance drops very close to the performance obtained with 2-core coalition. This is due to the memory access patterns of the benchmarks and the way Bahurupi executes the basic blocks in a distributed fashion.

In conclusion, in order to support 4-core coalition, we need high bandwidth to the L1 data cache that can be offered by multi-ported cache. However this has a huge impact on the area and the energy consumption. Alternatively, we can constrain the system to use less number of ports resulting in smaller area and energy consumption but reduced performance.

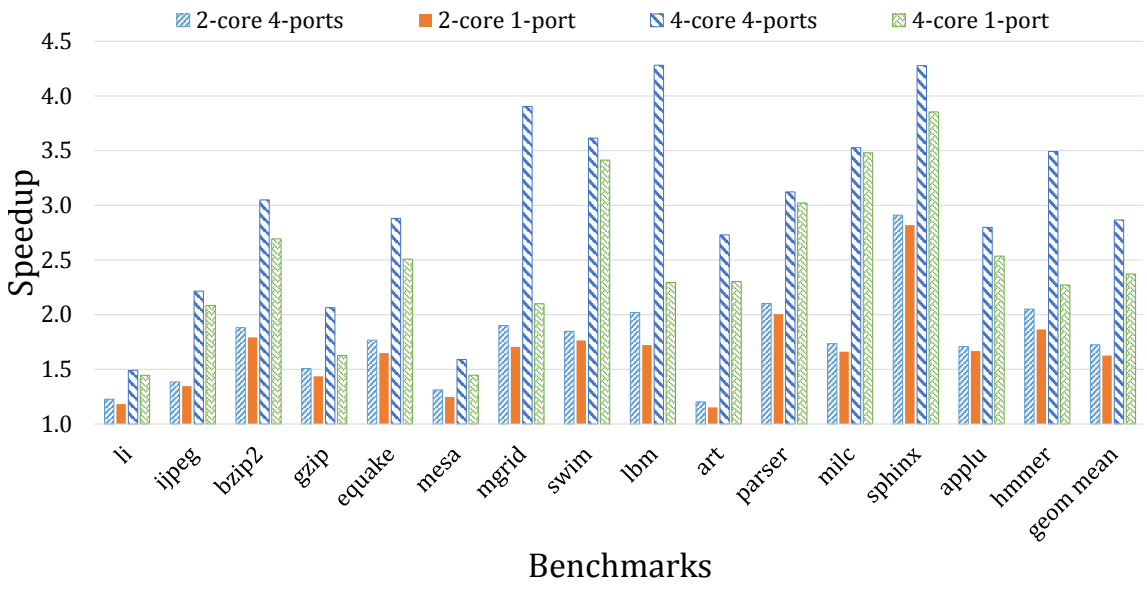


Figure 4.6: Performance impact for 2-core and 4-core coalition run on cache configurations with four ports and one port (normalized w.r.t performance of a baseline single core).

4.4 System reconfiguration

Using a multi-ported shared cache for adaptive multi-cores clearly shows significant overheads in terms of area and energy consumption. We overcome these limitations by proposing a novel reconfigurable L1 data cache architecture for adaptive multi-cores presented in Figure 4.7. Our design allows three types of reconfiguration described below.

An important property that the adaptive multi-cores must have is the ability to switch between *coalition mode* and *private mode* seamlessly. In coalition mode, the cores should connect to a shared L1 data cache, while in private mode the cores should connect traditionally to their own private caches that are connected through a coherent protocol. In our design we implement system reconfiguration through switches that control these two modes of execution by selecting the corresponding input for the four muxes shown in Figure 4.7.

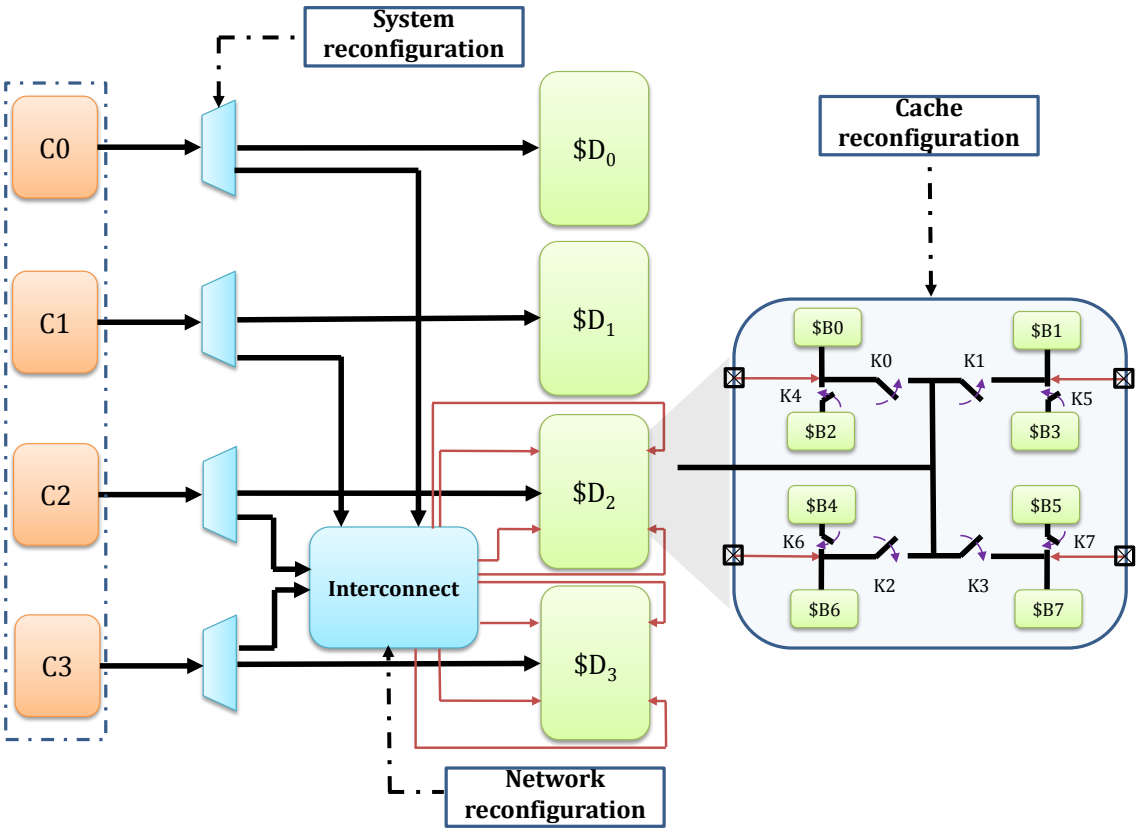
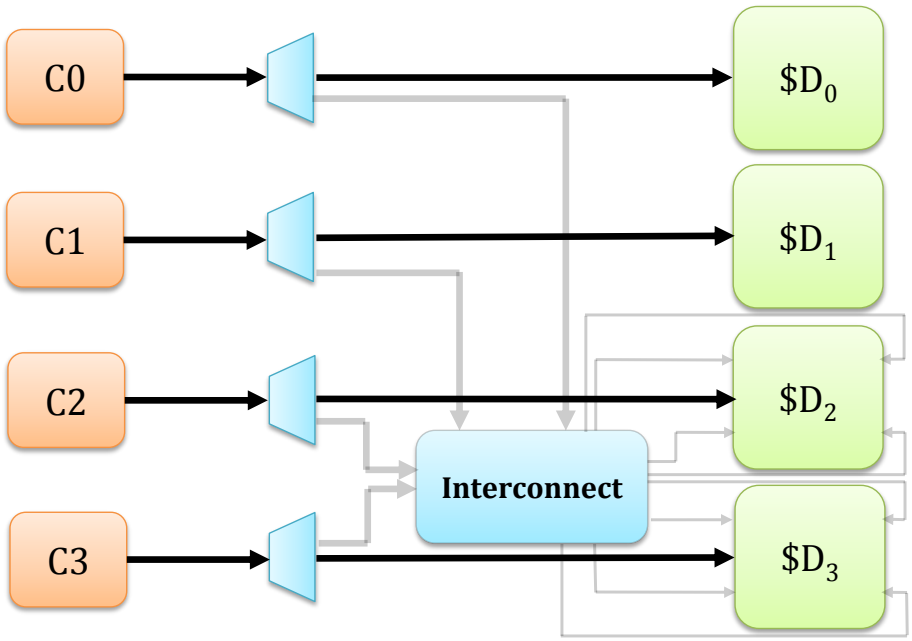
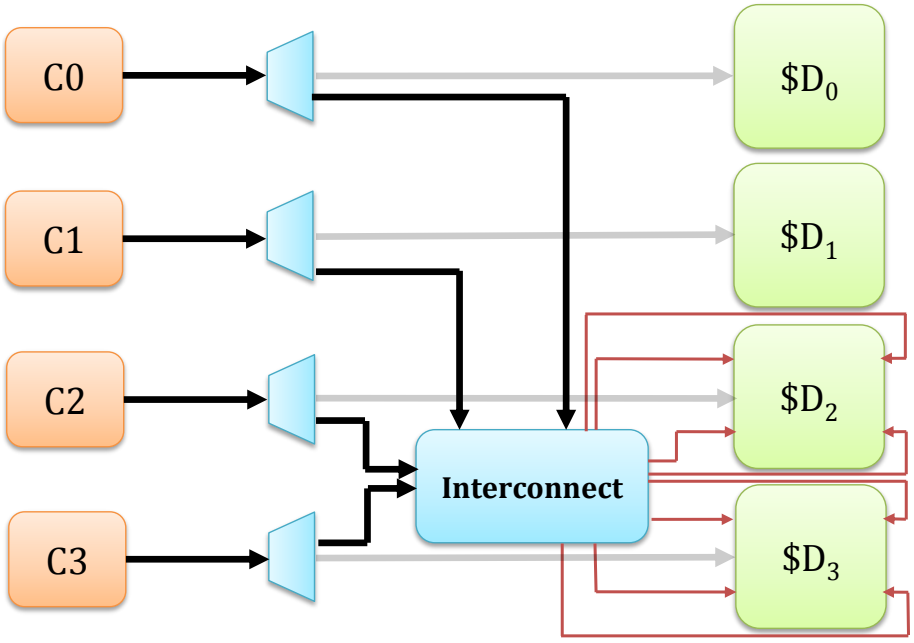


Figure 4.7: Novel L1 data cache architecture for dynamic heterogeneous multi-cores.



(a) System reconfiguration for *private mode*.



(b) System reconfiguration for *coalition mode*.

Figure 4.8: System reconfiguration for private and coalition modes in case of a 4-core coalition.

Bahurupi extends the ISA with a new instruction that can control the configuration of the muxes through the 4-bit *system reconfiguration* register. The decision of reconfiguration can be fully taken at the operating system side where performance counters (e.g., IPC and cache misses) can be combined with offline knowledge of the application to correctly predict that the application can run faster on a bigger coalition or the application should be moved to a smaller coalition. The register controls the selection input for each mux based on the subset of cores used for coalition. In Figure 4.8 we show how the system reconfiguration works for private and coalition modes in case of a 4-core coalition. In private mode (Figure 4.8a), an operating system routine configures the muxes to private mode such that each core accesses directly its own data cache. In this case, the interconnection network is not involved and it is practically switched off. When switching to coalition mode (Figure 4.8b), for all four cores, the muxes redirect the memory requests to the interconnection network that in turn sends the requests to the shared cache. In this case, the two private data caches D_0 and D_1 can be switched off, reducing the system power consumption. Of course, before switching off the two caches, their content must be flushed to the lower level of cache (e.g., L2 unified cache). Also, when doing system reconfiguration for creating coalition of cores, the content of the private caches corresponding to the cores engaged in the coalition must be flushed to the lower level cache.

4.5 Network reconfiguration and address mapping

Our main goal is to allow coalition of cores to be able to share the same amount of cache that is used by cores in private mode with minimal area, energy and performance overhead. In our case, our target is a 32KB, 8-way set associative shared cache. In default coalition mode, we only use four out of eight banks for each data cache. If necessary, we may use the extra banks for increasing associativity of the caches. In

order to obtain eight parallel access points to eight banks we need to use two data caches out of four. Thus, we make the network connect to already existing banks inside the two (D_2 and D_3) through the access points highlighted (small crossed rectangles) in Figure 4.7. We justify this decision by the fact that Bahurupi can fuse at most 4 cores at runtime and the network offers enough bandwidth to accommodate the requests coming from 4 cores. Moreover, the amount of shared cache is enough to offer a low miss rate.

The shared cache contains banks used in private mode in case of cores C2 and C3. As the system only allows one coalition of cores at a time, an operating system routine can ensure that the cores C2 and C3 will always be part of a coalition.

The architecture of the interconnection network is shown in Figure 4.9 where the four cores connect to eight access points. The network was first proposed in [84] where the main focus was on the architecture of the interconnect. It comprises of two layers: a layer of switches also known as *logarithmic network layer* and an *arbitration layer*. The logarithmic network layer connects to the memory ports of each core and carries the memory requests to the arbitration layer. Notice that up to the arbitration layer, the requests coming from different cores follow independent routes. Thus, the interconnect interacts with the outside world through four input ports and eight output ports. The arbitration layer grants access to the access points which are connected internally to the caches. In contrast to our proposed architecture, the original design did not support pipelined access to the cache banks and reconfiguration. The interconnect is a logarithmic network (or mash-of-trees) which comprises of two main parts: the combinational routing network and the arbitration stage.

The *combinational routing network* uses a simple routing scheme where a subset of the address bits are used to route the input of each switch (demux based implementation) S to one of the two outputs. The selection of the bits for routing highly influences the degree of interleaving of accesses to the cache banks.

The address mapping used in our proposal is presented in Figure 4.10 where we

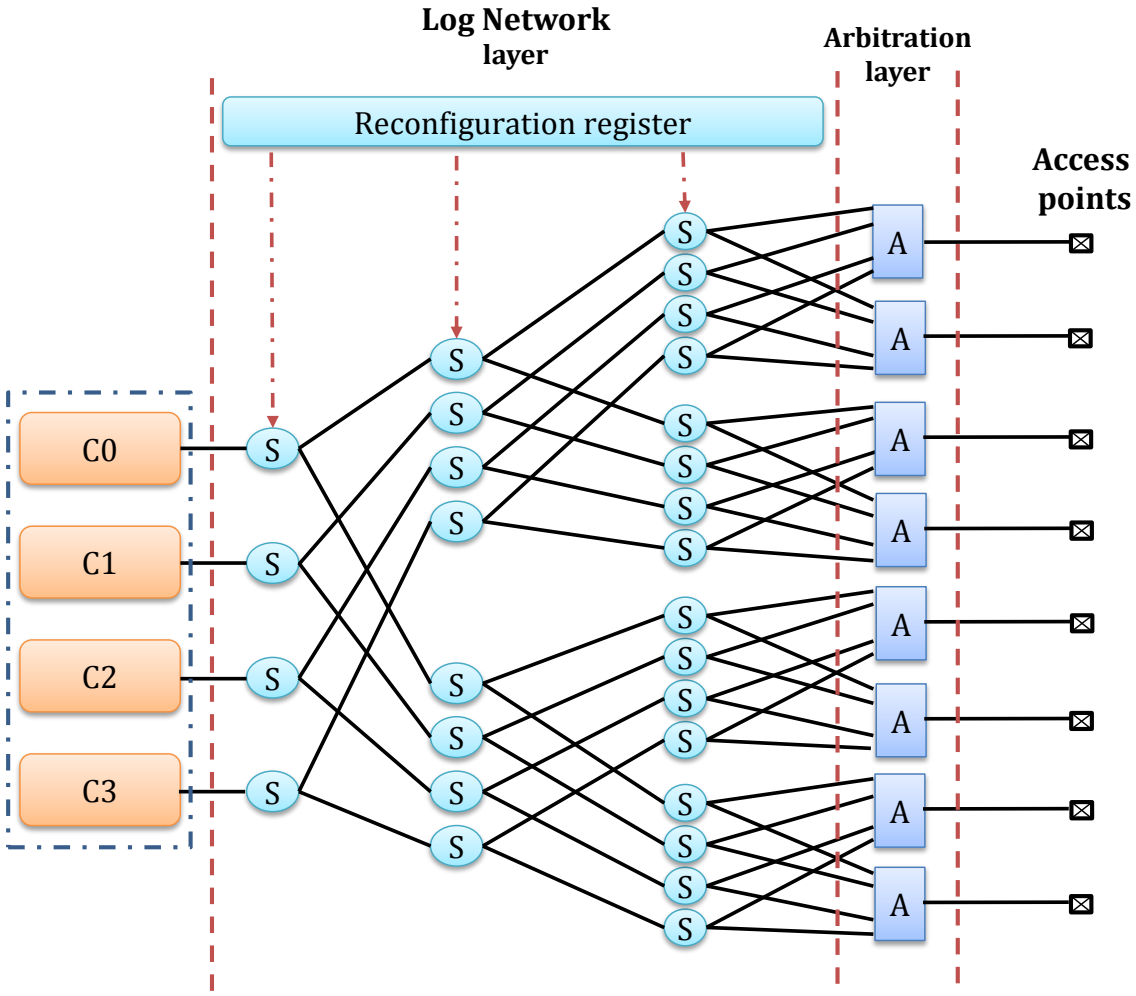


Figure 4.9: CPU to memory interconnection network architecture for 4-cores and 8 access points.

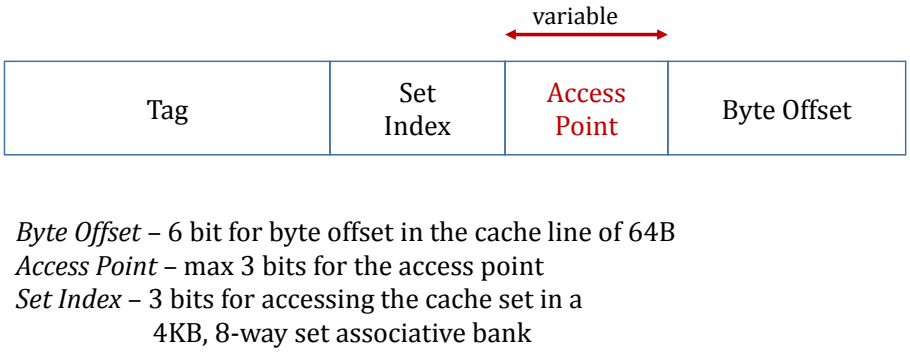


Figure 4.10: Address mapping in the interconnection network.

show the breakdown of the memory address issued by the core. The bitfield *Access Point* is used by the switches to route the packet to the destination access point and it has a maximum width of 3 bits as our network connects to a maximum of eight access points. The *Byte Offset* field is used to select the byte from the cache line, while the *Set Index* is used to select the set within a bank. The interconnection network has three columns of switches. By default, switches from the first column will check the *Access Point[2]* bit; if the bit is 0 the packet will be routed in the upward direction, otherwise it will be router in the downward direction. The second column and third column will do the same checking and routing by checking the bits *Access Point[1]* and *Access Point[0]* respectively.

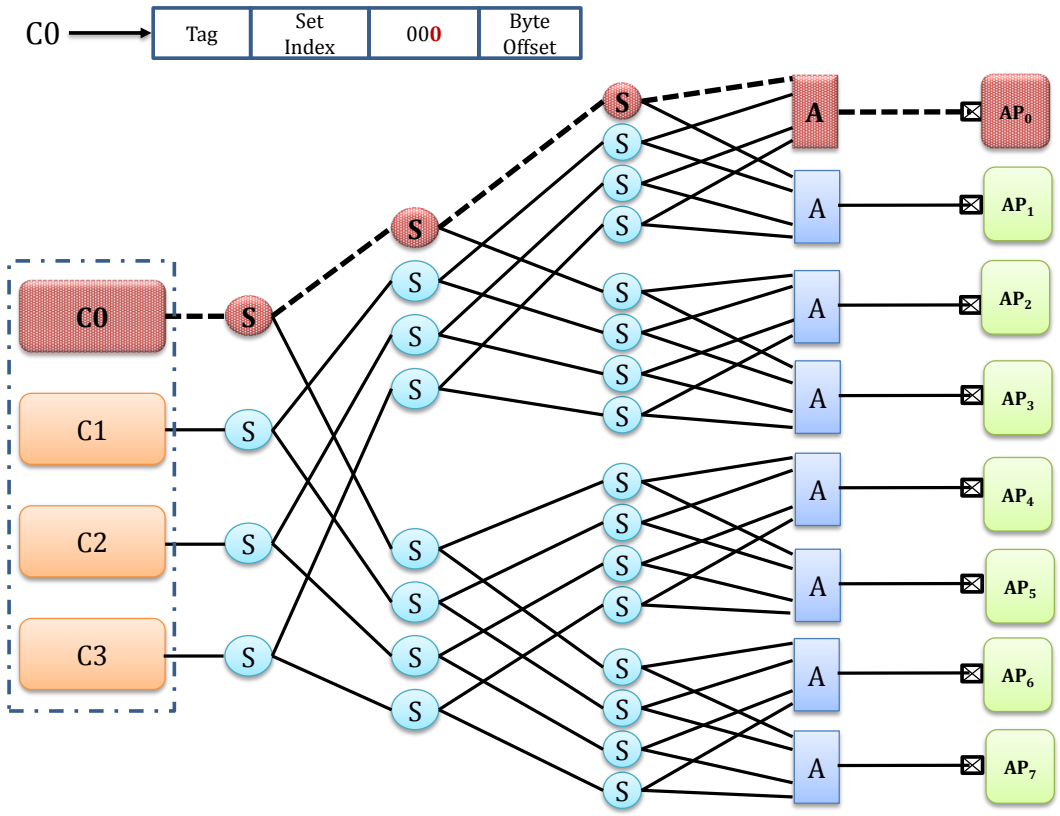
However, the routing decision can be reconfigured based on the number of access points the coalition of cores needs to access. This reconfiguration is done through a simple *Reconfiguration Register* that connects to all switches and can be dynamically written by the core. The reconfiguration means changing the mapping of the addresses by increasing or decreasing the width of the *Access Point* field in the address word. When moving to less than 8 access points, the *Reconfiguration Register* fixes the map for selected switches such that they simply forward the input to the fixed output. As a result of changing the address mapping, whenever a network reconfiguration is performed the shared cache must be flushed to lower level cache.

Moreover, when a network reconfiguration is performed the decoder corresponding to the cache bank must be configured for correctly choosing the set index bits from the instruction address. The length of the set index bitfield is the same in all configuration but its bits as a subset from the whole address can be shifter left or right by 3 positions based on the number of access points the system decides to use.

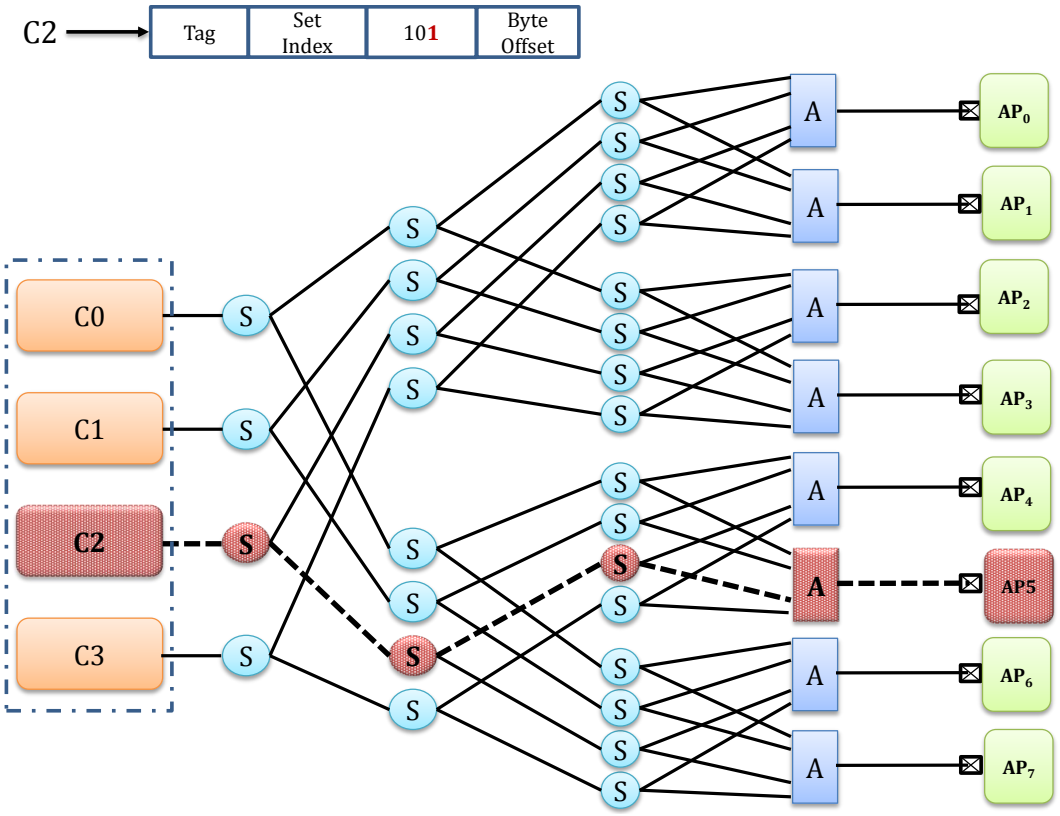
The second layer of the interconnection network is the *arbitration layer*. This layer takes care of the bank contentions that can happen when multiple cores access the same bank. The arbiters grant one input out of four inputs and consume one extra pipeline stage.

4.5.1 Network routing and reconfiguration examples

In Figure 4.11 we show two scenarios in which cores C0 and C2 send memory requests to the access points 0 and 5 respectively (the dashed lines highlight the paths from the cores to the access points). The length of the *Access Point* bitfield is three and each bit from MSB to LSB of the field is used by the switches respectively for routing the packet to the corresponding access point. The memory request is sent from the load-store queue and travels the normal memory translation process by using the internal TLBs. The translated request is then sent to our interconnection network that maps it to the corresponding access point. In the first case (Figure 4.11a) the switch connected directly to the core C0 will check the MSB bit in the access point bitfield. As the bit is zero, the switch will redirect the request in the upward direction to the second switch. Here, the middle bit is checked and as the bit is zero, the request will be directed up to the last switch that again checks the LSB bit and as this bit is zero will direct the packet in the upward direction to the arbiter. This entire routing process consumes one cycle. In the next cycle the arbitration layer decides what to do with the request. In this case, there is no arbitration to be done and the request will be directed to the access point which can forward it to a cache bank or two cache banks (in case of increased



(a) C0 sends request to access point 0.



(b) C2 sends request to access point 5.

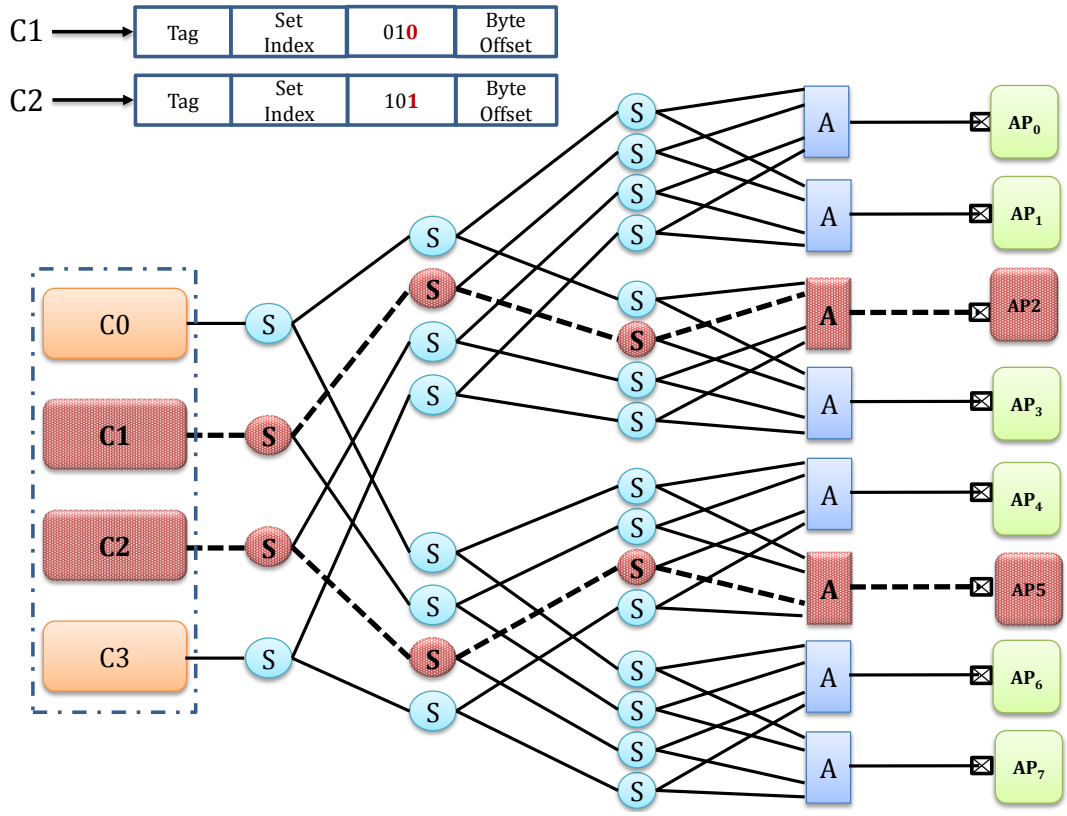
Figure 4.11: Address mapping examples for different scenarios.

associativity).

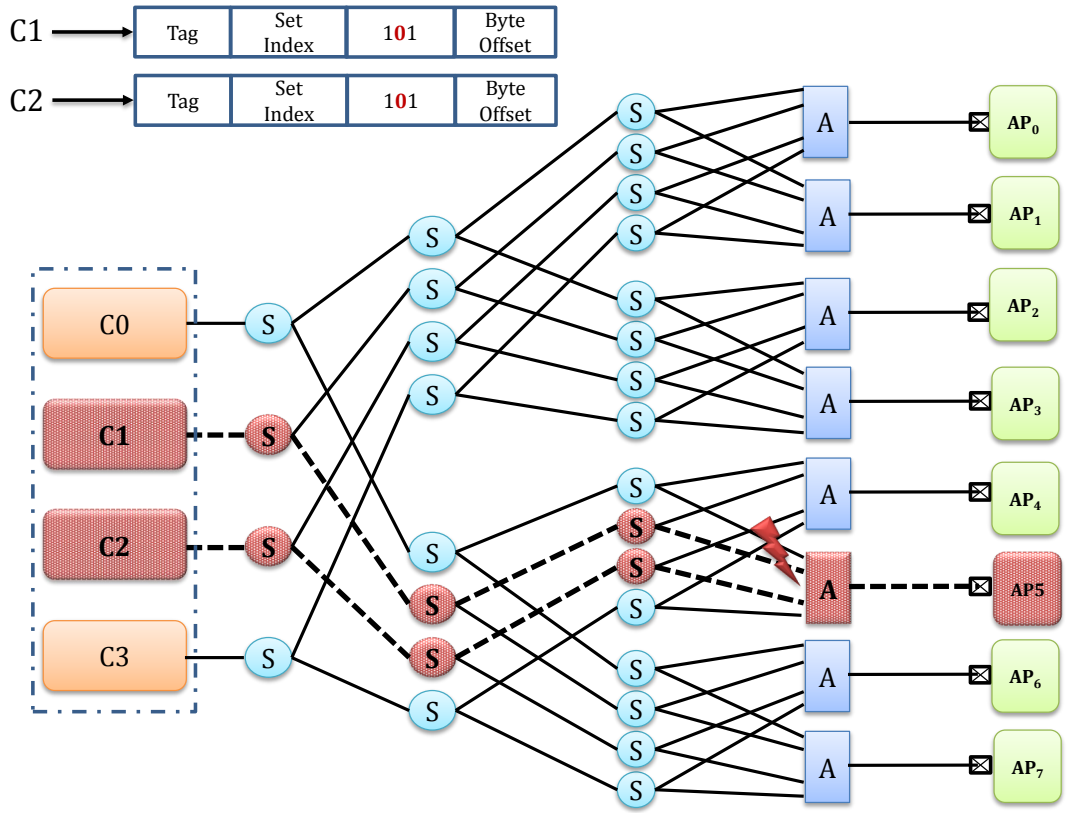
In the second case (Figure 4.11b), the core C2 is issuing the memory request. After translation, the immediate switch will check the MSB of the access point field. In this case the MSB is set to one and the switch will forward the packet in the downward direction to the second switch. Similarly, the second switch checks the middle bit and as the bit is zero the packet will be forwarded in the upward direction to the last switch that again, checks the LSB bit (set to one) and forwards the request in the downward direction to the arbiter. In the second cycle, the arbiter will detect no arbitration conflict and simply forwards the packet to the access point.

The powerful property of this interconnection network lies in the availability of bandwidth by allowing multiple requests to be serviced per cycle. In Figure 4.12 we show two scenarios in which cores C1 and C2 send simultaneous requests. In Figure 4.12a the requests are sent successfully to different access points (2 and 5). In this case, C1 issues the request to the access point 2. The MSB bit is zero so the first switch will direct the request upward to the second switch which will direct the request downward towards the last switch as the middle bit is one. Here, the switch will direct the request upward to the arbiter. C2 issues its request to the access point 5 through the three switches that redirect the request downward, upward and downward again based on the bits in the access point field. We notice that both arbiters allow the request in the second cycle with no arbitration conflict. If the requests generate a hit in the corresponding cache banks then both cores will obtain their response simultaneously.

It is possible that two or more cores will access the same access point. In case of Figure 4.12b the requests coming from C1 and C2 are sent at the same time to the same access point (5). In this case, both requests will have the same values for their Access Point bitfield (i.e., 101). Notice that the traversal of the network does not generate any conflict as it is done in parallel. There is an unique path from each core to each arbiter. After the two requests traverse the network, in the second clock cycle, the arbiter will



(a) C1 and C2 send requests to access points 2 and 5.



(b) C1 and C2 send request to access point 5 which generates an arbitration conflict.

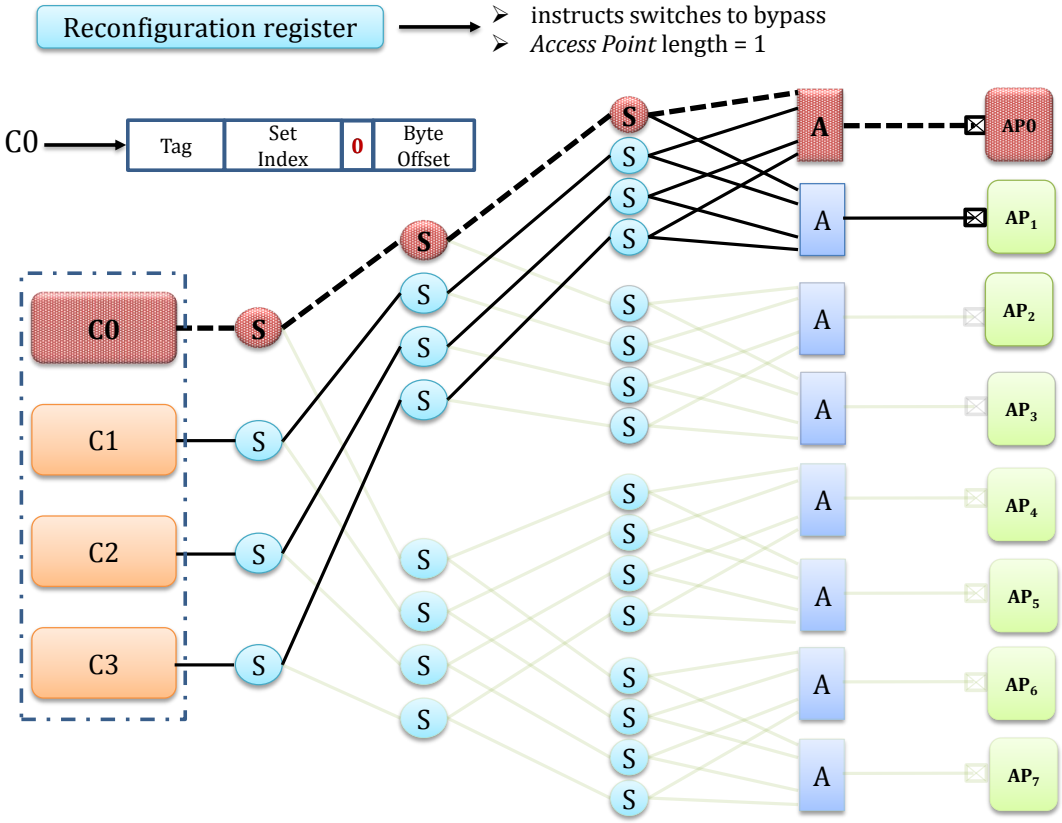
Figure 4.12: Address mapping examples for simultaneous requests.

decide which request to grant. In this case, the arbiter will use a simple round-robin scheme to select the request. Assuming that core C1 is granted, a retry response signal goes back to the load-store queue of C2 and announces that its request was not granted. In this case the core will re-issue its memory request in the next cycle.

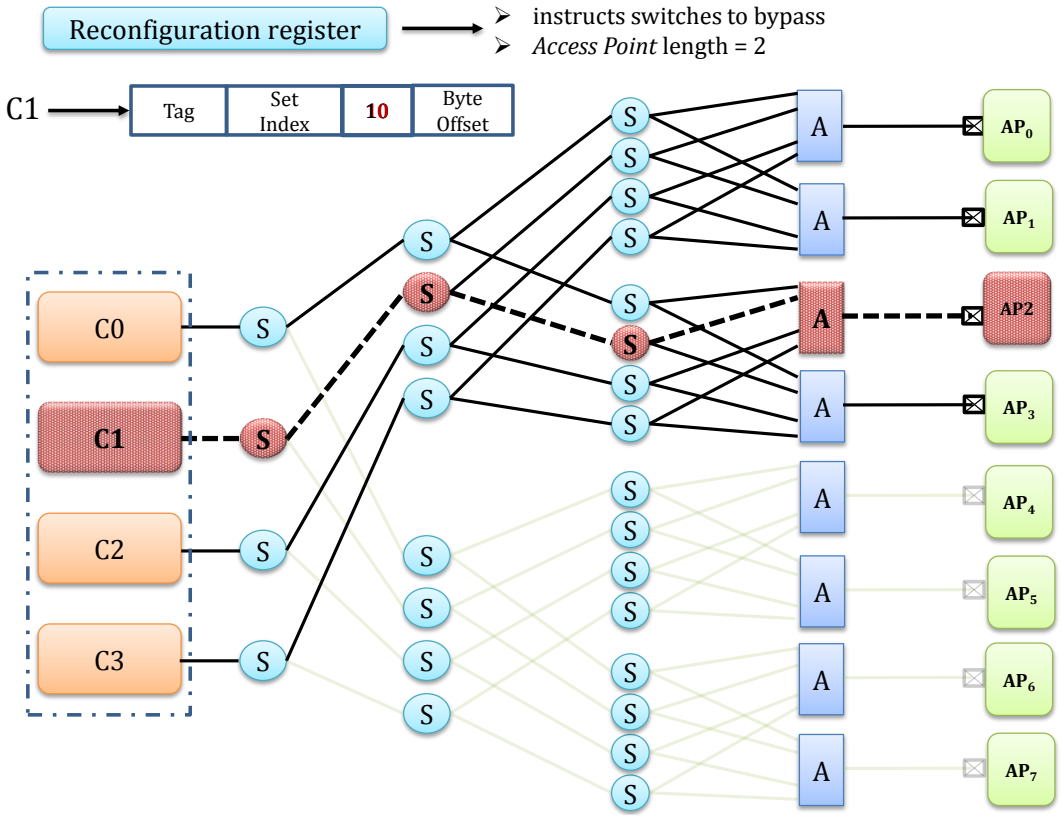
In Figure 4.13 we show how the network reconfiguration works by using the *Reconfiguration Register*. This register is accessible by the cores by adding a new instruction to the ISA. The register is a 56 bit wide (2 bits for each of the 28 switches). For each switch the system can specify one of three operation modes: *normal* mode in which the switch will use the bits from the Access Point bitfield to forward the request to the appropriate output. The second mode is *forward upward* in which the switch blindly forwards the packet upward and the third mode is *forward downward* which instructs the switch to blindly forward packets downward. The last two modes can be used when the network is reconfigured to use less than eight access points. In this cases, the length of the Access Point bitfield is reduced from three to two, one or zero.

In case of Figure 4.13a, the system reconfigures the network such that at most two access points can be addressed (0 and 1). Here, core C0 sends a request to access point 0. We can see that the length of the *Access Point* bitfield is set to one. In this case, the *Reconfiguration Register* will instruct the first two switches to work in the *forward upward* mode, while the last switch will operate in the *normal* mode. The first two switches will simply bypass the request in the upward direction to the last column of switches which will direct the request upward or downward based on the single bit entry. The bit is set to zero; so the request will be directed upward to the arbiter. In this case, we can see that there are only 12 switches used out of 28 and only two access points out of eight. A power-gating solution can be applied here in order to power down the remaining part of the network to reduce more energy. Similarly, the cache banks connected to the inactive access points can be shut down to save even more energy.

In case of Figure 4.13b, the system decides to reconfigure the network such that the



(a) At most 2 access points.



(b) At most 4 access points.

Figure 4.13: Reconfiguration of the number of access points example.

cores can address at most four access points. In this case, the length of the *Access Point* bitfield is two and core C1 sends a request to the access point 2. Here, the *Reconfiguration Register* instructs only the first switch to operate in the *forward upward* mode. The other two switches will operate in the *normal* mode. Half of the network is used in this case and the request goes upward, downward and then upward again to the arbiter which sends the request to the access point 2.

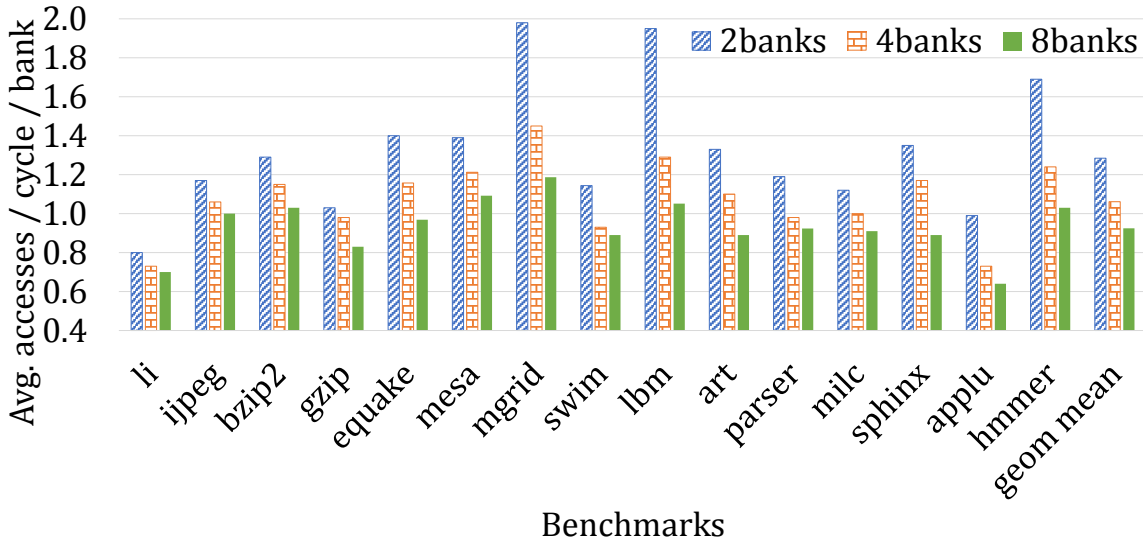


Figure 4.14: L1 data cache average accesses per cycle per banks for 4-core coalition connected to the log network.

4.5.2 Bank conflicts

We are interested to find out the number of access point conflicts that translates into cache bank conflicts. We measure the effect of the address interleaving on the bank conflicts by computing the average number of conflicts across the banks when using 2 banks, 4 banks and 8 banks. We assume that at each access point we attach a single cache bank. Figure 4.14 shows the average accesses per cycle per bank in case of a 4-core coalition connected to the interconnection network. As we see, when moving to more banks connected to the network, the address interleaving reduces the bank conflicts considerably. On average, the bank conflicts for 2 banks, 4 banks and 8 banks is 1.28, 1.06 and 0.92 respectively. We notice that the same two benchmarks *mgrid* and *lbm* are having high conflicts per bank when using 2 banks but this reduces to 1.45 and 1.29 respectively when using 4 banks and it reduces even more to 1.19 and 1.05 respectively when using 8 banks.

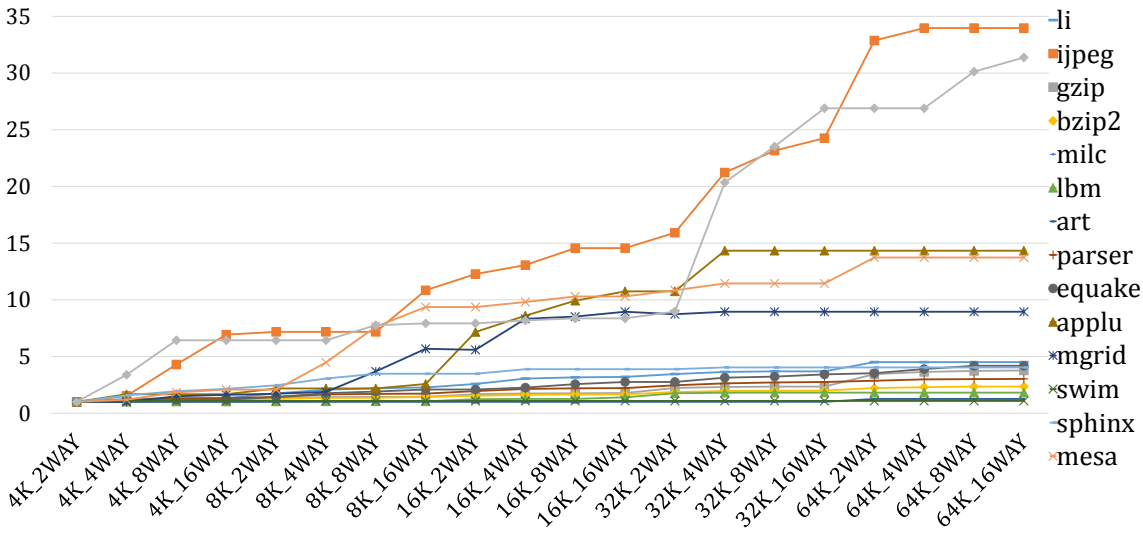


Figure 4.15: L1 data miss rates improvement relative to a 4KB, 2-way set associative cache.

4.6 Cache reconfiguration

4.6.1 L1 data cache miss rates

In our evaluation we also perform a study on the L1 data cache behavior of different benchmarks. In case of many individual applications, the cache parameters fixed at fabrication time are not the ideal ones in terms of energy and area efficiency. We have collected traces of L1 data cache accesses from different benchmarks using the reference inputs. We measured the miss rate of different cache configurations by using DineroIV [38].

In Figure 4.15 we show the L1 data cache miss rate improvement of the applications with different cache parameters relative to the miss rate obtained on a 4KB, 2-way set associative cache. In all the experiments the cache line size is set to 64 Bytes. For typical L1 data cache sizes found in modern processors from ARM or Intel (i.e., 32KB, 2-way set associative or 32KB, 8-way set associative respectively) we can see that many applications gain very little or nothing across many neighbouring cache configurations.

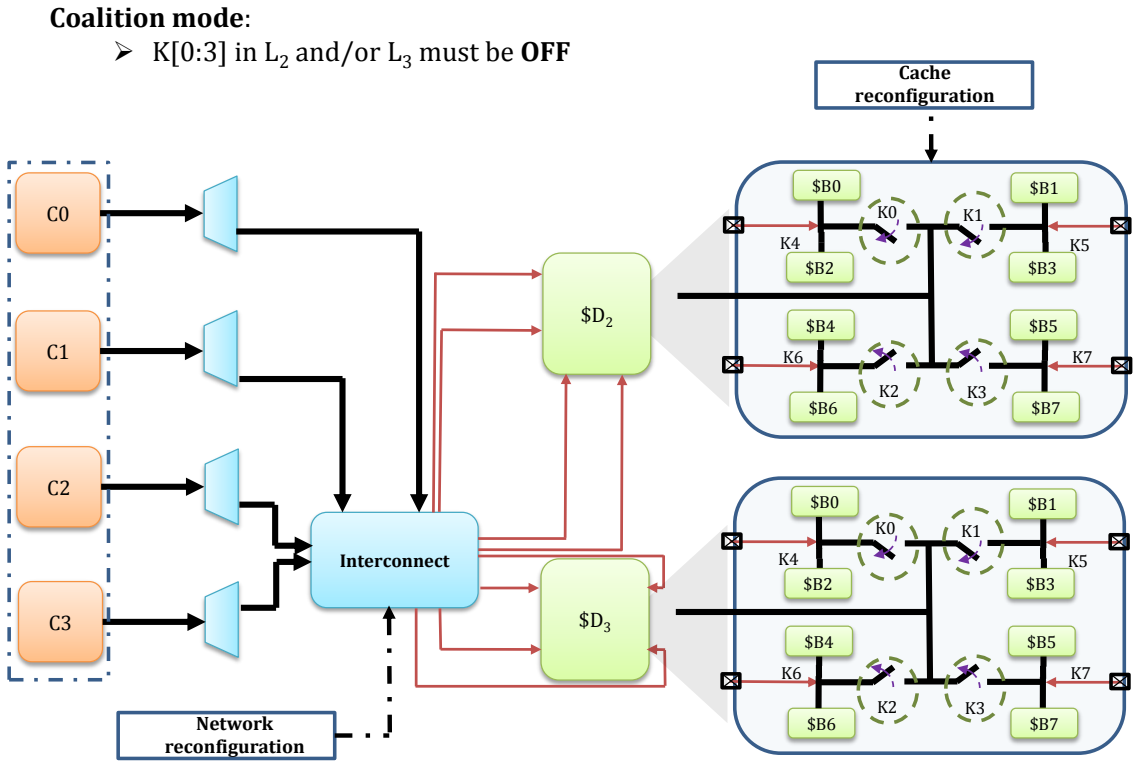


Figure 4.16: Cache reconfiguration for coalition mode.

For example benchmarks *mesa, swim and equake* have similar miss rate improvements around this configuration. In these cases, reducing the cache size would be desirable from energy efficiency point of view.

Now we will present how the cache reconfiguration works. In our design, the amount of cache shared by the cores can be reconfigured at runtime. We connect the access points of the caches D_2 and D_3 to the H-tree as shown in Figure 4.7 and make the H-tree reconfigurable through the switches $K0-K7$. When switches $K0-K3$ are off, the system and the h-tree runs in *coalition mode* (as shown in Figure 4.16). We add a new instruction to the Bahurupi ISA that allows control on the *Cache Reconfiguration Register* which is 16 bit wide (8 switches for each cache).

Switches $K4-K7$ introduce an additional reconfiguration knob to the cache, by which one can double the size and associativity of the cache bank. For example, by

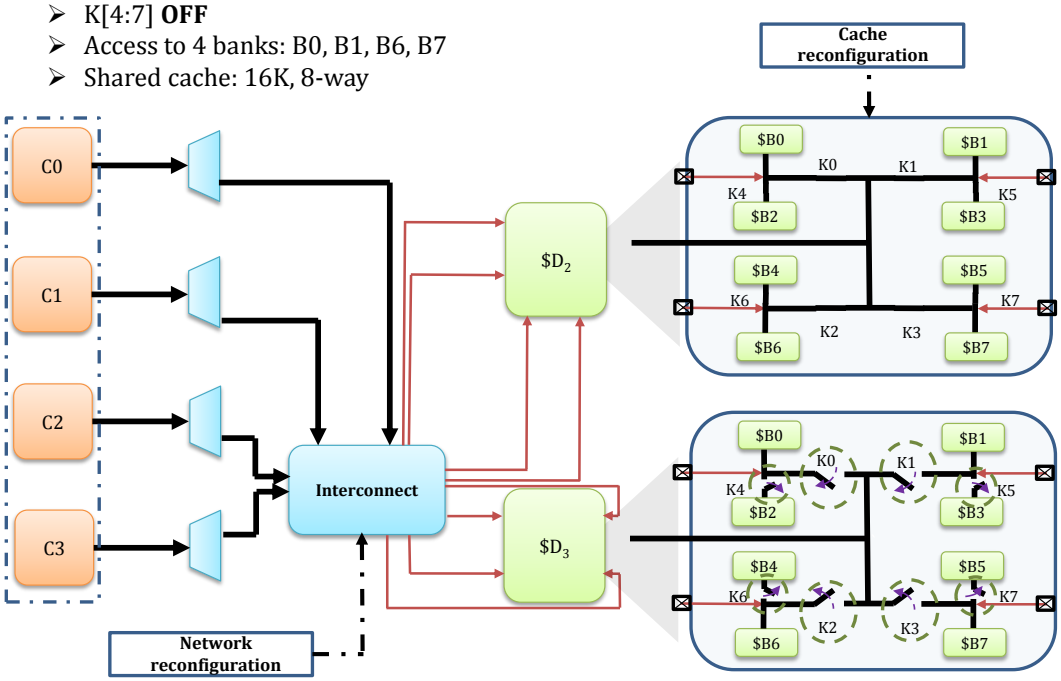
turning on the switch $K4$ the arbiter will access a 8KB, 16-way set associative formed by two banks ($B0$ and $B2$). The request coming from the corresponding access point is simply broadcast to both cache banks. This operation does not require any change in the number of cache sets of bank $B0$; hence the address mapping remains the same and the system does not need to flush the content of the bank to the lower level cache. When switches $K4 - K7$ are off, the banks $B2$, $B3$, $B6$ and $B7$ are powered down.

We enumerate all possible shared L1 data cache configurations obtained by doing network reconfiguration and cache reconfiguration in Table 4.2. For configurations that allow access to 16KB/16-way and 8KB/16-way caches the table does not show all possible switch combinations that can be used. For example, for accessing 8KB/16-way cache the switches K5, K6 or K7 could also be used depending on which access point the request is coming from and how the network was configured.

In Figure 4.17 we show two examples of cache reconfigurations. In the first case (Figure 4.17a), the system decides to use only four access points connected to the D_3 cache by doing network reconfiguration – the length of the Access Point bitfield is two, the first level of switches are set to the mode *forward downward* and the cache D_2 is switched off. The system also reconfigures the cache by accessing the *Cache Reconfiguration Register* such that all switches K0-K7 are off. In this case, the amount of shared cache allocated to the coalition of cores will be 16KB and the total associativity is equal to the associativity of a single bank (8-way set associative). For many benchmarks a 16KB, 8-way set associative L1 data cache offers a good hit rate (see Figure 4.15). If the number of access points offers a good interleaving factor (low arbitration conflicts) than this configuration can offer a very good energy-performance tradeoff.

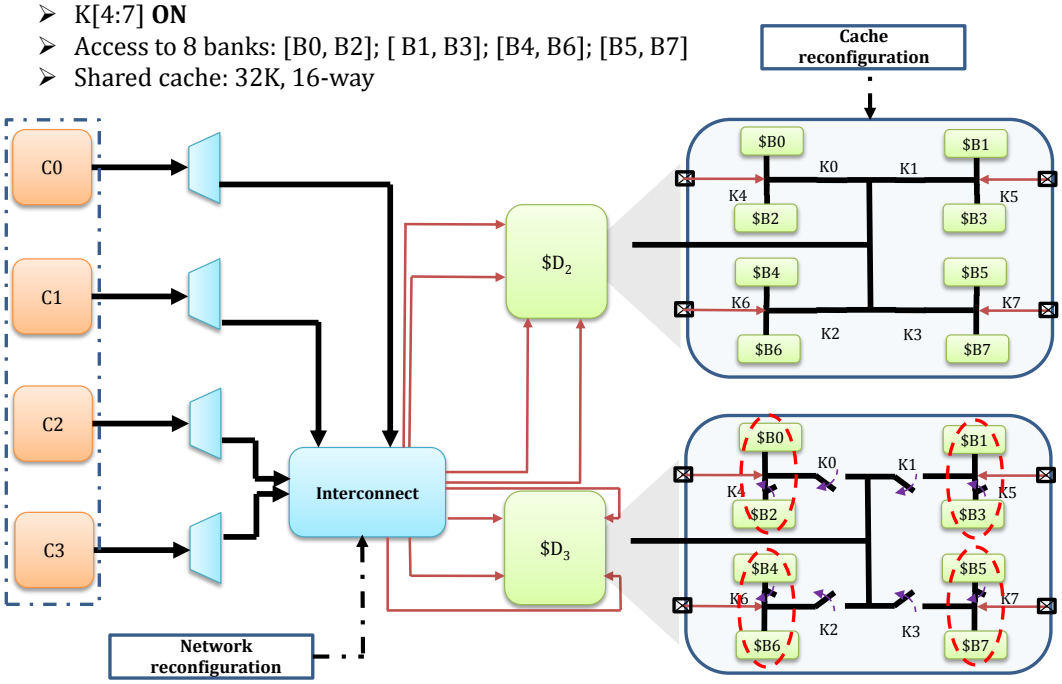
In the second case (Figure 4.17b), we show how the cache reconfiguration works when we want the core coalition to share a cache double in size and associativity. The system uses the *Cache Reconfiguration Register* to switch on K4-K7. As discussed previously, when doing this, we do not need to flush the contents of the cache banks

- *Network Reconfiguration*: 2 bits for the *Access Point*
- K[0:3] **OFF**
- K[4:7] **OFF**
- Access to 4 banks: B0, B1, B6, B7
- Shared cache: 16K, 8-way



(a) Cache reconfiguration to 4 access points.

- *Network Reconfiguration*: 2 bits for the *Access Point*
- K[0:7] **OFF**
- K[4:7] **ON**
- Access to 8 banks: [B0, B2]; [B1, B3]; [B4, B6]; [B5, B7]
- Shared cache: 32K, 16-way



(b) Cache reconfiguration to 4 access points and increased associaitvity.

Figure 4.17: Example of cache reconfiguration.

Access Point bitfield width	Shared cache size/assoc.	# access points	Banks used from	Switches on
3	64KB/16-way	8	$L1D_2$ & $L1D_3$	K4-K7
3	32KB/8-way	8	$L1D_2$ & $L1D_3$	–
2	32KB/16-way	4	$L1D_3$	K4-K7
2	16KB/8-way	4	$L1D_3$	–
1	16KB/16-way	2	$L1D_3$	K4&K5
1	8KB/8-way	2	$L1D_3$	–
0	8KB/16-way	1	$L1D_3$	K4
0	4KB/8-way	1	$L1D_3$	–

Table 4.2: Amount of shared cache accessed based on the *Access Point* bitfield width and the cache reconfiguration switches position.

B0, B1, B4 and B5. Thus, the access points will be connected to group of banks that double the associativity to 16-way: (B0-B2), (B1-B3), (B4-B6) and (B5-B7). The total amount of shared cache is 32KB which can offer a very good hit rate.

As we can see, increasing the *Access Point* bitfield width not only has the effect of increasing the interleaving factor by accessing more banks but also it increases the amount of shared cache. Consequently, our design can allow the adaptive multi-core to access a wide variety of cache sizes and associativity starting from 4KB, 8-way set associative and ending with a 64KB, 16-way set associative shared cache.

We now move on to show the area and energy consumption of our design and then we do a performance analysis.

4.6.2 Area and energy consumption

The total area occupied by our design is mainly given by the area occupied by the logarithmic network and the arbiters and the area occupied by the 8 single ported cache banks. We have implemented in Verilog and synthesised the network and the arbiters for both core-to-cache and cache-to-core using Synopsys [92] with 32nm Generic Library. The total area occupied by our design is 1.07 times bigger than the area occupied by a single-ported 32KB, 8-way associative cache. This gives us a great advantage over the

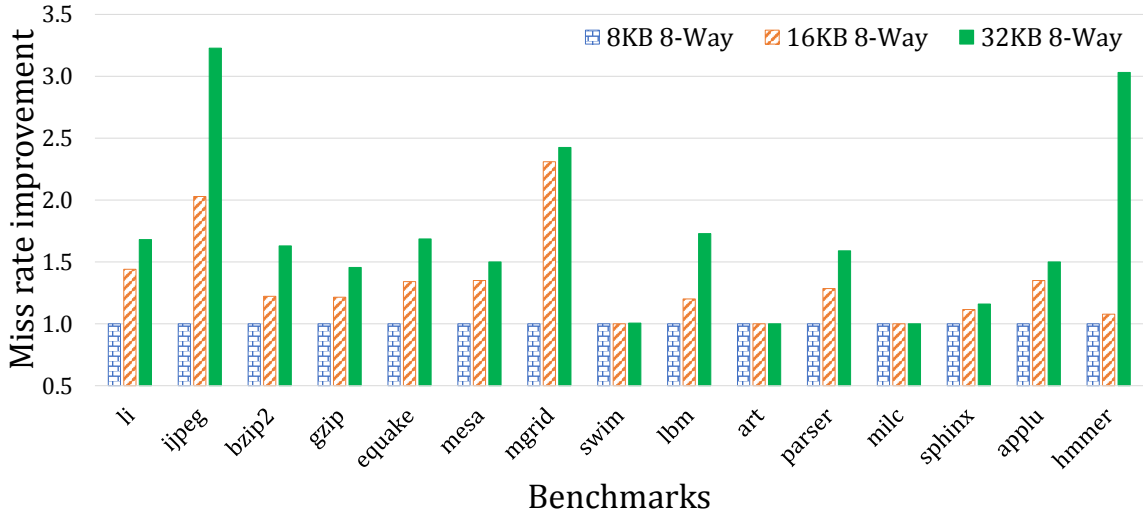


Figure 4.18: Miss rate improvement for 2 banks, 4 banks and 8 banks (normalized w.r.t miss rate of 2 banks).

multi-ported caches.

In terms of energy consumed per access, we assume that in coalition mode, the h-tree network (together with the predecoders) is mostly turned off. This saves considerably amount of power as the h-tree network is a power hungry component in a cache. The results show that our design consumes 1.18 times more energy than the single ported private cache.

4.6.3 Miss rate improvement and performance analysis

The network can be reconfigure to reduce the bank conflicts, the miss rate or both. In Figure 4.18 we show the improvement in miss rate when moving to higher number of banks. The numbers are normalised w.r.t the miss rate obtained when using 2 banks. Here, we do not switch on K4-K7, thus, each access point corresponds to a cache bank. We can see that there are benchmarks for which the increase in the cache size brings very small or no contribution to the improvement in the miss rate. In case of *milc* benchmark, there is no improvement in the cache miss rate and also there is small bank conflict reduction when moving to more banks (see Figure 4.14). In this case using

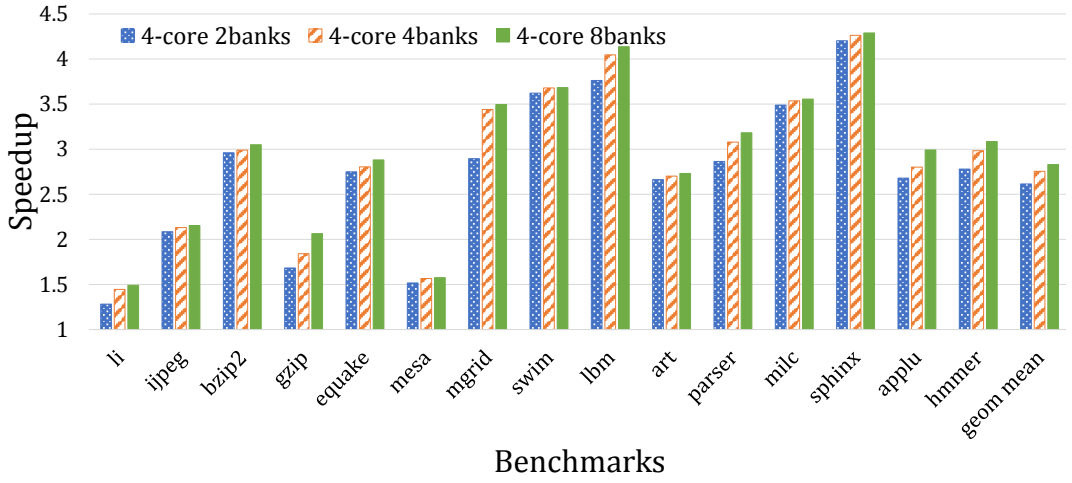


Figure 4.19: Speedup of 4-core coalition connected to 2, 4 and 8 banks using the log network (normalized w.r.t performance of a baseline single core).

only 2 access points (2 banks) would save significant amount of energy per access with minimal performance loss.

The reduction in bank conflicts plus the increase in the L1 data cache size has a positive impact on the performance of some of the applications. In Figure 4.19 we show the speedup (IPC ratio) obtained when running a 4-core coalition on the logarithmic network connected to 2, 4 and 8 banks normalized w.r.t the performance of a baseline 2-way out-of-order core. On average, when moving from 2 banks to 4 banks the increase in speedup is 14%, relative to the baseline core, while when moving from 4 banks to 8 banks the increase is 7% relative to the baseline core.

In Figure 4.15 we can see that there are benchmarks that are receiving great miss rate improvement when moving to higher cache sizes and associativity. For example, the benchmark *equake* running on 16KB, 16-way set associative cache improves the miss rate by 43% compared with 8K, 8-way set associative. In this case a cache reconfiguration can be done by turning the switches K4 and K5 on. Notice that having a higher cache size does not improve the miss rate any further for this benchmark. Thus, by only using two access points we can obtain a good miss rate with low energy con-

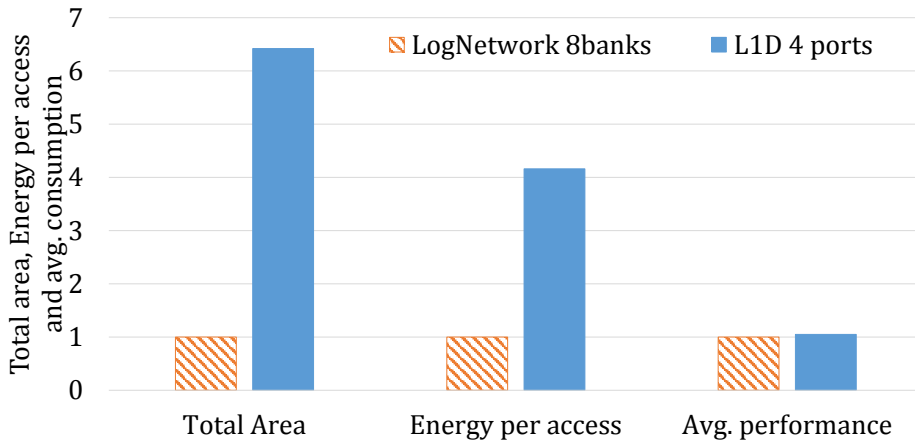


Figure 4.20: Comparison between our design (8banks) and four-ported cache.

sumption.

4.7 Comparison with multi-ported shared L1 cache

Our main goal is to have a design that can allow the coalition of cores to provide performance as close as possible to the one obtained when using a multi-ported cache. At the same time, we want our design to occupy very low area and consume very low energy per access. In fact we want these values to be closer to the area and energy consumed by the private caches.

Figure 4.20 summarises the comparison between the four-ported cache and our L1 data cache design in terms of area, energy and performance. The multi-ported values are normalized w.r.t the results obtained for our design. The multi-ported cache occupies 6.42X more area than our design. Moreover, our design consumes 4.16X less energy per access compared to the multi-ported design. In terms of performance, when using 8 access points and run on 4-core coalition, our design only looses 5% (normalized to the baseline core) compared with the multi-ported cache. In conclusion, our novel architecture consumes much less area and energy per access when compared with the multi-ported cache with minimal performance penalty.

As we can see, by only using 8 access points the network offers enough interleaving (low arbitration conflicts) to obtain a performance very close to the one obtained when connected to a four-ported cache. Thus, scaling the network to larger number of access points will not bring significant performance improvement. Moreover, scaling the network to more than 8 access points brings the risk of increasing the power consumption above the accepted limits.

4.8 Summary

In this chapter we have approached the architecture of the first level of data cache used by the dynamic heterogeneous multi-core architectures. A coalition of cores requires reconfiguration of the first level of data cache that must efficiently accommodate the bandwidth and latency needed by such architectures. We propose and implement a novel L1 data cache design that is able to overcome the limitations in terms of area, energy and performance of multi-ported shared caches. Results show that our design is able to accommodate successfully the memory demands of Bahurupi providing high performance with very low area and energy consumption. Additionally, our design offers dynamic reconfiguration capabilities. It can reconfigure at runtime the number of cache banks that can be accessed in parallel reducing the L1 data bank conflicts. At the same time, the design can also reconfigure the size and associativity of the shared cache dynamically.

Chapter 5

Scheduling on Bahurupi Architecture

In this chapter we conduct a performance limit study for Bahurupi by employing optimal schedulers in order to obtain the ideal speedup of applications on an ideal dynamic heterogeneous multi-core architecture that has no hardware limitations. We then conduct a performance limit study of Bahurupi and see how the real hardware constraints affect the speedup of the applications running on the dynamic architecture. In this thesis we consider a more realistic scenario where both parallel and sequential tasks coexist in the system.

Illustrative Example. Here, we present an example that provides a visual illustration of our scheduling problem. This example also concretely explains the challenges involved in conducting the performance limit study. For this example, we have chosen a set of five benchmarks: three sequential applications (*gobmk*, *quantum* and *fft*) from SPEC [2] and MiBench [43] benchmark suites that can exploit ILP through complex out-of-order cores, and two parallel applications (*bodytrack* and *blackscholes*) from PARSEC [16] benchmark suite that can exploit TLP through multiple simple cores. The experimental setup used to obtain the performance of each individual benchmark on different number of cores and configurations will be presented in Section 5.3.

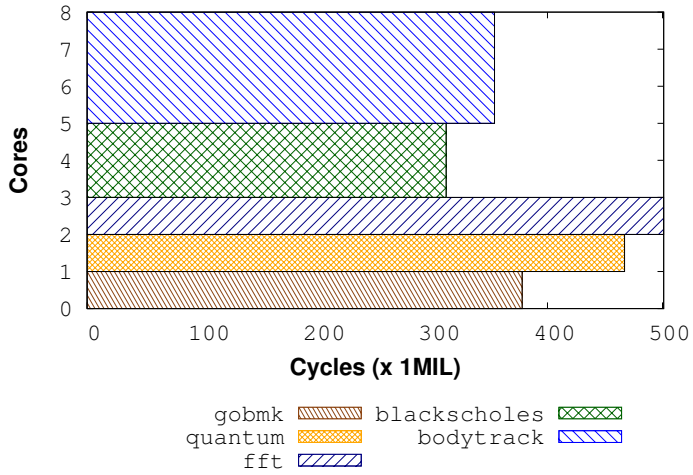
We consider three different multi-core architectures that were introduced in Chapter 1: (a) homogeneous multi-core architecture with eight 2-way out-of-order cores, (b) static heterogeneous multi-core architecture with one 8-way out-of-order core and four 2-way out-of-order cores, and (c) dynamic heterogeneous multi-core architecture with eight 2-way cores where the cores can be coalesced together to form 4-way, 6-way, or 8-way complex core. If we assume the area requirement of a 4-way core is roughly equivalent to that of two 2-way cores and the area of a 8-way core is roughly equal to that of two 4-way cores then all the architectures are area-equivalent to the homogeneous architecture with eight 2-way cores.

The Gantt charts presented in Figure 5.1 show the schedules for different architecture along with the makespan (the time when all the applications finish execution). The lower the makespan, the better is the throughput of the system.

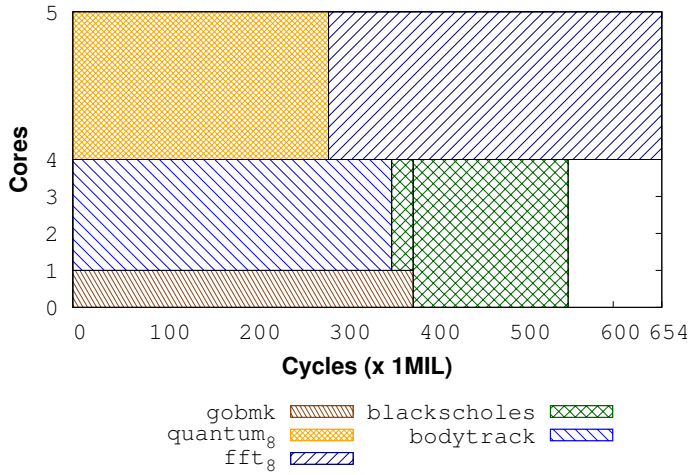
For the homogeneous multi-core, the sequential applications are restricted to using only one core, while the parallel applications can benefit from multiple cores. This severely restricts the performance and the makespan of 500 million cycles is defined by the sequential application *fft*.

In the static heterogeneous multi-core, we provide opportunity for the sequential applications to exploit ILP through one 8-way core. But the parallel applications are restricted to use only four simple cores. The sequential applications *fft* and *quantum* attempt to take advantage of the complex core to reduce execution time and reduce the demand for the simple cores. But this choice only leads to increased makespan. This example clearly shows that a fixed static heterogeneous solution may not always be the best replacement for the homogeneous architecture due to the lack of flexibility and availability of the number of cores.

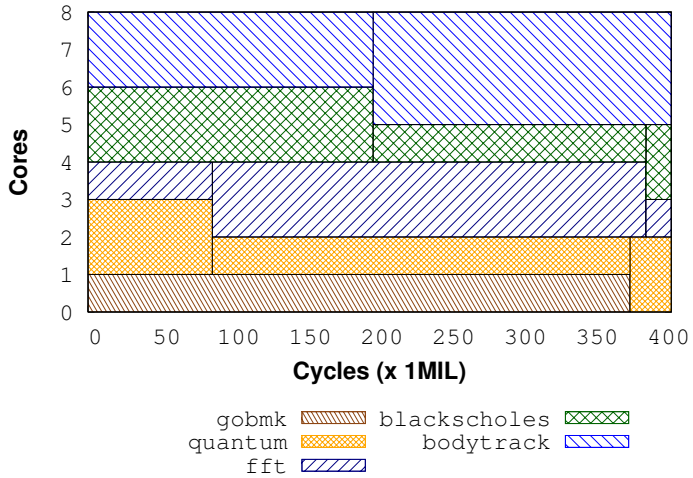
The dynamic heterogeneous architecture, on the other hand, carefully selects the number of cores allocated to each application. In this case, the sequential applications exploit ILP through core coalition, while the parallel applications exploit TLP by using



(a) Homogeneous multi-core



(b) Static heterogeneous multi-core



(c) Dynamic heterogeneous multi-core

Figure 5.1: Illustrative example showing the schedule of a mix of sequential and parallel applications on different architectures.

multiple simple cores. In fact, the dynamic heterogeneous architecture dynamically creates five different static heterogeneous multi-core configurations during the makespan, in contrast to rigid homogeneous and static heterogeneous solutions.

The example illustrates the challenges in forming the schedule for a dynamic heterogeneous multi-core architecture. For the optimal schedule in Figure 5.1, we have to first determine the allocation of the cores to the applications. The problem is even more challenging because an application can be allocated varying number of cores over time. For example, *fft* uses two cores for certain time interval and one core for the remaining time. Moreover, an application may be allowed to migrate from one core to another during its execution even if it uses fixed number of cores throughout execution. Finally, any realistic dynamic heterogeneous architecture imposes additional scheduling and allocation constraints that need to be included in our decision process.

5.1 Optimal schedule on ideal dynamic heterogeneous multi-core

The goal of this section is to conduct a quantitative performance limit study of the dynamic heterogeneous multi-core architectures. We design an efficient off-line scheduler to carry out this limit study. We first present an optimal scheduler for an ideal dynamic heterogeneous multi-core architecture that is not restricted by any physical or technological constraint. Next, we impose additional constraints to perform scheduling on our realistic dynamic heterogeneous multi-core Bahurupi.

The ideal dynamic heterogeneous multi-core architecture consists of m physical 2-way superscalar out-of-order cores supporting shared memory through hardware cache coherence. Any subset of these cores can be coalesced together to form one or more complex out-of-order cores. If r cores ($r \leq m$) form a coalition, then the resulting complex core supports $2r$-way superscalar out-of-order execution. The architecture can

support any number of coalitions as long as the total number of cores included in all the coalitions at any point in time does not exceed m. We assume that the core coalition does not incur any performance overhead, that is, the performance of a $2r$-way core coalition is identical to a native $2r$-way core. A parallel application can execute on any subset of the simple cores, while a sequential application can execute on any simple core or a core coalition.

The dynamic heterogeneous multi-core architecture allows both sequential and parallel applications to use time varying number of cores. Thus we model the applications as malleable workload [60], where the number of cores allocated per application is not fixed and can change during execution through preemption. For the limit study, our goal is to create the optimal schedule for the malleable tasks [1] on the ideal dynamic heterogeneous multi-core architecture.

The scheduling problem can be formulated as follows. We consider an ideal dynamic heterogeneous multi-core architecture consisting of m homogeneous independent physical processors $\{P_0, P_2, ..., P_{m-1}\}$ running a set of n ($n \leq m$) preemptive *malleable tasks* $\{T_0, T_2, ..., T_{n-1}\}$. We assume that all the tasks arrive at time zero. The objective is to allocate and schedule the tasks on the cores so as to minimize the makespan $C_{max} = max_j\{C_j\}$ where C_j is the finish time of task T_j.

5.1.1 Optimal schedule with continuous resources

We first determine the optimal C_{max} assuming that the number of cores allocated to a task need not be an integer. Then we transform this schedule to one that uses discrete resources.

Let us denote the number of processors assigned to a task T_j by r_j, where $0 < r_j \leq m$. If r_j is a continuous renewable resource (i.e., r_j can have real value), then we can adopt the solutions presented for the continuous resource allocation problem [100, 25] as our

[1] We use the terms *task* and *application* interchangeably.

starting point.

Each task has a fixed amount of processing work $p_j > 0$. In a time interval of length t, a task performs $t \times g(t)$ amount of work where $g(t) = f_j(r_j) \geq 0$ is a continuous non-decreasing processing *speedup function* that relates r_j to the processing speed of a task. The set of *feasible resource allocations* of processors to tasks is as follows.

$$R = \left\{ r = (r_0, ..., r_{n-1}) \mid r_j > 0, \sum_{j=0}^{n-1} r_j \leq m \right\}$$

Applying speedup function $f_j(r_j)$ over the elements of R we obtain the set of *feasible transformed resource allocations*

$$U = \left\{ u = (u_0, ..., u_{n-1}) \mid u_j = f_j(r_j), j = 0, ..., n-1, r_j \in R \right\}$$

Theorem 1. (Resource allocation theory [100])

Let $n \leq m$, convU be the convex hull of the set U, i.e, the set of all convex combinations of the elements of U, and $u = p/C$ be a straight line in the space of transformed resource allocations given by the parametric equations $u_j = p_j/C, j = 0, ..., n-1$. Then, the minimum schedule length is

$$C^0_{max} = min \left\{ C \mid C > 0, \frac{p}{C} \in convU \right\} \tag{5.1}$$

where $p = (p_0, ..., p_{n-1})$ is the processing work for the tasks.

From (5.1) it follows that, the minimum makespan value C^0_{max} is given by the intersection point of the line $u = p/C$ and the boundary of the convU set in the n-dimensional space of transformed resource allocations. The boundary of the convU set has a shape that depends on the convexity or concavity of the speedup functions f_j. The referred resource allocation solution is only valid for concave speedup functions f_j. In our evaluation, all the applications we tested can be approximated with concave speedup

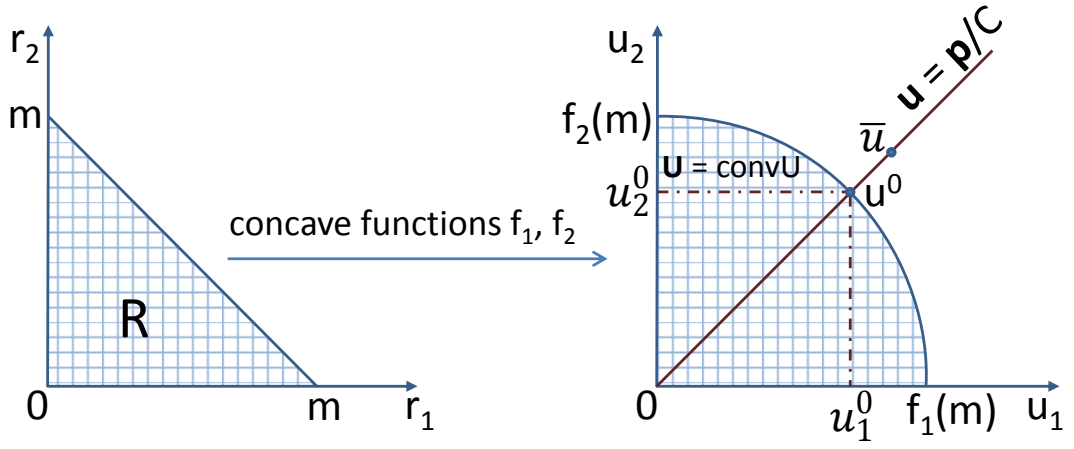


Figure 5.2: Resource transformation example for $n = 2$.

functions. In Figure 5.2, we give a geometrical interpretation of the resource allocation problem applied in the case of $n = 2$ tasks, m processors and concave speedup functions f_0, f_1. The set of feasible allocations R is transformed into the set of feasible transformed allocations U by applying the speedup functions.

The value of C^0_{max} is determined by the intersection point u^0, $C^0_{max} = p_j/u^0_j, j = 0,...,n-1$. Thus, in order to find the minimum schedule length for our problem, we have to find the point u^0. [25] presents an algorithm that finds the solution for the continuous resource allocation problem in $O(n\ max\{m, n\log^2 m\})$ time.

Normally, the speedup functions are only defined at integer points for a resource (discrete functions). The speedup functions f_j are extended with piecewise linear functions between consecutive r_j points. This way the monotonicity and concavity properties of the functions are maintained. The piecewise linear functions are described by the equations

$$\begin{gathered} f_j(r) = b_{j,s}r + d_{j,s},\ r \in [s-1,s], \\ s = 2,...,m,\ j = 0,...,n-1, \\ b_{j,0} = d_{j,0} = 0 \end{gathered} \tag{5.2}$$

Figure 5.3 shows an example of piecewise interpolation applied to the discrete speedup

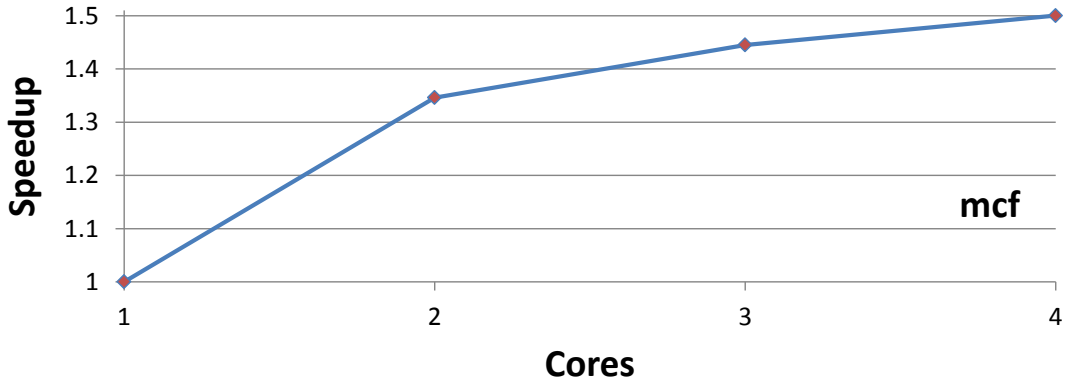


Figure 5.3: Piecewise interpolation of speedup function for *mcf*

function for the *mcf* benchmark from SPEC benchmark suite with $m = 4$. If using more than four cores, the observed speedup is minimal. Note that *mcf* is a sequential application. So the speedup on r cores correspond to the speedup on $2r$-way out-of-order core compared to the baseline 2-way out-of-order core.

5.1.2 Optimal schedule with discrete resources

Clearly, considering processors as continuous renewable resources is not a realistic assumption. Fortunately, the solution to the continuous problem can be transformed into a discrete solution with the same optimal makespan value C^0_{max} [24].

The discrete solution is obtained from the continuous version through a rectangle packing procedure where two rectangles are allocated to each task. The rectangles represent the processing work and the number of cores allocated to a task. The dimensions (height and width) of the rectangles (a_j, v_j), (b_j, w_j) are computed as follows

$$\begin{aligned} a_j &= \lfloor r^0_j \rfloor,\ b_j = \lceil r^0_j \rceil \\ v_j &= (b_j - r^0_j) p_j / f_j(r^0_j) \\ w_j &= (r^0_j - a_j) p_j / f_j(r^0_j) \end{aligned} \tag{5.3}$$

Essentially, Equation 5.3 rounds the processor allocation r^0_j up and down to integer

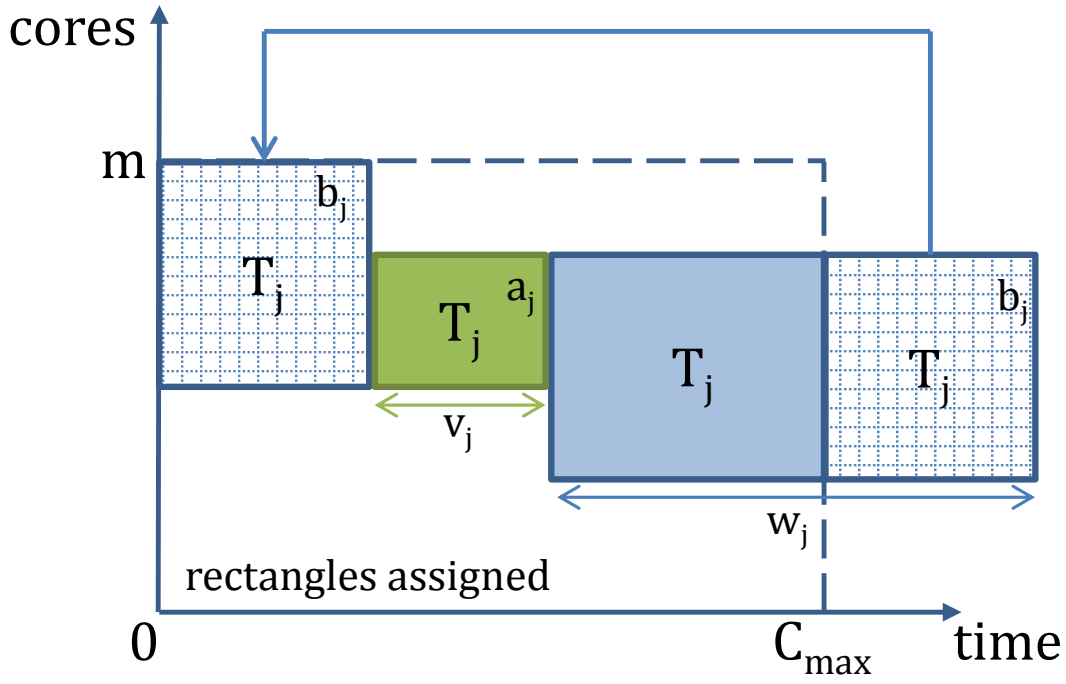


Figure 5.4: Rectangle packing for discrete resource problem.

values and represent r_j^0 as linear combinations of these two values. Consequently $b_j - a_j = 1$ for any task T_j. The rectangles are packed in (C_{max}^0, m) rectangle using the *rule of the southwest corner* in which a new rectangle is always assigned the leftmost position at the bottom of the unoccupied area. It can be proved [24] that the total width of the two rectangles can not exceed C_{max}^0. Also a rectangle always fits within the height m. Once a rectangle exceeds C_{max}^0 along the width, it is cut and the excess portion is moved back inside following the packing rule.

An example is shown in Figure 5.4, where two rectangles of height b_j and a_j are allocated to application T_j. The rectangle of height b_j is packed first and it exceeds C_{max}^0. The highlighted part of the rectangle is moved back and then the second rectangle allocated for this task is placed. The packing algorithm guarantees at most two preemption points for each task and requires $O(n)$ time. Consequently, the time complexity to optimally schedule the malleable tasks on an ideal adaptive multi-core is $O(n \max\{m, n \log^2 m\})$.

Algorithm 1: Malleable Task Scheduler on Bahurupi

```
AdaptiveScheduler(task_list, m, n) begin
    restart = FALSE;
    Apply_constraint_C1(task_list, m, n);
    Find_contiunous_cmax(task_list, m, n);
    Convert_to_discrete(task_list, m, n);
    Generate_and_pack_rectangles(task_list, m, n, restart);
    if restart == TRUE then
        AdaptiveScheduler(task_list, m, n);
    end
    Apply_constraint_C3(task_list, m, n);
end

Generate_and_pack_rectangles(task_list, m, n, restart)
begin
    constraint_violated = FALSE;
    current_task = 1;
    rectangles = Generate_rectangles(m, n);
    Order_rectangles(rectangles, n);
    while constraint_violated == FALSE do
        Place_rectangles(rectangles[current_task]);
        Apply_constraint_C2(task_list, m, n, constraint_violated);
        if constraint_violated == TRUE then
            restart = TRUE;
            violating_task = current_task;
            break;
        end
        current_task = current_task + 1;
    end
    if constraint_violated == TRUE then
        Constrain_remaining_tasks(current_task, n);
    end
end

Apply_constraint_C2(task_list, m, n, constraint_violated)
begin
    for all time intervals (Δ_j, Δ_{j+1}) do
        n_coalitions = Count_coalitions(Δ_j, Δ_{j+1});
        CL_j = Build_coalitions_list(Δ_j, Δ_{j+1});
        NCL_j = Build_non_coalitions_list(Δ_j, Δ_{j+1});
        if n_coalitions > (m/4) then
            constraint_violated = TRUE;
            break;
        end
    end
end

Apply_constraint_C3(task_list, m, n)
begin
    constraint_violated = FALSE;
    for all tasks in CL_j do
        for 1 ≤ k < (m/4) do
            if task uses cores P_{4k} and P_{4k-1} then
                constraint_violated = TRUE;
                break;
            end
        end
    end
    if constraint_violated == TRUE then
        Pack_coalitions(CL_j);
        Pack_non_coalitions(NCL_j);
    end
end
```

5.2 Task scheduling on Bahurupi

When scheduling tasks on a realistic dynamic heterogeneous multi-core architecture, we must take into consideration all the constraints and limitations imposed by the system. More concretely, for Bahurupi architecture, we need to consider the following constraints in forming core coalitions for sequential tasks. The constraints are actually quite generic and are present in almost all adaptive multi-core architectures in the literature even though the exact values for the constraints can be different.

C1. A sequential application can use at most four cores.

C2. We can form at most $m/4$ coalitions at any time.

C3. A sequential application can only use cores that belong to the same cluster.

There is no such constraints for the parallel tasks. A parallel task may use any number of available cores to minimize the overall makespan. The scheduling solution for Bahurupi needs to add the constraints to the optimal scheduling solution presented in Section 5.1 for the ideal dynamic heterogeneous multi-core architecture. Algorithm 1 presents the scheduling algorithm for Bahurupi.

5.2.1 Constraint C1

Bahurupi restricts any sequential application to use at most four cores due to the limited amount of ILP found in sequential applications and the increase in coalition overhead when using more than four cores. To implement this constraint we modify the set of *feasible resource allocation* such that, the system can allocate at most four cores for sequential applications.

$$R = \left\{ r = (r_0, ..., r_j, ..., r_{n-1}) \mid r_j > 0, \sum_{j=0}^{n-1} r_j \leq m \right\}$$

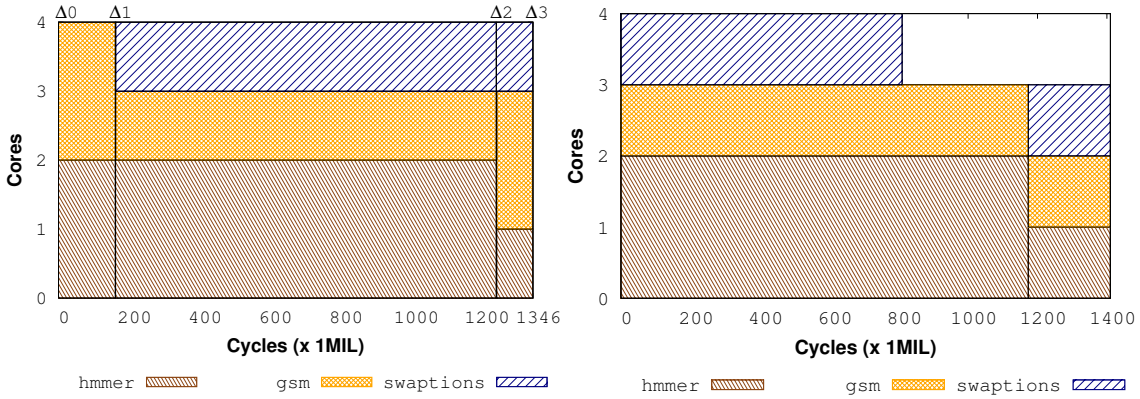


Figure 5.5: Example of imposing constraint C2.

$$max(r_j) = \begin{cases} 4 & \text{if application } T_j \text{ is sequential,} \\ \text{m} & \text{if application } T_j \text{ is parallel} \end{cases}$$

Function *Apply_constraint_C1* in Algorithm 1 implements this constraint for the sequential tasks.

5.2.2 Constraint C2

Bahurupi architecture can accommodate at most one coalition per cluster. For each cluster, Bahurupi uses the coalition logic, which can be allocated to at most one coalition. Figure 5.5 illustrates an example of a 4-core Bahurupi architecture running two sequential tasks, *hmmer* and *gsm* and one parallel task *swaptions*. This architecture can support at most one coalition. The time at which the tasks are preempted are marked on the top of the charts. The original schedule for the ideal adaptive architecture shown in Figure 5.5(a) violates the bound on number of coalitions in the interval $\Delta_0 - \Delta_1$ where there are two coalitions of two cores ($\{P_0, P_1\}$ and $\{P_2, P_3\}$) used simultaneously by the tasks *hmmer* and *gsm*.

The one coalition per cluster constraint is imposed through *Apply_constraint_C2* during the placement of the rectangles in the function *Generate_and_pack_rectangles*

in Algorithm 1. The rectangles are ordered initially by the function *Order_rectangles* in decreasing order of their heights. Rectangles having the same height are ordered in decreasing order of the corresponding processing work p_j. After placing a new rectangle, we scan each time interval (Δ_j–Δ_{j+1}) to count the number of coalitions used by the sequential tasks in that interval. If there are more than $m/4$ coalitions, then the constraint C2 is violated.

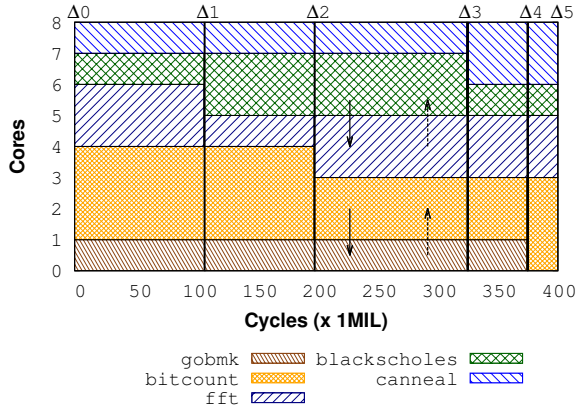
If the constraint C2 is violated, then we abort the rectangle packing and constrain the sequential tasks that are not placed yet (including the last placed task) to use only one processor, i.e., $r_j = 1$. The algorithm is then resumed with the new constraint. Imposing constraint C2 may lead to increased makespan. For example, in Figure 5.5(b), *gsm* is restricted to using only one core and the makespan is increased.

5.2.3 Constraint C3

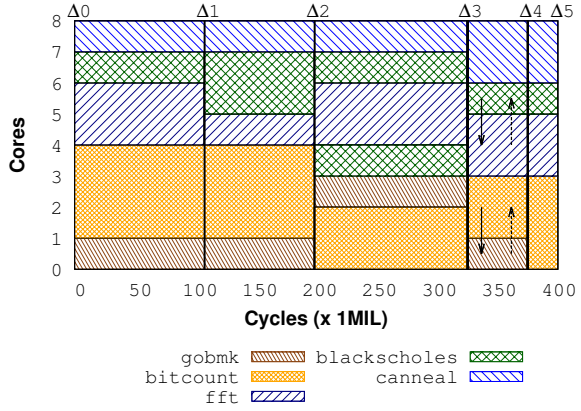
Constraint C3 ensures that a sequential task is restricted to using only the cores within a cluster. In Figure 5.6(a) we consider a set of three sequential applications *gobmk*, *bitcount* and *fft* and two parallel applications, *blackscholes* and *canneal*. We can see that the constraint C3 is violated for time intervals Δ_2–Δ_5 assuming that processors P_0 – P_3 belong to one cluster and P_4 – P_7 belong to another cluster.

Function *Apply_constraint_C3* in Algorithm 1 implements this constraint. To impose this constraint, we first assume that the schedule has already been modified to satisfy the constraints C1 and C2. For each time interval Δ_j–Δ_{j+1}, we check if any sequential task uses cores across clusters. This can be done by simply checking if any sequential task uses cores P_{4k} and P_{4k-1}, where $1 \leq k < (m/4)$. If the constraint is violated in any time interval, then we need to migrate the tasks within that interval.

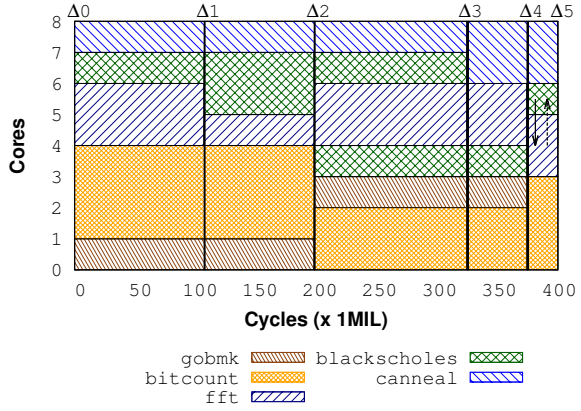
For each time interval Δ_j–Δ_{j+1}, let CL_j be the list of rectangles for which the height is greater than 1 and the corresponding tasks are sequential (i.e., sequential tasks that need coalitions) and NCL_j be the list with rectangles that correspond to sequential tasks



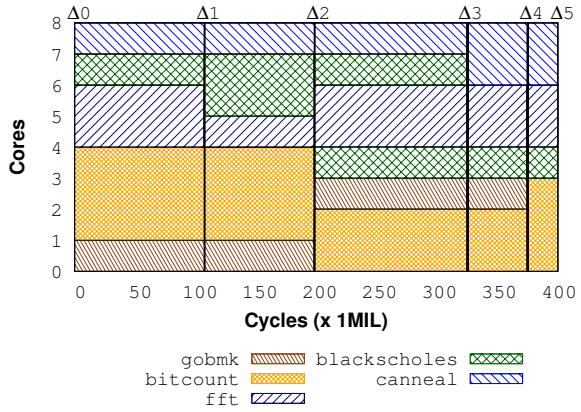
(a)



(b)



(c)



(d)

Figure 5.6: Example of imposing constraint C3.

without coalition or parallel tasks. These lists are built while the rectangles are packed. As constraints C1 and C2 have already been satisfied, it follows that we can fit each rectangle from CL_j list on a unique cluster $P_{4k} - P_{4k+3}$, where $0 \leq k < (m/4)$. This is implemented by function *Pack_coalitions*. The one-unit height rectangles from NCL_j can fit in the available free cores regardless of the clusters, while the rectangles (threads) corresponding to the parallel tasks can be scheduled such that they fill the remaining free cores. This is done in function *Pack_non_coalitions*.

Figure 5.6(b) shows the scheduling after applying the constraint C3 for the interval Δ_2–Δ_3. Here, $CL_2 = \{bitcount, fft\}$ and NCL_2=$\{canneal, blackscholes, gobmk\}$. The tasks *bitcount* and *fft* are placed on cores P_0–P_1 and P_4–P_5 as they are sequential applications running on coalitions. After this step, the cores P_2 and P_3 are free. We choose *gobmk* to run on core P_2 and one thread of parallel application *blackscholes* to run on core P_3. Figures 5.6(c)–(d) show the results for the rest of the intervals. As this step is only a rectangle rearrangement, application of the constraint C3 has no effect on the makespan.

Note that imposing constraints C1 and C3 maintain the optimality of the schedule. However, imposing constraint C2 may violate the optimality of the schedule. Among all the task sets we evaluated, only 1% of the task sets violate the constraint C2 in the optimal schedule on ideal adaptive multi-core. Thus, for most of the task sets, the schedule obtained for Bahurupi is the optimal schedule.

5.2.4 Online schedule for Bahurupi

The off-line schedule described in the previous section assumes all the tasks are ready at time zero. However, in a real system, the tasks can arrive at any point in time and the arrival times are not known beforehand. In this section, we present an online schedule for Bahurupi architecture to quantitatively evaluate the performance of a dynamic heterogeneous multi-core compared to homogeneous and static heterogeneous multi-

cores.

We allow the tasks to arrive in the system with different arrival times. Every task T_j is now defined by the tuple $< type_j,\ arr_j,\ f_{jk} >$, where $type_j$ is the type of the task (serial or parallel), arr_j is the arrival time and f_{jk} is its speedup function on k cores. The arrival times arr_j are randomly distributed in the interval $[0, \sum_{j=0}^{n-1} p_j]$ allowing the tasks to compete for limited number of free cores.

Algorithm 2 presents our online scheduler for Bahurupi multi-core. Here we model the workload as moldable tasks [60]. A moldable task can be scheduled on any number of cores just like malleable tasks but with the restriction that it cannot be preempted. This assumption makes it easy for us to integrate the scheduler in existing operating systems.

When a task arrives in the system, the only information required by the scheduler is its speedup function. This can be obtained by profiling the task on different number of cores. The scheduling decision is taken periodically at every system tick by the function *OnlineSchedule_tick*. As the moldable tasks cannot be preempted, once a task is scheduled on one or more cores using *Allocate_cores* function, it will run till completion. Once a task completes execution, the number of free cores is updated by *Update_free_cores* function.

The dynamic allocation of the tasks to the cores is handled by *Place_task* function. For most tasks, the speedup function tends to have a flat region when applied to a large number of cores (see Figure 5.7). In this region, the increase in performance on $r_j + 1$ cores is minimal compared to the performance on r_j cores. The function *Get_max_cores* returns the number of cores beyond which the speedup improvement is lower than 4%. This way we avoid allocating unnecessary cores that contribute little to performance improvement. Instead, these cores can be allocated to future tasks. The algorithm also ensures that the constraints C1, C2 and C3 mentioned earlier are satisfied. When constraint C3 is violated, function *Allocate_cores* migrates the tasks

Algorithm 2: Online scheduler for Bahurupi

```
InitAdaptiveOnlineScheduler( m, n) begin
    free_cores = m;
    free_clusters = m/4;
end

OnlineSchedule_tick()

begin
    if task_queue.empty() == FALSE then
        if free_cores > 0 then
            next_task = task_queue.front();
            Place_task(next_task);
        end
    end
    if current_task is finished then
        Update_free_cores(current_task, free_cores);
        free_clusters = free_cores/4;
    end
end

Place_task(task)

begin
    max_cores = Get_max_cores(task);
    use_cores = max_cores;
    if max_cores < free_cores then
        use_cores = free_cores;
    end
    if task.type == PARALLEL then
        Allocate_cores(task, use_cores);
    end
    else
        use_cluster = (use_cores > 1);
        if (free_clusters - use_cluster) > 0 then
            Allocate_cores(task, use_cores);
            free_clusters = free_clusters - use_cluster;
        end
        else
            use_cores = 1;
            Allocate_cores(task, use_cores);
        end
    end
    free_cores = free_cores - use_cores;
end
```

such that no sequential task spans across clusters.

5.3 Quantitative results

In this section, we first present quantitative characterization of the performance limit of ideal dynamic heterogeneous multi-core and realistic dynamic heterogeneous multi-core (Bahurupi) compared to homogeneous and static heterogeneous multi-cores. This is followed by performance comparison of Bahurupi with homogeneous and static heterogeneous multi-cores using online scheduler.

5.3.1 Workload

We select 27 sequential applications from SPEC2006, SPEC2000 and embedded MiBench benchmark suites and 6 parallel applications from PARSEC benchmark suite. The characteristics of the benchmarks appear in Table 5.1.

We generate different workload (task sets) consisting of varying mix of ILP and TLP tasks. We ensure that the tasks within a task set have similar processing workload so that all the tasks are competing for the resources throughout execution. This restriction in variability of processing workload is achieved as follows. Given the workload p_j for each task T_j, we compute the average workload and the standard deviation. We ensure that the ratio of standard deviation and average does not exceed 0.35 for a task set. Across all the tasks sets, the ratio of sequential tasks ranges from 25% to 80%; so the ratio of parallel tasks ranges from 20% to 75%.

5.3.2 Multi-core configurations

We model seven different static and dynamic multi-core configurations in our study as shown in Table 5.2. All these configurations are roughly area equivalent to eight 2-way multi-core architecture (S1) under the assumption that the area of a $2r$-way core is

Benchmarks	Inputs	Suite	Type
gzip	input.source	SPEC2000	sequential
mesa	mesa.ppm		
mcf	inp.in		
equake	inp.in		
crafty	crafty.in		
ammp	ammp.in		
parser	test.in		
perlbmk	diffmail.pl		
bzip	input.program	SPEC2006	
gobmk	capture.tst		
calculix	beampic		
hmm	bombesin.hmm		
sjeng	test.txt		
quantum	50 5		
lbm	reference.dat		
sphinx	an4.ctl		
basicmath	runme_large.sh	MiBench	
bicount			
qsort			
susan			
dijkstra			
patricia			
sha			
adpcm			
fft			
gsm			
stringsearch			
blackscholes	simsmall	PARSEC	parallel
swaptions			
canneal			
vips			
bodytrack			
raytrace			

Table 5.1: Characteristics of benchmarks used in our study.

Configuration	Description
(S1) 8x2-way	Homogeneous eight 2-way cores
(S2) 4x4-way	Homogeneous four 4-way cores
(A1) 2x4-way + 4x2-way	Static heterogeneous two 4-way + four 2-way cores
(A2) 1x8-way + 4x2-way	Static heterogeneous one 8-way + four 2-way cores
(A3) 1x8-way + 2x4-way	Static heterogeneous one 8-way + two 4-way cores
(Ideal) 8x2-way	Ideal dynamic heterogeneous multi-core
(Bahurupi) 8x2-way	Bahurupi dynamic heterogeneous multi-core

Table 5.2: Multi-core configurations used in our study.

Type	Issue width	Commit width	Dispatch width	ROB size	LSQ size	ALU cnt	FP cnt	LSU cnt	I/D TLB size	I/D L1$ size	I/D L2$ size
2-way	2	2	2	64	32	2	1	1	16	128K	2MB
4-way	4	4	4	128	64	3	2	2	32	256K	2MB
6-way	6	6	6	192	96	4	3	3	48	384K	2MB
8-way	8	8	8	256	128	5	4	4	64	512K	2MB

Table 5.3: Configuration parameters for out-of-order cores: issue, commit, dispatch width; reorder buffer (ROB) size; load-store queue (LSQ) size; number of ALU, floating point (FP), and load-store (LSU) units; instruction-data TLB size, L1 instruction-data cache size, and unified L2 cache size.

equivalent to that of r 2-way cores.

The homogeneous eight 2-way multi-core architecture S1 is treated as the baseline. We also consider another homogeneous multi-core S2 with medium complexity cores: four 4-way cores. The static heterogeneous multi-core architectures A1, A2, and A3 employ different combination of small, medium, and large complexity cores. For example, A1 has more number of cores compared to A3 and hence is more suitable for TLP tasks, while A3 can accelerate ILP tasks better than A1.

The ideal dynamic heterogeneous multi-core and Bahurupi require the same amount of physical area as the baseline homogeneous multi-core. However, they can be morphed at runtime to form various different homogeneous or static heterogeneous configurations. As mentioned earlier, the ideal dynamic heterogeneous architecture has no restriction on how core coalitions can be formed while Bahurupi imposes certain constraints driven by implementation considerations.

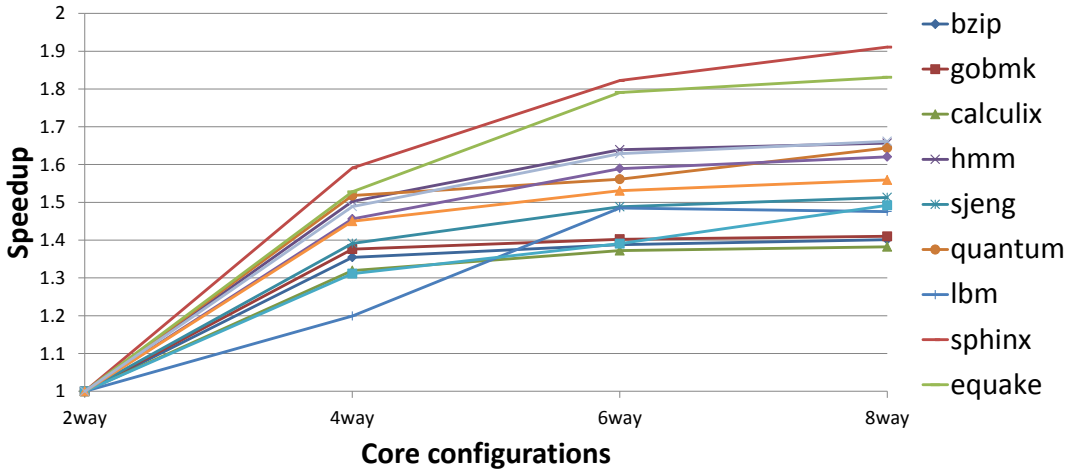
We use MARSS cycle-accurate multi-core simulator [73] for our quantitative char-

acterization work. The configuration parameters for out-of-order cores with different superscalarity values are shown in Table 5.3. The resources available to a core increases with increasing superscalarity.

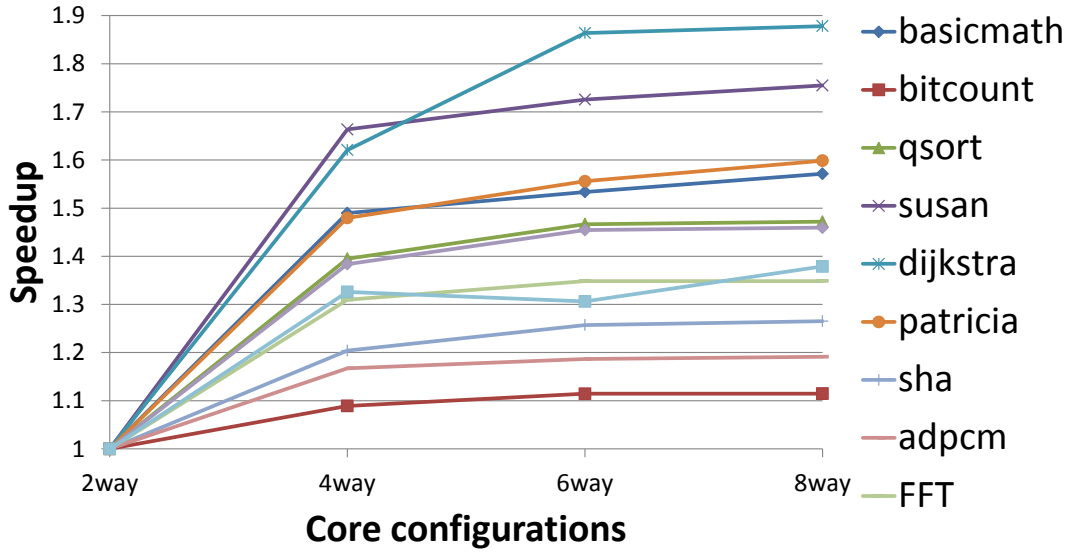
5.3.3 Speedup functions

All architectures allow parallel tasks to use any number of cores. The speedup function for parallel tasks on different number of cores are shown in Figure 5.7. We compile the parallel task with r threads and execute the threads on r cores to obtain the speedup. In other words, the speedup function represents the ideal scenario where the number of threads is equal to the number of cores. In reality, a parallel task compiled with r threads may need to use a different number of cores during execution. Similarly, for an adaptive architecture, a task can be allocated varying number of cores during execution. However, we noticed little difference in performance when an application compiled with m threads executed on r cores where $r < m$.

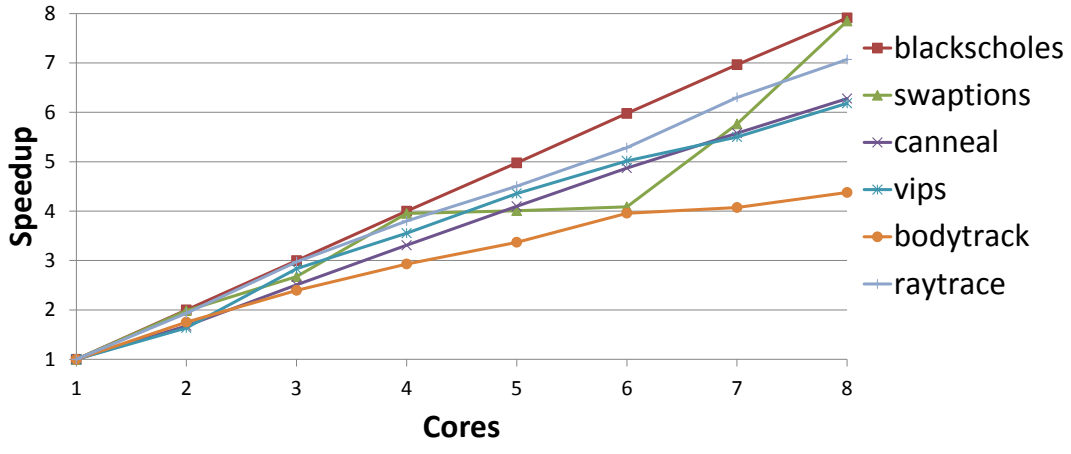
The homogeneous and static heterogeneous multi-cores use native 4-way and 8-way cores. The speedup of serial tasks on native 4-way and 8-way cores are obtained from MARSS cycle-accurate simulator [73]. Both ideal dynamic heterogeneous multi-core and Bahurupi architecture, on the other hand, employ core coalition to create virtual $2r$-way cores from r 2-way physical cores. We have established before [74] that the performance of a virtual $2r$-way core through core coalition in Bahurupi is either close to or even surpasses the performance of native $2r$-way core. For serial tasks running on virtual cores in dynamic heterogeneous architectures (Ideal and Bahurupi), we use speedup obtained from core coalition. As going beyond 8-way cores does not provide further speedup due to limited ILP, we restrict the speedup function for serial tasks to 8-way core as shown in Figure 5.7.



(a) SPEC



(b) MiBench



(c) PARSEC

Figure 5.7: Speedup functions for sequential and parallel tasks.

5.3.4 Scheduling on homogeneous and static heterogeneous multi-cores

The scheduling algorithms presented in Section 5.1 are used to obtain the makespan for Ideal and Bahurupi architectures. For the homogeneous architectures S1 and S2, we can employ the same optimal scheduling algorithm used for Ideal by simply restricting the sequential applications to use only one core.

For static heterogeneous architectures, however, we need to modify the scheduling algorithm. We first obtain the optimal makespan C^0_{max} assuming continuous resource allocation. Then for each task T_j, we round up or down the resource allocated r^0_j to match an available simple or complex core, except for $r^0_j < 1$ in which case r^0_j becomes 1. Finally, we perform strip packing [65] to optimally schedule the tasks. Note that scheduling using strip-packing is computationally expensive for static heterogeneous multi-cores; but scheduling on static heterogeneous multi-cores is not the focus here. We merely use it for comparison purposes.

For online schedule on homogeneous and static heterogeneous multi-cores, we adapt the online algorithm presented for Bahurupi in Section 5.2.4. In all the schedules, we assume a preemption penalty of 100 cycles and reconfiguration penalty of 100 cycles [74].

5.3.5 Limit study of dynamic heterogeneous multi-core

We now proceed to characterize the performance limit of dynamic heterogeneous multi-cores compared to homogeneous and static heterogeneous multi-cores. As mentioned earlier, we generate 850 task sets and compute the makespan of each task set on different multi-core architectures. The results are presented in Figure 5.8. The speedup on Y-axis is defined w.r.t. the makespan on baseline homogeneous S1 architecture consisting of eight 2-way cores. We plot the speedup on six different architectures for each task set represented on the X-axis. That is, each point in this graph represents the speedup of

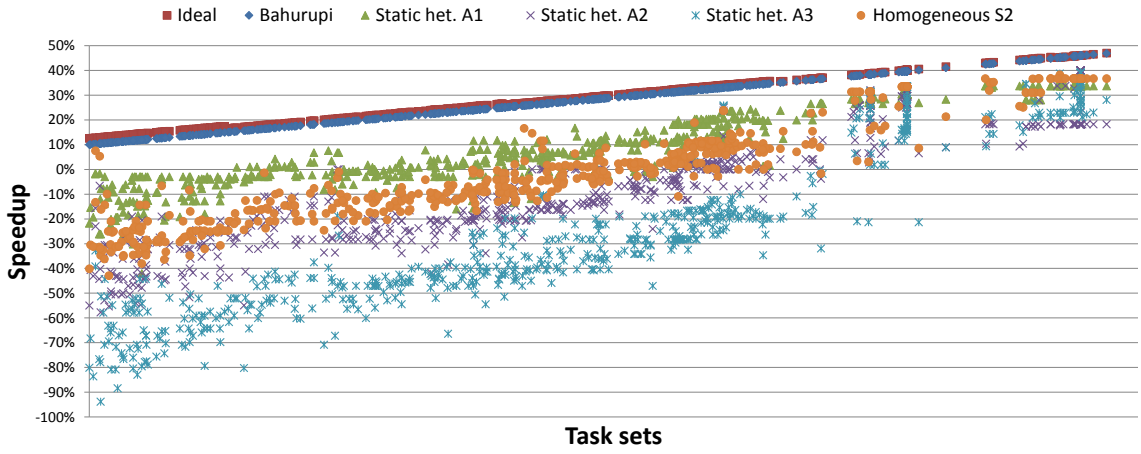


Figure 5.8: Comparison of dynamic and static multi-cores under off-line schedule. The speedup is w.r.t. the baseline homogeneous S1.

a particular task set on a particular architecture compared to baseline S1. For ease of presentation, the task sets along X-axis are sorted in non-decreasing order of speedup on Bahurupi.

The results clearly demonstrate that dynamic heterogeneous architectures (Ideal and Bahurupi) perform significantly better when compared to homogeneous and static heterogeneous architectures. It is interesting to note that the performance of Bahurupi is practically identical to that of Ideal dynamic heterogeneous architecture even though Bahurupi imposes certain constraints on core coalition. Thus, a cluster-based dynamic heterogeneous architecture like Bahurupi is quite effective in reaching the speedup limit set by ideal dynamic heterogeneous architecture.

The normalized speedup of dynamic heterogeneous architectures ranges from 10% to 49%. When the speedup on dynamic heterogeneous architecture is low, the speedup on the other multi-core architectures is also very low or they perform even worse than the baseline homogeneous architecture S1 due to lack of resources. On average, dynamic heterogeneous architectures outperform the static heterogeneous configurations A1, A2 and A3 by 18%, 35% and 52% respectively. When compared with the homogeneous configuration S2, the dynamic heterogeneous architecture performs 26% better,

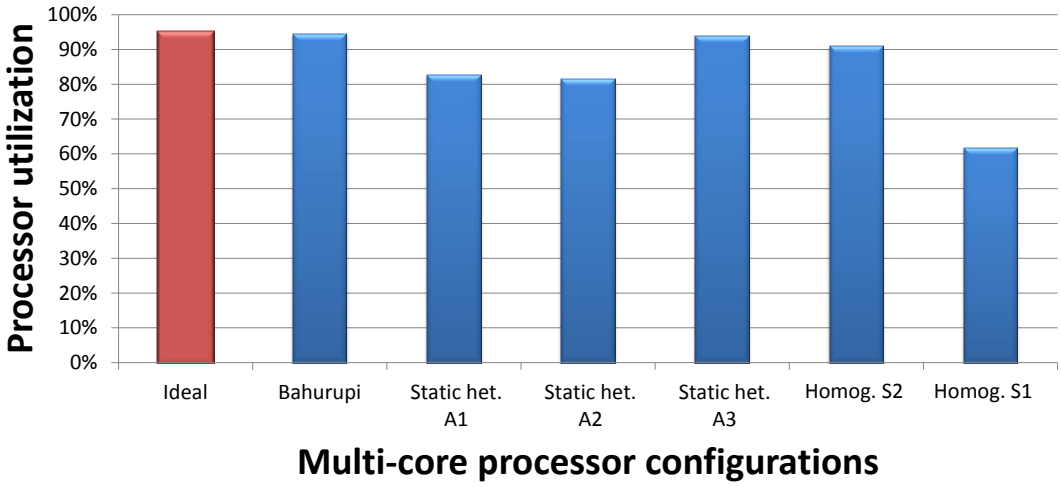


Figure 5.9: Utilization of different multi-cores in offline schedule.

which makes the homogeneous configuration S2 a better option than the static heterogeneous configurations A2 and A3. This is due to the availability of a number of powerful cores in S2 that can accelerate both the sequential and parallel tasks.

The results also anticipate the performance benefit of the announced static heterogeneous big.LITTLE multi-core [41], which includes two 3-way Cortex A-15 out-of-order cores and four dual-issue in-order Cortex A-7 cores on the same die. This configuration is close to the A1 configuration, which performs the best out of all static heterogeneous configurations.

Figure 5.9 reports the processor utilization (averaged across all tasks sets) for different static and dynamic multi-core architectures. The dynamic heterogeneous architectures have the best utilization (94%) and performance making them the most efficient architectures. In contrast, the static heterogeneous multi-core A3 has a high utilization (92%) but low performance, making it the least efficient multi-core architecture. The homogeneous configuration S1 has low utilization (61%) as it can only exploit TLP from parallel tasks. The serial tasks keep only a subset of the cores busy. The static heterogeneous configuration A1 has good utilization (82%) making it the most efficient static heterogeneous multi-core configuration.

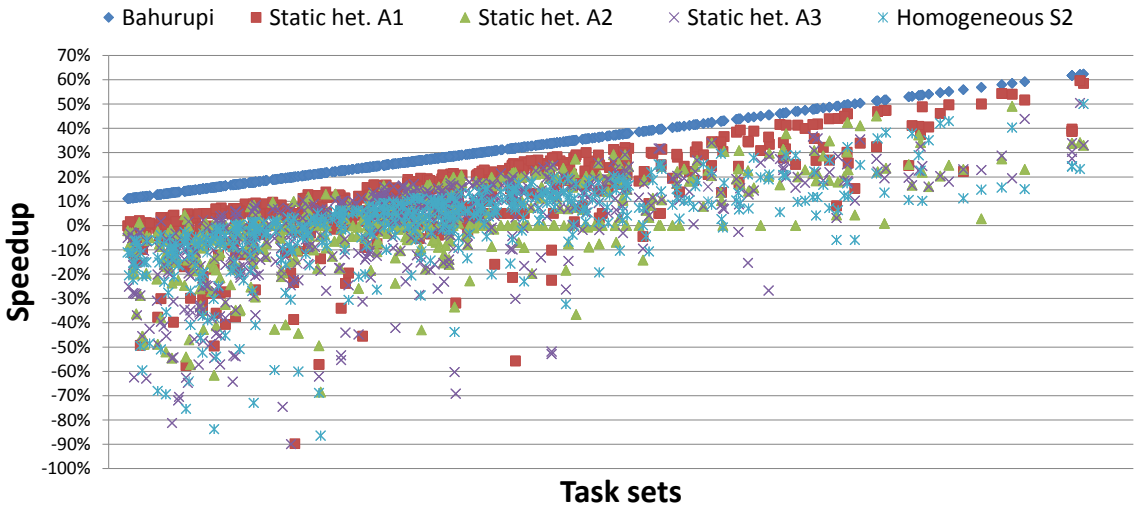


Figure 5.10: Comparison of Bahurupi with static multi-cores under online schedule. The speedup is w.r.t. baseline homogeneous S1.

5.3.6 Realistic performance benefit of dynamic heterogeneous multi-core

We now present the performance benefit of dynamic heterogeneous multi-core architecture such as Bahurupi in the context of realistic online scheduling where tasks can arrive at arbitrary point in time. The results are shown in Figure 5.10. The X-axis and Y-axis are defined similar to Figure 5.8. We perform the experiments for the same 850 task sets used for offline schedule; however, we run each task set with five different arrival times to create a total of 4,250 task sets.

Again Bahurupi outperforms homogeneous and static heterogeneous multi-core architectures with speedup ranging from 10% to 62%. On average, Bahurupi outperforms static heterogeneous A1, A2 and A3 configurations by 17%, 26%, and 28%, respectively. The homogeneous architecture S2 shows a loss in performance of 26% when compared to Bahurupi. This shows that in the case of online scheduling, the speedup trend for different configurations is almost the same as that of off-line schedule (Figure 5.8). The static heterogeneous configuration A1 offers the best performance out of all

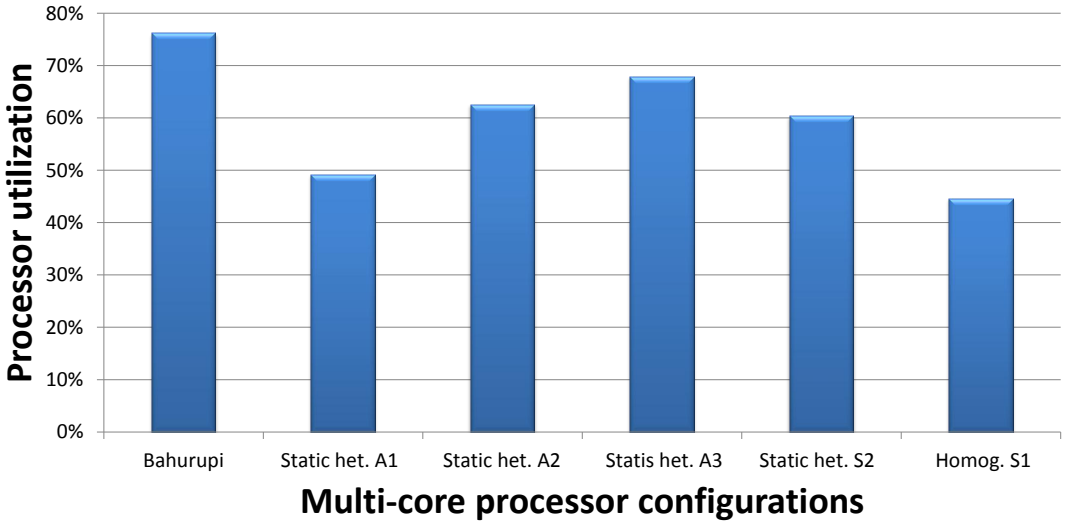


Figure 5.11: Utilization of different multi-cores in online schedule.

static heterogeneous multi-cores, while the configuration A3 offers the worst performance.

Figure 5.11 plots the average processor utilization of the different architectures in online scheduling. The utilization trend is similar to that of offline schedule (Figure 5.9). Bahurupi shows very good efficiency with 76% average utilization, while the static heterogeneous configuration A3 shows the worst efficiency with 67% average utilization. Similarly, homogeneous configuration S1 has the lowest utilization (44%) due to the large number of simple cores that can only benefit the parallel applications, whereas the serial applications keep the occupied cores busy for a long time.

The measured average *competitive ratio* between our online scheduler and an optimal online scheduler (obtained using strip packing) is 1.14.

5.3.7 Reconciling ILP and TLP

The main objective of dynamic heterogeneous multi-core architectures such as Bahurupi is to reconcile the conflict between ILP and TLP tasks and provide performance benefit for both. We now provide quantitative validation that Bahurupi indeed manages

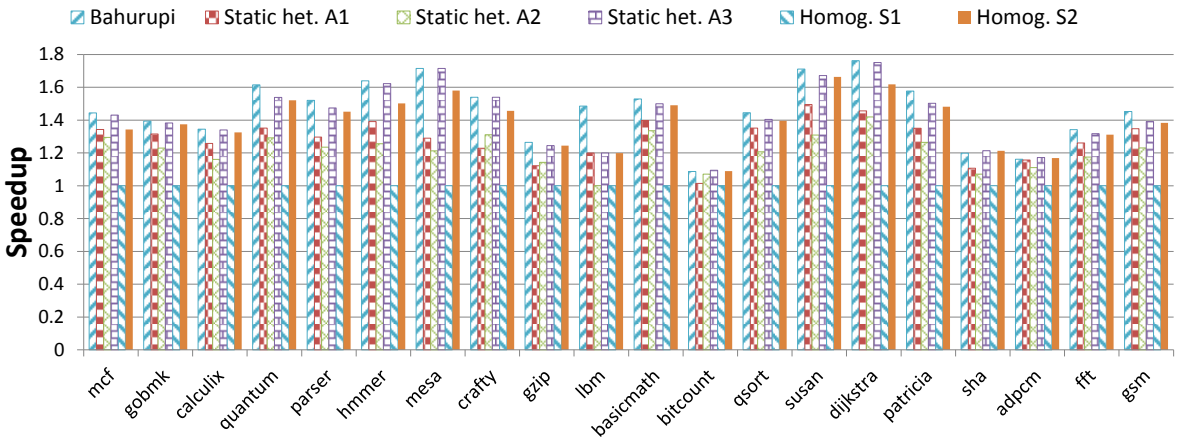


Figure 5.12: Speedup of sequential applications averaged across all task sets normalized w.r.t. execution on native 2-way core.

to accelerate both sequential and parallel tasks.

As mentioned before, we use 27 sequential and 6 parallel applications to create 850 different task sets for our online scheduling experiment. Each task set is run with 5 different randomly selected arrival times to create a total of 850×5 different task sets. For each task and multi-core configuration used in our study, we compute the average speedup of the task across all the online schedules in which the task participates. The speedup is computed w.r.t. the execution time of the task on a single 2-way core.

Figure 5.12 shows the speedup for the sequential applications. Bahurupi is the clear winner here and provides the best speedup for each application among all the multi-core configurations. Static heterogeneous A3 using native 8-way core is close to Bahurupi. Also as expected, homogeneous S2 deploys 4-way cores and hence has better speedup for serial tasks compared to homogeneous S1 using 2-way cores. The performance of static heterogeneous A1 and A2 for serial applications appear somewhere in between.

In contrast, Figure 5.13 shows the speedup for parallel applications. Again, the speedup of Bahurupi is close to that of baseline homogeneous S1, which understandably has the best speedup because it has a large number of simple cores. The other homogeneous and static heterogeneous configuration perform quite badly for parallel

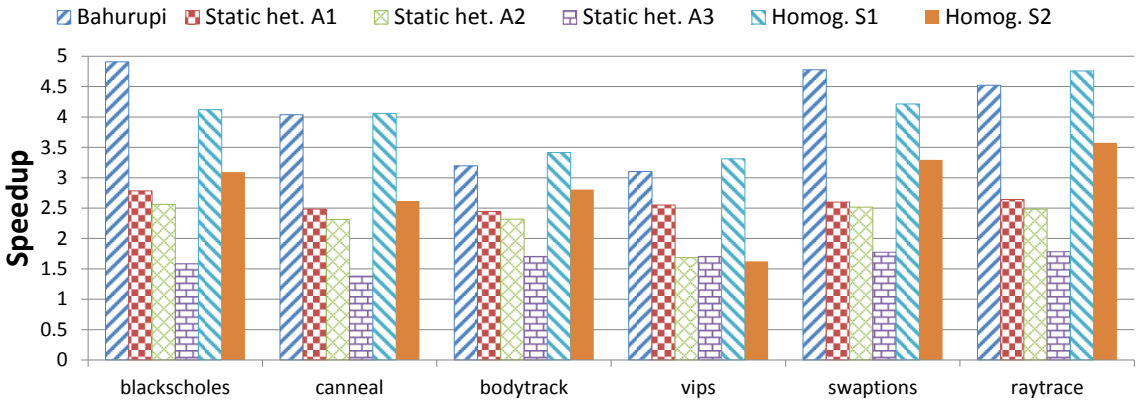


Figure 5.13: Speedup of parallel applications averaged across all task sets normalized w.r.t. execution on one 2-way core.

applications.

So in summary, Bahurupi dynamic heterogeneous multi-core architecture is successful in accelerating both serial and parallel tasks. While homogeneous S1 with large number of simple cores is quite effective for TLP, it shows poor performance for serial tasks. Static heterogeneous architecture A3 can perform well for serial tasks due to the presence of a complex core but suffers badly for parallel tasks. Among the static heterogeneous configurations, the configuration A1 provides the best balance of ILP and TLP speedup; but is far behind dynamic heterogeneous multi-core architecture Bahurupi.

5.4 Summary

In this chapter we have presented a comprehensive quantitative approach to establish the performance potential of dynamic heterogeneous multi-core architectures compared to homogeneous and static heterogeneous multi-cores. This is a performance study that considers a mix of sequential and parallel workloads to observe the capability of dynamic heterogeneous multi-cores in exploiting both ILP and TLP. We employ an optimal algorithm that allocates and schedules the tasks on varying number of cores so as to minimize the makespan. This optimal schedule allows us to define the perfor-

mance limit of ideal dynamic heterogeneous multi-cores for realistic workloads. We then modify this optimal schedule to satisfy the constraints imposed by a realistic dynamic heterogeneous multi-core, namely Bahurupi. The experiments reveal that both the ideal and the realistic dynamic heterogeneous architecture provide significant reduction in makespan for mixed workload compared to homogeneous and static heterogeneous architectures. Finally, we compare the performance of dynamic heterogeneous, homogeneous and static heterogeneous multi-cores in an online scheduling policy and demonstrate the same performance trend.

Chapter 6

Conclusions

6.1 Summary of the thesis

The Moore's law is still valid, but recent research already show that the current architectural trends of including more and more simple cores on the same die will soon come to an end. We already see the first [41] steps towards static heterogeneous designs where cores with different power and performance characteristics sharing the same ISA are placed on the same die. This is to offer a better energy-performance tradeoff when compared to the homogeneous designs. But these static heterogeneous multi-cores lack the adaptability to support changing requirements coming from the existing and emerging applications. The next logical step would be to move to dynamic heterogeneous architectures. These architectures are not fixed at the fabrication time, but they can adapt dynamically to the new performance requirements.

The major contribution of this thesis is the design and evaluation of a novel reconfigurable multi-core architecture, called *Bahurupi*, that can dynamically adapt itself to support both multi-threaded code with explicit thread-level parallelism as well as sequential code with instruction-level parallelism. Bahurupi can dynamically merge the baseline 2-way out-of-order execution engines to achieve the performance of 4-way

or even 8-way out-of-order processors. Bahurupi is a hardware-software cooperative solution that requires minimal additional hardware resources and compiler support for coalition. In this thesis we have also included the prototype implementation of the Bahurupi multi-core architecture in FPGAs.

We also propose and implement a novel L1 data cache architecture for our dynamic heterogeneous multi-core. When Bahurupi creates coalition of cores, the newly created core becomes more demanding in terms of memory requests. In order to satisfy the memory demands of such architecture, a high bandwidth access to the first level of cache must be provided. Our novel architecture is able to overcome the limitations in terms of area, energy and performance of multi-ported shared caches. Results show that our design is able to accommodate successfully the memory demands of dynamic heterogeneous architectures providing high performance with very low area and energy consumption. Additionally, our design offers dynamic reconfiguration capabilities. At runtime, the design can switch between two modes of execution: *private* mode and *coalition* mode. In private mode, the cores access their private caches in the traditional way and our design is not used. In coalition mode, the cores connect to a high bandwidth and low latency interconnect that can reconfigure at runtime the number of banks the adaptive multi-core accesses reducing the L1 data bank conflicts. At the same time, the design can also reconfigure the size and associativity of the cache that the virtual core accesses.

In this thesis, we have also presented a comprehensive quantitative approach to establish the performance potential of dynamic heterogeneous multi-core architectures compared to homogeneous and static heterogeneous multi-cores. This study considers a mix of sequential and parallel workloads to observe the capability of dynamic heterogeneous multi-cores in exploiting both ILP and TLP. We employ an optimal algorithm that allocates and schedules the tasks on varying number of cores so as to minimize the makespan. This optimal schedule allows us to define the performance limit of ideal dy-

namic heterogeneous multi-cores for realistic workloads. We then modify this optimal schedule to satisfy the constraints imposed by our realistic adaptive multi-core, Bahurupi. The experiments reveal that both the ideal and the realistic dynamic heterogeneous architectures provide significant improvement in throughput for mixed workload compared to homogeneous and static heterogeneous architectures. Finally, we compare the performance of dynamic heterogeneous, homogeneous and static heterogeneous multi-cores in an online scheduling policy and demonstrate the same performance trend.

6.2 Future work

In future, more research can be done on the portability of Bahurupi to any ISA. Many ISAs prevent the designers from adding any new instruction to the instruction set. Circumventing the *sentinel instruction* need would have a very good impact in the applicability of Bahurupi on existing architectures and ISAs. More research can also be conducted to analyse the benchmarks characteristics that impact the performance of Bahurupi.

In addition to our L1 data cache architecture design, we wish to investigate the effect of the coalition on the last level of unified cache (LLC) with combination of parallel applications. Usually, modern LLCs are caches with big banks, that are accessed in a non-uniform fashion (NUCA caches). By creating the coalition, the virtual core becomes more aggressive in terms of memory access as it basically accesses memory across multiple loop iterations. This may impact the behaviour of the other applications sharing the LLC. New research can propose solutions that can overcome these problems by proposing novel core-to-bank allocation policies.

In addition, we also wish to dedicate more research on energy-aware online schedulers for Bahurupi. We believe that a performance and power modeling scheme similar to the one in [77] plus a controller-based solution similar to the one in [70] can lead to

a very efficient scheduling system that can optimally balance the resource allocation to the sequential and parallel tasks coexisting in the system. A prediction method would be very useful at runtime to predict when the application will need coalition of cores and how big the coalition should be. The system would be able to decide when to create the coalition of cores with the help of performance counters (e.g., on-line measuring of IPC, cache misses). The available set of performance counters should be enriched with ones that can monitor the amount of dependency existing between basic blocks and inside the basic blocks. This information together with the overall system utilization and memory usage can make an operating system decide when to coalesce and how many cores to coalesce.

Similarly, in terms of L1 data cache architecture reconfiguration, the system needs to smartly decide when to add more access points or (and) to increase the size and associativity of the shared cache seen by the coalition of cores. Performance counters can be added to monitor the miss rates per cache bank, the arbitration conflicts per arbiter and total energy consumption. The reconfiguration penalty plays an important role when making these decisions. When reconfiguring the network, the address mapping changes; thus, the cache banks must flush their content to the next level of cache. This is a costly process, so the number of reconfigurations during execution of an application should be wisely decided. On the other hand, when increasing the associativity of the cache at runtime, there is no need for flushing the caches to the next level. However, in this case the energy consumption increases.

In the future, we also plan to implement our Bahurupi design in a more robust and easy to manage simulator (i.e., GEM5 simulator [17]) which will offer the possibility of running a full modern operating system. This will bring more opportunities for the operating systems research in the context of flexible multi-core architectures.

In addition, we also plan to implement multi-clusters of Bahurupi dynamic heterogeneous multi-cores by using multiple FPGAs and study the benefits of such architec-

ture in a many-core environment.

References

[1] ARM Ltd., The Advanced Microcontroller Bus Architecture (AMBA). www.arm.com/products/solutions/AMBAHomePage.html.

[2] SPEC CPU Benchmarks. http://www.spec.org/benchmarks.html.

[3] Stmicroelectronics, The STBus Interconnect., note = www.st.com.

[4] Xilinx. Virtex-6 FPGA ML605 Evaluation Kit. http://www.xilinx.com/products/boards-and-kits/EK-V6-ML605-G.htm.

[5] Nvidia. The Benefits of Multiple CPU Cores in Mobile Devices, 2010. http://www.nvidia.com/content/PDF/tegra_white_papers/Benefits-of-Multi-core-CPUs-in-Mobile-Devices_Ver1.2.pdf.

[6] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, February 2002.

[7] T. M. Austin and G. S. Sohi. High-bandwidth Address Translation for Multiple-issue Processors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ISCA, pages 158–167. ACM, 1996.

[8] Balakrishnan, Saisanthosh, Rajwar, Ravi, Upton, Mike, Lai, and Konrad. The Impact of Performance Asymmetry in Emerging Multicore Architectures. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA, pages 506–517. ACM, 2005.

[9] S. Baldawa and R. Sangireddy. CMP-SIM: A flexible CMP architectural simulation environment.

[10] A. O. Balkan, G. Qu, and U. Vishkin. A Mesh-of-Trees Interconnection Network for Single-Chip Parallel Processing. In *Application-specific Systems, Architectures and Processors*, ASAP, pages 73–80. IEEE, 2006.

[11] M. Becchi and P. Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In *Proceedings of the 3rd Conference on Computing Frontiers*, CF, pages 29–40. ACM, 2006.

[12] B. M. Beckmann, M. R. Marty, and D. A. Wood. ASR: Adaptive Selective Replication for CMP Caches. In *Proceedings of the 39th Annual International Symposium on Microarchitecture*, MICRO, pages 443–454. IEEE, 2006.

[13] L. Benini and G. De Micheli. Networks on Chips: A New SoC Paradigm. *Computer*, 35(1):70–78, January 2002.

[14] D. Benitez, J. C. Moure, D. I. Rexachs, and E. Luque. Evaluation of the Field-programmable Cache: Performance and Energy Consumption. In *Proceedings of the 3rd Conference on Computing Frontiers*, CF, pages 361–372. ACM, 2006.

[15] P.-E. Bernard, T. Gautier, and D. Trystram. Large Scale Simulation of Parallel Molecular Dynamics. In *Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, pages 638–644, 1999.

[16] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 72–81. ACM, 2008.

[17] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The Gem5 Simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, August 2011.

[18] T. Bjerregaard and S. Mahadevan. A Survey of Research and Practices of Network-on-chip. *ACM Computing Surveys*, 38(1), June 2006.

[19] E. Blayo and L. Debreu. Adaptive Mesh Refinement for Finite-Difference Ocean Models: First Experiments. *Journal of Physical Oceanography*, 29:1239–1250, 1999.

[20] J. Blazewicz, T. C. E. Cheng, M. Machowiak, and C. Oguz. Berth and Quay Crane Allocation: A Moldable Task Scheduling Model. *Journal of the Operational Research Society*, 62(7):1189–1197, 2011.

[21] J. Blazewicz, M. Drabowski, and J. Weglarz. Scheduling Multiprocessor Tasks to Minimize Schedule Length. *IEEE Transactions on Computers*, C-35(5):389–393, 1986.

[22] J. Blazewicz, M. Drozdowski, G. Schmidt, and D. de Werra. Scheduling Independent Multiprocessor Tasks on a Uniform k-processor System. *Parallel Computing*, 20(1):15–28, January 1994.

[23] J. Blazewicz, K. Ecker, and D. Trystram. *Handbook on Parallel and Distributed Processing*. Springer, 2000.

[24] J. Blazewicz, M. Y. Kovalyov, M. Machowiak, D. Trystram, and J. Weglarz. Preemptable Malleable Task Scheduling Problem. *IEEE Transactions on Computers*, 55(4):486–490, April 2006.

[25] J. Blazewicz, M. Machowiak, J. Weglarz, M. Y. Kovalyov, and D. Trystram. Scheduling Malleable Tasks on Parallel Processors to Minimize the Makespan. *Annals OR*, 129(1-4):65–80, 2004.

[26] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA, pages 83–94, june 2000.

[27] D. Burger, J. R. Goodman, and A. Kagi. Memory Bandwidth Limitations of Future Microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ISCA, pages 78–89. ACM, 1996.

[28] J. Chang and G. S. Sohi. Cooperative Caching for Chip Multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA, pages 264–276. IEEE, 2006.

[29] G.-I. Chen and T.-H. Lai. Preemptive Scheduling of Independent Jobs on a Hypercube. *Information Processing Letters*, 28(4):201–206, July 1988.

[30] D. Chiou, S. Devadas, L. Rudolph, and B. Ang. Dynamic Cache Partitioning via Columnization. Technical report, MIT, 1999.

[31] N. K. Choudhary, S. V. Wadhavkar, T. A. Shah, H. Mayukh, J. Gandhi, B. H. Dwiel, S. Navada, H. H. Najaf-abadi, and E. Rotenberg. FabScalar: Composing Synthesizable RTL Designs of Arbitrary Cores Within a Canonical Superscalar Template. In *Proceedings of the 38th International Symposium on Computer Architecture*, ISCA, pages 11–22. ACM, 2011.

[32] J. Cong and B. Yuan. Energy-efficient Scheduling on Heterogeneous Multi-core Architectures. In *Proceedings of the International Symposium on Low Power Electronics and Design*, ISLPED, pages 345–350. ACM, 2012.

[33] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Elsevier Morgan Kaufmann, 1999. Page 337.

[34] W. J. Dally and B. Towles. Route Packets, Not Wires: On-chip Inteconnection Networks. In *Proceedings of the 38th Annual Design Automation Conference*, DAC, pages 684–689. ACM, 2001.

[35] J. J. Dongarra, L. S. Duff, D. C. Sorensen, and H. A. V. Vorst. *Numerical Linear Algebra for High Performance Computers*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[36] J. Du and J. Y.-T. Leung. Complexity of Scheduling Parallel Task Systems. *SIAM Journal on Discrete Mathematics*, 2(4):473–487, November 1989.

[37] P.-F. Dutot and D. Trystram. Scheduling on Hierarchical Clusters Using Malleable Tasks. In *Proceedings of the Thirteenth Annual Symposium on Parallel Algorithms and Architectures*, SPAA, pages 199–208. ACM, 2001.

[38] J. Edler and M. D. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator. University of Wisconsin Computer Sciences.

[39] FreePDK, 2011. `http://www.eda.ncsu.edu/wiki/FreePDK`.

[40] A. Gordon-Ross, F. Vahid, and N. Dutt. Fast Configurable-cache Tuning with a Unified Second-level Cache. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, ISLPED, pages 323–326. ACM, 2005.

[41] P. Greenhalg. Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. Technical report, ARM, 2011.

[42] D. P. Gulati, C. Kim, S. Sethumadhavan, S. W. Keckler, and D. Burger. Multitasking Workload Scheduling on Flexible Core Chip Multiprocessors. *SIGARCH Computer Architecture News*, 36(2):46–55, May 2008.

[43] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the Workload Characterization*, WWC, pages 3–14. IEEE, 2001.

[44] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA, pages 184–195. ACM, 2009.

[45] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[46] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7):33–38, July 2008.

[47] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 13–22. ACM, 2006.

[48] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *Proceedings of the 34th*

Annual International Symposium on Computer Architecture, ISCA, pages 186–197, 2007.

[49] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive Insertion Policies for Managing Shared Caches. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 208–219. ACM, 2008.

[50] T. Juan, J. J. Navarro, and O. Temam. Data Caches for Superscalar Processors. In *Proceedings of the 11th International Conference on Supercomputing*, ICS, pages 60–67. ACM, 1997.

[51] Khubaib, M. A. Suleman, M. Hashemi, C. Wilkerson, and Y. N. Patt. Morphcore: An Energy-Efficient Microarchitecture for High Performance ILP and High Throughput TLP. In *Proceedings of the 45th Annual International Symposium on Microarchitecture*, MICRO, pages 305–316. IEEE, 2012.

[52] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable Lightweight Processors. In *Proceedings of the 40th Annual International Symposium on Microarchitecture*, MICRO, pages 381–394. IEEE, 2007.

[53] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 111–122. IEEE, 2004.

[54] D. Koufaty, D. Reddy, and S. Hahn. Bias Scheduling in Heterogeneous Multicore Architectures. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys, pages 125–138. ACM, 2010.

[55] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th annual International Symposium on Microarchitecture*, MICRO, pages 81–. IEEE, 2003.

[56] R. Kumar, N. P. Jouppi, and D. M. Tullsen. Conjoined-Core Chip Multiprocessing. In *Proceedings of the 37th Annual International Symposium on Microarchitecture*, MICRO, pages 195–206. IEEE, 2004.

[57] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA, pages 64–, 2004.

[58] C. E. LaForest and J. G. Steffan. Efficient Multi-ported Memories for FPGAs. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA, pages 41–50. ACM, 2010.

[59] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, MICRO, pages 330–335. IEEE, 1997.

[60] J. Leung, L. Kelly, and J. H. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. 2004.

[61] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient Operating System Scheduling for Performance-asymmetric Multi-core Architectures. In *Proceedings of the Conference on Supercomputing*, SC, pages 1–11, 2007.

[62] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating System Support for Overlapping-ISA Heterogeneous Multi-core Architectures. In *Preceedings of the 16th International Symposium on High Performance Computer Architecture*, HPCA, pages 1–12. IEEE, 2010.

[63] W. T. Ludwig. Algorithms for Scheduling Malleable and Non-malleable Parallel Tasks, 1995. PhD thesis. Department of Computer Science, University of Wisconsin-Madison.

[64] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke. Composite Cores: Pushing Heterogeneity Into a Core. In *Proceedings of the 45th Annual International Symposium on Microarchitecture*, MICRO, pages 317–328. IEEE, 2012.

[65] S. Martello, M. Monaci, and D. Vigo. An Exact Approach to the Strip-Packing Problem. *Institute for Operations Research and the Management Sciences Journal on Computing*, 15(3):310–319, July 2003.

[66] T. Morad, U. Weiser, and A. Kolodny. ACCMP Asymmetric Cluster Chip Multi-Processing, 2004. CCIT Technical Report.

[67] G. Mounie, C. Rapine, and D. Trystram. Efficient Approximation Algorithms for Scheduling Malleable Tasks. In *Proceedings of the eleventh Annual Symposium on Parallel Algorithms and Architectures*, pages 23–32. ACM, 1999.

[68] G. Mounie, C. Rapine, and D. Trystram. A $\frac{3}{2}$-Approximation Algorithm for Scheduling Independent Monotonic Malleable Tasks. *SIAM Journal of Computing*, 37(2):401–412, May 2007.

[69] N. Muralimanohar and R. Balasubramonian. CACTI 6.0: A Tool to Understand Large Caches. University of Utah and Hewlett Packard Laboratories.

[70] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin. Hierarchical Power Management for Asymmetric Multi-core in Dark Silicon Era. In *Proceedings of the 50th Annual Design Automation Conference*, DAC, pages 174:1–174:9. ACM, 2013.

[71] K. Mysur, M. Pricopi, T. Marconi, and T. Mitra. Implementation of Core Coalition on FPGAs. In *Preceedings of the 21st International Conference on Very Large Scale Integration*, VLSI-SoC, pages 198–203. IFIP/IEEE, 2013.

[72] J. Nickolls and W. J. Dally. The GPU Computing Era. *IEEE Micro*, 30(2):56–69, March 2010.

[73] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSS: A Full System Simulator for Multicore x86 CPUs. In *Proceedings of the 48th Design Automation Conference*, DAC, pages 1050–1055. ACM, 2011.

[74] M. Pricopi and T. Mitra. Bahurupi: A Polymorphic Heterogeneous Multi-core Architecture. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):22:1–22:21, January 2012. Presented at 7th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC) 2012.

[75] M. Pricopi and T. Mitra. Polymorphic Heterogeneous Multi-Core Architecture, 2012. Patent number: PCT/SG2012/000454.

[76] M. Pricopi and T. Mitra. Task Scheduling on Adaptive Multi-core. *IEEE Transactions on Computers (TC)*, (PrePrints), 2013.

[77] M. Pricopi, T. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin. Power-Performance Modeling on Asymmetric Multi-cores. In *International*

Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES, pages 1–10, 2013.

[78] M. Pricopi, Y. Yao, Z. Ghe, T. Mitra, N. Zhang, and W. Chen. L1 Data Cache Architecture for Adaptive Multi-Cores. In *Design Automation Conference*, DAC, 2014. Under review.

[79] M. Qureshi. Adaptive Spill-Receive for Robust High-performance Caching in CMPs. In *International Symposium on High Performance Computer Architecture*, HPCA, pages 45–54. IEEE, 2009.

[80] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA, pages 381–391. ACM, 2007.

[81] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual International Symposium on Microarchitecture*, MICRO, pages 423–432. IEEE, 2006.

[82] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural Support for Operating System-driven CMP Cache Management. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 2–12. ACM, 2006.

[83] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No "Power" Struggles: Coordinated Multi-level Power Management for the Data Center. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 48–59. ACM, 2008.

[84] A. Rahimi, I. Loi, M. R. Kakoee, and L. Benini. A Fully-Synthesizable Single-Cycle Interconnection Network for Shared-L1 Processor Clusters. In *Design Automation and Test in Europe*, DATE, pages 491–496. IEEE, 2011.

[85] E. Rotenberg. Fabscalar Project. `http://www.tinker.ncsu.edu/ericro/research/fabscalar.htm`.

[86] P. Salverda and C. Zilles. Fundamental Performance Constraints in Horizontal Fusion of In-order Cores. In *Preceedings of the 14th International Symposium on High Performance Computer Architecture*, HPCA, pages 252 –263, February 2008.

[87] P. Shivakumar and N. P. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model, 2001. WRL Technical Report.

[88] G. S. Sohi and M. Franklin. High-bandwidth Data Memory Systems for Superscalar Processors. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 53–62. ACM, 1991.

[89] S. Srikantaiah, E. Kultursay, T. Zhang, M. Kandemir, M. Irwin, and Y. Xie. Morphcache: A Reconfigurable Adaptive Multi-level Cache Hierarchy. In *Preceedings of the 17th International Symposium on High Performance Computer Architecture*, HPCA, pages 231–242. IEEE, 2011.

[90] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Partitioning of Shared Cache Memory. *Journal of Supercomputing*, 28(1):7–26, April 2004.

[91] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating Critical Section Execution with Asymmetric Multi-core Architectures. In *Proceedings*

of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, pages 253–264, 2009.

[92] Synopsys, 2010. `http://www.synopsys.com`.

[93] D. Tarjan, M. Boyer, and K. Skadron. Federation: Repurposing Scalar Cores for Out-of-Order Instruction Issue. In *Proceedings of the 45th Annual Design Automation Conference*, DAC, pages 772–775, 2008.

[94] J. Turek, J. L. Wolf, and P. S. Yu. Approximate Algorithms Scheduling Parallelizable Tasks. In *Proceedings of the fourth Annual Symposium on Parallel Algorithms and Architectures*, pages 323–332. ACM, 1992.

[95] S. Vajapeyam, B. Rychlik, and J. P. Shen. Dependence-chain Processing using Trace Descriptors having Dependency Descriptors, 2008. Intel Corporation Patent No.: US 7363467 B2.

[96] A. Vajda. *Programming Many-Core Chips*. Springer Publishing Company, Incorporated, 2011. Chapter 4 – The Fundamental Laws of Parallelism.

[97] K. Varadarajan, S. K. Nandy, V. Sharda, A. Bharadwaj, R. Iyer, S. Makineni, and D. Newell. Molecular Caches: A Caching Structure for Dynamic Creation of Application-specific Heterogeneous Cache Regions. In *Proceedings of the 39th Annual International Symposium on Microarchitecture*, MICRO, pages 433–442. IEEE, 2006.

[98] D. W. Wall. Limits of Instruction-level Parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 176–188. ACM, 1991.

[99] Q. Wang and K. H. Cheng. A Heuristic of Scheduling Parallel Tasks and its Analysis. *SIAM Journal on Computing*, 21(2):281–294, April 1992.

[100] J. Weglarz. Modelling and Control of Dynamic Resource Allocation Project Scheduling Systems. In S. G. Tzafestas, editor, *Optimization and Control of Dynamic Operational Research Models*. Amsterdam: North-Holland, 1982.

[101] K. M. Wilson, K. Olukotun, and M. Rosenblum. Increasing Cache Port Efficiency for Dynamic Superscalar Microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ISCA, pages 147–157. ACM, 1996.

[102] J. L. Wolf, P. S. Yu, J. Turek, and D. M. Dias. A Parallel Hash Join Algorithm for Managing Data Skew. *IEEE Transactions on Parallel and Distributed Systems*, 4(12):1355–1371, 1993.

[103] M. Zhang and K. Asanovic. Fine-grain CAM-tag Cache Resizing Using Miss Tags. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design*, ISLPED, pages 130–135. ACM, 2002.

[104] H. Zhong, S. A. Lieberman, and S. A. Mahlke. Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, HPCA, pages 25–36, 2007.