UNIVERSITY OF CALIFORNIA, SAN DIEGO

Holistic Design for Multi-core Architectures

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Rakesh Kumar

Committee in charge:

Professor Dean Tullsen, Chair
Professor Brad Calder
Professor Fred Chong
Professor Rajesh Gupta
Dr. Norman P. Jouppi
Professor Andrew Kahng

2006

The dissertation of Rakesh Kumar is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

University of California, San Diego

2006

DEDICATIONS

This dissertation is dedicated to friends, family, labmates, and mentors – the ones who taught me, indulged me, loved me, challenged me, and laughed with me, while I was also busy working on my thesis.

To Professor Dean Tullsen for teaching me the values of humility, kindness,and caring while trying to teach me football and computer architecture. For always encouraging me to do the right thing. For always letting me be myself. For always believing in me. For always challenging me to dream big. For all his wisdom. And for being an adviser in the truest sense, and more.

To Professor Brad Calder. For always caring about me. For being an inspiration. For his trust. For making me believe in myself. For his lies about me getting better at system administration and foosball even though I never did.

To Dr Partha Ranganathan. For always being there for me when I would get down on myself. And that happened often. For the long discussions on life, work, and happiness. For always being willing to listen. For all his infectious optimism, enthusiasm, and energy.

To Dr. Norman Jouppi. For his wisdom. For his humility. For his boundless energy and desire to teach. For always wanting me to get better. Most importantly, for getting my jokes and laughing even at the bad ones.

To Drs. Andrew Kahng and Rajesh Gupta for the always constructive feedback and discussions, and for holding high standards for excellence. To Dr. Fred Chong for his invaluable comments. To Drs. Ravi Nair, Victor Zyuban, and Keith Farkas for being excellent mentors.

To my labmates for the memories and the friendships. To Jeremy for always listening to my rants, and for the countless hours spent discussing with me the ‘ “what are the heck are we doing” question. To Jeff for teaching me much of what I know about computer architecture and America. To Satish for his sense

of balance and his ever willingness to be my sounding board. To Erez for always helping me with my perspective on life. To John, Eric, and Jamison for being my mentors in so many ways. To Wei for his enthusiasm. To Mike and Weifeng for their sincerity. To Jack for his sense of humor. To Matt for tennis and the beautiful Easter party. To Shubhro for being a great roommate, labmate, and friend. To Ganesh for his infectious laughter. And to so many other past and present members of the lab for shaping it into what it is.

To my friends who helped me keep my sanity outside work. To Puneet, Satya, Sagnik, Sharma, Swamy, Sameer, and so many others. For the many hours of fun. For sharing sorrow and laughter. For their invaluable friendships. For letting me talk them into going out for a lunch, dinner, or a movie with me always. Even when they wanted to cook at home or had work to do.

Finally, to my family. For their love, support, encouragement, guidance, and checks. For their character. For my character. For laughing with me. For crying with me. For crying for me. For bliss. For happiness. They are the ones responsible for everything that I am and everything that I want to be. And to them I dedicate this dissertation.

"Karmanye va dhikarasthe, Ma phaleshu kadachana
Karma Phala heturbhur ma te sangastva karmani"

TABLE OF CONTENTS

Signature Page iii

Dedication Page iv

Epigraph vi

Table of Contents vii

List of Figures xi

List of Tables xiv

Acknowledgments xv

Vita and Publications xvii

Abstract xx

I Introduction 1
A. Design Methodology for Multi-cores 2
1. Holistic Design 3
B. Overview of Dissertation 7

II Background 9
A. Why Multi-cores 9
B. Chronicling Multi-core Efforts 11

III Holistic Design for Adaptability: Single-ISA Heterogeneous Multi-core Architectures 17
A. Inefficiency due to Workload Diversity 17
B. Single-ISA Heterogeneous Multi-core Architectures 20
C. Evaluation Methodology 24
D. Scheduling for Throughput: Analysis and Results 30
E. Acknowledgment 44

IV Holistic Design for Adaptability: Power Advantages of Heterogeneity . 45
A. Discussion of Core Switching 46
B. Choice of cores 47
C. Switching applications between cores 49
D. Evaluation Methodology 50
1. Modeling of CPU Cores 51

2. Modeling Power . . . 53
3. Estimating Chip Area . . . 57
4. Modeling Performance . . . 57
E. Scheduling for Power: Analysis and Results . . . 59
1. Variation in Core Performance and Power . . . 59
2. Oracle Heuristics for Dynamic Core Selection . . . 62
3. Static Core Selection . . . 64
4. Realistic Dynamic Switching Heuristics . . . 66
5. Practical heterogeneous architectures . . . 69
F. Summary . . . 70
G. Acknowledgment . . . 70

V Holistic Design for Adaptability: Designing Heterogeneous Multi-cores From the Ground Up . . . 71
A. Overview of Related Proposals . . . 71
B. Benefits of Ground-up Design . . . 75
C. From Workloads to Multi-core Design . . . 77
D. Customizing Cores to Workloads . . . 79
E. Methodology . . . 81
1. Modeling of CPU Cores . . . 81
2. Modeling Power and Area . . . 82
3. Modeling Performance . . . 84
F. Analysis and Results . . . 88
1. Fixed Area Budget . . . 88
2. Fixed Power Budget . . . 95
3. Impact of Non-monotonic Design . . . 97
4. Varying Thread-Level Parallelism . . . 97
5. Dynamic Switching . . . 98
6. Efficient Search Techniques . . . 99
G. Validating Results . . . 101
H. Acknowledgment . . . 102

VI Obviating Overprovisioning: Conjoined-core Multiprocessing Architectures . . . 107
A. Related Work . . . 109
B. Baseline Architecture . . . 110
1. Baseline processor model . . . 111
2. Die floorplan and area model . . . 112
C. Conjoined-core Architectures . . . 114
1. ICache sharing . . . 116
2. DCache sharing . . . 117

3. Crossbar sharing . . . 117
4. FPU sharing . . . 119
5. Summary of sharing . . . 120
D. Experimental Methodology . . . 120
E. Simple Sharing . . . 122
1. Sharing the ICache . . . 123
2. DCache sharing . . . 125
3. FPU sharing . . . 126
4. Crossbar sharing . . . 126
5. Simple sharing summary . . . 127
F. Intelligent Sharing of Resources . . . 128
1. ICache sharing . . . 128
2. DCache sharing . . . 130
3. Symbiotic assignment of threads . . . 132
G. A Unified Conjoined-Core Architecture . . . 132
H. Acknowledgment . . . 134

VII The Interconnect Problem and the Need for Co-design . . . 140
A. Related Work . . . 141
B. Interconnection Mechanisms . . . 142
1. Shared Bus Fabric . . . 142
2. P2P Links . . . 147
3. Crossbar Interconnection System . . . 147
C. Modeling Area, Power, and Latency . . . 149
1. Wiring Area Overhead . . . 149
2. Logic Area Overhead . . . 150
3. Power . . . 151
4. Latency . . . 152
D. Modeling Multi-core Architectures . . . 152
1. Workload . . . 154
E. Shared Bus Fabric: Overheads and Design Issues . . . 155
1. Area . . . 156
2. Power . . . 158
3. Performance . . . 158
F. Shared Caches and the Crossbar . . . 161
1. Area and power overhead . . . 161
2. Performance . . . 163
G. Scaling with Technological Parameters . . . 165
H. An Example Holistic Approach to Interconnection . . . 166
I. Acknowledgment . . . 167

VIII Summary and Future Work 174
A. Holistic Design for Adaptability 175
B. Obviating Overprovisioning in Multi-cores 178
C. Interconnection-aware Co-design 179
D. Future Work 180

Bibliography 183

LIST OF FIGURES

III.1 Performance of *applu* over time on Alpha cores 19

III.2 Exploring the potential of heterogeneity: Comparing the throughput of six-core homogeneous and heterogeneous architectures for different area budgets 23

III.3 Benefits from heterogeneity - static scheduling for inter-thread diversity 31

III.4 Three strategies for evaluating the performance an application will realize on a different core 35

III.5 Sensitivity to sampling frequency for time-based trigger mechanisms using the sample-avg core-sampling strategy 38

III.6 Comparison of event-based triggers using the sample-avg core-sampling strategy 40

III.7 Limiting response-time for various loads on comparable budget homogeneous and heterogeneous architectures 43

IV.1 Relative sizes of the Alpha cores when implemented in 0.10 micron technology 48

IV.2 (a) Performance of *applu* on the four cores (b) Oracle switching for energy (c) Oracle switching for energy-delay product. 60

IV.3 *applu* energy efficiency. IPS^2/W varies inversely with energy-delay product 61

IV.4 Results for realistic switching heuristics for heterogeneous multi-cores - the last one is a constraint-less dynamic oracle 68

V.1 Area and Power of the cores 85

V.2 Throughput for *all-same* (top) and *all-different* (bottom) workloads, area budget=$40mm^2$ 89

V.3 Throughput for *all-different* workloads for an area budget of (left to right) $20mm^2$, $30mm^2$, $50mm^2$, and $60mm^2$. 91

V.4 Throughput for *all-same* (left) and *all-different* (right) workloads, power budget=30W 96

V.5 Throughput for *all-different* workloads for a power budget of (left to right) 20W, 40W, 50W, and 60W. 103

V.6 Benefits due to non-monotonicity of cores;area budget=$40mm^2$,power budget =30W 104

V.7 Throughput for *all-same* and *all-different* workloads for different TLPs, area budget=$40mm^2$, power budget=30W 105

V.8 Benefits due to dynamic switching; area budget = $40mm^2$,power budget=30W 106

V.9 Comparing the results using assumed methodology against full simulation results: area budget = $40mm^2$, power budget = 30W 106

VI.1 Baseline die floorplan for studying conjoining, with L2 cache banks in the middle of the cluster, and processor cores (including L1 caches) distributed around the outside 113

VI.2 (a)Floorplan of the original core (b)Layout of a conjoined-core pair, both showing FPU routing. Routing and register files are schematic and not drawn to scale 135

VI.3 A die floorplan with crossbar sharing 136

VI.4 Impact of ICache sharing for various threading levels 136

VI.5 ICache sharing when no extra latency overhead is assumed, cache structure bandwidth is not doubled, and cache is doubly banked 137

VI.6 Impact of Dcache sharing for various threading levels 137

VI.7 DCache sharing when no extra latency overhead is assumed . . 137

VI.8 Impact of FPU sharing for various threading levels 138

VI.9 Impact of private FP divide sub-units 138

VI.10 Reducing crossbar area through width reduction and port sharing138

VI.11 ICache assertive access results when the original structure bandwidth is not doubled 139

VI.12 Fetch-combining results 139

VI.13 Effect of assertive access and static assignment 139

VII.1 The assumed shared bus fabric for our interconnection study . 143

VII.2 A typical crossbar 148

VII.3 Floorplans for 4, 8 and 16 core processors 168

VII.4 Area overhead for shared bus fabric. 169

VII.5 Power overhead for shared bus fabric 169

VII.6 Performance overhead due to shared bus fabric. 170

VII.7 Trading off interconnection bandwidth with area. 170

VII.8 Area overhead for cache sharing – results for crossbar routed over L2 assume uniform cache density. 170

VII.9 Power overhead for cache sharing (the three bars, left to right, correspond to 2-way, 4-way and full sharing). 171

VII.10 Evaluating cache sharing for a fixed cache size for different crossbar implementations – no area overhead is assumed 171

VII.11 Evaluating cache sharing for a fixed die area – area overhead taken into account 172

VII.12 Scaling of interconnection overhead with pipelining and technology 172

VII.13 Hierarchical approach (splitting SBFs) 173

VII.14 Split vs Monolithic SBF 173

LIST OF TABLES

III.1 Configuration and area of the EV4 and EV6 cores. 26

III.2 Benchmarks simulated for evaluating heterogeneous multi-cores for throughput 28

IV.1 Configuration of the cores used for power evaluation of heterogeneous multi-cores 52

IV.2 Power and area statistics of the Alpha cores 56

IV.3 Benchmarks simulated for power evaluation of heterogeneous multi-cores 58

IV.4 Summary for dynamic *oracle* switching for energy on heterogeneous multi-cores 65

IV.5 Summary for dynamic *oracle* switching for energy-delay on heterogeneous multi-cores 65

IV.6 Oracle heuristic for static core selection on heterogeneous multi-cores – energy metric. Rightmost two columns are for dynamic selection 66

V.1 Various Parameters and their possible values for configuration of the cores 82

V.2 Area and power estimation methodology and relevant assumptions for various hardware structures. Renaming for OOO cores is assumed to be done using RAM tables. IW refers to issue-width, WP to a write-port, and RP to a read-port. 83

V.3 Derived Area and Power Estimates for Processor Components . 84

V.4 Benchmarks used for design space exploration of heterogeneous multi-cores 86

VI.1 Simulated Baseline Processor for studying Conjoining 111

VI.2 Benchmarks simulated for evaluating conjoining 121

VI.3 Results with multiple sharings. 133

VII.1 Design parameters for wires in different metal planes 149

VII.2 Interconnection-related Logic overhead 158

ACKNOWLEDGEMENTS

The text of Chapter III is in part a reprint of the material as it appears in the proceedings of the Thirty-first International Symposium on Computer Architecture (pp64-75, June 2004). The dissertation author was the primary researcher and author and the co-authors involved in the submission directed the research which forms the basis for Chapter III.

The text of Chapter IV is in part a reprint of the material as it appears in the proceedings of the Thirty-sixth International Symposium on Microarchitecture (pp81-92, December 2003). The dissertation author was the primary researcher and author and the co-authors involved in the submission directed the research which forms the basis for Chapter IV.

The text of Chapter V is in part a reprint of the material as it appears in the proceedings of the Fifteenth International Conference on Parallel Architectures and Compilation Techniques (September 2006). The dissertation author was the primary researcher and author and the co-authors involved in the submission directed the research which forms the basis for Chapter V.

The text of Chapter VI is in part a reprint of the material as it appears in the proceedings of the Thirty-seventh International Symposium on Microarchitecture (pp195-206, December 2004). The dissertation author was the primary researcher and author and the co-authors involved in the submission directed the research which forms the basis for Chapter VI.

The text of Chapter VII is in part a reprint of the material as it appears in the proceedings of the Thirty-second International Symposium on Computer Architecture (pp408-419, June 2005). The dissertation author was the primary researcher and author and the co-authors involved in the submission directed, supervised, and assisted the research which forms the basis for Chapter VII.

VITA

1981	Born, Pusa, Bihar (INDIA)
1997	High School Certificate Pusa, Bihar (INDIA)
2001	BTech. in Computer Science & Engineering Indian Institute of Technology (IIT), Kharagpur
2002	Internship, Hewlett-Packard Labs, WRL Group, Palo Alto, California
2004	Internship, IBM TJ Watson Research Center, Yorktown Heights, New York
2006	Doctor of Philosophy University of California, San Diego

PUBLICATIONS

David Sheldon, Rakesh Kumar, Frank Vahid, Dean Tullsen, and Roman Lysecky. "Conjoining Soft-Core FPGA Processors". In International Conference on Computer-Aided Design (ICCAD). November, 2006.

David Sheldon, Rakesh Kumar, Roman Lysecky, Frank Vahid, and Dean Tullsen. "Application-Specific Customization of Parameterized FPGA Soft-Core Processors". In International Conference on Computer-Aided Design (ICCAD). November, 2006.

Rakesh Kumar, Dean M. Tullsen, and Norman P. Jouppi. "Core Architecture Optimization for Heterogeneous Chip Multiprocessors". In 15th International Symposium on Parallel Architecture and Compilation Techniques (PACT). September, 2006.

Matt Devuyst, Rakesh Kumar, and Dean Tullsen. "Scheduling for Energy and Performance for a CMP of SMT Processors". In International Parallel & Distributed Processing Symposium (IPDPS), April, 2006.

Rakesh Kumar, Dean Tullsen, Norman Jouppi, and Parthasarathy Ranganathan. "Heterogeneous Chip Multiprocessors". IEEE Computer. November 2005.

Rakesh Kumar, Victor Zyuban, and Dean M. Tullsen. "Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads and Scaling". In the 32nd International Symposium on Computer Architecture, ISCA-32, June, 2005.

Yannakis Sazeidis, Rakesh Kumar, Dean Tullsen, and Theophanis Konstantinou. "The Danger of Interval-Based Power-Efficiency Metrics: When Worst is Best". Computer Architecture Letters, Volume 4, January 2005.

Rakesh Kumar, Norman Jouppi, and Dean Tullsen. "Conjoined-core Chip Multiprocessing". In the 37th International Symposium on Microarchitecture, MICRO-37, December, 2004.

Eric Tune, Rakesh Kumar, Dean Tullsen, Brad Calder "Balanced Multithreading: Increasing Throughput via a Low Cost Multithreading Hierarchy". In the 37th International Symposium on Microarchitecture, MICRO-37, December, 2004.

Rakesh Kumar, Dean Tullsen, Parthasarathy Ranganathan, Norman Jouppi, and Keith Farkas. "Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance". In the 31st International Symposium on Computer Architecture, ISCA-31, June, 2004.

Rakesh Kumar, Keith Farkas, Norman Jouppi, Parthasarathy Ranganathan, and Dean Tullsen. "Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction". In the 36th International Symposium on Microarchitecture , MICRO-36, December, 2003.

Rakesh Kumar, Keith Farkas, Norman Jouppi, Parthasarathy Ranganathan, and Dean Tullsen. "A Multi-Core Approach to Addressing the Energy-Complexity Problem in Microprocessors". Workshop on Complexity-Effective Design(WCED), June 2003.

Rakesh Kumar, Keith Farkas, Norman Jouppi, Parthasarathy Ranganathan, and Dean Tullsen. "Processor Power Reduction Via Single-ISA Heterogeneous Multi-Core Architectures". Computer Architecture Letters, Volume 2, April 2003.

Rakesh Kumar and Dean Tullsen. "Compiling for Instruction Cache Performance on a Multithreaded Architecture". In the 35th International Symposium on Microarchitecture, MICRO-35, November, 2002.

V. Ramakrishna, Rakesh Kumar, and Anupam Basu. "Switching Activity Minimization by Efficient Instruction Set Architecture Design". In the 45th IEEE International Midwest Symposium on Circuits and Systems, MWSCAS2002, August, 2002.

Rakesh Kumar, Tushar Kanti Patra, and Anupam Basu. "Software Energy Optimization for Preemptive Realtime Systems". In the 4th International Symposium on High-Performance Computing, ISHPC-IV, May, 2002.

Rakesh Kumar and Sudeshna Sarkar. "Three-staged Refinement Model for Information Retrieval with Application to Newspaper Articles and Online Documents". In International Symposium on Artificial Intelligence, ISAI-2001, December, 2001.

Tusharkanti Patra, Rakesh Kumar, and Anupam Basu. "Cache Optimization for Minimizing Software energy in Embedded Systems". In International Conference on Communications, Computers and Devices, ICCCD, December, 2000.

Rakesh Kumar and D. Dutta Majumdar. "A Multi-processing Database Model for Efficient Storage and Retrieval of Medical Images". Journal of Computer Science & Informatics, Volume30, No 3, page 31-38.

Rakesh Kumar and D.Dutta Majumdar. "A Multi-staged Database Model for Efficient Storage and Retrieval of Medical Images". In All India Seminar on Information Technology organized by Institution of Engineers (India),March, 2000.

ABSTRACT OF THE DISSERTATION

Holistic Design for Multi-core Architectures

by

Rakesh Kumar
Doctor of Philosophy in Computer Science (Computer Engineering)
University of California, San Diego, 2006
Professor Dean Tullsen, Chair

Increasing design complexity and diminishing marginal utility of monolithic processor designs has resulted in integration of multiple loosely-coupled processing cores on the same die. However, fundamental questions remain about the right form, implementation, and methodology for multi-core designs. This thesis addresses these questions.

A popular methodology for designing a multi-core architecture is to replicate an off-the-shelf core design multiple times, and then connect the cores together using an interconnect mechanism. However, this methodology is "multi-core oblivious" as subsystems are designed/optimized unaware of the overall chip-multiprocessing system they would become parts of. This thesis demonstrates that this methodology is very inefficient in terms of area/power, and recommends a holistic approach where the subsystems are designed from the ground up as different components of a full system.

Inefficiency in "multi-core oblivious" multi-core designs comes at different levels. Having multiple replicated cores results in an inability to adapt to the demands of execution workloads, and results in either underutilization or overutilization of processor resources. This thesis proposes *single-ISA (instruction-set architecture) heterogeneous multi-core architectures* where the die hosts cores of

varying power/performance characteristics, but all capable of running the same ISA. Such a processor can result in significant power savings and performance improvements if the applications are mapped to cores judiciously. The thesis also presents holistic design methodologies for such architectures.

Another source of inefficiency is blind replication of over-provisioned hardware structures. To that effect, the thesis proposes *conjoined-core chip multiprocessing* where the adjacent cores of a multi-core architecture share some resources. The thesis shows that this can result in significant area savings without much performance degradation. The thesis also proposes novel optimizations for minimizing the already small degradation.

Yet another source of inefficiency is the interconnection. This thesis shows that the interconnection overheads can be very significant for a "multi-core oblivious"" multi-core design – especially as the number of cores increases and the pipelines get deeper. The thesis demonstrates the need to co-design the cores, the memory and the interconnection to obviate the inefficiency problem, and also makes several suggestions regarding co-design.

I

Introduction

The processor industry has seen a tremendous growth since its inception. The performance of processors has increased by over 5000 times since the time Intel introduced the first general-purpose microprocessor [8]. This increase in processor performance has been fueled by several technology shifts at various levels of the processor design flow – architecture, tools and techniques, circuits, processes, and materials. These technology shifts not only provide a quantum jump in performance, but also require us to rethink system architecture and revisit fundamental questions regarding the scope and applicability of computing.

Specifically, at the architectural level, we have moved from scalar processing, where a processor could execute one instruction every clock cycle, to superscalar processing, where a processor could execute multiple instructions every clock cycle, to out-of-order processing, where the processors could now also execute instructions out-of-order, to on-chip multithreading (e.g., simultaneous multithreading), where a processor could support multiple streams of execution simultaneously. We are now at the cusp of another major technology shift at the architectural level. This technology shift is towards multi-core architectures – i.e., architectures with multiple processing nodes on the same die. Such processors, also called chip multiprocessors, can not only support multiple streams

of program execution at the same time, but provide productivity advantages over monolithic processors due to the relative simplicity of the cores, and hence shorter design cycles. Such processors can also make better use of the hardware resources, as the marginal utility of transistors is higher for a smaller processing node with a smaller number of transistors.

While the potential technological advantages of such architectures are apparent, one of the major questions facing computer architects right now is – how should one design a multi-core architecture? That is, what should a multi-core architecture look like? The final form and implementation will depend heavily on the design methodology as well. Hence, another fundamental question that needs to be addressed is – what should be the methodology for doing multi-core design?

This thesis seeks to address these questions.

I.A Design Methodology for Multi-cores

One suggested (as well as practiced) methodology for multi-core design is to take an off-the-shelf core design, optimize it for power and/or performance, replicate it multiple times, and then connect the cores together using an interconnection mechanism in a way that maximizes performance for a given area and/or power budget. This is a clean, relatively easy way to design a multi-core because one design team can work on the core, the other can work on the caches, the third can work on the interconnection, and then there can be a team of chip-integrators who will put them all together to create a multi-core (*aka* chip multiprocessor). Such a methodology encourages modularity as well as reuse, and serves to keep the design costs manageable.

However, this methodology is "multi-core oblivious". This is because each subsystem that constitutes the final chip multiprocessing system is designed

and optimized without any cognizance of the overall system it would become a part of. For example, a methodology like the above forces each subsystem to target the entire universe of applications (i.e, a set of all possible applications that a processor is expected to run). This is an overly stringent constraint as the real requirement is on the full system to target the universe where the individual subsystems can be working cooperatively to that effect.

As this thesis shows, "multi-core oblivious" designs result in highly inefficient processors in terms of area and power. This is because the above constraint results in either overutilization or underutilization of processor resources. For example, Pentium Extreme is a dual-core Intel processor that is constructed by replicating two identical off-the-shelf cores. While the area and power cost of duplicating cores is 2X (in fact, even more considering the glue logic required), the performance benefits are significantly lower [93]. The thesis shows that while the costs are superlinear with the number of cores for all "multi-core oblivious" multi-core designs, the benefits tend to be highly sublinear. In fact, this sublinearity increases with increasing number of cores on the die, and it is becoming increasingly clear that we need to design multi-cores differently.

I.A.1 Holistic Design

This thesis proposes a holistic approach to designing multi-core architectures. A holistic multi-core design methodology involves various processor subsystems being designed from the ground up as different components of a full system. This enables the various subsystems to work cooperatively in executing a certain task efficiently. Specifically, this thesis identifies three sources of inefficiency in "multi-core oblivious" multi-core design and proposes the corresponding holistic approaches that can significantly obviate inefficiency. The three sources of inefficiency are inadaptability to workloads, resource overprovisioning,

and high interconnection overheads.

Workload Adaptability

There is diversity in workloads that a typical processor is expected to run. This diversity can be due to diversity among applications or different threads of the same application. It can also be due to diversity across varying program phases within an application or varying processor load. If a multi-core is constructed by replicating an off-the-shelf core design, each core might be able to target a certain class of applications well; however, the processor as a whole will not be able to adapt to application diversity. Alternatively, a multi-core can be constructed using "mediocre" cores [37], where a core is designed to perform adequately over the entire universe of applications. Such a processor, however, will be overprovisioned or underprovisioned for most individual applications. While overprovisioning leads to wasted power and real estate, underprovisioning leads to reduced performance.

This thesis recommends a holistic approach for adapting to workload diversity. Instead of constructing a chip multiprocessor through replication of one core design, the thesis advocates *single-ISA heterogeneous multi-core architectures*. That is, architectures with multiple types of cores on the same die. These cores can all execute the same ISA (instruction-set architecture), but represent different points in the power performance continuum. For example, a high-performance, high-power core and a low-performance, low-power core on the same die. Applications can then be mapped to cores judiciously in a way that each application or execution thread runs on the core whose resources match the current execution needs. This significantly enhances the efficiency of computation and can result in processors that have up to 63% higher throughput than the equivalent-budget "multi-core oblivious" homogeneous processors. The flex-

ibility to do resource matching also enables power reduction. The thesis shows that more than three-fold power savings are possible.

Resource Overprovisioning

Resource overprovisioning causes inefficiency as even the unused transistors consume power and occupy real estate. Designers usually provision the CPU for a few important applications that each stress a particular resource. Similarly, hardware often gets added to provide a feature even if not many applications use it. This results in overprovisioning for most applications. When a multi-core is constructed by blindly replicating such overprovisioned cores, the cost of overprovisioning is exacerbated as it gets multiplied by the number of compute cores. What is really needed to maintain is the same level of provisioning for any single thread without scaling the costs by the number of cores.

This thesis proposes *conjoined-core chip multiprocessing* – a holistic approach to obviating the overprovisioning problem by allowing topologically feasible sharing of overprovisioned structures between adjacent cores of a chip multiprocessor. The shared structures need to be accessible by both cores in a way that does not require additional queues or pipeline stalls. The thesis investigates several sharing policies consistent with the constraints of modern high-frequency core designs. Sharing results are presented for instruction and data caches, the floating-point unit, and the input ports of the crossbar connecting the cores to the shared L2 cache. This thesis shows that intelligent policies to schedule access to shared structures can minimize the performance degradation of conjoining to 10-12% while saving roughly half the area.

Interconnections Overheads

While inefficiency due to inadaptability and overprovisioning afflict even monolithic processors, interconnection overheads result in a design problem unique to multi-cores. This thesis examines the area, power, performance, and design issues for the interconnects on a chip multiprocessor. It attempts to present a comprehensive view of a class of interconnect architectures. It shows that the design choices for the interconnect have significant effect on the rest of the chip, potentially consuming a big fraction of the real estate and power budget. Since these budgets are shared with the cores and the caches, the number, the size, and the design of cores gets affected anytime the interconnection is made aggressive. Conversely, anytime the number, the size, or the design of cores are scaled up, it is going to place conflicting demands on the interconnect – requiring higher bandwidth, but providing even less real estate.

This thesis shows that designs that treat the interconnect as an entity that can be independently architected and optimized (the "multi-core oblivious" designs) would not arrive at the best multi-core design. Several examples are presented showing the need for careful, holistic co-design of multi-core processors. For instance, increasing interconnect bandwidth requires area that then constrains the number of cores or cache sizes, and does not necessarily increase performance. Also, shared level-2 caches become significantly less attractive when the overhead of the resulting crossbar is accounted for.

As an example of holistic design, a hierarchical bus structure is examined which negates some of the performance costs of the assumed baseline architectures. Hierarchical interconnection architectures recognize the diversity in applications running on a multi-core and provide shorter coherence paths for interactions between the physically adjacent cores than the cores that are located far from each other. Such interconnection architectures can provide lower

average-case latency for coherence transactions at the expense of worse-case latency.

I.B Overview of Dissertation

Chapter II gives background information on architecture and design of multi-core processors. It details the reasons for the advent of multi-core architectures and discusses some prominent multi-core efforts. It also identifies the the research issues that need to be addressed for a more widespread adoption of multi-core technology.

Chapter III discusses how "multi-core oblivious" multi-core designs cannot adapt to the diversity in workloads. It presents *single-ISA heterogeneous multi-core architectures* as a holistic solution to adapting to diversity. These architectures can provide significantly higher throughput for a given area or power budget. Chapter IV shows how they can also be used to reducing processor power. Chapter V discusses methodologies for holistic, ground-up design of multi-core architecture and demonstrates their benefits over processors designed using off-the-shelf components.

Chapter VI introduces overprovisioning as another source of significant inefficiency in multi-core architectures that are designed by blindly replicating cores. Such architectures unnecessarily multiply the cost of overprovisiong by the number of compute nodes. The chapter introduces a holistic approach to addressing overprovisioning through *conjoined-core chip-multiprocessing*. Conjoined-core multi-cores have adjacent cores sharing large, overprovisioned structures. Intelligently scheduling accesses to the shared resources enables conjoined-core multi-cores to achieve significantly higher efficiency ($throughput/area$) than their "multi-core oblivious" multi-core counterparts.

Chapter VII details the overheads that conventional interconnection

mechanisms entail, especially as the number of cores increase and as transistors get faster. The chapter shows that overheads become unmanageable very soon and require a holistic approach to designing multi-cores where the interconnect is co-designed with the cores, and the caches. Several examples are presented for the need to co-design.

II

Background

This chapter provides background information on topics related to this thesis. Chapter II.A explains why we are seeing an advent of multi-core architectures. II.B provides an overview of some groundbreaking multi-core efforts.

II.A Why Multi-cores

The processor industry has made giant strides in terms of speed and performance. The first microprocessor, Intel 4004 [8], ran at 784 KHz while the microprocessors of today run easily in the GHz range due to significantly smaller and faster transistors. The increase in performance has been historically consistent with Moore's law that states that the number of transistors on the processor die keeps doubling every eighteen months due to the transistors getting smaller every successive process technology.

However, the price that one pays for getting performance has been going up rapidly as well. For example, as Horowitz *et al* [65] show, the power cost for squeezing a given amount of performance has been going up linearly with the the performance of the processor. This is super-exponential increase over time. Similarly, the area cost for squeezing a given amount of performance has been

going up as well.

In fact, one thing that the progress in processor architecture and technology has taught us is that the marginal utility of transistors is decreasing. While area and power are roughly linear with the number of transistors, performance is highly sublinear with the number of transistors. Empirically, it has been close to the square root of the number of transistors [63, 62]. The main reason why we are on the wrong side of the square law is that we have already extracted the easy ILP (instruction-level parallelism) through techniques like superscalar processing, out-of-order processing, etc.. The ILP that is left is difficult to extract. However, technology keeps making transistors available to us at the rate predicted by Moore's Law [101] (though it has slowed down, of late). We have reached a point where we have more transistors available than we know how to make effective use of in a conventional monolithic processor environment.

This quandary gives rise to multi-core computing. Instead of using all the transistors to construct a monolithic processor targeting high single-thread performance, we can use the transistors to construct multiple simpler cores where each core can execute a program (or a thread of execution). Once we do that, we can jump to the right side of the square law (see below). Such cores can collectively provide higher many-thread performance (or throughput) than the baseline monolithic processor at the expense of single-thread performance.

Consider, for example, the Alpha 21164 and Alpha 21264 cores. Alpha 21164 (also called, and henceforth referred to as, EV5) is an inorder processor that was originally implemented in 0.5 micron technology [18]. Alpha 21264 (also called, and henceforth referred to as, EV6) is an out-of-order processor that was originally implemented in 0.35 micron technology [19]. If we assume both the processors to be mapped to the same 0.10 micron technology, an EV6 core would be roughly five times bigger than an EV5 core (methodological details for

technology mapping in Chapter IV). If one were to take a monolithic processor like EV6, and replace it with EV5 cores, one could construct a multi-core that can support five streams of execution for the same area budget (ignoring the cost of interconnection and glue logic). However, for the same technology, the single-thread performance of an EV6 core is only roughly 2.0-2.2 times that of an EV5 core (assuming performance is proportional to the square root of the number of transistors). Hence, if we replaced an EV6 monolithic processor by a processor with five EV5 cores, the aggregate throughput would be more than a factor of two higher than the monolithic design for the same area budget. Similar throughput gains can be shown even for a fixed power budget. This potential to get significantly higher aggregate performance for the same budget is the main motivation for multi-core architectures.

Another advantage of multi-cores over monolithic designs is improved design productivity. The more complex a core is, the higher the design and verification costs in terms of time, opportunity, and money. Several recent monolithic designs have taken several thousand man years worth of work. A multi-core enables deployment of pre-existing cores thereby bringing down the design and verification costs. Even when the cores are designed from the ground up, the simplicity of cores can keep the costs low. With increasing market competition and declining hardware profit margins, the time-to-market of processors is more important than before, and multi-cores help to that effect.

Other benefits of multi-cores over monolithic designs can be explained as derivatives of the above two advantages.

II.B Chronicling Multi-core Efforts

This chapter provides an overview of some of the visible general-purpose multi-core projects that have been undertaken in academia and industry. The

chapter does not claim completeness and is biased towards the first few general-purpose multi-core processors that broke ground for mainstream multi-core computing.

The first multi-core design was the Hydra [59]. Hydra was a 4-way chip multiprocessor that integrated four 250 MHz MIPS cores on the same die. The cores each had 8 KB private instruction and data caches and share a 128 KB level-2 cache. The L1 caches were write-through and inclusion was maintained between the L1s and the L2. Coherence was maintained by having all the caches connected through a shared write bus and a read bus. Hydra was focused not only on providing hardware parallelism for throughput-oriented applications, but also on providing high single-thread performance for applications that can be parallelized into threads by a compiler. Significant amount of support was provided in the hardware to aid the thread-level speculation efforts. Hydra never got implemented, but an implementation of Hydra (0.25 micron technology) was estimated to take up 88 mm^2 of area.

One of the earliest commercial multi-cores, Piranha [26] was a 8-way chip multiprocessor designed at DEC/Compaq WRL. It was targeted at commercial, throughput-oriented workloads whose performance is not limited by instruction-level parallelism. It integrated eight simple, inorder processor cores on the die. Each core consisted of private 64 KB instruction and data caches and shared a 1 MB L2 cache. Inclusion was not enforced between the L1s and the L2. The connection between cores is through a high-bandwidth switch instead of buses. The processor also integrated on the chip functionality required to support scalability of the processor to large multiprocessing systems. Also, unlike Hydra, the cores did not provide support for data speculation due to the nature of the expected workload. Piranha never got implemented.

Around the same time as Piranha, Sun started the design of a multi-

core processor MAJC 5200 [126]. MAJC 5200 was a two-way chip-multiprocessor where each core was a four-way issue VLIW (very large instruction word) processor. The processor was targeted at multimedia and Java applications. Each core had a private 16KB L1 instruction cache. The cores shared a 16KB dual-ported data cache. The shared L2 data cache aided in efficient space-time computing (where one of the cores speculatively executes future instructions) as well as vertical multithreading (where a running thread get swapped out of the core on a cache miss and is replaced by a waiting thread). The processor also provided other required functionality on the chip for speculative multithreading. Small L1 caches and the lack of an on-chip L2 cache made the processor unsuitable for commercial workloads. One implementation of MAJC 5200 (0.22 micron technology) took 15W of power and 220 mm^2 of area.

Sun later came out with other multi-core products, like UltraSparc-IV [123] and Niagara [80]. UltraSparc-IV is a dual-core processor where each core is four-way superscalar and is two-way simultaneously multithreaded (SMT). Cores have private L1 caches (32 KB 4-way instruction, 64 KB 4-way data). Each core also has a private 8MB L2 cache. One implementation of UltraSparc-IV (in 0.13 micron technology) operated at 1.2GHz, took a maximum of 198W of power, and consisted of 66 million transistors. Niagara is an eight-core processor where each core is four-way multithreaded. The processor is targeted to applications with abundant thread-level parallelism. Single-thread performance and instruction-level parallelism are only second-order concerns as each core is in-order and scalar. Cores each have a 4-way 16KB private L1 instruction cache and a 4-way 8KB data cache. The relatively small data cache is based on the expectation that hardware multithreading (among the four threads) can effective hide L1 miss latency. The cores are connected to a shared four-banked 3MB L2 cache where each bank is 12-way set-associative. Connection is through a

200GB/s crossbar. One implementation of Niagara (in 0.90 micron technology) operates at 1.2GHz, takes up 72W at 1.3V, has a die area of $37mm^2$, and consists of 279 million transistors.

IBM's multi-core efforts started with Power4 [68]. Power4, a contemporary of MAJC 5200 and Piranha processors, was a dual-core processor running at 1GHz where each core was a five-issue out-of-order superscalar processor. Each core consisted of a private direct-mapped 32KB instruction cache and a private 2-way 32KB data cache. The cores were connected to a shared triply-banked 8-way set-associative L2 cache. The connection was through a high-bandwidth crossbar switch (called crossbar-interface unit). Four Power4 chips could be connected together within a multi-chip module and made to logically share the L2. One implementation of Power4 (in 0.13 micron technology) consisted of 184 million transistors and took up $267mm^2$ in die area.

Power5 [69] was a successor to Power4. The most significant difference was that the individual cores now were two-way simultaneously multithreaded for a total of four threads per chip. Also, there was an increased level of integration where the memory controllers were brought on-chip. Also, the interface to the off-chip L3 was brought on-chip and coupling between the CPU and the L3 was made stronger. A 130 nm implementation of Power5 was $389mm^2$ in size and consisted of 287 million transistors.

IBM's most ambitious multi-core offering arguably has been Cell [75]. Cell is a multi-ISA heterogeneous chip-multiprocessor that consists of one two-way SMT dual-issue Power core and eight dual-issue SIMD (single instruction, multiple data)-style Synergistic Processing Element (SPE) cores on the same die. While the Power core executes the PowerPC instruction set (while supporting the vector SIMD instruction set at the same time), the SPEs execute SIMD instructions of variable widths. The Power core has a multi-level storage hierarchy

– 32KB instructions and data caches, and a 512KB L2. Unlike the Power core, the SPEs operate only on their local memory (local store or LS). Code and data must be transferred into the associated LS for an SPE to operate on. LS addresses have an alias in the Power core address map, and transfers to/from an individual LS and the global memory are kept coherent and done through DMAs (direct memory accesses). An implementation of Cell in 90nm operated at 3.2GHz, consisted of 234 million transistors and took $229mm^2$ of die area.

The first x86 multi-core processors were introduced by AMD last year. The relatively slow adoption of multi-core technology for the most dominant ISA in the market can be attributed to the then-perpetual success of the microprocessor industry in continually driving the single-thread performance of processors up by increasing clock speed. The diminishing marginal utility of transistors and increasing power budgets eventually forced a move towards on-chip multithreading and eventually chip multiprocessing. At the time of writing this thesis, AMD offers Opteron [2], Athlon [3], and Turion [1] dual-core processors serving different market segments. Intel's dual-core offerings include Pentium D [10], Pentium Extreme [11], Xeon [12], and Core Duo [9] processors.

One constraint for all the above multi-core designs was that they had to be capable of running legacy code in their respective ISAs. This restricted the degree of freedom in architecture and design of these processors. Three academic multi-core projects that were not bound by such constraints were RAW, TRIPS, and WaveScalar.

The RAW [130] processor consists of sixteen identical tiles spread across the die in a regular two-dimensional pattern. Each tile consists of communication routers, one scalar MIPS-style core, an FPU (floating-point unit), a 32KB DCache, and a software-managed 96KB Icache. Tiles are sized such that the latency of communication between two adjacent tiles is always one cycle. Tiles

are connected using on-chip networks that interface with the tiles through the routers. Hardware resources on a RAW processor (tiles, pins, and the interconnect) are exposed to the ISA. This enables the compiler to generate aggressive code that map more efficiently to the underlying computation substrate.

The TRIPS [112] processor consists of four large cores that are constructed out of small decentralized structures and are accompanied by resources (like memory and wires) that are polymorphous. The cores can be partitioned into small execution nodes when a program with high data-level parallelism needs to be executed. These execution nodes can also be logically chained when executing streaming programs. Like RAW, the microarchitecture is exposed to the ISA. Unlike RAW and other multi-cores discussed above, the unit of execution is a hyperblock. A hyperblock commits only when all instructions belonging to a hyperblock finish execution.

WaveScalar [124] is an attempt at moving away from Von-Neumann processing in order to get the full advantage of multi-cores. It has a dataflow instruction set architecture that allows for traditional memory ordering semantics. Each instruction executes on an ALU (arithmetic logic unit) that sits inside a cache, and explicitly forwards the result to the dependent instructions. The $ALU + cache$ is arranged as regular tiles, thereby allowing the communication overheads to be exposed to hardware. Like RAW and TRIPS, wavescalar also supports all the traditional imperative languages.

There have been multi-core offerings in non-mainstream computing markets as well. A few examples are Broadcom SiByte (SB1250, SB1255, SB1455) [5], PA-RISC (PA-8800) [7], Raza Microelectronics' XLR processor [13] that has eight MIPS cores, Cavium Networks' Octeon [6] processor that has 16 MIPS cores, Arm's MPCore processor [4], and Microsoft's Xbox 360 game console [14] that uses a triple core PowerPC microprocessor.

III

Holistic Design for Adaptability: Single-ISA Heterogeneous Multi-core Architectures

This chapter discusses the implication of workload diversity on multi-core design and presents a new class of holistically-designed architectures that are significantly more efficient than "multi-core oblivious" designs. Section III.A discusses the diversity present in computer workloads and how naively-designed multi-core architectures cannot adapt to them. Section III.B introduces *single-ISA heterogeneous multi-core architectures* that can adapt to workload diversity and result in highly efficient computation. The chapter also discusses the scheduling policies and mechanisms that are required to effect adaptability.

III.A Inefficiency due to Workload Diversity

Workloads can be diverse in various ways. There can be diversity among applications (or different threads belonging to the same application). There can be diversity within applications (data dependent or between different execution

phases of the same application). Then, there can be diversity in the number of applications constituting a workload (or the number of threads for a given application). Each has implications for multi-core design.

The amount of diversity among applications that a typical computer is expected to run can be considerable. For example, there can often be more than a factor of ten difference in the average performance of *gcc* and *mcf* on a given processor [116]. The difference is due to a combination of the difference in their semantics and resource requirements, and diversity in their program inputs. Even in the server domain there can be diversity among threads. Here the diversity can be because of batch processing, different threads processing different inputs, or threads having different priorities.

A "multi-core oblivious" multi-core design (or a homogeneous design that consists of identical cores) can target only a single point efficiently in the diversity spectrum. For example, a decision might need to be made beforehand if the core should be designed to target *gcc* or *mcf*. In either case, an application whose resource demands are different from the amount of resources provided by a core will suffer – the resource mismatch will either result in underutilization of the resulting processor or it will result in low program performance. Note that this is the same problem that a general-purpose uniprocessor faces as well, as the same design is expected to perform well for the entire universe of applications.

The same is true of diversity within applications as well. There is significant diversity among different phases of the same application. The diversity is again due to difference in semantics, inputs and execution phases. Consider, for example, *applu*. It is a computational fluid dynamics workload that performs non-linear partial differential equations. Figure III.1 shows the performance of *applu* over time when run on different cores belonging to the Alpha processor family (the cores are assumed to be implemented in the same 0.10 micron tech-

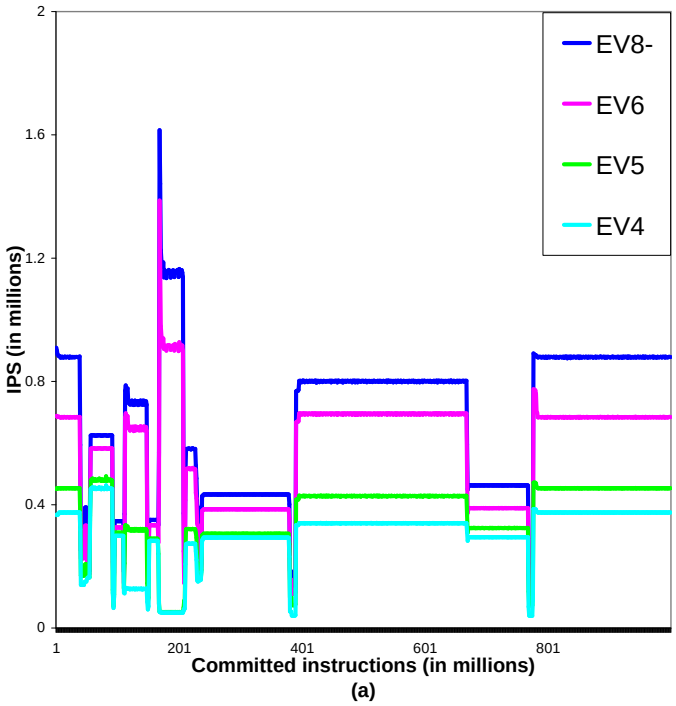

Figure III.1: Performance of *applu* over time on Alpha cores

nology, methodological details later in the chapter). As the graph shows, not only does the performance of *applu* change over time, so does the ratio of performance of *applu* on different cores. There are phases of execution where the difference in performance between the most complex core and the simplest core is only a few percent. On the other hand, there are phases where the difference is more than a factor of 10. Hence, even for the same application, there are inherently slow phases (the ones with low instruction-level parallelism) that will be happy enough on a simple core, while there are other phases that have high instruction-level parallelism and can execute fast when provided with a high performance core. A homogeneous multi-core design cannot target multiple types of phases and hence intra-program diversity results in inefficient execution for such architectures.

Homogeneous multi-core architectures also suffer from the inability to adjust to the diversity due to the varying number of threads in a workload. De-

pending on the workload, compute conditions, or objective functions, there are scenarios where the number of threads is small, applications need high single-thread performance, and peak throughput is not much of a concern. Such scenarios include single-threaded workloads, serial phases of a parallel program, low server loads, high priority threads, etc. On the other hand, there are other scenarios where the number of threads is high, high peak throughput is a requirement, and single-thread performance is not much of a concern. Such scenarios include multiprogrammed/multithreaded workloads, parallel phases of a parallel application, high server loads, etc. The bane of a processor is that it is expected to run both these classes of applications at one time or the other. A homogeneous multi-core architecture is forced to make a choice between high single-thread performance and high peak throughput (as determined by the number and complexity of cores). This inability to adjust to varying levels of thread level parallelism also results in processor inefficiency.

III.B Single-ISA Heterogeneous Multi-core Architectures

With the inadaptability of homogeneous multi-core architectures in mind, we propose *single-ISA heterogeneous multi-core architectures*. That is, architectures with multiple core types on the same die. The cores are all capable of executing the same ISA (instruction-set architecture), but represent different points in the power/performance continuum – a low-power, low-performance core and a high-power, high-performance core on the same die, for example.

The advantages of single-ISA heterogeneous architectures stem from two sources. The first advantage results from a more *efficient adaptation to application diversity*. As discussed above, applications (or different phases of a single application) place different demands on the architecture, stemming from the nature of the computation [87]. While some applications take good advantage of

the most advanced processors, others often under-utilize that hardware and suffer little performance loss when run on a less aggressive processor. For example, a floating-point application with regular code might make good use of an out-of-order pipeline with high issue width; however, a bandwidth-bound, control-sensitive application might perform almost as well on an in-order core with low issue width. Given a set of diverse applications and heterogeneous cores, we can assign applications (or phases of applications) to cores such that those that benefit the most from complex cores are assigned to them, while those that benefit little from complex cores are assigned to smaller, simpler cores. This allows us to approach the performance of an architecture with a larger number of complex cores.

The second advantage from heterogeneity results from a more *efficient use of die area for a given thread-level parallelism*. Successive generations of microprocessors have been providing diminishing performance returns per chip area. This is evident from the following. Microprocessor implementation technology has been scaling for many years according to Moore's Law [101] for lithography and roughly according to MOS scaling theory [38]. For a given $O(n)$ scaling of lithography, one can expect an equivalent $O(n)$ increase in transistor speed and an $O(n^2)$ increase in the number of transistors per unit area. If the increases in transistor speed and transistor count were to directly translate to performance, one would expect an $O(n^3)$ increase in performance. However, past microprocessor performance has only been increasing at an $O(n^2)$ rate [63, 62]. This is not too surprising, since the performance improvement of many microprocessor structures (e.g., cache memories) is less than linear with their size.

Therefore, in an environment with large amounts of process or thread-level parallelism, such a nonlinear relationship between transistor count and microprocessor speed means that higher throughputs could be obtained by building

a large number of small processors, rather than a small number of large processors. However, in practice the amount of process or thread level parallelism in most systems will vary with time. This implies that building chip-level multiprocessors with a mix of cores – some large cores with high single-thread performance and some small cores with high throughput per die area – is a potentially attractive approach.

To explore the potential from heterogeneity, we model a number of chip multiprocessing configurations that can be derived from combinations of two existing off-the-shelf processors from the Alpha architecture family – the EV5 (21164) and the EV6 (21264) processors. Figure III.2 compares the various combinations in terms of their performance and their total chip area. In this figure, performance is that obtained from the best static mapping of applications constituting multiprogrammed SPEC workloads to the processor cores. The staircase represents the maximum throughput obtainable using a homogeneous configuration for a given area.

We see from this figure that over a large portion of the graph the highest performance architecture for a given area limit, often by a significant margin, is a heterogeneous configuration. The increased throughput is due to the increased number of contexts as well as improved processor utilization.

Constant-area comparisons do not tell the whole story, because equivalent area does not necessarily imply equal cost, power, or complexity. But the results are instructive, nonetheless. Note also that we explore a smaller subset of the design space than possible for obtaining results for the graph because of our constraint of focusing on just two generations of a commodity processor family (studies with more generations of cores in Chapter IV). However, even with this limited approach, our results in the graph make a case for the general advantages for heterogeneity. For example, on the far left part of the graph, the area is

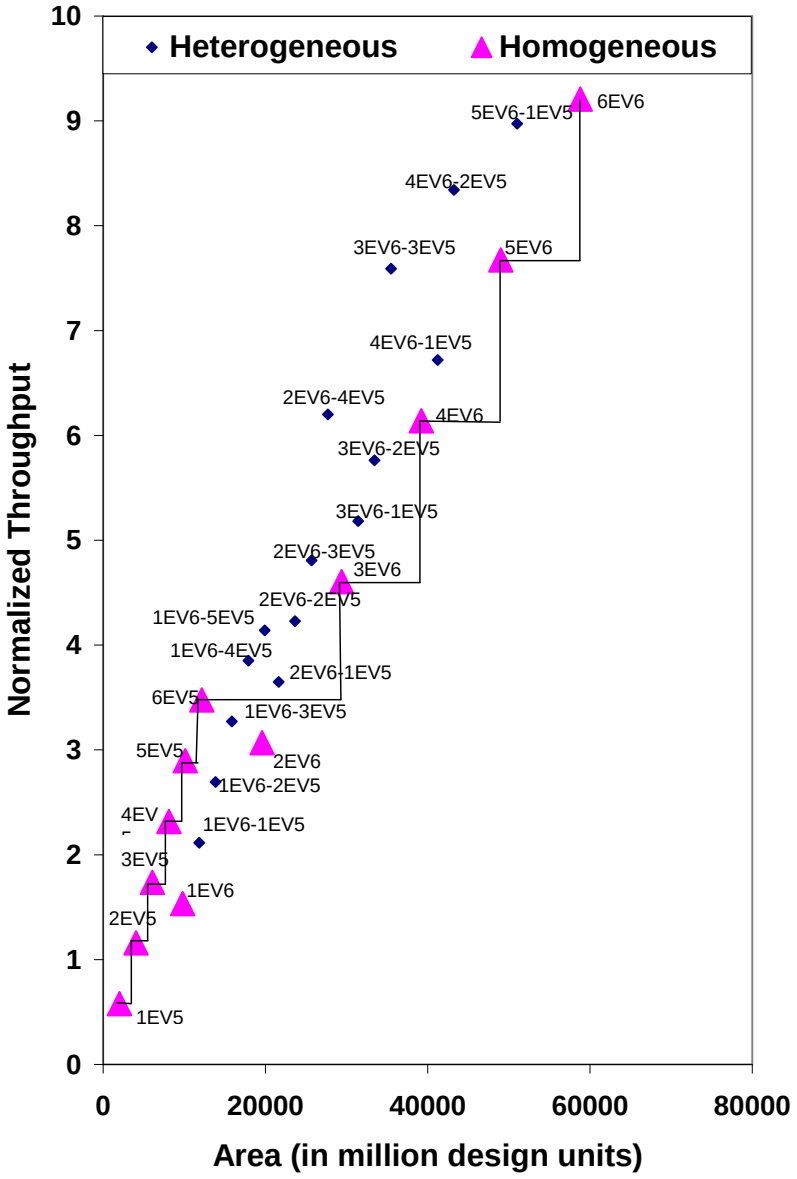

Figure III.2: Exploring the potential of heterogeneity: Comparing the throughput of six-core homogeneous and heterogeneous architectures for different area budgets

insufficient to support a heterogeneous configuration of the EV6 and EV5 cores; however, our other data (not plotted here) on heterogeneous architectures using EV5 and EV4 (21064) cores confirm that these designs are superior in this region.

III.C Evaluation Methodology

This section discusses the methodological details for evaluating the benefits of heterogeneous multi-core architectures over their homogeneous counterparts. It discusses how intelligent application-to-core scheduling is needed to exploit the throughput benefits. It also details the hardware assumptions made for the evaluation and provides the methodology for constructing multiprogrammed workloads for evaluation. The last two sub-sections discuss the simulation details and the evaluation metrics respectively.

Supporting multi-programming

The primary issue when using heterogeneous cores for greater throughput is with the scheduling, or assignment, of jobs to particular cores. We assume a scheduler at the operating system level that has the ability to observe coarse-grain program behavior over particular intervals, and move jobs between cores. Since the phase lengths of applications are typically large [118], this enables the cost of core switching to be piggybacked with the operating system context-switch overhead. Core switching overheads are modeled in detail for the evaluation of dynamic scheduling policies presented in this chapter – ones where jobs can move throughout execution.

With multiple jobs and multiple cores, the task of the scheduler is to find the best global assignment. All of our policies in this chapter strive to maximize average performance gain over all applications in the workload. Fairness is not taken into consideration explicitly. All threads make good progress, but

if further guarantees are needed, we assume the priorities of those threads that need performance guarantees will reflect that. The heterogeneous architecture is also ideally suited to manage varied priority levels, but that advantage is not explored here.

An additional issue with heterogeneous multi-core architectures supporting multiple concurrently executing programs is cache coherence. In this chapter, we study multi-programmed workloads with disjoint address spaces, so the particular cache coherence protocol is not an issue (even though we do model the performance effects of the writeback of dirty cache data during core switching). However, when there are differences in cache line sizes and/or per-core protocols, the cache coherence protocol might need some redesign. We believe that even in those cases, cache coherence can be accomplished with minimal additional overhead.

Minor differences in the ISA between processor generations can be handled easily. Either programs are compiled to the least common denominator (the EV5), or we use software traps for the older cores. If extensive use is made of the software traps, our mechanisms will naturally shy away from those cores, due to the low performance.

Hardware assumptions

Table III.C summarizes the configurations used for the EV5 (Alpha 21164) and the EV6 (Alpha 21264) cores that we use for our throughput-related evaluations. The cores are assumed to be implemented in 0.10 micron technology and are clocked at 2.1 GHz (the EV6 frequency when scaled to 0.10 micron).

In addition to the individual L1 caches, all the cores share an on-chip 4 MB, 4-way set-associative, 16-way L2 cache. The cache line size is 128 bytes. Each bank of the L2 cache has a memory controller and an associated RDRAM

Table III.1: Configuration and area of the EV4 and EV6 cores.

Processor	**EV5**	**EV6**
Issue-width	4	6 (Out-of-order)
I-Cache	8KB, Direct-mapped	64KB, 2-way
D-Cache	8KB, Direct-mapped	64KB, 2-way
Branch Pred.	2K-gshare	hybrid 2-level
Number of MSHRs	4	8
Number of threads	1	1
Area (in mm^2)	5.06	24.5

channel. The memory bus is assumed to be clocked at 533Mhz, with data being transferred on both edges of the clock for an effective frequency of 1GHz and an effective bandwidth of 2GB/s per bank. Note that for any reasonable assumption about power and ground pins, the total number of pins that this memory organization would require would be well within the ITRS limits[15] for the cost/performance market. A fully-connected matrix crossbar interconnect is assumed between the cores and the L2 banks. All L2 banks can be accessed simultaneously, and bank conflicts are modeled. The access time is assumed to be 10 cycles. Memory latency was set to be 150 ns. We assume a snoopy bus-based MESI coherence protocol and model the writeback of dirty cache lines for every core-switch.

Table III.C also presents the area occupied by the cores. These were computed using a methodology outlined in Chapter IV. As can be seen from the table, a single EV6 core occupies as much area as 5 EV5 cores.

To evaluate the performance of heterogeneous architectures, we perform comparisons against homogeneous architectures occupying equivalent area. We

assume that the total area available for cores is around 100 mm^2. This area can accommodate a maximum of 4 EV6 cores or 20 EV5 cores. We expect that while a 4-EV6 homogeneous configuration would be suitable for low-TLP (thread-level parallelism) environments, the 20-EV5 configuration would be a better match for the cases where TLP is high. For studying heterogeneous architectures, we choose a configuration with 3 EV6 cores and 5 EV5 cores with the expectation that it would perform well over a wide range of available thread-level parallelism. It would also occupy roughly the same area.

For the chosen cache configuration, the area occupied by the L2 cache would be around 135 mm^2. The rest of the logic (e.g. 16 memory-controllers, crossbar interconnect etc.) might occupy up to 50 mm^2 (crossbar area calculations assume 300 bit wide links implemented in the M3/M5 layer; memory-controller area assumptions are consistent with Piranha [26] estimates). Hence, the total die size would be approximately 285 mm^2. Note that actual area might be dependent on the layout and other issues, but the above assumptions provide a first-order model adequate for this study.

Workload construction

All our evaluations are done for various thread counts ranging from one through the maximum number of available processor contexts. Instead of choosing a large number of benchmarks and then evaluating each number of threads using workloads with completely unique composition, we instead choose a relatively small number of SPEC2000 benchmarks (8) and then construct workloads using these benchmarks. Table III.2 summarizes the benchmarks used. These benchmarks are evenly distributed between integer benchmarks (*crafty, mcf, eon, bzip2*) and floating-point benchmarks (*applu, wupwise, art, ammp*). Also, half of them (*applu, bzip2, mcf, wupwise*) have a large memory footprint (over 175MB),

Table III.2: Benchmarks simulated for evaluating heterogeneous multi-cores for throughput

Program	Description	fast-forward (billion instr)
ammp	Computational Chemistry	8.75
applu	Parabolic/Elliptic Partial Diff. Equations	116
art	Image Recognition/Neural Networks	15
bzip2	Compression	65
crafty	Game Playing:Chess	83
eon	Computer Visualization	55
mcf	Combinatorial Optimization	55
wupwise	Physics/Quantum Chromodynamics	88

while the other half (*ammp, art, crafty, eon*) have memory footprints of less than 30MB. All the data points are generated by evaluating 8 workloads for each case and then averaging the results. A workload consisting of n threads is constructed by selecting the benchmarks using a sliding window (with wraparound) of size n and then shifting the window right by one. Since there are 8 distinct benchmarks, the window selects eight distinct workloads (except for cases when the window-size is a multiple of 8, in those cases all the selected workloads have identical composition). All of these workloads are run, ensuring that each benchmark is equally represented at every data point. This methodology for workload construction is similar to that used in [129, 120].

Simulation approach

Benchmarks are simulated using SMTSIM, a cycle-accurate execution-driven simulator that simulates an out-of-order, simultaneous multithreading processor [127]. SMTSIM executes unmodified, statically linked Alpha binaries. The simulator was modified to simulate the various multi-core architectures.

The Simpoint tool [118] was used to find good representative fast-forward distances for each benchmark. Fast-forwarding involves skipping the initial instructions of a benchmark during simulation so that only representative portions of the benchmark are accounted during measurements. Table III.2 also shows the distance each benchmark was fast-forwarded before beginning simulation. Unless otherwise stated, all simulations involving n threads were done for $500 \times n$ million total instructions. All the benchmarks are simulated using *ref* inputs provided by the SPEC.

Evaluation metrics

In a study like this, IPC (number of total instructions committed per cycle) is not a reliable metric as it would inordinately bias all the heuristics (and policies) against inherently slow-running threads. Any policy that favors high-IPC threads boosts the reported IPC by increasing the contribution from the favored threads. But this does not necessarily represent an improvement. While the IPC over a particular measurement interval might be higher, in a real system the machine would eventually have to run a workload inordinately heavy in low-IPC threads, and the artificially-generated gains would disappear. Hence, we use weighted speedup [120, 128] for our evaluations. In this chapter, weighted speedup measures the arithmetic sum of the individual IPCs of the threads constituting a workload divided by their IPC on a baseline configuration when running alone. This metric makes it difficult to produce artificial speedups

by simply favoring high-IPC threads.

As another axis of comparison, we also present results from open system experiments where jobs enter and leave the system at random rates. This represents a real system with variable job-arrival rates and variable service times. The systems are then compared in terms of average response time of applications as well as system queue lengths. Response time of an application, as used in this chapter, is the time between job submission and job completion, and hence accounts for the queuing delays that might be incurred when the processor is busy. We believe that this is a better metric than throughput to quantify the performance of a real system with variable job inter-arrival rates and/or variable job service times.

III.D Scheduling for Throughput: Analysis and Results

In this section, we demonstrate the performance advantage of the heterogeneous multi-core architectures for multithreaded workloads and demonstrate job-to-core assignment mechanisms that allow the architecture to deliver on its promise. The first two subsections focus on the former, and the rest of the section demonstrates the further gains available from a good job assignment mechanism.

Static scheduling for inter-thread diversity

A heterogeneous architecture can exploit two dimensions of diversity in an application mix. The first is diversity between applications. The second is diversity over time within a single application. Prior work [118, 131] has shown that both these dimensions of diversity occur in common workloads. In this section, we attempt to separate these two effects by first looking at the performance of a static assignment of applications to cores. Note that the static assignment approach may not eliminate the need for core switching in several

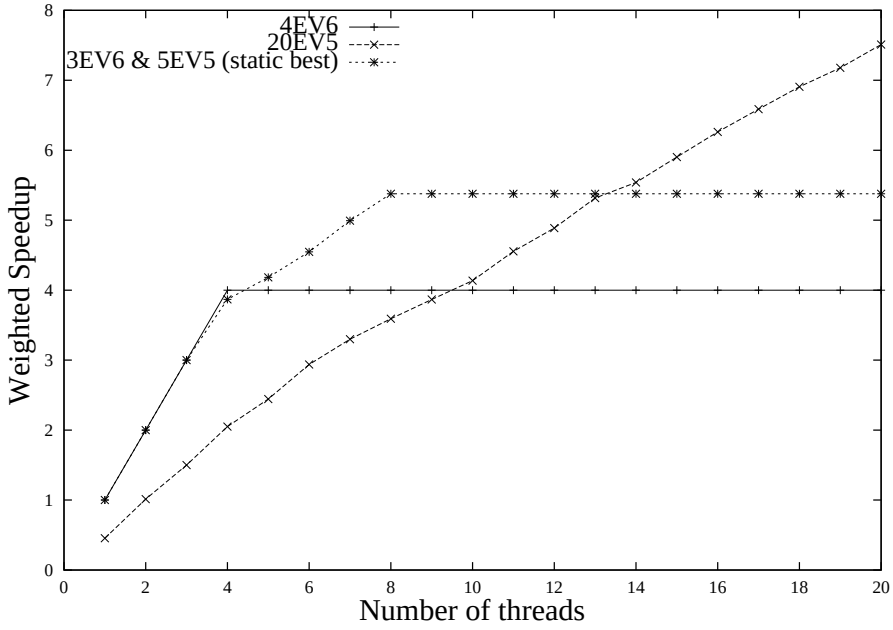

Figure III.3: Benefits from heterogeneity - static scheduling for inter-thread diversity

cases, because the best assignment of jobs to cores will change as jobs enter and exit the system.

Figure III.3 shows the results comparing one heterogeneous architecture against two homogeneous architectures all requiring approximately the same area. The heterogeneous architecture that we evaluate includes 3 EV6 cores and 5 EV5 cores, while the two homogeneous architectures that we study have 4 EV6 cores or 20 EV5 cores, respectively. For each architecture, the graph shows the variation of the average weighted speedup for varying number of threads.

For the homogeneous CMP configuration, we assume a straightforward scheduling policy, where as long as a core is available, any workload can be assigned to any core. For the heterogeneous case, we use an assignment that seeks to match the optimal static configuration as closely as possible. The optimal configuration would factor in even the potential shared L2 cache interactions.

However, determining this configuration is only possible by running *all possible* combinations. Instead, as a simplifying assumption, our scheduling policy assumes no knowledge of L2 interactions (only for determining core assignments – the interactions are still simulated) when determining the static assignment of workloads to cores. This simplification allows us to find the best configuration (defined as the one which maximizes weighted speedup) by simply running each job alone on each of our unique cores and using that to guide our core assignment. This results in consistently good, if not optimal, assignments. For a few cases, we compared this approach to an exhaustive exploration of all combinations; our results indicated that this results in performance close to the optimal assignment.

The use of weighted speedup as the metric ensures that those jobs assigned to the EV5 are those that are least affected (in relative IPC) by the difference between EV6 and EV5. In both the homogeneous and heterogeneous cases, once all the contexts of a processor get used, we just assume that the weighted speedup will level out as shown in the Figure III.3. The effects when the number of jobs exceeds the number of cores in the system (e.g., additional context switching) is modeled more exactly in the following section.

As can be seen from Figure III.3, even with a simple static approach, the results show a strong advantage for heterogeneity over the homogeneous designs, for most levels of threading. The heterogeneous architecture attempts to combine the strengths of both the homogeneous configurations - CMPs of a few powerful processors (EV6 CMP) and CMPs of many less powerful processors (EV5 CMP). While for low threading levels, the applications can run on powerful EV6 cores resulting in high performance for each of the few threads, for higher threading levels the applications can run on the added EV5 contexts enabled by heterogeneity, resulting in higher overall throughput.

The results in Figure III.3 show that the heterogeneous configuration

achieves performance identical to the homogeneous EV6 CMP from 1 to 3 threads. At 4 threads, the optimum point for the EV6 CMP, that configuration shows a slight advantage over the heterogeneous case. However, this advantage is very small because with 4 threads, the heterogeneous configuration is nearly always able to find one thread that is impacted little by having to run on an EV5 instead of EV6. As soon as we have more than 4 threads, however, the heterogeneous processor shows clear advantage.

The superior performance of the heterogeneous architecture is directly attributable to the diversity of the workload mix. For example, *mcf* underutilizes the EV6 pipeline due to its poor memory behavior. On the other hand, benchmarks like *crafty* and *applu* have much higher EV6 utilization. Static scheduling on heterogeneous architectures enables the mapping of these benchmarks to the cores in such a way that overall processor utilization (average of individual core utilization values) is high.

The heterogeneous design remains superior to the EV5 CMP out to 13 threads, well beyond the point where the heterogeneous architecture runs out of processors and is forced to queue jobs. Beyond that, the raw throughput of the homogeneous design with 20 EV5 cores wins out. This is primarily because of the particular heterogeneous designs we chose. However, we can always come up with a different configuration that is competitive with more threads (e.g., fewer EV6's, more EV5's), if that is the desired design point.

Compared to a homogeneous processor with 4 EV6 cores, the heterogeneous processor performs up to 37% better with an average 26% improvement over the configurations considering 1-20 threads. Relative to 20 EV5 cores, it performs up to 2.3 times better, and averages 23% better over that same range.

These results demonstrate that over a range of threading levels, a heterogeneous architecture can outperform comparable homogeneous architectures.

Although the results are shown here only for a particular area and two core types, our experiments with other configurations (at different processor areas and core types) indicate that these results are representative of other heterogeneous configurations as well.

Dynamic scheduling for intra-thread diversity

The previous sections demonstrated the performance advantages of the heterogeneous architecture when exploiting core diversity for inter-workload variation. However, that analysis has two weaknesses – it used unimplementable assignment policies in some cases (e.g., the static assignment oracle) and ignored variations in the resource demands of individual applications. This section solves each of these problems, and demonstrates the importance of good dynamic job assignment policies.

Prior work has shown that an application's demand for processor resources varies across phases of the application. Thus, the best match of applications to cores will change as those applications transition between phases. In this section, we examine implementable heuristics that dynamically adjust the mapping to improve performance.

These heuristics are sampling-based. During the execution of a workload, every so often, a trigger is generated that initiates a *sampling phase*. In the *sampling* phase, the scheduler permutes the assignment of applications to cores. During this phase, the dynamic execution profiles of the applications being run are gathered by referencing hardware performance counters. These profiles are then used to create a new assignment, which is then employed during a much longer phase of execution, the *steady* phase. The steady phase continues until the next trigger. Note that applications continue to make forward progress during the sampling phase, albeit perhaps non-optimally.

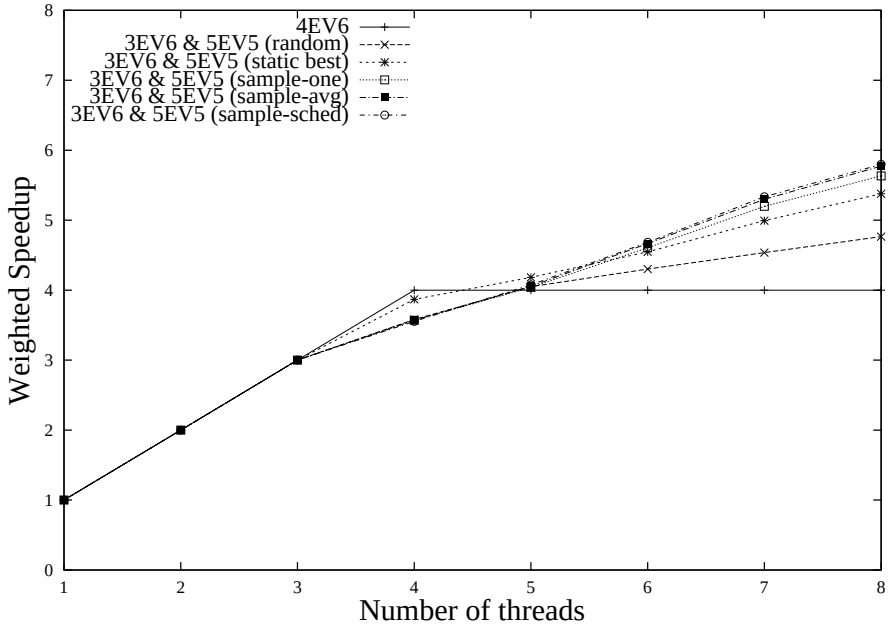

Figure III.4: Three strategies for evaluating the performance an application will realize on a different core

In terms of the core sampling strategies, There are a large number of application-to-core assignment permutations possible. We prune the number of permutations significantly by assuming that we would never run an application on a less powerful core when doing so would leave a more powerful core idle (for either the sampling phase or the steady phase). Thus, with four threads on our 3 EV6/5 EV5 configuration, four possible assignments are possible based on which thread gets allocated to the EV5. With more threads, the number of permutations increase, up to 56 potential choices with eight threads. Rather than evaluating all these possible alternatives, our heuristics only sample a subset of possible assignments. Each of these assignments are run for 2 million cycles. At the end of the sampling phase, we use the collected data to make assignments.

Selection of the assignments to be sampled depends on how much we account for interactions at the L2 cache level (which, if large, can color the data collected for all threads and lead to inappropriate decisions). We evaluated three

strategies for sampling the assignment space.

The first strategy, *sample-one*, samples as many assignments as is needed to run each thread once on each core-type. This assumes that the single sample is accurate, regardless of what other jobs are doing. Then the assignment is made, maximizing weighted speedup under the assumption future performance will be the same as our one sample for each thread. The assignment that maximizes weighted speedup is simply the one that assigns to the EV5s those jobs whose ratio of average EV5 throughput to EV6 throughput is highest.

The second strategy, *sample-avg*, assumes we need multiple samples to get the average behavior of a job on each core. In this case, we sample as many times as there are threads running. The samples are distinct and are done such that we get at least two runs of each thread on each core type, then base the assignment (again maximizing expected weighted speedup) on the average performance of each thread on each core.

The third strategy, *sample-sched*, assumes we know little about a particular assignment unless we have actually run it. It thus samples a number of possible assignments, and then is constrained to choose one of the assignments it sampled. In fact, we sample $4 \times n$ representative assignments for a n-threaded workload (bounded by the maximum possible for that configuration). Selection of the best core assignment, of those sampled, is the one that maximizes total weighted speedup, using average EV5 throughput for each thread as the baseline.

Figure III.4 presents a quantitative comparison of the effectiveness of the three strategies. The average weighted speedup values reported here were obtained using a default time-based trigger that resulted in a sampling phase being triggered every 500 million processor cycles; we later evaluate the impact of other time intervals. Also included in the graph, for comparison, are (1) the results obtained using the homogeneous multi-core processor, (2) the random assignment

policy described in the previous section, and (3) the best static assignment found previously.

As suggested by the graph, the *sample-sched* strategy performs the best, although *sample-avg* has very similar performance (within 2%). Even *sample-one* is not much worse. We observed a similar trend for other time intervals and for other trigger types. We conclude from this result that for our workload and L2 cache configuration, the level of interaction at the L2 cache is not sufficient to affect overall performance unduly. We use *sample-avg* as the basis for our trigger evaluation in the discussion to follow as it not only has lower overhead than *sample-sched*, but is also more robust than both *sample-sched* and *sample-one* against worst-case events like phase changes *during* sampling.

The second significant result that the graph shows is that the intelligent assignment policies make a significant performance difference, allowing us to outperform the random core assignment strategy by up to 22%. Perhaps more surprising is the importance of the dynamic sampling and reconfiguration, as we outperform the static best by as much as 10%. We take a hit at 4 threads, when the sampling overhead first kicks in, but quickly recover that loss as the number of threads increases, maximizing the scheduler's flexibility. The results also suggest that fairly good decisions can be made about an application's relative performance on various cores even if it runs for no more than 2 million cycles on each core. This also indicates that the cold-start effect on core-switching is *much less* than the running time on a core during sampling – it would be difficult to make good decisions during a short sampling period if cold-start effects are high.

Sampling overhead depends significantly on the trigger mechanisms. Sampling effectively requires juggling two conflicting goals – minimizing sampling overhead and reacting quickly to changes in workload behavior. To manage this tradeoff, we compare two classes of trigger mechanisms, one based on a pe-

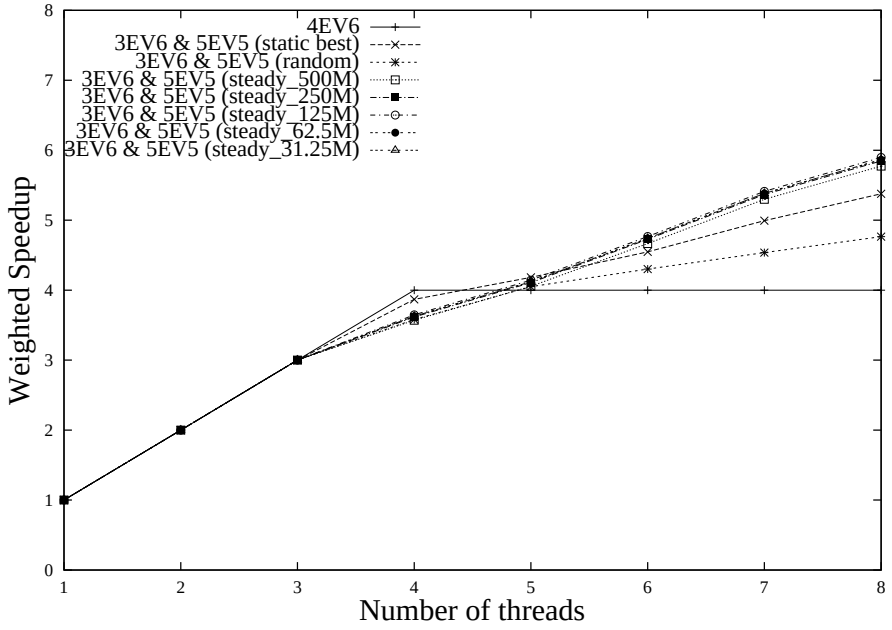

Figure III.5: Sensitivity to sampling frequency for time-based trigger mechanisms using the sample-avg core-sampling strategy

riodic timer, and the second based on events indicating significant changes in performance.

We begin by evaluating the first class and the performance impact of varying the amount of time between sampling phases, that is, the length of the steady phase. For smaller steady phase lengths, a greater fraction of the total time is spent in the sampling phases, thus contributing overhead. The overhead derives from, first, the overhead of application core switching each time we sample a different configuration, and second, the fact that sampling is usually running non-ideal configurations.

Figure III.5 presents a comparison of the average weighted speedup obtained with steady-phase lengths between 31.25 million and 500 million cycles for the sample-average strategy. We note from this graph that the sampling frequency has a second-order impact on performance while the steady-phase length

of 125 million cycles performs best overall. Also, in a similar observation as in [87], we found that the act of core-switching has relatively small overhead. So, the optimal sampling frequency is determined by the average phase length for the applications constituting the various workloads, as well as the ratio of the lengths of the steady phase and the sampling phase.

While time-triggered sampling is very simple to implement, it does not capture either inter-thread or intra-thread diversity fully. A fixed sampling frequency is inadequate when the phase lengths of different applications in the workload mix are different. Also, each application can demonstrate multiple phases each with its own phase length. For example, in our simulation window, while *art* demonstrates a periodic behavior with phase length of 80 million instructions, *mcf* demonstrates at least two distinct phases with one of the phases at least 350 million instructions long. Any sampling-based heuristic that hopes to capture phase changes for both *art* and *mcf*, with minimal overhead, needs to be adaptive.

Therefore, we consider the second class of trigger mechanisms. Here, we monitor the run-time behavior of the workload and detect when sufficiently significant changes have occurred. We consider three instantiations of this trigger class. With the *individual-event* trigger, a sampling phase is triggered every time the steady-phase IPC of an individual thread changes by more than 50%. In contrast, with the *global-event* trigger, we sum the absolute values of the percent changes in IPC for each application, and trigger a sampling phase when this value exceeds 100%. The last heuristic, *bounded-global-event*, modifies the *global-event* trigger by initiating a sampling phase if more than 300 million cycles has elapsed since the last sampling phase, and avoiding sampling if the global event trigger occurs within 50 million cycles since the last sampling phase. All the thresholds were determined by observing the execution characteristics of the

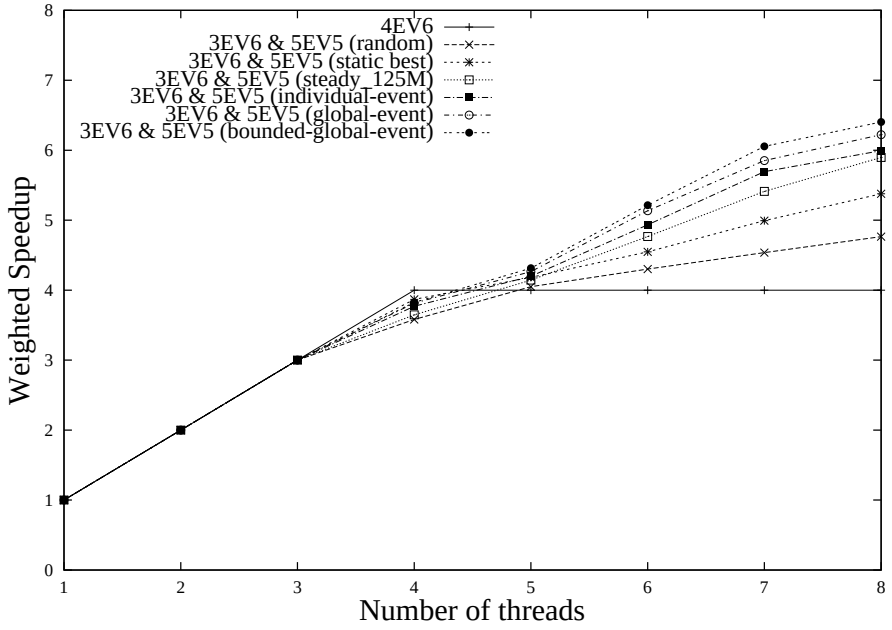

Figure III.6: Comparison of event-based triggers using the sample-avg core-sampling strategy

simulated applications.

Figure III.6 presents a comparison of these three event-based triggers, along with the time-based trigger mechanism using a steady-state length of 125 million cycles (the best one from the previous discussion). We continue to use the *sample-avg* core-sampling strategy. The graph also includes the static-best heuristic from Section III.D, and the homogeneous core. As we can see from the figure, the event-based triggers out-perform the best timer-based trigger and the static assignment approach. This mechanism effectively meets our two goals of reacting quickly to workload changes and minimizing sampling.

While the individual event trigger performs well in general, using a global event trigger achieves better performance. This is because a change in the behavior of a single application might often not result in changing the workload-to-cores mapping. Using a global event trigger guards against these false pos-

itives. The bounded-global-event trigger achieves the best performance (close to a 20% performance improvement over static) indicating the benefits from a hybrid timer-based and event-based approach. It has all the advantages of a global-event trigger, but the bounds also help to guard against oversampling and undersampling. By eliminating most of the sampling overhead, this hybrid scheme also closes the gap again with the homogeneous processor at 4 threads. Similar trends were observed for different values of parameters embodied in the event-based triggers.

To summarize this section, the results presented indicate that dynamic heuristics which intelligently adapt the assignment of applications to cores can better leverage the diversity advantages of a heterogeneous architecture. Compared to the base homogeneous architecture, the best dynamic heuristic achieves close to a 63% improvement in throughput in the best case (for 8 threads) and an average improvement in throughput of 17% over configurations running 1-8 threads. Even more interesting, the best dynamic heuristic achieves a weighted speedup of 6.5 for eight threads, which is close to 80% of the optimal speedup (8) achievable for this configuration (despite the fact that over half of our cores have roughly half the raw computation power of the baseline core!). In contrast, the homogeneous configuration achieves only 50% of the optimal speedup. We have also demonstrated the importance of an intelligent dynamic assignment, which achieves up to 31% improvement over a random scheduler.

While our relatively simple dynamic heuristics are effective, there are clearly many other heuristics that could be considered. For example, event-based sampling could be based on other metrics aside from IPC, such as changes in ILP, cache or branch behavior, or basic block profiles as suggested in [118]. Further, rather than using past behavior as a simple approximation for future behavior, more complex models for phase identification and prediction [118] could also be

used.

Open system experiments

Graphs of performance at various threading levels are instructive, but do not necessarily reflect real system performance accurately. Real systems typically operate at a variety of threading levels (run queue sizes), and observed performance is a factor of the whole range. Thus, while a particular architecture may appear to be optimal at a single point (even if that design point represents the expected average behavior), it may not be optimal on a system that experiences a range of demand levels. This section explores the performance of heterogeneous architectures on an open system. It does so with a sophisticated simulation framework that models random job arrivals and random job lengths. This addresses some methodological issues that remain even when using the weighted speedup metric. In this experiment, we are able to guarantee that every simulation executes the exact same set of instructions. Additionally, we are able to use average response time as our performance metric.

We model a system where jobs enter and leave the system with exponentially distributed arrival rate λ and exponentially distributed average time to complete a job T. We study the two systems for varying values of λ and observe the effects on mean response time, queue length, and stability of the systems. Whenever a system is stable, it is better to measure response time rather than throughput, since throughput cannot possibly exceed the rate of job arrival. If two stable systems are compared and one is faster, the faster one will complete jobs more quickly and thus typically have fewer jobs queued up waiting to run.

For these experiments, we randomly generate jobs (using a Poisson model) centered around an average expected execution time of 200 million cycles on an EV6. Jobs are generated by first generating random numbers with average

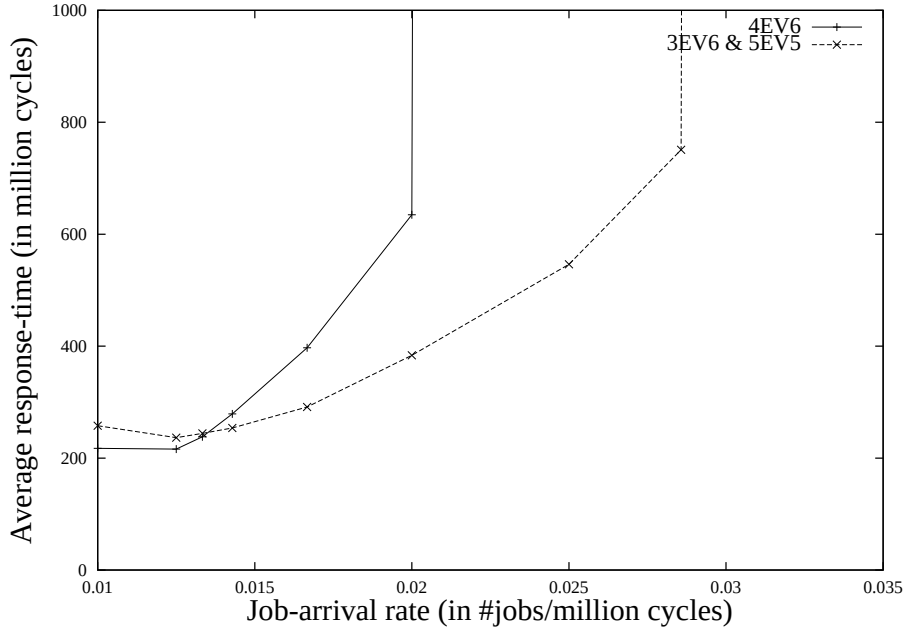

Figure III.7: Limiting response-time for various loads on comparable budget homogeneous and heterogeneous architectures

distribution centered around 200 million cycles and then executing that many instructions multiplied by the single-threaded IPC of the benchmarks on EV6. We then simulate different mean job arrival rates with exponential distributions. To model a random system but produce repeatable results, for each point on the job arrival rate axis, we feed the same jobs in the same order with the same arrival times to each of the systems under experimentation.

For the heterogeneous configuration, we use a naive scheduling heuristic which simply assigns jobs randomly, only ensuring that the more powerful processors get used before the less powerful. Significant improvements over this heuristic will be demonstrated in the following section.

Figure III.7 shows the results for these experiments. The most profound difference between the homogeneous and the heterogeneous architectures is that they saturate at very different throughputs. The homogeneous architecture sees

unbounded response times as the arrival rate approaches its maximum throughput around 2 jobs per 100 million cycles. At this point, its run queue becomes (if we ran the simulations long enough) infinite.

However, the heterogeneous architecture remains stable well beyond this point. Furthermore, the scheduling heuristics we will demonstrate in the following section will actually increase the maximum throughput of the architecture, so its saturation point would be even further out. The heterogeneous architecture also sees average response time improvements well before the other architecture becomes saturated. There is only a very narrow region where the homogeneous architecture sees no queuing beyond 4 jobs, but the heterogeneous is forced to use an EV5 occasionally, where the homogeneous architecture sees some slight advantage. As soon as the probability of queue lengths beyond four becomes non-insignificant, the heterogeneous architecture is superior.

Another interesting point to note is that, besides supporting greater throughput in peak load conditions, heterogeneous chip-level multiprocessor response time degrades more gracefully under heavier loads than for homogeneous processors. This should enhance system reliability in transient high load conditions. This is particularly important as the reliability and availability of systems become more important with the maturity of computer technology.

III.E Acknowledgment

The text of Chapter III is in part a reprint of the material as it appears in the proceedings of the Thirty-first International Symposium on Computer Architecture (pp64-75, June 2004). The dissertation author was the primary researcher and author and the co-authors involved in the submission directed the research which forms the basis for Chapter III.

IV

Holistic Design for Adaptability: Power Advantages of Heterogeneity

The ability of *single-ISA heterogeneous multi-core architectures* to adapt to workload diversity can also be used for improving the energy efficiency of processors. Again, the architecture consists of a chip-level multiprocessor with multiple, diverse processor cores. These cores all execute the same instruction set, but include significantly different resources and achieve different performance and energy efficiency on the same application. During an application's execution, the operating system software tries to match the application to the different cores, attempting to meet a defined objective function. For example, it may be trying to meet a particular performance requirement or goal, but doing so with maximum energy efficiency.

IV.A Discussion of Core Switching

There are many reasons, some discussed in previous sections, why the best core for execution may vary over time. The demands of executing code vary widely between applications; thus, the best core for one application will often not be the best for the next, given a particular objective function (assumed to be some combination of energy and performance). In addition, the demands of a single application can also vary across phases of the program.

Even the objective function can change over time, as the processor changes power conditions (e.g., plugged vs. unplugged, full battery vs. low battery, thermal emergencies), as applications switch (e.g., low priority vs. high priority job), or even within an application (e.g., a real-time application is behind or ahead of schedule).

The experiments in the following sections explore only a subset of these possible changing conditions. Specifically, we examine adaptation to phase changes in single applications. However, by simulating multiple applications and several objective functions, it also indirectly examines the potential to adapt to changing applications and objective functions. We believe a real system would see far greater opportunities to switch cores to adapt to changing execution and environmental conditions than the narrow set of experiments exhibited here.

This work examines a diverse set of execution cores. In a processor where the objective function is static (and perhaps the workload is well known), some of our results indicate that a smaller set of cores (often two) will suffice to achieve very significant gains. However, if the objective function varies over time or workload, a large set of cores has even greater benefit.

IV.B Choice of cores

To provide an effective platform for a wide variety of application execution characteristics and/or system priority functions, the cores on the heterogeneous multi-core processor should cover both a wide and evenly spaced range of the complexity/performance design space.

In this study, we consider a design that takes a series of previously implemented processor cores with slight changes to their interface – this choice reflects one of the key advantages of the CMP architecture, namely the effective amortization of design and verification effort. We include four Alpha cores – EV4 (Alpha 21064), EV5 (Alpha 21164), EV6 (Alpha 21264) and a single-threaded version of the EV8 (Alpha 21464), referred to as EV8-. These cores demonstrate strict gradation in terms of complexity and are capable of sharing a single executable. We assume the four cores have private L1 data and instruction caches and share a common L2 cache, phase-lock loop circuitry, and pins.

We chose the cores of these off-the-shelf processors due to the availability of real power and area data for these processors, except for the EV8 where we use projected numbers [39, 44, 79, 100]. All these processors feature 64-bit architectures. Note that mapping cores from one process generation to another has been shown to be feasible across a few generations [82].

Figure IV.1 shows the relative sizes of the cores used in the study, assuming they are all implemented in a 0.10 micron technology (the methodology to obtain this figure is described in the next section). It can be seen that the resulting processor is only modestly (within 15%) larger than the EV8- core by itself.

For this research, to simplify the initial analysis of this new execution paradigm, we assume only one application runs at a time on only one core. This design point could either represent an environment targeted at a single application

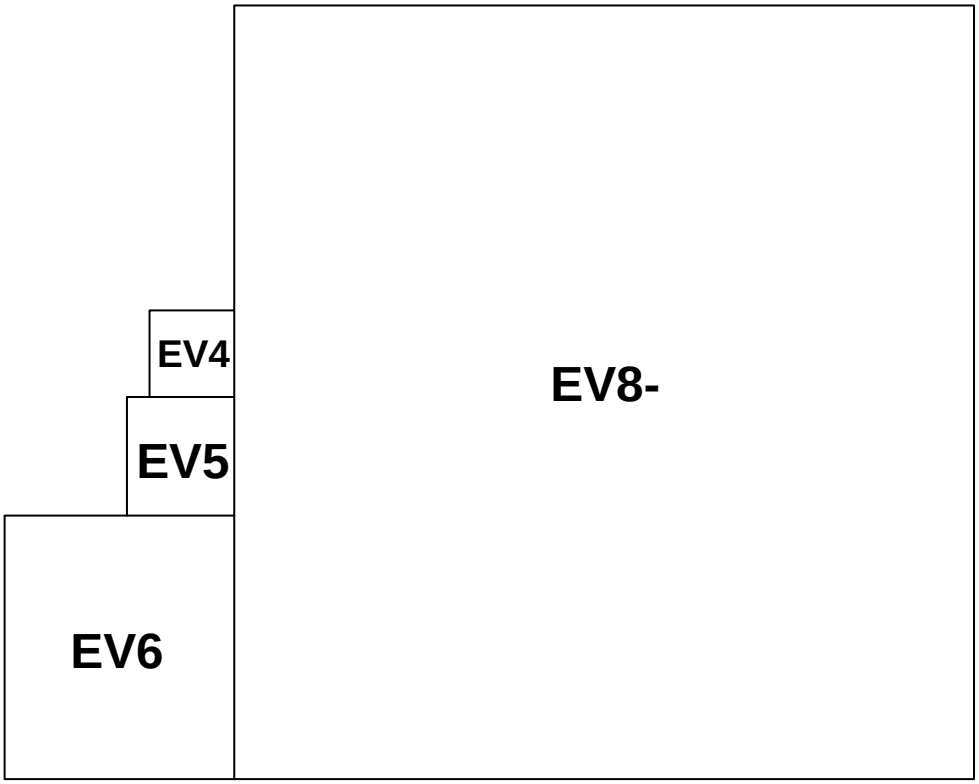

Figure IV.1: Relative sizes of the Alpha cores when implemented in 0.10 micron technology

at a time, or modeling policies that might be employed when a multithreaded multi-core configuration lacks thread parallelism. Because we assume a maximum of one thread running, the multithreaded features of EV8 are not needed. Hence, these are subtracted from the model, as discussed in Section IV.D. In addition, this assumption means that we do not need more than one of any core type. Finally, since only one core is active at a time, we implement cache coherence by ensuring that dirty data is flushed from the current core's L1 data cache before execution is migrated to another core.

This particular choice of architectures also gives a clear ordering in both power dissipation and expected performance. This simplifies the design of core-switching algorithms and is a natural outcome of choosing existing cores from a family. However, in Chapter V, we consider a more unordered set of cores.

IV.C Switching applications between cores

Typical programs go through phases with different execution characteristics [115, 131]. Therefore, the best core during one phase may not be best for the next phase. This observation motivates the ability to dynamically switch cores in mid execution to take full advantage of our heterogeneous architecture.

There is a cost to switching cores, so we must restrict the granularity of switching. One method for doing this would switch only at operating system timeslice intervals, when execution is in the operating system, with user state already saved to memory. If the OS decides a switch is in order, it powers up the new core, triggers a cache flush to save all dirty cache data to the shared L2, and signals the new core to start at a predefined OS entry point. The new core would then power down the old core and return from the timer interrupt handler. The user state saved by the old core would be loaded from memory into the new core at that time, as a normal consequence of returning from the operating system.

Alternatively, we could switch to different cores at the granularity of the entire application, possibly chosen statically. In this study, we consider both these options.

In this work, we assume that unused cores are completely powered down, rather than left idle. Thus, unused cores suffer no static leakage or dynamic switching power. This does, however, introduce a latency for powering a new core up. We estimate that a given processor core can be powered up in approximately one thousand cycles of the 2.1GHz clock. This assumption is based on the observation that when we power down a processor core we do not power down the phase-lock loop that generates the clock for the core. Rather, in our multi-core architecture, the same phase-lock loop generates the clocks for all cores. Consequently, the power-up time of a core is determined by the time required for the power buses to charge and stabilize. In addition, to avoid injecting excessive noise on the power bus bars of the multi-core processor, we assume a staged power up would be used.

In addition, our experiments confirm that switching cores at operating-system timer intervals ensures that the switching overhead has almost no impact on performance, even with the most pessimistic assumptions about power-up time, software overhead, and cache cold start effects. However, these overheads are still modeled in our experiments.

IV.D Evaluation Methodology

This section discusses the various methodological challenges of this research, including modeling the power, the real estate, and the performance of the heterogeneous multi-core architecture.

IV.D.1 Modeling of CPU Cores

The cores we simulate are roughly modeled after cores of EV4 (Alpha 21064), EV5 (Alpha 21164), EV6 (Alpha 21264) and EV8-. EV8- is a hypothetical single-threaded version of EV8 (Alpha 21464). The data on the resources for EV8 was based on predictions made by Joel Emer [44] and Artur Klauser [79], conversations with people from the Alpha design team, and other reported data [39, 100]. The data on the resources of the other cores are based on published literature on these processors [17, 18, 19].

The multi-core processor is assumed to be implemented in a 0.10 micron technology. The cores have private first-level caches, and share an on-chip 3.5 MB 7-way set-associative L2 cache. At 0.10 micron, this cache will occupy an area just under half the die size of the Pentium 4. All the cores are assumed to run at 2.1GHz. This is the frequency at which an EV6 core would run if its 600MHz, 0.35 micron implementation was scaled to a 0.10 micron technology. In the Alpha design, the amount of work per pipe stage was relatively constant across processor generations [28, 40, 44, 52]; therefore, it is reasonable to assume they can all be clocked at the same rate when implemented in the same technology (if not as designed, processors with similar characteristics certainly could). The input voltage for all the cores is assumed to be 1.2V.

Note that while we took care to model real architectures that have been available in the past, we could consider these as just initial sample design points in the continuum of processor designs that could be integrated into a heterogeneous multiple-core architecture. These existing designs already display the diversity of performance and power consumption desired. However, a custom or partially custom design would have much greater flexibility in ensuring that the performance and power space is covered in the most appropriate manner. We discuss custom designs in Chapter V.

Table IV.1: Configuration of the cores used for power evaluation of heterogeneous multi-cores

Processor	EV4	EV5	EV6	EV8-
Issue-width	2	4	6 (OOO)	8 (OOO)
I-Cache	8KB, DM	8KB, DM	64KB, 2-way	64KB, 4-way
D-Cache	8KB, DM	8KB, DM	64KB, 2-way	64KB, 4-way
Branch Pred.	2KB,1-bit	2K-gshare	hybrid 2-level	hybrid 2-level (2X EV6 size)
Number of MSHRs	2	4	8	16

Table IV.D.1 summarizes the configurations that were modeled for various cores. All architectures are modeled as accurately as possible, given the parameters in Table IV.D.1, on a highly detailed instruction-level simulator. However, we did not faithfully model every detail of each actual architecture; we were most concerned with modeling the approximate spaces each core covers in our complexity/performance continuum.

Specific instances of deviations from exact design parameters include the following. Associativity of the EV8- caches is double the associativity of equally-sized EV6 caches. EV8- uses a tournament predictor double the size of the EV6 branch predictor. All the caches are assumed to be non-blocking, but the number of MSHRs is assumed to double with successive cores to adjust to increasing issue width. All the out-of-order cores are assumed to have big enough re-order buffers and large enough load/store queues to ensure no conflicts for these structures.

The various miss penalties and L2 cache access latencies for the simulated cores were determined using CACTI. CACTI [119] provides an integrated model of cache access time, cycle time, area, aspect ratio, and power. To calculate the penalties, we used CACTI to get access times and then added one cycle each for L1 miss detection, going to L2, and coming from L2. For calculating the

L2 access time, we assume that the L2 data and tag access are serialized so that the data memories don't have to be cycled on a miss and only the required set is cycled on a hit. Memory latency was set to be 150ns.

IV.D.2 Modeling Power

Modeling power for this type of study is a challenge. We need to consider cores designed over the time span of more than a decade. Power depends not only on the configuration of a processor, but also on the circuit design style and process parameters. Also, actual power dissipation varies with activity, though the degree of variability again depends on the technology parameters as well as the gating style used.

No existing architecture-level power modeling framework accounts for all of these factors. Current power models like Wattch [30] are primarily meant for activity-based architectural level power analysis and optimizations within a single processor generation, not as a tool to compare the absolute power consumption of widely varied architectures. We integrated Wattch into our architectural simulator and simulated the configuration of various cores implemented in their original technologies to get an estimate of the maximum power consumption of these cores as well as the typical power consumption running various applications. We found that Wattch did not, in general, reproduce published peak and typical power for the variety of processor configurations we are using.

Therefore we use a hybrid power model that uses estimates from Wattch, along with additional scaling and offset factors to calibrate for technology factors. This model not only accounts for activity-based dissipation, but also accounts for the design style and process parameter differences by relying on measured datapoints from the manufacturers.

To solve for the calibration factors, this methodology requires peak and

typical power values for the actual processors and the corresponding values reported by Wattch. This allows us to establish scaling factors that use the output of Wattch to estimate the actual power dissipation within the expected range for each core. To obtain the values for the processor cores, we derive the values from the literature. For the corresponding Wattch values, we estimate peak power for each core given peak activity assumptions for all the hardware structures, and use the simulator to derive typical power consumed for SPEC2000 benchmarks.

This methodology then both reproduces published results and scales reasonably accurately with activity. While this is not a perfect power model, it will be far more accurate than using Wattch alone, or relying simply on reported average power.

Now we detail the methodology for estimating peak power dissipation of the cores. Table IV.D.2 shows our power and area estimates for the cores. We start with the peak power data of the processors obtained from data sheets and conference publications [17, 18, 19, 39, 79]. To derive the peak power dissipation in the core of a processor from the published numbers, the power consumed in the L2 caches and at the output pins of the processor must be subtracted from the published value. Power consumption in the L2 caches under peak load was determined using CACTI, starting by finding the energy consumed per access and dividing by the effective access time. Details on *bitouts*, the extent of pipelining during accesses, etc. were obtained from data sheets (except for EV8-). For the EV8 L2, we assume 32 byte (288 bits including ECC) transfers on reads and writes to the L1 cache. We also assume the L2 cache is doubly pumped.

The power dissipation at the output pins is calculated using the formula: $P = (1/2)CV^2f$.

The values of V (bus voltage), f (effective bus frequency) and C (load capacitance) were obtained from data sheets. Effective bus frequency was calcu-

lated by dividing the peak bandwidth of the data bus by the maximum number of data output pins which are active per cycle. The address bus was assumed to operate at the same effective frequency. For processors like the EV4, the effective frequency of the bus connecting to the off-chip cache is different from the effective frequency of the system bus, so power must be calculated separately for those buses. We assume the probability that a bus line changes state is 0.5. For calculating the power at the output pins of EV8, we used the projected values for V and f. We assumed that half of the pins are input pins. Also, we assume that pin capacitance scales as the square root of the technology scaling factor. Due to reduced resources, we assumed that the EV8- core consumes 80% of the calculated EV8 core-power. This reduction is primarily due to smaller issue queues and register files. The power data was then scaled to the 0.10 micron process. For scaling, we assumed that power dissipation varies directly with frequency, quadratically with input voltage, and is proportional to feature-size.

The second column in Table IV.D.2 summarizes the power consumed by the cores at 0.10 micron technology. As can be seen from the table, the EV8-core consumes almost 20 times the peak power and more than 80 times the real estate of the EV4 core.

CACTI was also used to derive the energy per access of the shared L2 cache, for use in our simulations. We also estimated power dissipation at the output pins of the L2 cache due to L2 misses. For this, we assume 400 output pins. We assume a load capacitance of 50pF and a bus voltage of 2.5V. Again, an activity factor of 0.5 for bit-line transitions is assumed. We also ran some experiments with a detailed model of off-chip memory access power, but found that the level of off-chip activity is highly constant across cores, and did not impact our results.

Values for typical power are more difficult to obtain, so we rely on a

Table IV.2: Power and area statistics of the Alpha cores

Core	**Peak-power** (Watts)	**Core-area** (mm^2)	**Typical-power** (Watts)	**Range** (%)
EV4	4.97	2.87	3.73	92-107
EV5	9.83	5.06	6.88	89-109
EV6	17.80	24.5	10.68	86-113
EV8-	92.88	236	46.44	82-128

variety of techniques and sources to arrive at these values.

Typical power for the EV6 and EV8- assume similar peak to typical ratios as published data for Intel processors of the same generation (the 0.13 micron Pentium 4 [70] for EV8-, and the 0.35 micron late-release Pentium Pro [57, 73] for the EV6).

EV4 and EV5 typical power is extrapolated from these results and available thermal data [17, 18] assuming an approximately linear increase in power variation over time, due to wider issue processors and increased application of clock gating.

These typical values are then scaled in similar ways to the peak values (but using measured typical activity) to derive the power for the cores alone. Table IV.D.2 gives the derived typical power for each of our cores. Also shown, for each core, is the range in power demand for the actual applications we run, expressed as a percentage of typical power.

While our methodology includes several assumptions based on common rules-of-thumb used in typical processor design, we performed several sensitivity experiments with widely different assumptions about the range of power dissipation in the core. Our results show very little difference in the qualitative results

in this research. *For any reasonable assumptions about the range, the power differences between cores still dominates the power difference between applications on the same core.* Furthermore, as noted previously, the cores can be considered as just sample design points in the continuum of processor designs that could be integrated into a heterogeneous multiple-core architecture.

IV.D.3 Estimating Chip Area

Table IV.D.2 also summarizes the area occupied by the cores at 0.10 micron (also shown in Figure IV.1). The area of the cores (except EV8-) is derived from published photos of the dies after subtracting the area occupied by I/O pads, interconnection wires, the bus-interface unit, L2 cache, and control logic. Area of the L2 cache of the multi-core processor is estimated using CACTI.

The die size of EV8 was predicted to be 400 mm^2 [110]. To determine the core size of EV8-, we subtract out the estimated area of the L2 cache (using CACTI). We also account for reduction in the size of register files, instruction queues, reorder buffer, and renaming tables to account for the single-threaded EV8-. For this, we use detailed models of the register bit equivalents (rbe) [104] for register files, reorder buffer and renaming tables at the original and reduced sizes. The sizes of the original and reduced instruction queue sizes were estimated from examination of MIPS R10000 and HP PA-8000 data [32, 84], assuming that the area grows more than linear with respect to the number of entries ($num\_entries^{1.5}$). The area data is then scaled for the 0.10 micron process.

IV.D.4 Modeling Performance

In this study, we simulate the execution of 14 benchmarks from the SPEC2000 benchmark suite, including 7 from SPECint and 7 from SPECfp. These are listed in Table IV.D.4.

Table IV.3: Benchmarks simulated for power evaluation of heterogeneous multi-cores

Program	Description
ammp	Computational Chemistry
applu	Parabolic/Elliptic Partial Differential Equations
apsi	Meteorology:Pollutant Distribution
art	Image Recognition/Neural Networks
bzip2	Compression
crafty	Game Playing:Chess
eon	Computer Visualization
equake	Seismic Wave Propagation Simulation
fma3d	Finite-element Crash Simulation
gzip	Compression
mcf	Combinatorial Optimization
twolf	Place and Route Simulator
vortex	Object-oriented Database
wupwise	Physics/Quantum Chromodynamics

Benchmarks are simulated using SMTSIM [127]. The simulator was modified to simulate a multi-core processor comprising four heterogeneous cores sharing an on-chip L2 cache and the memory subsystem. In all simulations in this research we assume a single thread of execution running on one core at a time. Switching execution between cores involves flushing the pipeline of the "active" core and writing back all its dirty L1 cache lines to the L2 cache. The next instruction is then fetched into the pipeline of the new core. The execution time and energy of this overhead, as well as the startup effects on the new core, are accounted for in our simulations of the dynamic switching heuristics in Section IV.E. The simpoint tool [116] is used to determine the number of committed instructions which need to be fast-forwarded so as to capture the representative program behavior during simulation. After fast-forwarding, we simulate 1 billion instructions. All benchmarks are simulated using *ref* inputs.

IV.E Scheduling for Power: Analysis and Results

This section examines the effectiveness of single-ISA heterogeneous multi-core designs in reducing the power dissipation of processors. We first examine the relative energy efficiency across cores, and how it varies by application and phase. Later sections use this variance, demonstrating both oracle and realistic core switching heuristics to maximize particular objective functions.

IV.E.1 Variation in Core Performance and Power

As discussed in Section IV.A, this work assumes that the performance ratios between our processor cores is not constant, but varies across benchmarks, as well as over time on a single benchmark. This section verifies that premise.

Figure IV.2(a) shows the performance measured in million instructions committed per second (IPS) of one representative benchmark, *applu*. In the

figure, a separate curve is shown for each of the five cores, with each data point representing the IPS over the preceding 1 million committed instructions.

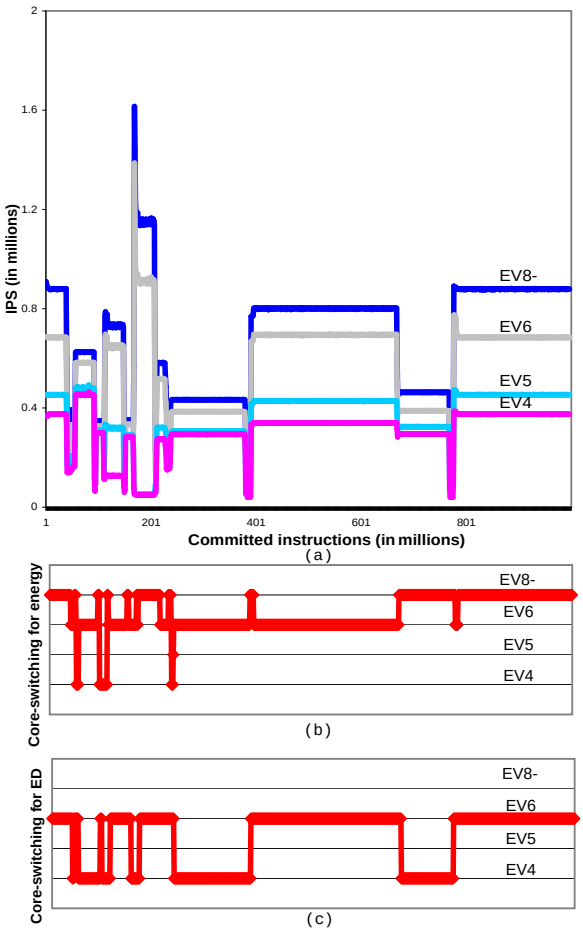

Figure IV.2: (a) Performance of *applu* on the four cores (b) Oracle switching for energy (c) Oracle switching for energy-delay product.

With *applu*, there are very clear and distinct phases of performance on each core, and the relative performance of the cores varies significantly between these phases. Nearly all programs show clear phased behavior, although the frequency and variety of phases varies significantly.

If relative performance of the cores varies over time, it follows that en-

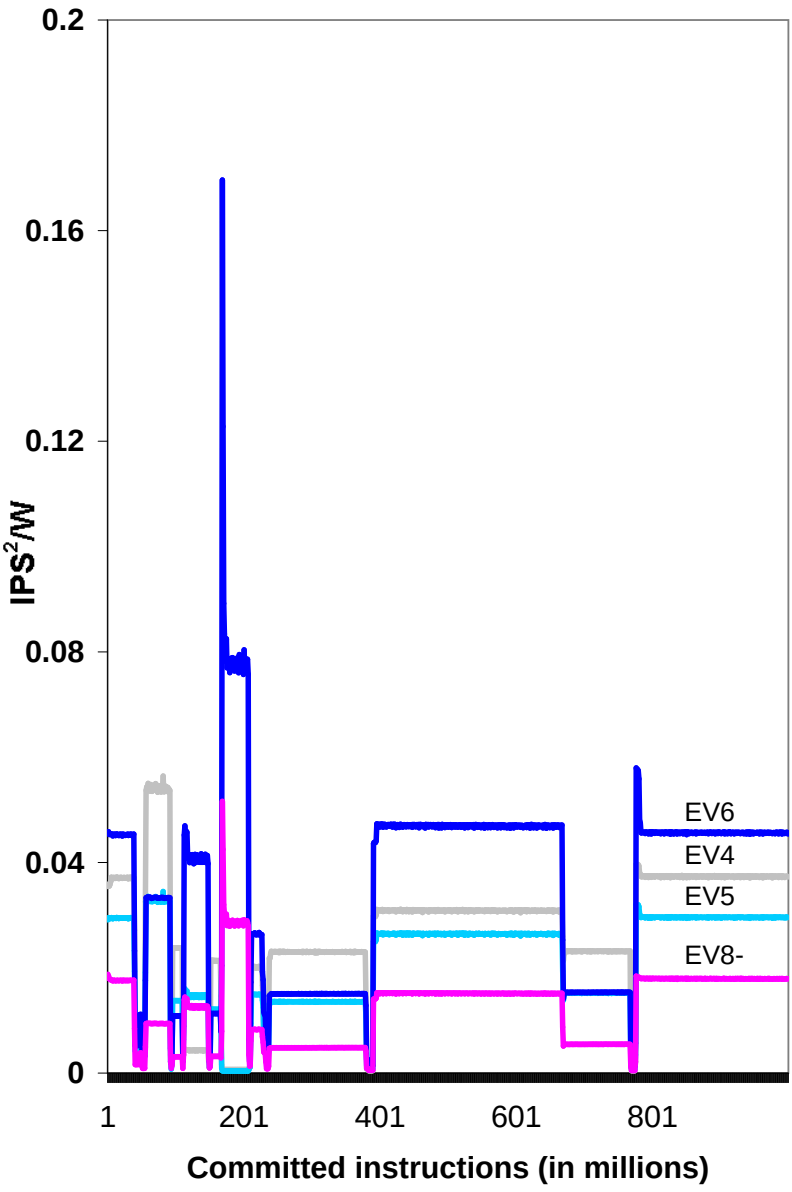

Figure IV.3: *applu* energy efficiency. IPS^2/W varies inversely with energy-delay product

ergy efficiency will also vary. Figure IV.3 shows one metric of energy efficiency (defined in this case as $IPS^2/Watt$) of the various cores for the same benchmark. $IPS^2/Watt$ is merely the inverse of Energy-Delay product. Energy-Delay product [53], as the name suggests, is the product of average energy taken up by a instruction during its execution and the time of execution. The lower the Energy-Delay product, the more efficient a core is. As can be seen, the relative value of the energy-delay product among cores, and even the ordering of the cores, varies from phase to phase.

IV.E.2 Oracle Heuristics for Dynamic Core Selection

This section examines the limits of power and efficiency improvements possible with a heterogeneous multi-core architecture. The ideal core-selection algorithm depends heavily on the particular goals of the architecture or application. This section demonstrates oracle algorithms that maximize two sample objective functions. The first optimizes for energy efficiency with a tight performance threshold. The second optimizes for energy-delay product with a looser performance constraint.

These algorithms assume perfect knowledge of the performance and power characteristics at the granularity of intervals of one million instructions (corresponding roughly to an OS time-slice interval). It should be noted that choosing the core that minimizes energy or the energy-delay product over each interval subject to performance constraints does not give an optimal solution for the global energy or energy-delay product; however, the algorithms do produce good results.

The first oracle that we study seeks to minimize the energy per committed instruction (and thus, the energy used by the entire program). For each interval, the oracle chooses the core that has the lowest energy consumption,

given the constraint that performance has always to be maintained within 10% of the EV8- core for each interval. This constraint assumes that we are willing to give up performance to save energy but only up to a point. Figure IV.2(b) shows the core selected in each interval for *applu*.

For *applu*, we observe that the oracle chooses to switch to EV6 in several phases even though EV8- performs better. This is because EV6 is the less power-consuming core and still performs within the threshold. The oracle even switches to EV4 and EV5 in a small number of phases. Table IV.E.2 shows the results for all benchmarks. In this, and all following results, performance degradation and energy savings are always given relative to EV8- performance. As can be seen, this heuristic achieves an average energy reduction of 38% (see column 8) with less than 4% average performance degradation (column 9). Five benchmarks (*ammp, fma3d, mcf, twolf, crafty*) achieve no gain because switching was denied by the performance constraint. Excluding these benchmarks, the heuristic achieves an average energy reduction of 60% with about 5% performance degradation.

Our second oracle utilizes the *energy-delay product* metric. The energy-delay product seeks to characterize the importance of both energy and response time in a single metric, under the assumption that they have equal importance. Our oracle minimizes energy-delay product by always selecting the core that maximizes $IPS^2/Watt$ over an interval. We again impose a performance threshold, but relax it due to the fact that energy-delay product already accounts for performance degradation. In this case, we require that each interval maintains performance within 50% of EV8-.

Figure IV.2(c) shows the cores chosen for *applu*. Table IV.E.2 shows the results for all benchmarks. As can be seen, the average reduction in energy-delay is about 63%; the average energy reductions are 73% and the average performance degradation is 22%. All but one of the fourteen benchmarks have fairly significant

(47% to 78%) reductions in energy-delay savings. The corresponding reductions in performance ranges from 4% to 45%. As before, switching activity and the usage of the cores varies. This time, EV8 never gets used. EV6 emerges as the dominant core. Given our relaxed performance constraint, there is a greater usage of the lower-power cores compared to the previous experiment.

Both Tables IV.E.2 and IV.E.2 also show results for Energy-Delay^2 [135] improvements. Improvements are 35-50% on average. This is instructive because chip-wide voltage/frequency scaling can do no better than break even on this metric, demonstrating that this approach has the potential to go well beyond the capabilities of that technique. In other experiments specifically targeting the ED^2 metric (again with the 50% performance threshold), we saw 53.3% reduction in energy-delay^2 with 14.8% degradation in performance.

IV.E.3 Static Core Selection

This section examines the necessity of dynamic switching between cores by measuring the effectiveness of an oracle-based static assignment of benchmark to core (for just one of our sample objective functions). This models a system that accurately selects a single core to run for the duration of execution, perhaps based on compiler analysis, profiling, past history, or simple sampling.

Table IV.6 summarizes the results when a static oracle selects the best core for energy. As in the earlier dynamic results, a performance threshold (this time over the duration of the benchmark) is applied. EV6 is the only core other than EV8 which gets used. This is because of the stringent performance constraint. Average energy savings is 32%. Excluding the benchmarks which remain on EV8, average energy savings is 74.3%. Average performance degradation is 2.6%. This low performance loss leads to particularly high savings for both energy-delay and energy-delay^2. Average energy savings is 31% and average

Table IV.4: Summary for dynamic *oracle* switching for energy on heterogeneous multi-cores

Benchmark	Total switches	% of instructions per core				Energy Savings(%)	ED Savings(%)	ED^2 Savings(%)	Perf. Loss (%)
		EV4	EV5	EV6	EV8-				
ammp	0	0	0	0	100	0	0	0	0
applu	27	2.2	0.1	54.5	43.2	42.7	38.6	33.6	7.1
apsi	2	0	0	62.2	37.8	27.6	25.3	22.9	3.1
art	0	0	0	100	0	74.4	73.5	72.6	3.3
equake	20	0	0	97.9	2.1	72.4	71.3	70.1	3.9
fma3d	0	0	0	0	100	0	0	0	0
wupwise	16	0	0	99	1	72.6	69.9	66.2	10.0
bzip	13	0	0.1	84.0	15.9	40.1	38.7	37.2	2.3
crafty	0	0	0	0	100	0	0	0	0
eon	0	0	0	100	0	77.3	76.3	75.3	4.2
gzip	82	0	0	95.9	4.1	74.0	73.0	71.8	3.9
mcf	0	0	0	0	100	0	0	0	0
twolf	0	0	0	0	100	0	0	0	0
vortex	364	0	0	73.8	26.2	56.2	51.9	46.2	9.8
Average	1(median)	0.2%	0%	54.8%	45.0%	38.5%	37.0%	35.4%	3.4%

Table IV.5: Summary for dynamic *oracle* switching for energy-delay on heterogeneous multi-cores

Benchmark	Total switches	% of instructions per core				Energy-delay Savings(%)	Energy Savings(%)	Energy-delay^2 Savings(%)	Perf. Loss (%)
		EV4	EV5	EV6	EV8-				
ammp	0	0	0	100	0	63.7	70.3	55.7	18.1
applu	12	32.3	0	67.7	0	69.8	77.1	59.9	24.4
apsi	0	0	0	100	0	60.1	69.1	48.7	22.4
art	619	65.4	0	34.5	0	78.0	84.0	69.6	27.4
equake	73	55.8	0	44.2	0	72.3	81.0	59.2	31.7
fma3d	0	0	0	100	0	63.2	73.6	48.9	28.1
wupwise	0	0	0	100	0	68.8	73.2	66.9	10.0
bzip	18	0	1.2	98.8	0	60.5	70.3	47.5	24.8
crafty	0	0	0	100	0	55.4	69.9	33.9	32.5
eon	0	0	0	100	0	76.2	77.3	75.3	4.2
gzip	0	0	0	100	0	74.6	75.7	73.5	4.2
mcf	0	0	0	100	0	46.9	62.8	37.2	24.3
twolf	0	0	0	100	0	26.4	59.7	-34.2	45.2
vortex	0	0	0	100	0	68.7	73.0	66.7	9.9
Average	0(median)	11.0%	0.1%	88.9%	0%	63.2%	72.6%	50.6%	22.0%

Table IV.6: Oracle heuristic for static core selection on heterogeneous multi-cores – energy metric. Rightmost two columns are for dynamic selection

Benchmark	Core	Static Selection				Dynamic Selection	
		Energy savings (%)	Energy-delay savings (%)	Energy-delay^2 savings (%)	Perf loss (%)	Energy savings(%)	Perf. loss (%)
ammp	EV8-	None	None	None	None	36.1	10.0
applu	EV8-	None	None	None	None	49.9	10.0
apsi	EV8-	None	None	None	None	42.9	10.0
art	EV6	74.4	73.5	72.6	3.3	75.7	10.0
equake	EV6	73.4	72.3	70.8	4.5	74.4	10.0
fma3d	EV8-	None	None	None	None	28.1	10.0
wupwise	EV6	73.2	70.5	66.9	10.0	49.5	10.0
bzip	EV8-	None	None	None	None	47.7	10.0
crafty	EV8-	None	None	None	None	17.6	10.0
eon	EV6	77.3	76.3	75.3	4.2	77.3	9.8
gzip	EV6	75.7	74.6	73.5	4.3	76.0	10.0
mcf	EV8-	None	None	None	None	19.9	10.0
twolf	EV8-	None	None	None	None	8.1	10.0
vortex	EV6	73.0	70.3	66.7	9.9	52.0	10.0
Average	-	31.9%	31.3%	30.4%	2.6%	46.8%	10.0

energy-delay^2 savings is 30%.

Also shown is a corresponding dynamic technique. For a fair comparison, we apply a global runtime performance constraint rather than a per-interval constraint. That is, any core can be chosen in an interval as long as the accumulated runtime up to this point (including all core choices made on earlier intervals) remains within 10% of the EV8- alone. This gives a more fair comparison with the static technique.

IV.E.4 Realistic Dynamic Switching Heuristics

This section examines the extent to which the energy benefits in the earlier sections can be achieved with a real system implementation that does not depend on oracular future knowledge. We do, however, assume an ability to track both the accumulated performance and energy over a past interval. This functionality either already exists or is easy to implement. This section is intended to be an existence proof of effective core selection algorithms, rather

than a complete evaluation of the switching design space. We only demonstrate a few simple heuristics for selecting the core to run on. The heuristics seek to minimize overall energy-delay product during program execution.

Our previous oracle results were idealized not only with respect to switching algorithms, but also ignored the cost of switching (power-up time, flushing dirty pages to the L2 cache and experiencing cold-start misses in the new L1 cache and TLB) both in performance and power. The simulations in this section account for both, although our switching intervals are long enough and switchings infrequent enough that the impact of both effects is under 1%.

In this section, we measure the effectiveness of several heuristics for selecting a core. The common elements of each of the heuristics are these: every 100 time intervals (one time interval consists of 1 million instructions in these experiments), one or more cores are sampled for five intervals each (with the results during the first interval ignored to avoid cold start effects). Based on measurements done during sampling, the heuristic selects one core. For the case when one other core is sampled, the switching overhead is incurred once if the new core is selected, or twice if the old core is chosen. The switching overhead is greater if more cores are sampled. The dynamic heuristics studied here are:

- **neighbor**. One of the two neighboring cores in the performance continuum is randomly selected for sampling. A switch is made if that core has lower energy-delay over the sample interval than the current core over the last run interval.

- **neighbor-global**. Similar to neighbor, except that the selected core is the one that would be expected to produce the lowest accumulated energy-delay product to this point in the application's execution. In some cases this is different than the core that minimizes the energy-delay product for this interval.

- **random**. One other randomly-chosen core is sampled, and a switch is made if that core has lower energy-delay over the sample interval.
- **all** All other cores are sampled.

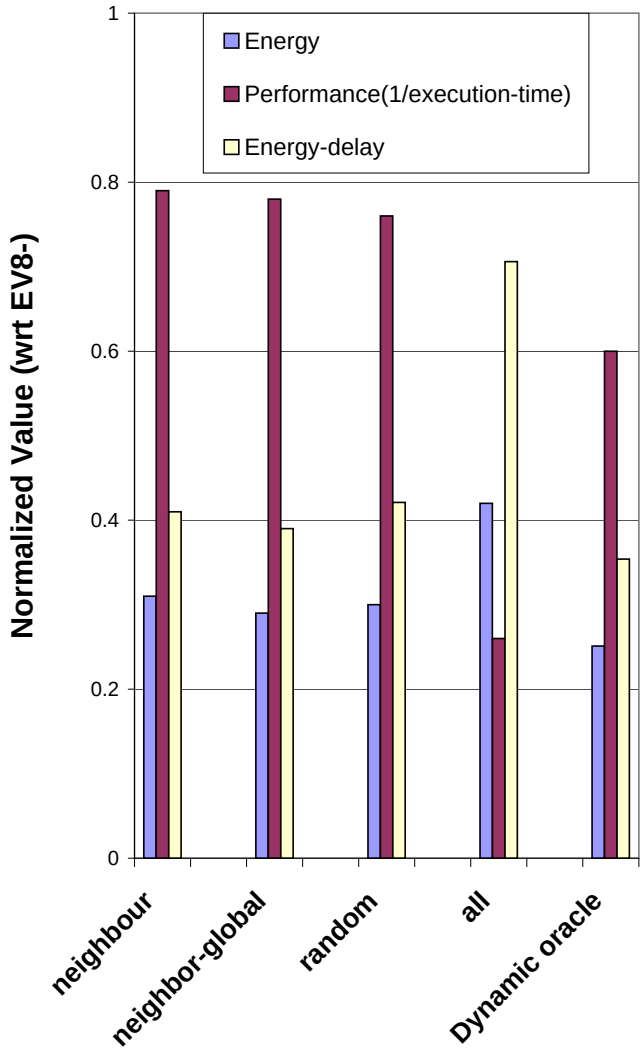

Figure IV.4: Results for realistic switching heuristics for heterogeneous multi-cores - the last one is a constraint-less dynamic oracle

The results are shown in Figure IV.4. The results are all normalized to EV8- values. This figure also includes oracle results for dynamic switching

based on the energy-delay metric when core selection is not hampered with performance constraints. Lower bars for energy and energy-delay, and higher bars for performance are desirable.

Our heuristics achieve up to 93% of the energy-delay gains achieved by the oracle-based switcher, despite modeling the switching overhead, sampling overhead, and non-oracle selection. The performance degradation on applying our dynamic heuristics is, on average, *less* than the degradation found by the oracle-based scheme. Also, although not shown in the figure, there is a greater variety in core-usage between applications.

It should be noted that switching for this particular objective function is not heavy; thus, heuristics that find the best core quickly, and minimize sampling overhead after that, tend to work best. The best heuristic for a different objective function, or a dynamically varying objective function may be different. These results do show, however, that for a given objective function, very effective realtime and predictive core switching heuristics can be found.

IV.E.5 Practical heterogeneous architectures

Although our use of existing cores limits design and verification overheads, these overheads do scale with the number of distinct cores supported. Some or our results indicate that in specific instances, two cores can introduce sufficient heterogeneity to produce significant gains. For example the (minimize energy, maintain performance within 10%) objective function relied heavily on the EV8- and the EV6 cores. The (energy-delay, performance within 50%) objective function favored the EV6 and EV4. However, if the objective function is allowed to vary over time, or if the workload is more diverse than what we model, wider heterogeneity than 2 cores will be useful. Presumably, other objective functions than those we model may also use more than 2 cores.

IV.F Summary

We introduce and seek to gain some insights into the energy benefits available for single-ISA heterogeneous multi-core architectures. The particular opportunity examined is a single application switching among cores to optimize some function of energy and performance.

We show that a sample heterogeneous multi-core design with four complexity-graded cores has the potential to increase energy efficiency (defined as energy-delay product, in this case) by a factor of three, in one experiment, without dramatic losses in performance. Energy efficiency improvements significantly outdistance chip-wide voltage/frequency scaling. It is shown that most of these gains are possible even by using as few as two cores.

These results indicate that not only is there significant potential for this style of architecture, but that reasonable runtime heuristics for switching cores, using limited runtime information, can achieve most of that potential.

IV.G Acknowledgment

The text of Chapter IV is in part a reprint of the material as it appears in the proceedings of the Thirty-sixth International Symposium on Microarchitecture (pp81-92, December 2003). The dissertation author was the primary researcher and author and the co-authors involved in the submission directed the research which forms the basis for Chapter IV.

V

Holistic Design for Adaptability: Designing Heterogeneous Multi-cores From the Ground Up

This chapter first provides an overview of other proposals related to heterogeneous multi-cores and then discusses a holistic design methodology for heterogeneous multi-core architectures.

V.A Overview of Related Proposals

There have been other proposals studying the advantages of on-chip heterogeneity. This chapter provides an overview of other work directly dealing with heterogeneous multi-core architectures. We also discuss previous work with similar goals – i.e., adapting to workload diversity to improve processor efficiency.

Morad, *et al.* [102] explore the theoretical advantages of placing asymmetric core clusters in multiprocessor chips. The asymmetric processor executed the serial phases of a parallel program on large high-performance cores, while the parallel phases get executed on the simpler, smaller cores. They show that

asymmetric core clusters are expected to achieve higher performance per area and higher performance for a given power envelope. The analysis is extended in [103] and experimental results presented through emulation of such processors.

Annavaram, *et al.* [23] evaluate the benefits of heterogeneous multiprocessing to minimize the execution times of multi-threaded programs containing nontrivial parallel and sequential phases, while keeping the CMP's total power consumption within a fixed budget. Whenever a parallel program is in its serial phase, it is executed on a core running at high frequency. During the parallel phases, threads are executed on cores after reducing their frequency to meet the power budget. They report significant speedups over a a chip multiprocessor where all the cores are on during the entire execution of the program and their frequency is set constant to meet the power budget.

Balakrishanan, *et al.* [25] seek to understand the impact of such an architecture on software. They show, using a hardware prototype, that asymmetry can have significant impact on the performance, stability, and scalability of a wide range of commercial applications. They also demonstrate that in addition to heterogeneity-aware kernels, several commercial applications may themselves need to be aware of heterogeneity at the hardware level.

The energy benefits of heterogeneous multi-core architectures is also explored by Ghiasi and Grunwald [50]. They consider single-ISA, heterogeneous cores of different frequencies belonging to the x86 family for controlling the thermal characteristics of a system. Applications run simultaneously on multiple cores and the operating system monitors and directs applications to the appropriate job queues. They report significantly better thermal and power characteristics for heterogeneous processors.

Grochowsky, *et al.* [55] compare voltage/frequency scaling, asymmetric (heterogeneous) cores, variable-sized cores, and speculation as means to re-

duce the energy per instruction (EPI) during the execution of a program. They find that the EPI range for asymmetric chip-multiprocessors using x86 cores was 4-6X, significantly more than the next best technique (which incidentally was voltage/frequency scaling).

There have also been proposals for multi-ISA heterogeneous multi-core architectures. The proposed Tarantula processor [45] is one such example of integrated heterogeneity. It consists of a large vector unit unit sharing the die with an EV8 core. The Alpha ISA is extended to include vector instructions that operate on the new architectural state. The unit is targeted towards applications with high data-level parallelism. IBM Cell [75] is another example of a heterogeneous chip multiprocessor with cores belonging to different ISA (more details on cell in Section II.B).

Having custom ISAs for different kinds of workloads can result in potentially higher benefits as compared to single-ISA heterogeneous multi-core architectures, especially when an "exact" workload-to-core mapping is possible. However, the downside of this approach is that little design reuse between the cores and also a poor resource utilization when the application mix contains a balance different than that ideally suited to the underlying heterogeneous architecture. Single-ISA heterogeneous multi-core architectures do not have the same disadvantages because of the use of off-the-shelf cores which can all execute the same code, though with different performance. Having the same ISA also enables them to adjust to dynamic changes in program behavior or workload mix. Also, compilation effort would arguably be larger for multi-ISA chips.

The main objective of heterogeneous multi-core architectures is to adapt to workload diversity. An alternative way of handling different amounts of parallelism is by configuring the processor to the demands of the workload. There have been several papers on processor reconfiguration [22, 46, 49, 56, 71,

97, 99, 54, 108, 72, 114] for minimizing power. Gating-based power optimizations [22, 46, 49, 56, 71, 97, 99] provide the option to turn off (gate) portions of the processor core that are not useful for the currently executing phases of the workload. Similarly, voltage and frequency scaling reduces the parameters of the entire core [54, 108] or portions of the core [72, 114] to adapt to reduced workload demands.

However, for all these techniques, benefits are limited by the granularity of structures that can be gated or scaled down, and the inability to change the overall size and complexity of the processor. Also, these designs are still susceptible to static leakage inefficiencies. Furthermore, voltage and frequency scaling is fundamentally limited by the process technology in which the processor is built. Heterogeneous multi-core designs address both these deficiencies.

Single-ISA heterogeneous multi-core architecture also enhance throughout by addressing both low-TLP and high-TLP using the same architecture. There have been other architectures proposed for the same purpose. Speculative multithreading architectures [98, 122, 121, 134, 35, 60, 129, 59] which employ additional contexts of a multi-context processor(chip multiprocessors or simultaneously multithreaded processors) to enhance single-thread performance fall into this category. When TLP is high, all the contexts are used to run different threads; when TLP is low, the spare contexts are used to run speculative/helper threads to enhance the single-thread performance of the main thread(s). Single-ISA heterogeneous architectures differ from the speculative multithreading architectures in that when TLP is low, single-thread performance is enhanced by running the application(s) on more powerful processors and hence ensuring better performance instead of relying on the goodness of speculative/helper threads to accelerate the main thread. However, we believe that speculative multithreading is a complimentary approach and one could envision heterogeneous speculative

multithreading architectures where the speculative threads run on the slower processors while the primary threads run on the more powerful ones. Little work has been done previously on reconfiguration for multi-purpose processors for maximizing throughput. The TRIPS [112] architecture contains mechanisms that enable the processing cores and the on-chip memory system to be configured and combined in different modes for different granularities and types of parallelism. To adapt to small and large-grain concurrency, the architecture contains some number out-of-order, wide-issue Grid Processor cores, which can be partitioned when easily extractable fine-grained parallelism exists. Our work is different in that we primarily target commodity heterogeneity with no special demands on the compiler or the programmer. Also, we exploit not only inter-thread diversity by matching each thread to a right core but also intra-thread diversity by dynamically switching the applications depending on the current job-execution profile. The potential and overheads for exploiting intra-thread diversity are unclear for TRIPS architectures.

Overall, having heterogeneous processor cores provides potentially greater power savings compared to previous approaches and greater flexibility and scalability of architecture design. Moreover, these previous approaches can still be used in a multi-core processor to greater advantage.

V.B Benefits of Ground-up Design

While the previous proposals demonstrated the benefits of heterogeneity, they gave no insight into what constitutes, or how to arrive at, a good heterogeneous design. Previous work (including Sections III.B and IV of this thesis) assumed a given heterogeneous architecture. More specifically, those architectures were composed of existing architectures, either different generations of the same processor family [90, 87, 50, 55], or voltage and frequency scaled editions

of a single processor [23, 25, 51, 81]. While these architectures surpassed similar homogeneous designs, they failed to reach the full potential of heterogeneity, for three reasons. First, the use of pre-existing designs presents low flexibility in choice of cores. Second, core choices maintain a monotonic relationship, both in design and performance – for example, the most powerful core is bigger or more complex in every dimension and the performance-ordering of the cores is the same for every application. Third, all cores considered perform well for a wide variety of applications — we show that the best heterogeneous designs are composed of specialized core architectures.

A heterogeneous architecture, and particularly a fully custom heterogeneous processor not necessarily composed of pre-existing cores, incurs additional costs in design, verification, and testing. A key goal of the research presented in this chapter is to evaluate the full benefits of these architectures, so that this tradeoff can be more appropriately evaluated by processor manufacturers.

In actually deriving the best designs for a variety of multiprogramming workloads, power and area constraints, level of threading, etc., this chapter makes three significant contributions. First, it re-evaluates the benefits of heterogeneity in power and area efficient architectures, showing new benefits and higher gains. Performance improvements of up to 40% are shown. Second, it demonstrates methodologies for arriving at good heterogeneous designs – we examine both those that find the best designs but do not scale well to larger design spaces, and those that scale yet still find good architectures. Third, by actually finding the best designs across many different assumptions and constraints, it identifies a number of key principles critical to the effective design of future chip multiprocessors.

V.C From Workloads to Multi-core Design

The goal of this research is to identify the characteristics of cores that combine to form the best heterogeneous architectures, and also demonstrate principles for designing such an architecture. Such a methodology would start with a set of applications and a set of constraints on the processor. It should then identify the best architecture for that workload, given some objective function to evaluate the goodness of an architecture.

Because this methodology requires that we accurately reflect the wide diversity of applications (their parallelism, their memory behavior), running on widely varying architectural parameters, there is no real shortcut to using simulation to characterize these combinations.

The design space for even a single processor is large, given the flexibility to change various architectural parameters; however, the design space explodes when considering the combined performance of multiple different cores on arbitrary permutations of the applications. Hence, we make some simplifying assumptions that make this problem tractable so that we can navigate through the search space faster; however, we show that the resulting methodology still results in the discovery of very effective multi-core design points.

First, we assume that the performance of individual cores is separable – that is, that the performance of a four-core design, running four applications, is the sum (or the sum divided by a constant factor) of the individual cores running those applications in isolation. This is an accurate assumption if the cores do not share L2 caches or memory controllers (which we validate in Section V.G). However, we also show in Section V.G that this methodology still makes good design decisions with shared L2 caches for our workloads. This assumption dramatically accelerates the search because now the single-thread performance of each core (found using simulation) can be used to estimate the performance of

the processor as a whole without the need to simulate all 4-thread permutations.

Since we are interested in the highest performance that a processor can offer, we assume good static scheduling of threads to cores. Thus, the performance of four particular threads on four particular cores is the performance of the best static mapping. However, this actually represents a *lower bound* on performance. Section III.B has shown that the ability to migrate threads dynamically during execution only increases the benefits of heterogeneity as it exploits intra-thread diversity – we show in Section V.F that it continues to hold true for the best heterogeneous designs that we come up with under the static scheduling assumption.

To further accelerate the search, we consider only major blocks to be configurable, and only consider discrete points. For example, we consider 2 instruction queue sizes (rather than all the intermediate values) and 4 cache configurations (per cache). But we consider only a single branch predictor, because the area/performance tradeoffs of different sizes had little effect in our experiments. Values that are expected to be correlated (e.g., size of re-order buffer and number of physical registers) are scaled together instead of separately. This methodology might appear to be crude for an important commercial design, but we believe that even in that environment this methodology would find a design very much in the neighborhood of the best design. Then, a more careful analysis could be done of the immediate neighborhood, considering structure sizes at a finer granularity and considering particular choices for smaller blocks we did not vary.

We only consider and compare processors with a fixed number (4) of cores. It would be interesting to also relax that constraint in our designs, but we did not do so for the following reasons. Accurate comparisons would be more difficult, because the interconnect and cache costs would vary. Second,

it is shown both in this work (Section V.F) and in previous work (III.B) that heterogeneous designs are much more tolerant than homogeneous when running a different number of threads than the processor is optimized for. However, the methodology shown here need only be applied multiple times (once for each possible core count) to fully explore the larger design space, assuming that an accurate model of the off-core resources was available.

The above assumptions allow us to model performance for various combinations of cores for various permutations of our benchmarks, thus evaluating the expected performance of the possible homogeneous and heterogeneous processors for various area and power budgets.

To search through the design space for a given set of workloads we follow two techniques – exhaustive search and efficient search. Our algorithm for finding the best design typically is an exhaustive search of all core combinations, accounting for every permutation of our benchmarks on each combination. This approach ensures that we do indeed find the best combination in each case. While this approach works for our workloads and architectural variables, considering more benchmarks and more architectural options will quickly make the exhaustive approach impractical. In Section V.F, we examine more efficient search algorithms and quantify how closely they come to identifying the best design.

V.D Customizing Cores to Workloads

One of the biggest advantages of creating a heterogeneous processor from the ground up is that the cores can be chosen in an unconstrained manner as long as the processor budgetary constraints are satisfied. We define *monotonicity* to be a property of a multi-core architecture where there is a total ordering among the cores in terms of performance and this ordering remains the same for all applications. For example, a multiprocessor consisting of EV5 and EV6 cores

is a monotonic multiprocessor. This is because EV6 is strictly superior to EV5 in terms of hardware resources and virtually always performs better than EV5 for a given application given the same cycle time and latencies. Similarly, for a multi-core architecture with identical cores, if the voltage/frequency of a core is set lower than the voltage/frequency of some other core, it will always provide less performance, regardless of application. Fully customized monotonic designs represent the upper bound (albeit a high one) on the benefits possible through previously proposed heterogeneous architectures.

As we show in this chapter, monotonic multiprocessors may not provide the "best fit" for various workloads and hence result in inefficient mapping of applications to cores. For example, in the results shown in [87], *mcf*, despite having very low ILP, consistently gets mapped to the EV6 or EV8- core for various energy-related objective functions, because of the larger caches on these cores. Yet it fails to take advantage of the complex execution capabilities of these cores, and thus still wastes energy unnecessarily.

Doing a custom design of a heterogeneous multi-core architecture allows the monotonicity constraint to be relaxed. That is, it is possible for a particular core of the multiprocessor to be the highest performing core for some application but not for others. For example, if one core is in-order, scalar, with 8KB caches, and another core is out-of-order, dual-issue, with 16KB caches, applications will always run best on the latter. However, if the scalar core had 64KB L1 caches, then it might perform better for applications with low ILP and large working sets, while the other would likely be best for jobs with high ILP and smaller working sets.

The advantage of non-monotonicity is that now different cores on the same die can be customized to different classes of applications, which was not the case with previously studied designs.

V.E Methodology

This section discusses the various methodological challenges of this study, including modeling power, real estate, and performance of the heterogeneous multi-core architectures.

V.E.1 Modeling of CPU Cores

For all our studies in this chapter, we model 4-core multiprocessors assumed to be implemented in 0.10 micron, 1.2V technology. Each core on a multiprocessor, either homogeneous or heterogeneous, has a private L2 cache and each L2 bank has a corresponding memory controller. The ITRS roadmap [15] confirms that sufficient pins are available to support four memory controllers for the assumed technology. Assuming private L2 caches reduces the dimensions of the design; however, we also consider a shared L2 cache (of the same total size) in Section V.G.

We consider both in-order cores and out-of-order cores for this study. We base our OOO processor microarchitecture model on the MIPS R10000, and our in-order cores on the Alpha EV5 (21164). We evaluate 480 cores as possible building blocks for constructing the multiprocessors. This represents all possible distinct cores that can be constructed by changing the parameters listed in Table V.E.1. The various values that were considered are listed in the table as well. We assumed a gshare branch-predictor with 8k entries for all the cores. Out of these 480 cores, there are 96 distinct in-order cores and 384 distinct out-of-order cores. The number of distinct 4-core multiprocessors that can be constructed out of 480 distinct cores is over 2.2 billion.

Other parameters that are kept fixed for all the cores are also listed in Table V.E.1. The various miss penalties and L2 cache access latencies for the simulated cores were determined using CACTI [119].

Table V.1: Various Parameters and their possible values for configuration of the cores

Issue-width	1, 2, 4
I-Cache	8KB-DM, 16KB-2way, 32KB-4way, 64KB-4way
D-Cache	8KB-DM, 16KB-2way, 32KB-4way, 64KB-4way dual ported
FP-IntMul-ALU units.	1-1-2, 2-2-4
IntQ-fpQ (OOO)	32-16, 64-32
Int-FP PhysReg-ROB (OOO)	64-64-32, 128-128-64
L2 Cache	1MB/core, 4-way, 12cycle access
Memory Channel	533MHz, doubly-pumped, RDRAM
ITLB-DTLB	64, 28 entries
Ld/St Queue	32entries

All evaluations are done for multiprocessors satisfying a given aggregate area and power budget for the 4 cores. We do not expect the memory and interconnection subsystem to vary significantly with the core type for a given number of cores. We also confirmed that the contribution of L2s to overall power consumption did not vary significantly between four-core designs taking up the same area, even when the total number of serviced memory requests differed. Hence, we do not concern ourselves with the area and power consumption of anything other than the cores for this study.

V.E.2 Modeling Power and Area

In this chapter, the area budget refers to the sum of the area of the 4 cores of a processor (the L1 cache being part of the core), and the power budget refers to the sum of the worst case power of the cores of a processor. Specifically, we consider peak activity (dynamic) power, as this is a critical constraint in the architecture and design phase of a processor. Static power is not considered explicitly in this chapter (though it is typically proportional to area, which we do consider).

We model the peak activity power and area consumption of each of the key structures in a processor core using a variety of techniques. Table V.E.2 lists the methodology and assumptions used for estimating area and power overheads for various structures. Table V.E.2 shows the area and power values for various parameterized hardware structures. Notice that some of the structures listed are for out-of-order (OOO) cores only.

Table V.2: Area and power estimation methodology and relevant assumptions for various hardware structures. Renaming for OOO cores is assumed to be done using RAM tables. IW refers to issue-width, WP to a write-port, and RP to a read-port.

Structure	Methodology	Assumptions
L1 caches	[119]	Parallel data/tag access
TLBs	[119],[58]	
RegFiles	[119],[104]	$2 \times IW$ RP, IW WP
Execution Units	[58]	
RenameTables	[119][104]	$3 \times IW$ RP, IW WP
ROBs	[119]	IW RP, IW WP, 20b-entry,6b-tag
IQs(CAM arrays)	[119]	IW RP, IW WP, 40b-entry,8b-tag
Ld/St Queues	[119]	64b-addressing,40b-data

To get total area and power estimates, we assume that the area and power of a core can be approximated as the sum of its major pieces. In reality, we expect that the unaccounted-for overheads will scale our estimates by constant factors. In that case, all our results will still be valid.

Figure V.1 shows the area and power of the 480 cores used for this study. As can be seen, the cores represent a significant range in terms of power (4.1-16.3W) as well as area (3.3-22mm^2). For this study, we consider 4-core multiprocessors with different area and peak power budgets. There is a significant range in the area and power budget of the 4-core multiprocessors that can be

Table V.3: Derived Area and Power Estimates for Processor Components

Structure	Area (mm^2)	Power (W)
8KB-DM cache	0.4	0.638
16KB-2-way cache	0.745	1.018
32KB-4-way cache	1.495	1.744
64KB-4-way cache	2.6	1.869
64KB-4-way dual-ported cache	5.05	3.932
64-entry ITLB	0.119	0.126
128-entry ITLB	0.238	0.186
16-entry InstQ (Int/FP)	0.063, 0.203, 0.721 ($IW = 1, 2, 4$)	0.129, 0.266, 0.565 ($IW = 1, 2, 4$)
32-entry InstQ (Int/FP)	0.086, 0.273, 0.991 ($IW = 1, 2, 4$)	0.144, 0.301, 0.655 ($IW = 1, 2, 4$)
64-entry InstQ	0.16, 0.505, 2.596 ($IW = 1, 2, 4$)	0.186, 0.394, 0.899 ($IW = 1, 2, 4$)
32-entry lsQ single-port (Int/FP)	0.1	0.161
32-entry lsQ dual-ported (Int/FP)	0.319	0.333
Branch Predictor	0.2	0.3
1 ALU	0.385	0.45
1 IntMul	0.295	0.45
1 FPU	0.728	0.9
32-entry Regfile	0.1, 0.339, 1.244 ($IW = 1, 2, 4$)	0.212, 0.439, 0.953 ($IW = 1, 2, 4$)
64-entry Regfile	0.137, 0.411, 1.5 ($IW = 1, 2, 4$)	0.367, 0.854, 1.897 ($IW = 1, 2, 4$)
128-entry Regfile	0.192, 0.611, 2.15 ($IW = 1, 2, 4$)	0.517, 1.154, 2.788 ($IW = 1, 2, 4$)
32-entry RAM Rename Table	0.049, 0.176, 0.668 ($IW = 1, 2, 4$)	0.137, 0.284, 0.606 ($IW = 1, 2, 4$)
32-entry ROB	0.04, 0.158, 0.533 ($IW = 1, 2, 4$)	0.07, 0.157, 0.311 ($IW = 1, 2, 4$)
64-entry ROB	0.06, 0.218, 0.753 ($IW = 1, 2, 4$)	0.1, 0.209, 0.451 ($IW = 1, 2, 4$)

constructed out of these cores. Area can range from $13.2mm^2$ to $88mm^2$. Power can range from 16.4W to 65.2W.

V.E.3 Modeling Performance

This section describes the workloads used for evaluation, the performance evaluation methodology, and the evaluation metric.

All our evaluations are done for multiprogrammed workloads. Table V.E.3 lists the ten benchmarks used for constructing workloads. Seven benchmarks are from the SPEC suite. These benchmarks are chosen in the following

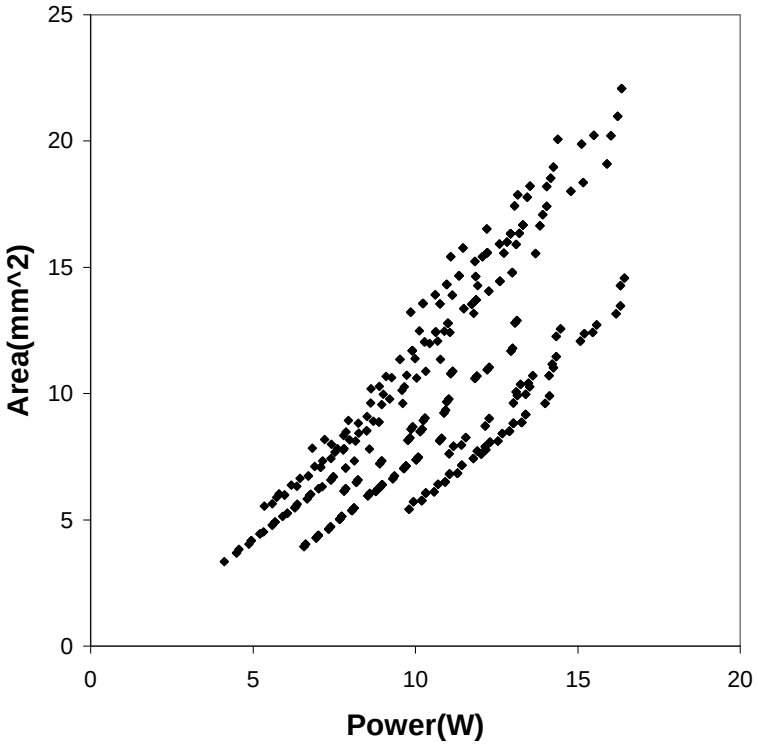

Figure V.1: Area and Power of the cores

way. We simulated *all* 26 benchmarks from the SPEC suite for 250 million cycles using the EV5 processor model after fast-forwarding for an appropriate number of instructions [117]. Then benchmarks were classified into *processor bound* or *bandwidth bound* based on the number of main memory references per instruction. Seven benchmarks were then chosen from these two sets in proportion to the occurrence of these classes of benchmarks in the SPEC suite. Hence, the chosen SPEC benchmarks are intended to represent the entire SPEC suite. We also chose *groff*, *deltablue* and *adpcmc* from the IBS, OOCSB and Mediabench suites respectively. Choosing these three additional benchmarks recognizes the existence of other kinds of application behavior that are not displayed by the SPEC benchmarks, while still considering SPEC representative of a wide variety of applications.

Table V.4: Benchmarks used for design space exploration of heterogeneous multi-cores

Program	Description
ammp	Computational Chemistry
crafty	Game Playing:Chess
eon	Computer Visualization
mcf	Combinatorial Optimization
twolf	Place and Route Simulator
mgrid	Multi-grid Solver: 3D Potential Field
mesa	3-D Graphics Library
groff	Typesetting package
deltablue	Constraint Hierarchy Solver
adpcmc	Encoder for Adaptive Differential Pulse Code Modulation

Every multiprocessor is evaluated on two classes of workloads. The *all different* class consists of all possible 4-threaded combinations that can be constructed such that each of the 4 threads running at a time is different. The *all same* consists of all possible 4-threaded combinations that can be constructed such that all the 4 threads running at a time are the same. For example, *a,b,c,d* is an *all different* workload while *a,a,a,a* is an *all same* workload. This effectively brackets the expected diversity in any workload – including server, parallel, and multithreaded workloads. Hence, we expect our results to be generalizable across a wide range of environments.

As discussed before, there are over 2.2 billion distinct 4-core multiprocessors that can be constructed using our 480 distinct cores. We assume that the performance of a multiprocessor is the sum of the performance of each core of the multiprocessor, as described in Section V.C. Note that this is a reasonable assumption (and validated in Section V.G) because each core is assumed to have a private L2 cache as well as a memory channel. This is the same architecture (private L2s) assumed in [67] and is supported by recent research comparing private and shared L2 caches for multi-core architectures [91]. We also validate that the results made with these assumptions still apply with shared L2 caches for our benchmarks (see Section V.G).

We find the single thread performance of each application on each core. This represents 4800 simulations. Simulations use a modified version of SMTSIM [127]. Scripts are used to calculate the performance of the multiprocessors using these single-thread performance numbers.

All results are presented for the best (oracular) static mapping of applications to cores. Note that realistic dynamic mapping can do better (III.B) – we show in Section V.F that dynamic mapping continues being useful for the best heterogeneous designs that our methodology produces. However, evaluating 2.2

billion multiprocessors becomes intractable if dynamic mapping is assumed.

We use weighted speedup [120] for our evaluations. In this chapter, weighted speedup measures the arithmetic sum of each running thread's IPC, divided by its IPC on the simplest core considered in this study when running alone. The IPC is derived by running a thread for a fixed amount of time. We believe that this metric guards against multiprocessor design points that produce artificial speedups by simply favoring high-IPC threads.

V.F Analysis and Results

This section presents the results of our heterogeneous multi-core architecture design space search. We present these results for a variety of different area and power constraints, allowing us to observe how the benefits of heterogeneity vary across area and power domains. We also examine the effect of different levels of thread level parallelism and the impact of dynamic thread switching mechanisms. We quantify the gains observed due to allowing non-monotonic cores on the processor. Last, we show that the results we observe are applicable not only for the private L2 cache configurations we assume, but even when a shared L2 cache is considered.

V.F.1 Fixed Area Budget

This section presents results for fixed area budgets. For every fixed area limit, a complete design space exploration is done to find the highest performing 4-core multiprocessor. In fact, for each area budget, we find the best architectures across a range of power constraints — the best architecture overall for a given area limit, regardless of power, will always be the highest line on the graph.

Figure V.2 shows the weighted speedup for the highest performing 4-core multiprocessors within an area budget of $40mm^2$. The three lines correspond

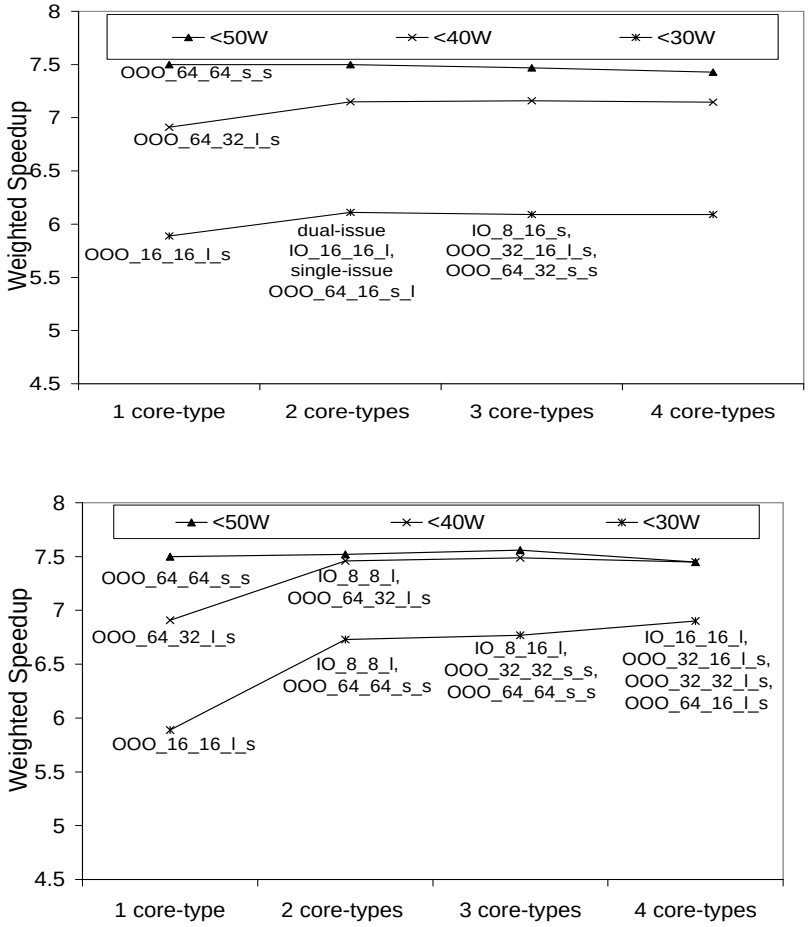

Figure V.2: Throughput for *all-same* (top) and *all-different* (bottom) workloads, area budget=$40mm^2$

to different power budgets for the cores. The top line represents the highest performing 4-core multiprocessors with total power due to cores *not exceeding* 50W. The middle line corresponds to 4-core multiprocessors with total power due to cores not exceeding 40W. The line at the bottom corresponds to 4-core multiprocessors with total power due to cores not exceeding 30W.

Figure V.3 shows the results for different area budgets.

The performance of these 4-core multiprocessors is shown for different amounts of on-chip diversity. *One core type*, for example, implies that all cores

on the die are of the same type, and hence refers to a homogeneous multiprocessor. There are two ways to construct 4-core multiprocessors with *two core types*. One is when three cores are of one type and the fourth core is of another type. Alternatively, one can have two cores of one type and the other two cores of another type. We graph these separately, and refer to these possibilities as *2 core types (3-1)* and *2 core types (2-2)* respectively. Similarly, *3 core types* refers to multiprocessors with three types of cores on the die and *4 core types* refers to multiprocessors with all different cores. Note that the *4 core types* result, for example, only considers processors with four unique cores. Thus, if the best heterogeneous configuration has two unique cores, the three-core and four-core results will show as lower.

Performance, as discussed in the methodology section, is expressed in terms of weighted speedup where the baseline is the performance of the simplest core considered in this study.

The results in Figures V.3 and V.2 lead to several interesting observations. First, we notice that the advantages of diversity are lower with *all same* than the *all different* workload, but they do exist. This is non-intuitive for our artificially-homogeneous workload; however, we find that even these workloads achieve their best performance when at least one of the cores is well-suited for the application — a heterogeneous design ensures that whatever application is being used for the homogeneous runs, such a core likely exists. For example, the best homogeneous CMP for *all same* workloads for an area budget of $40mm^2$ and a power budget of 30W consists of 4 single-issue OOO cores with 16KB L1 caches and double the functional units than the simplest core. This multiprocessor does not perform well when running applications with high cache requirements. On the other hand, the best heterogeneous multiprocessor with 3 core types for *all same* workloads for identical budgets consists of two single-issue in-order cores with

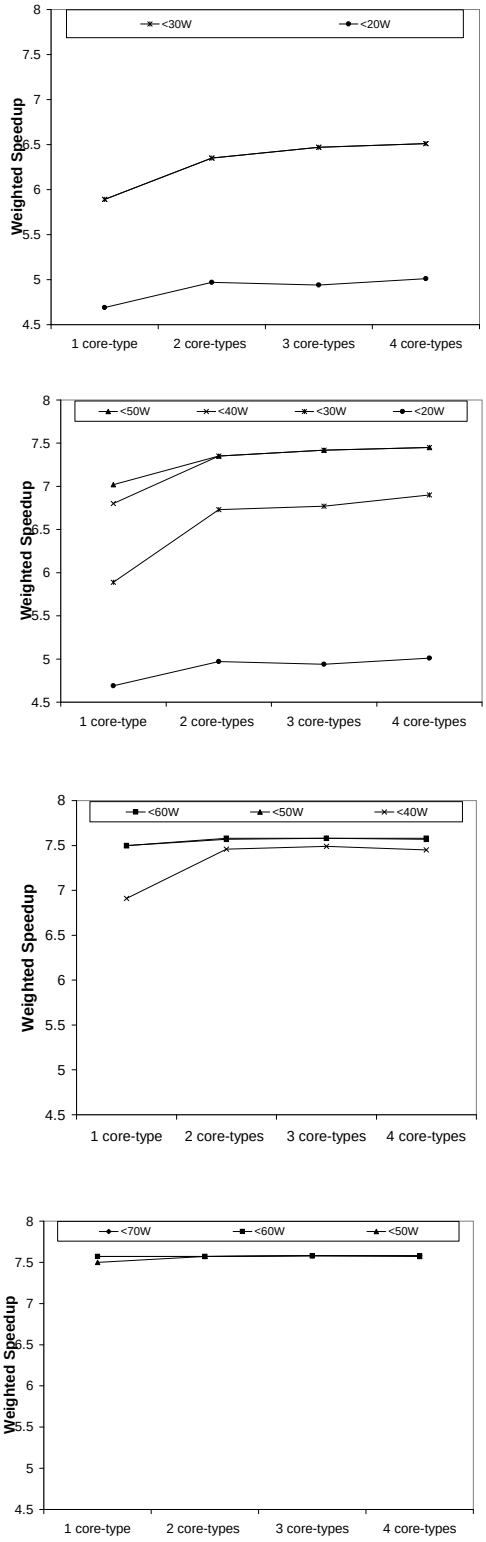

Figure V.3: Throughput for *all-different* workloads for an area budget of (left to right) $20mm^2$, $30mm^2$, $50mm^2$, and $60mm^2$.

8KB ICache and 16KB DCache, one scalar OOO core with 32KB ICache, 16KB DCache and double the functional units, and one scalar OOO core with 64KB ICache and 32KB DCache. The three core types cover the spectrum of application requirements better and hence result in outperforming the best homogeneous CMP by 3.6%. In general, if there is benefit to be had from heterogeneity (as shown in the *all-different* results), it typically also exists in the *all same* case, but to a lesser degree.

Second, we observe that the advantages due to heterogeneity for a fixed area budget depend largely on the power budget available — as shown by the shape of the lines corresponding to different power budgets. In this case (Figure V.2), heterogeneity buys little additional performance with a generous power budget (50W), but is increasingly important as the budget becomes more tightly constrained. We see this pattern throughout our results, whenever either power or area is constrained. What we find is that without constraints, the homogeneous architecture can create "envelope" cores — cores that are over-provisioned for any single application, but able to run most applications with high performance. For example, for an area budget of $40mm^2$, if the power budget is set high (50W), the "best" homogeneous architectures consists of 4 OOO cores with 64KB ICache, 32KB DCache and double the number of functional units than the simplest core. This architecture is able to run both the memory-bound as well as processor-bound applications well. When the design is more constrained, we can only meet the needs of each application through heterogeneous designs that are customized to subsets of the applications. It is likely that in the space where homogeneous designs are most effective, a heterogeneous design that contained more cores would be even better; however, we did not explore this axis of the design space.

We see these same trends in Figure V.3, which shows results for four

other area budgets. There is significant benefit to a diversity of cores as long as either area or power are reasonably constrained.

The power and area budgets also determine the amount of diversity needed for a multi-core architecture. In general, the more constrained the budget, the more benefits are accrued due to increased diversity. For example, considering the *all different* results in Figure V.2, while having 4 core types results in the best performance when the power limit is 30W, two core types (or one) are sufficient to get all the potential benefits for higher power limits. Similar trends can be observed for other area budgets as well. In some of the regions where moderate diversity is sufficient, two unique cores not only matches configurations with higher diversity, but even beats them. In cases where higher diversity is optimal, the gains must still be compared against the design and test costs of more unique cores. For example, in the example above, the marginal performance of 4 core types over the best 2-type result is 2.5%, and probably does not justify the extra effort in this particular example.

These results do underscore the increasing importance of single-ISA heterogeneous multi-core architectures for current and future processor designs. As designs become more aggressive, we will want to place more cores on the die (placing area pressure on the design), and power budgets per core will likely tighten even more severely. Our results show that having two core types is sufficient for getting most of the potential out of moderately power-limited designs, increased diversity results in significantly better performance for highly power-limited designs.

Even when two core types is sufficient, the best combination of those types varies. There is often a substantial difference in the performance of the best *2 core-types(3-1)* and *2 core-types(2-2)* processors for a given area and power budget, and which is better varies.

Another way to interpret these results is that heterogeneous designs dampen the effects of constrained power budgets significantly. For example, in the $40mm^2$ results, both homogeneous and heterogeneous solutions provide good performance with a 50W budget. However, the homogeneous design loses 9% performance with a 40W budget and 23% with a 30W budget. Conversely, with a heterogeneous design, we can drop to 40W with only a 2% penalty and to 30W with a 9% loss.

Perhaps more illuminating than the raw performance of the best designs is what architectures actually provide the best design for a given area and power budget. We observe that there can be a significant difference between the cores of the best heterogeneous multiprocessor and the cores constituting the best homogeneous CMP. That is, the best heterogeneous multiprocessors cannot be constructed only by making slight modifications to the best homogeneous CMP design. Rather, they need to be designed from the ground up. Consider, for example, the best multiprocessors for an area budget of $40mm^2$ and a power budget of 30W. The best homogeneous CMP consists of single-issue OOO cores with 16KB L1 caches, few functional units (1-1-2) and a large number of registers (128). On the other hand, the best heterogeneous CMP with two types of cores, for *all different* workloads, consists of two single-issue in-order cores with 8KB L1 caches and two single-issue OOO cores with 64KB ICache, 32KB DCache and double the number of functional units. Clearly, these cores are significantly different from each other.

Another interesting observation is the reliance on non-monotonicity. Prior work on heterogeneous multi-core architectures [87, 51, 95, 55] assumed configurations where every core was either a subset or superset of every other core (in terms of processor parameters). However, in several of our best heterogeneous configurations, we see that no core is a subset of any other core. For

example, in the same example as above, the best heterogeneous CMP for two core types for *all same* workloads, consists of superscalar in-order cores (issue-width=2) and scalar out-of-order cores (issue-width=1). Even when all the cores are different, the "best" multiprocessor for *all different* workloads consists of one single-issue in-order core with 16KB L1 caches, one single-issue OOO core with 32KB ICache and 16KB DCache, one single-issue in-order core with 32KB L1 caches and one single-issue OOO core with 64KB ICache and 16KB DCache. Thus, the real power of heterogeneity is not in combining "big" and "little" cores, but rather in providing cores each well tuned for a class of applications. This was a common result, and we will explore the importance of non-monotonic designs further shortly.

An interesting tangential observation is that some interesting core types keep cropping up in our results. For example, scalar out-of-order cores occurred frequently in the configurations quoted above. Increased issue-width results in a significant jump in area and power consumption of a core and hence there is a resistance to going superscalar under very constrained area and power budgets. Additional performance is gained by out-of-order issue and larger queues rather than superscalar issue. While our search methodology finds this to be a fairly robust engine, no general-purpose microprocessor ever used this design point.

V.F.2 Fixed Power Budget

While the previous section presented results for fixed area budgets, this section discusses results for fixed power budgets. Figure V.4 (power budget of 30W, both homogeneous and heterogeneous workloads) and Figure V.5 (other power budgets, heterogeneous workloads) give these results. Note that these figures are drawn using the same set of data used for results in the previous section.

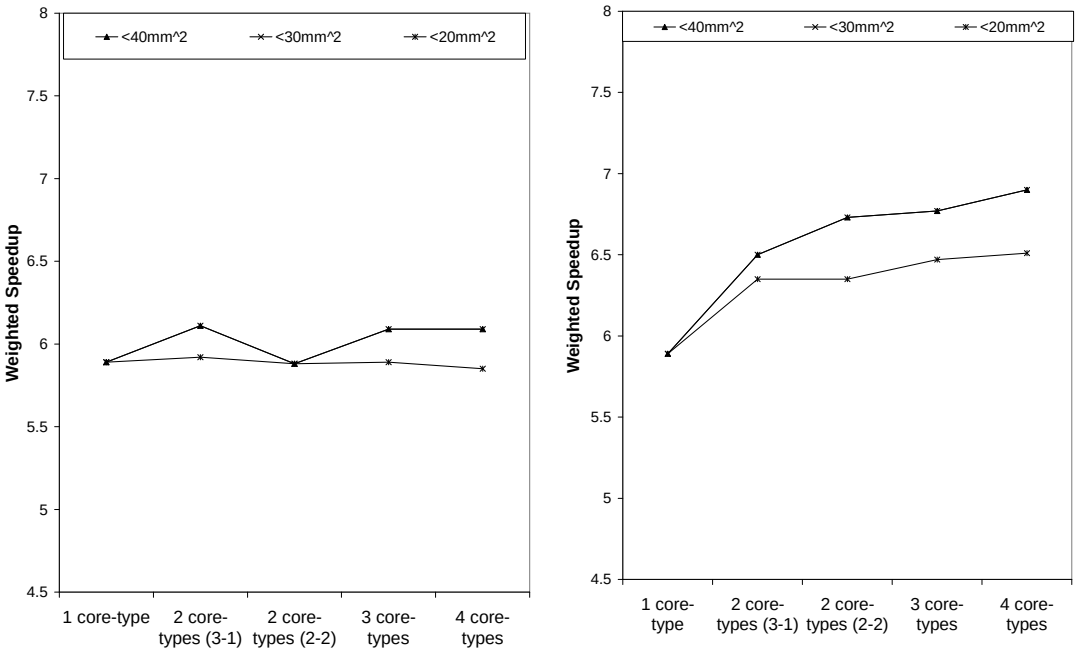

Figure V.4: Throughput for *all-same* (left) and *all-different* (right) workloads, power budget=30W

These results show many of the same trends described in the previous section, and in other cases illustrate a corresponding principle. For example, here we can see that in most cases, heterogeneity also dampens the effect of a decreased area budget – an application of this being that it should be easier to move to more cores when the design is heterogeneous. With a 50W power budget the cost of dropping the area budget from 50 to 30 mm^2 is much greater for the homogeneous designs. Notice that there are a couple points where there appears to be no loss for the homogeneous architecture when reducing the power budget. This is because both power budgets actually chose the exact same homogeneous design. This is a result of the loss of granularity available in choosing homogeneous designs.

Thus, as was the case with power budget, area budget also determines the amount of diversity on the chip.

V.F.3 Impact of Non-monotonic Design

As discussed above, we observed non-monotonicity in several of the highest performing multiprocessors for various area and power budgets. In this section, we analyze this phenomenon further and also try to quantify the advantages due to non-monotonic design.

The reason this feature is particularly interesting is that any design that starts with pre-existing cores from a given architectural family is likely to be monotonic [90, 87]. Additionally, a heterogeneous design that is achieved with multiple copies of a single core, but each with separate frequencies, is also by definition monotonic. Thus, the monotonic results we show in this section serve as a generous upper bound (given the much greater number of configurations we consider) to what can be achieved with an architecture constructed from existing same-ISA cores that are monotonic.

Figure V.6 shows the results for a single set of area and power budgets. In this case, we see that for the *all-same* workload, the benefits from non-monotonic configurations is small, but with the heterogeneous workload, the non-monotonic designs outperform the monotonic much more significantly. More generally (results not shown here), we find that the cost of monotonicity in terms of performance is greater when budgets are constrained. In fact, diversity beyond two core types has benefits only for non-monotonic designs for very constrained power and area budgets.

V.F.4 Varying Thread-Level Parallelism

All results shown thus far in this thesis for heterogeneous architectures have been conservative, for two reasons. They ignore two axes of diversity that also drive the need for heterogeneity – varying thread-level parallelism (TLP) and intra-program (phase) diversity. These two effects are explored in this section and

the next.

While all our studies are done for 4-core processors, it is unrealistic to assume that the processor will always have four threads to run. Thus, we want the same processor to run efficiently with TLP less than four. We see this effect in Figure V.7. Here, TLP=1 means one thread is running at all times, TLP=4 means four threads are always running, and TLP=2 and 3 represent a mix of environments with 1, 2, 3, and 4 threads such that the average TLP is 2.0 or 3.0. Those two results represent the most realistic expected workloads.

Here we see that the benefits of heterogeneity are definitely more pronounced with mixed TLP, particularly with the *all-same* workload. In these particular experiments, we see nearly a 50% advantage from heterogeneity. Of particular interest is the marked gain incurred by a single thread (TLP=1) when going from 2 to 3 unique core types. It would not be unexpected for the best one-thread configuration to be one large core and three tiny cores, with the tiny cores never being used – this is similar to a large monolithic uniprocessor running a single thread. However, we beat such a processor with a more balanced heterogeneous design, with different cores being used by different threads when they run alone.

V.F.5 Dynamic Switching

The other source of diversity that has been ignored thus far in this study is intra-thread diversity. This is a significant factor in prior studies (see Sections III.B, IV, [51]), and is exploited by allowing threads to move between cores as they run. Even finding the best static mapping of threads to cores is difficult without the ability to move threads to find the best long-term mapping.

If threads can switch cores, we can exploit intra-thread diversity by moving to a more suitable core when the thread changes execution character-

istics, due to a change in phase. We did not fully explore this characteristic because the number of simulations would have been enormous – our current methodology treats performance on the cores as separable, and is thus driven by single-thread simulations. Finding the best architectures for dynamic switching would require a full multithreaded simulation for every processor configuration and every workload mix – that is completely impractical, even given significantly larger computational resources than we had available.

Despite the fact that our methodology does not necessarily find the best set of cores given dynamic switching, it is still a reasonable design methodology. This is because we know that (1) the designs we arrive at are *good* designs even when dynamic switching is applied, and (2) the gains over homogeneous designs will only increase. Point (1) is true because in the worst case, the dynamic switching performance mirrors the static case, which we have seen typically beats homogeneous designs. Point (2) is true because the homogeneous processor never gains from dynamic switching.

The results for the case where the applications are mapped to cores dynamically using the *bounded event-triggered* heuristic (see Section III.D for details) are shown in Figure V.8. The results are for a simulation time of 1 billion cycles, a sampling phase of 10 million cycles (with five samples of 2 million cycles) and a steady-phase of 125 million cycles. As can be seen, dynamic switching results in higher performance, and larger marginal gains from higher levels of diversity.

V.F.6 Efficient Search Techniques

For this study, we use exhaustive search to find the best multiprocessors for given area and power budgets. This is because one of our goals was to find and analyze the very best designs. If we instead started with non-optimal search

heuristics, we would have no means to evaluate the quality of the results. The full search considers over 2.2 billion distinct multiprocessor options. Also, each multiprocessor is evaluated on thousands of 4-thread workloads. Hence, while the reported best multiprocessors are indeed the best multiprocessors out of the ones considered, the design space exploration itself was slow. Also, this methodology is not scalable. In particular, the methodology becomes impractical when the number of distinct cores used to construct multiprocessors is increased or when each multiprocessor is evaluated using many more workloads. In such scenarios, alternate methodologies for design space exploration need to be considered.

In those cases, a more efficient search algorithm is needed to navigate the design space. That is, instead of exploring every multiprocessor design, one can evaluate only a subset of the space (e.g., a path through the space, seeking increasingly good designs). The choice of the subset determines the accuracy of evaluation.

As an example, we used a simple well known search algorithm, *hill climbing* [111], to redo the design space exploration for multiprocessors with an area budget of $40mm^2$ and a power budget of 30W. The starting point was the four simplest cores, and we then made modifications to the multiprocessor, one at a time, such that the chosen modification resulted in the highest incremental performance per unit area out of all the incremental modifications possible. The modifications are made in small steps. Also, a change once made is never undone. This has the danger of the search getting stuck in local minima, but results in much faster searches than *generalized hill climbing* where the changes can be undone. The search stops when the available area budget or power budget is exceeded. We observed that the best multiprocessor yielded by *hill climbing* is 11% better that the best homogeneous CMP found using exhaustive search and is only 4.5% worse than the best heterogeneous CMP found using exhaustive search.

It is the subject of future work to experiment with other search algorithms, with which we hope to approach the effectiveness of exhaustive search more closely.

Note that these search algorithms can be used even for fast design space exploration for uniprocessors.

V.G Validating Results

Our results assume that the performance of a multiprocessor is simply the sum of the single-thread performance of the individual cores. Our confidence in this methodology stems from (1) each core on the multiprocessor gets a private 1MB 4-way L2 cache and a private memory controller, (2) we are using multi-programmed, not parallel, workloads, and (3) the SPEC benchmarks (and others used in the study) do not generate heavy off-chip traffic. In this section, we validate the accuracy of this assumption with full multithreaded simulation of particular points, accounting for interactions beyond the L2 cache.

Less obvious is whether these results, and the methodology we follow, applies to the case where L2 caches are shared and interactions between cores are greater. We cannot fully validate all results without full multithreaded simulation of all configurations. Thus, we reduce the full validation to two questions.

First, is the performance of the configurations found to be "best" accurate, particular with respect to the relative performance of homogeneous and heterogeneous designs? Second, is the performance ordering of configurations competing for the designation of "best" configuration preserved with more accurate simulation?

To answer the first question, we consider the highest performing multiprocessors for various number of core types for an area budget of $40mm^2$ and a power budget of 30W. We performed full simulation for the four multiprocessors for various 4-threaded *all different* workloads. We performed simulations

assuming that each core has a private 1MB 4-way L2 cache. We also performed simulations assuming that all cores shared a 4MB, 4-banked 4-way L2 cache. Figure V.9 shows the results.

As can be seen from the graph, the relative ordering of the four multiprocessors remains the same. In fact, the simulations assuming private L2 caches lies on top of the line drawn using the methodology assumed in the paper. Even when the L2 cache is shared, the average performance differs by no more than 2%.

For checking the second condition, we chose 5 multiprocessors with two core-types (3-1) whose performance, as determined by the assumed methodology, fell within the top 50 percentile. The multiprocessors are chosen such that they are equally spread through that range (one from the 90-100th percentile, one from the 80-90th, etc.). Then we performed full simulation for those multiprocessors and compared them against the performance of the multiprocessors using the assumed methodology. Choosing multiprocessors with only two core types helped in keeping the simulation time manageable. We found that the relative ordering of these multiprocessors in terms of performance remains the same in all three cases (performance using assumed methodology, full simulation assuming private L2 caches and full simulation assuming a shared L2 cache). We considered other datapoints as well and observed no significant difference in the trends or analysis.

V.H Acknowledgment

The text of Chapter V is in part a reprint of the material as it appears in the proceedings of the Fifteenth International Conference on Parallel Architectures and Compilation Techniques (September 2006). The dissertation author was the primary researcher and author and the co-authors involved in the submission directed the research which forms the basis for Chapter V.

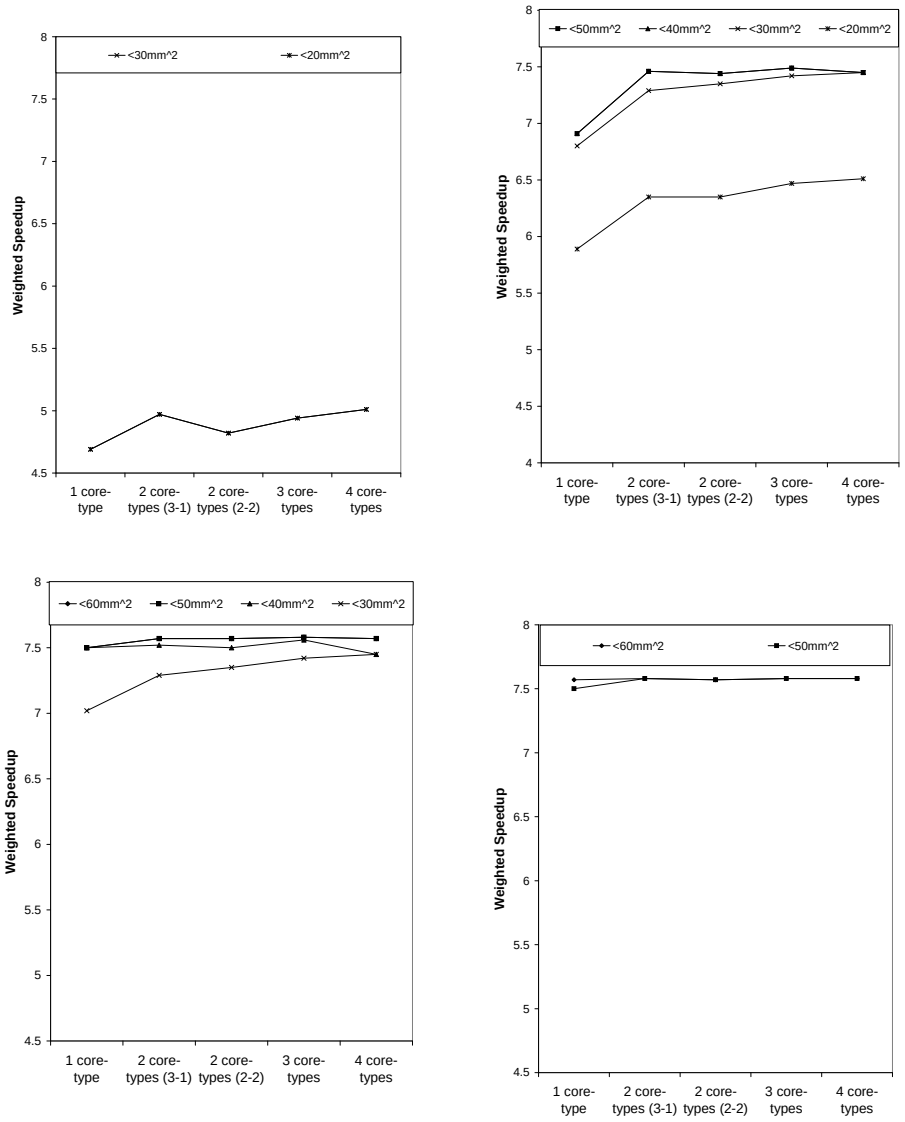

Figure V.5: Throughput for *all-different* workloads for a power budget of (left to right) 20W, 40W, 50W, and 60W.

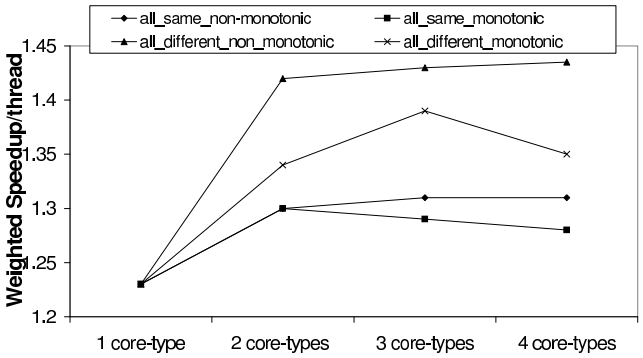

Figure V.6: Benefits due to non-monotonicity of cores;area budget=$40mm^2$,power budget =30W

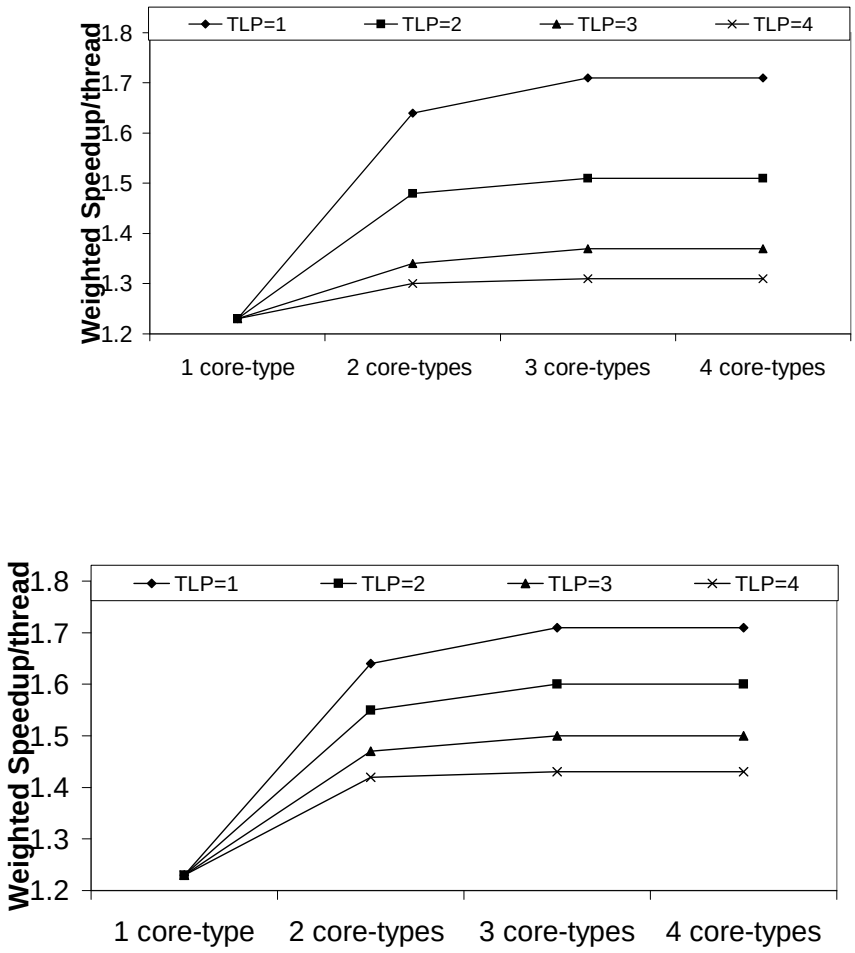

Figure V.7: Throughput for *all-same* and *all-different* workloads for different TLPs, area budget=$40mm^2$, power budget=30W

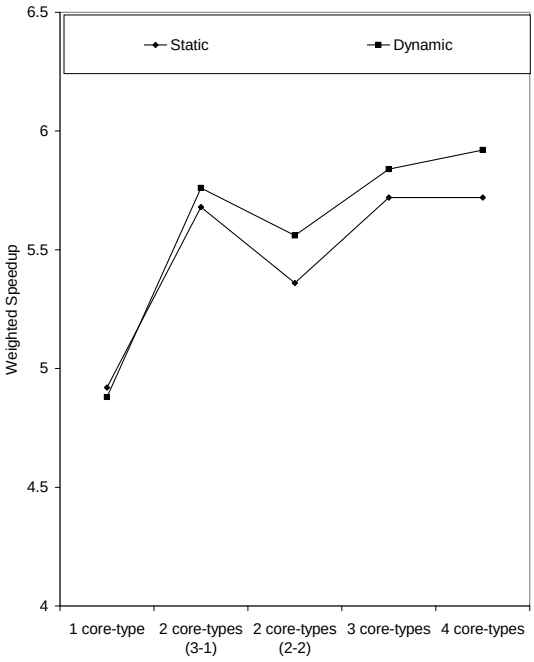

Figure V.8: Benefits due to dynamic switching; area budget = $40mm^2$,power budget=30W

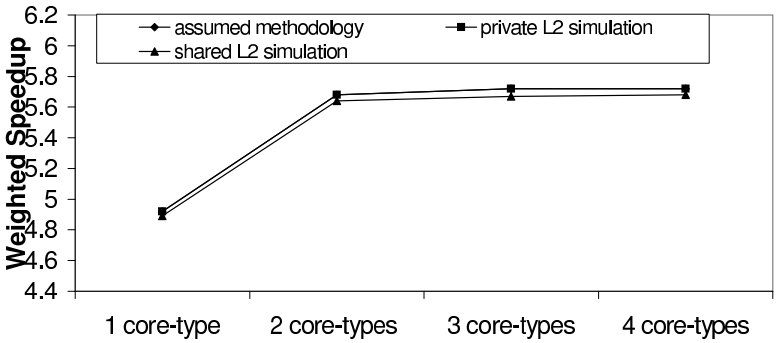

Figure V.9: Comparing the results using assumed methodology against full simulation results: area budget = $40mm^2$, power budget = 30W

VI

Obviating Overprovisioning: Conjoined-core Multiprocessing Architectures

Most modern processors are highly overprovisioned. Designers usually provision the CPU for a few important applications that stress a particular resource. For example, the vector processing unit (VMX) of a processor can often take up more than 10% of the die area, gets used by only a few applications, but the functionality still needs to be there.

"Multi-core-oblivious" multi-core designs exacerbate the overprovisioning problem because the blind replication of cores results in multiplying the cost of overprovisiong by the number of cores. What is really needed is the same level of overprovisioning for any single thread without multiplying the cost by the number of cores.

In this study, we propose a holistic approach to designing chip multiprocessors where the adjacent cores of a multi-core share large, over-provisioned resources. There are several benefits to sharing hardware between more than

one processor or thread. Time-sharing a lightly-utilized resource saves area, increases efficiency, and reduces leakage. Dynamically sharing a large resource can also yield better performance than having distributed small private resources, statically partitioned [129, 41].

Topology is a significant factor in determining what resources are feasible to share and what are the area, complexity, and performance costs of sharing. For example, in the case of sharing entire floating-point units (FPUs), since processor floorplans often have the FPU on one side and the integer datapath on the other side, by mirroring adjacent processors FPU sharing could present minimal disruption to the floorplan. For the design of a resource-sharing core, the floorplan must be co-designed with the architecture, otherwise the architecture may specify sharings that are not physically possible or have high communication costs. In general, resources to be shared should be large enough that the additional wiring needed to share them does not outweigh the area benefits obtained by sharing.

With these factors in mind we have investigated the possible sharing of FPUs, crossbar ports, first-level instruction caches, and first-level data caches between adjacent pairs of processors. Resources could potentially be shared among more than two processors, but this creates more topological problems. Because we primarily investigate sharing between pairs of processors, we call our approach *conjoined-core chip multiprocessors*.

There are many ways that the shared resources can be allocated to the processors in a conjoined configuration. We consider both simple mechanisms, such as fixed allocation based on odd and even cycles, as well as more intelligent sharing arrangements we have developed as part of this work. All of these sharing mechanisms must respect the constraints imposed by long-distance on-chip communication. In fact, we assume in all of our sharing mechanisms that core-to-core

delays are too long to enable cycle-by-cycle arbitration of any shared resource.

It is also possible that the best organization of a shared resource is different than the best organization of a private resource. For example, the right banking strategy may be different for shared memory structures than for private memory structures. Therefore, we also examine tradeoffs in the design of the shared resources as part of this study.

The chief advantage of our proposal is a significant reduction in per-core real estate with minimal impact on per-core performance, providing a higher computational capability per unit area. This can either be used to decrease the area of the whole die, increasing the yield, or to support more cores given a fixed die size. Ancillary benefits include a reduction in leakage power due to a fewer number of transistors for a given computational capability.

VI.A Related Work

Prior work has evaluated design space issues for allocating resources to thread execution engines, both at a higher level and at a lower level than is the target of this chapter. At a higher level, CMP, SMT, and CMPs composed of SMT cores have been compared. At a lower level, previous work has investigated both multithreaded and single-threaded clustered architectures that break out portions of a single core and make them more or less accessible to certain instructions or threads within the core.

Krishnan and Torrellas study the tradeoffs of building multithreaded processors as either a group of single-threaded CMP cores, a monolithic SMT core, or a hybrid design of multiple SMT cores in [83]. Burns and Gaudiot [31] study this as well. Both the studies conclude that the hybrid design, a chip multiprocessor where the individual cores are SMT, represents a good performance-complexity design point. They do not share resources between cores, however.

There has been some work on exploring clustering and hardware partitioning for multithreaded processors. Collins and Tullsen [34] evaluate various clustered multithreaded architectures to enhance both IPC as well as cycle time. They show that the synergistic combination of clustering and simultaneous multithreading minimizes the performance impact of the clustered architecture, and even permits more aggressive clustering of the processor than is possible with a single-threaded processor.

Dolbeau and Seznec [41] propose the CASH architecture as an intermediate design point between CMP and SMT architectures for improving performance. This work is probably the closest prior work to ours. CASH shares caches, branch predictors, and divide units between dynamically-scheduled cores. CASH pools resources from two to four cores to create larger dynamically shared structures with the goal of higher per-core performance. However, the CASH work did not evaluate the area and latency implications of wire routing required by sharing. In our work we consider sharing entire FPUs and crossbar ports as well as caches, and attempt to accurately account for the latency and area of wiring required by sharing. We also consider more sophisticated scheduling techniques for sharing which are consistent with the limitations of global chip communication.

VI.B Baseline Architecture

Conjoined-core chip multiprocessing deviates from a conventional chip multiprocessor design by sharing selected hardware structures between adjacent cores to improve processor efficiency. The choice of the structures to be shared depends not only on the area occupied by the structures but also whether it is topologically feasible without significant disruption to the floorplan or wiring overheads. In this section, we discuss the baseline chip multiprocessor architec-

ture and derive a reasonable floorplan for the processor, estimating area for the various on-chip structures.

VI.B.1 Baseline processor model

For our evaluations, we assume a processor similar to Piranha [26], with eight cores sharing a 4MB, 8-banked, 4-way set-associative, 128B L2 cache. The cores are modeled after Alpha 21164 (EV5). EV5 is a 4-issue in-order processor. The various parameters of the processor are given in Table VI.1. The processor was assumed to be implemented in 0.07 micron technology and clocked at 3.0 GHz.

Table VI.1: Simulated Baseline Processor for studying Conjoining

2K-gshare branch predictor
Issues 4 integer instrs per cycle, including up to 2 Load/Store
Issues 2 FP instructions per cycle
4 MSHRs
64 Byte linesize for L1 caches, 128 Byte linesize for L2 cache
64k 2-way 3 cycle L1 Instruction cache (1 access/cycle)
64k 2-way 3 cycle L1 Data cache (2 access/cycle)
4MB 4-way set-associative, 8-bank 10 cycle L2 cache (3 cycle/access)
4 cycle L1-L2 data transfer time plus 3 cycle transfer latency
450 cycle memory access time
64 entry DTLB, fully associative, 256 entry L2 DTLB
48 entry ITLB, fully associative
8KB pages

For the baseline processor, each core has 64KB, 2-way associative L1 instruction and data caches. The ICache is single-ported while the DCache is dual-ported (2 R/W ports). The L1 cache sizes are similar to those of Piranha cores. A maximum of 4 instructions can be fetched in a given cycle from the ICache. Linesize for both the L1 caches is 64 bytes. Each core has a private

FPU. Floating point divide and square root are non-pipelined. All other floating point operations are fully pipelined. The latency for all operations is modeled after EV5 latencies.

Cores are connected to the L2 cache using a point-to-point fully-connected blocking matrix crossbar such that each core can issue a request to any of the L2 cache banks every cycle. However, one bank can entertain a request from only one of the cores any given cycle. Crossbar link latency is assumed to be 3 cycles, and the data transfer time is 4 cycles.

Each bank of the L2 cache has a memory controller and an associated RDRAM channel. The memory bus is assumed to be clocked at 750MHz, with data being transferred on both edges of the clock for an effective frequency of 1.5GHz and an effective bandwidth of 3GB/s per bank (considering that each RDRAM memory channel supports 30 pins and 2 data bytes). Note that for any reasonable assumption about power and ground pins, the total number of pins that this memory organization would require would be well within the ITRS [15] limits for the cost/performance market. Memory latency is set to 150ns.

VI.B.2 Die floorplan and area model

The baseline architecture and its floorplan is shown in Figure VI.1. We use CACTI to estimate the size and dimensions of the L2 cache. Each 512KB bank is $8.08mm^2$. CACTI gives the aspect ratio to be 2.73. So, each bank is $1.7mm \times 4.7mm$. Total L2 cache area is $64.64mm^2$. The area of the EV5-like core (excluding L1 caches) was calculated using similar assumptions and methodology as was used in [87], which also featured multiple Alpha cores on a die and technology scaling. Each core excluding caches is $2.12mm^2$. CACTI gives the area of of the L1 64KB, 2-way ICache to be $1.15mm^2$ and the 64KB, 2-way DCache to be $2.59mm^2$. Hence, including the area occupied by private L1 caches,

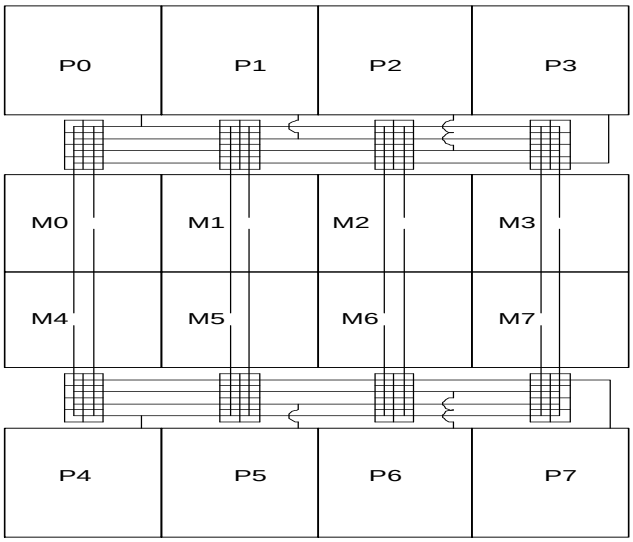

Figure VI.1: Baseline die floorplan for studying conjoining, with L2 cache banks in the middle of the cluster, and processor cores (including L1 caches) distributed around the outside

core area is $5.86mm^2$. If we assume an aspect-ratio of 1, it is $2.41mm \times 2.41mm$. The total area for the eight cores is $46.9mm^2$.

The crossbar area calculations measure the area occupied by the interconnect wires. Each link from a core to a cache bank consists of roughly 300 lines. Of those, 256 lines correspond to a set of 128 unidirectional wires from the L2 to the cores and another 128-bit data bus from the cores to the L2 cache. We assume 20 lines correspond to the 20-bit unidirectional addressing signals while the rest correspond to control signals. Since each of the cores needs to be able to talk to each of the banks, there is a switched repeater corresponding to each core-to-bank interface. Therefore, the number of horizontal tracks required per link would be approximately 300. The total number of input ports is equal to the number of cores. So, the number of horizontal tracks required will be $8 \times 300 = 2400$. This would determine the height of the crossbar. For the lay-

out of the baseline processor (shown in Figure VI.1), the crossbar lies between the cores and the L2 banks. Also, there are two clusters of interconnects. The clusters are assumed to be connected by vertical wires in the crossbar routing channels and by vertical wires in upper metal layers that run over the top of the L2 cache banks (see Figure VI.1).

We assume that all the (horizontal) connecting lines are implemented in the M3/M5 layer. ITRS [15] and the "Future of Wires" paper by Horowitz, et al. [64] predict that wire pitch for a semi-global layer is 8-10λ. Assuming 10λ for 0.07 micron, the pitch is 350nm. The width of the crossbar then is $2400 \times 350nm = 0.84mm$. Therefore, the area occupied by the crossbar for the baseline processor is $16.22mm^2$. This methodology of crossbar area estimation is similar to that used in [76].

Therefore, the total area of the processor is $127.76mm^2$ out of which $46.9mm^2$ is occupied by the cores, $16.22mm^2$ by the crossbar and $64.64mm^2$ by the L2 cache.

VI.C Conjoined-core Architectures

For the conjoined-core chip multiprocessor, we consider four optimizations – instruction cache sharing, data cache sharing, FPU sharing, and crossbar sharing. For each kind of sharing, two adjacent cores share the hardware structure. In this section, we investigate the mechanism for each kind of sharing and discuss the area benefits that they accrue. We talk about the performance impact of sharing in Section VI.E. The usage of the shared resource can be based on a policy decided either statically, such that it can be accessed only during fixed cycles by a certain core, or the accesses can be determined based on certain dynamic conditions visible to both cores (given adequate propagation time). The initial mechanisms discussed in this section all assume the simplest

and most naive static scheduling, where one of the cores gets access to the shared resource during odd cycles while the other core gets access during even cycles. More intelligent sharing techniques/policies are discussed in Section VI.F. All of our sharing policies, however, maintain the assumption that communication distances between cores are too great to allow any kind of dynamic cycle-level arbitration for shared resources.

Note that in modern high-performance pipelines (beginning with the DEC Alpha 21064), variable operation latency past the issue point as a result of FIFO-type structures is not possible. This is because in modern pipelines, each pipestage is only a small number of FO4 delays, and global communication plus control logic overhead for implementing stalling on a cycle-by-cycle basis would drastically increase the cycle time. Such stalling is required by variable delays because instructions must be issued assuming results of previous operations are available when expected. Instead any delay (such as that required by a DCache miss instead of an expected hit) results in a flush and replay of the missing reference plus the following instructions. Although flush and replay overhead is acceptable for rare long latency events such as cache misses, it is unacceptable for routine operation of the pipeline. By assuming very simple fixed scheduling in the baseline sharing case we guarantee that the later pipe stages do not need to be stalled, and the cycle time of the pipeline is not adversely affected. Later we examine more complex techniques for scheduling sharing that remain compatible with high-speed pipeline design.

Due to wiring overheads, it only makes sense to share relatively large structures that already have routing overhead. FPUs, crossbars, and caches all have this property. In contrast, ALUs in a datapath normally fit under the datapath operand and result busses. Thus, placing something small like an individual ALU remotely would actually result in a very significant increase in bus

wiring and chip area instead of a savings, as well as increased latency and power dissipation.

VI.C.1 ICache sharing

We implement ICache sharing between two cores by providing a shared fetch path from the ICache to both the pipelines. Figure VI.2 shows a floorplan of two adjacent cores sharing a 64KB, 2-way associative ICache. Because the layout of memories is a function of the number of rows and columns, we have increased the number of columns but reduced the number of rows in the shared memory. This gives a wider aspect ratio that can span two cores.

As mentioned, the ICache is time-shared every other cycle. We investigate two ICache fetch widths. In the *double fetch width* case, the fetch width is changed to 8 instructions every other cycle (compared to 4 instructions every cycle in the unshared case). The time-averaged effective fetch bandwidth of all cores (ignoring branch effects) remains unchanged in this case. In the *original fetch width* case, we leave the fetch width to be the same. In this case the effective per-core fetch bandwidth is halved. Finally, we also investigate a banked architecture, where cores can fetch 4 instructions every cycle, but only if their desired bank is allocated to them that cycle (a single core gets access to the low bank on even cycle and high bank on odd cycles).

In the *double fetch width* case, sharing requires a wider instruction fetch path, wider multiplexors and extra instruction buffers before decode for the instruction front end. We have modeled this area increase and we also assume that sharing increases the access latency by 1 cycle. The double fetch width solution would also result in higher power consumption per fetch. Furthermore, since longer fetch blocks are more likely to include taken branches out of the block, the fetch efficiency is somewhat reduced. We also evaluate two cases corresponding to

a shared instruction cache with an unchanged fetch width – one with the access time extended by a cycle and another when it remains unchanged.

Based on modeling with CACTI, in the baseline case each ICache takes up $1.15mm^2$. In the double fetch width case, the ICache has double the bandwidth (BITOUT=256), and requires $1.16mm^2$. However, instead of 8 ICaches on the die, there are just four of them. This results in a core area savings of 9.8%. In the normal fetch width case (BITOUT=128), sharing results in core area savings of 9.9%.

VI.C.2 DCache sharing

Even though the DCaches occupy a significant area, DCache sharing is not an obvious candidate for sharing because of its relatively high utilization. In our DCache sharing experiments, two adjacent cores share a 64KB, 2-way set-associative L1 DCache. Each core can issue memory instructions only every other cycle.

Sharing entails lengthened wires that increase access latency slightly. This latency may or may not be able to be hidden in the pipeline. Thus, we evaluate two cases – one where the access time is lengthened by one cycle and another where the access time remains unchanged.

Based on modeling with CACTI, each dual-ported DCache takes up $2.59mm^2$ in the baseline processor. In the shared case, it takes up the area of just one cache for every two cores, but with some additional wiring. This results in core area savings of 22.09%.

VI.C.3 Crossbar sharing

As shown in VI.B, the crossbar occupies a significant fraction (13%) of the die area. The configuration and complexity of the crossbar is strongly tied to

the number of cores. Therefore, we also study how crossbar sharing can be used to free up die area.

Crossbar sharing involves two adjacent cores sharing an input port to the L2 cache's crossbar interconnect. This halves the number of rows (or columns) in the crossbar matrix resulting in linear area savings. Crossbar sharing entails that only one of the two conjoined cores can issue a request to a particular L2 cache bank in a given cycle. Again, we assume a baseline implementation where one of the conjoined cores can issue requests to a bank every odd cycle, while the other conjoined core can issue requests only on even cycles. There would also be some overhead in routing signal and data to the shared input port. Hence, we assume the point-to-point communication latency will be lengthened by one cycle for the conjoined core case. Figure VI.10 shows conjoined core pairs sharing input ports to the crossbar.

Crossbar sharing results in halving the area occupied by the interconnect and results in 6.43% die area savings. This is equivalent to 1.38 times the size of a single core.

Note that this is not the only way to reduce the area occupied by the crossbar interconnect. One can alternatively halve the number of wires for a given point-to-point link to (approximately) halve the area occupied by that link. This would, though, double the transfer latency for each connection. In Section VI.E, we compare both these approaches and show that this performs worse than our port-sharing solution.

Finally, if the DCache and ICache are already shared between two cores, sharing the crossbar port between the same two cores is very straightforward since the cores have already been joined together before reaching the crossbar.

VI.C.4 FPU sharing

Processor floorplans often have the FPU on one side and the integer datapath on the other side. So, FPU sharing can be enabled by simply mirroring adjacent processors without significant disruption to the floorplan. Wires connecting the FPU to the left core and the right core can be interdigitated, so no additional horizontal wiring tracks are required (see Figure VI.2). This also does not significantly increase the length of wires in comparison the non-conjoined case. In our baseline FPU sharing model, each conjoined core can issue floating-point instructions to the fully-pipelined floating-point sub-units only every other cycle. Based on our design experience, we believe that there would be no operation latency increase when sharing pipelined FPU sub-units between the cores. This is because for arithmetic operations the FP registers remain local to the FPU. For transfers and load/store operations, the routing distances from the integer datapath and caches to the FPU remain largely unchanged (see Figure VI.2). For the non-pipelined sub-units (e.g., divides and square root) we assume alternating three cycle scheduling windows for each core. If a non-pipelined unit is available at the start of its three-cycle window, the core may start using it, and has the remainder of the scheduling window to communicate this to the other core. Thus, when the non-pipelined units are idle, each core can only start a non-pipelined operation once every six cycles. However, since operations have a known long latency, there is no additional scheduling overhead needed at the end of non-pipelined operations. Thus, when a non-pipelined unit is in use, another core waiting for it can begin using the non-pipelined unit on the first cycle it becomes available.

The FPU area for EV5 is derived from published die photos, scaling the numbers to 0.07micron technology and then subtracting the area occupied by the FP register file. The EV5 FPU takes up $1.05mm^2$ including the FP register file.

We estimate the area taken up by a 5 exclusive read port (ERP), 4 exclusive write port (EWP), 32-entry FP register file using *register-bit equivalents* (rbe). The total area of the FPU (excluding the register file) is $0.72mm^2$. Sharing results in halving the number of units and results in area savings of 6.1%.

We also consider a case where each core has its own copy of the divide sub-unit, while the other FPU sub-units are shared. We estimated the area of the divide sub-unit to be $0.0524mm^2$. Total area savings in that case is 5.7%.

VI.C.5 Summary of sharing

To sum up, ICache sharing results in core area savings of 9.9%, DCache sharing results in core area savings of 22%, FPU sharing saves 6.1% of the core area, and sharing the input ports to the crossbar can result in a savings of 1.4 cores. Statically deciding to let each conjoined core access a shared hardware structure only every other cycle provides an upper-bound on the possible degradation. As our results in VI.E indicate, even these conservative assumptions lead to relatively small performance degradation and hence reinforce the argument for conjoined-core chip multiprocessing.

VI.D Experimental Methodology

Benchmarks are simulated using SMTSIM [127]. The simulator was modified to simulate the various chip multiprocessor (conjoined as well as conventional) architectures.

Several of our evaluations are done for various numbers of threads ranging from one through a maximum number of available processor contexts. Each result corresponds to one of three sets of eight benchmarks, where each data point is the average of several permutations of those benchmarks.

Table VI.D shows the subset of the SPEC CPU2000 benchmark suite

Table VI.2: Benchmarks simulated for evaluating conjoining

Program	Description	FF Dist (in millions)
bzip2	Compression	5200
crafty	Game Playing:Chess	100
eon	Computer Visualization	1900
gzip	Compression	400
mcf	Combinatorial Optimization	3170
perl	PERL Programming Language	200
twolf	Place and Route Simulator	3200
vpr	FPGA Circuit Placement and Routing	7200
applu	Parabolic/Elliptic Partial Diff. Eqn.	1900
apsi	Meteorology:Pollutant Distribution	4700
art	Image Recognition/Neural Networks	6800
equake	Seismic Wave Propagation Simulation	19500
facerec	Image Processing: Face Recognition	13700
fma3d	Finite-element Crash Simulation	29900
mesa	3-D Graphics Library	9000
wupwise	Physics/Quantum Chromodynamics	58500

that was used. The benchmarks are chosen such that out of the 8 CINT2000 benchmarks, half of them (*vpr,crafty,eon,twolf*) have a dataset of less than 100MB while the remaining half have datasets bigger than 100MB. Similarly, for CFP2000 benchmarks, half of them (*wupwise,applu,apsi,fma3d*) have datasets bigger than 100MB while the remaining half have datasets of less than 100MB. We also perform all our evaluations for mixed workloads which are generated using 4 integer benchmarks (*bzip2,mcf,crafty,eon*) and 4 floating-point benchmarks (*wupwise, applu, art, mesa*). Again, the subsetting was done based on application datasets.

All the data points are generated by evaluating 8 workloads for each case and then averaging the results. A workload consisting of n threads is constructed by selecting the benchmarks using a sliding window (with wraparound) of size n and then shifting the window right by one. Since there are 8 distinct

benchmarks, the window selects eight distinct workloads (except for cases when the window-size is a multiple of 8, in those cases all the selected workloads have identical composition). All of these workloads are run, ensuring that each benchmark is equally represented at every data point. This methodology for workload construction is similar to that used in [120].

We also perform evaluations using the parallel benchmark *water* from the SPLASH benchmark suite and use the STREAM benchmark for crossbar evaluations. We change the problem size of STREAM to 16,384 elements. At this size, when running eight copies of STREAM, the working set fits into the L2-cache and hence it acts as a worst-case test of L1-L2 bandwidth (and hence crossbar interconnect). We also removed the timing statistics collection routines.

The Simpoint tool [118] was used to find good representative fast-forward distances for each SPEC benchmark. Early simpoints are used. Table VI.D also shows the distance to which each benchmark was fast-forwarded before beginning simulation. For *water*, fast-forwarding is done just enough so that the parallel threads get forked. We do not fast forward for STREAM.

All simulations involving n threads are preceded by a warmup of $10 \times n$ million cycles. Simulation length was 800 million cycles. All the SPEC benchmarks are simulated using *ref* inputs. All the performance results are in terms of throughput.

VI.E Simple Sharing

This section examines the performance impact of conjoining cores assuming simple time-slicing of the shared resources on alternate cycles. More intelligent sharing techniques are discussed in the next section.

In this section, we show results for various threading levels. We schedule the workloads statically and randomly such that two threads are run together on a

conjoined-core pair only if one of them cannot be placed elsewhere. Hence, for the given architecture, for 1 to 4 threads, there is no other thread that is competing for the shared resource. If we have 5 runnable threads, one of the threads needs to be put on a conjoined-core pair that is already running a thread. And so on. However, even if there is no other thread running on the other core belonging to a conjoined-core pair, we still assume, in this section, that accesses can be made to the shared resource by a core only every other cycle.

VI.E.1 Sharing the ICache

Results are shown as performance degradation relative to the the baseline conventional CMP architecture. Performance degradation experienced with ICache sharing comes from three sources: increased access latency, reduced effective fetch bandwidth, and inter-thread conflicts. Effective fetch bandwidth can be reduced even if the fetch width is doubled because of the decreased likelihood of filling an eight-wide fetch with useful instructions, relative to a four-wide fetch.

Figure VI.4 shows the performance impact of ICache sharing for varied threading levels for SPEC-based workloads. The results are shown for a fetch width of 8 instructions and assuming that there is an extra cycle latency for ICache access due to sharing. We assume the extra cycle is required since in the worst case the round-trip distance to read an ICache bit has gone up by two times the original core width due to sharing. We observe a performance degradation of 5% for integer workloads, 1.2% for FP workloads and 2.2% for mixed workloads. The performance degradation does not change significantly when the number of threads is increased from 1 to 8. This indicates that inter-thread conflicts are not a problem for this workload and these caches. The SPEC benchmarks are known to have relatively small instruction working sets.

To identify the main cause for performance degradation on ICache shar-

ing, we also show results assuming that there is no extra cycle increase in the latency. Figure VI.5 shows the 8-thread results for both integer and floating-point workloads. Performance degradation becomes less than 0.25%. Two conclusions can be drawn from this. First, the extra latency is the main reason for degradation on ICache sharing (note that the latency does not introduce a bubble in the pipeline – the performance degradation comes from the increased branch mispredict penalty due to the pipeline being extended by a cycle). The integer benchmarks are most affected by the extra cycle latency, being more sensitive to the branch mispredict penalty.

Increasing fetch width to 8 instructions ensures that the potential fetch bandwidth remains the same for the sharing case as the baseline case, but it increases the size of the ICache (relative to a single ICache in the base case) and results in increased power consumption. This is because doubling the output width doubles both the number of sense amps and the data output lines being driven, and these structures account for much of the power in the original cache. Thus, we also investigate the case where fetch width is kept the same. In that case, only up to 4 instructions can be fetched every other cycle (effectively halving the per-core fetch bandwidth). Figure VI.5 shows the results for 8-thread workloads. As can be seen, degradation jumps up to 16% for integer workloads and 10.2% for floating-point workloads. This is because at effective fetch bandwidth of 2 instructions every cycle (per core), execution starts becoming fetch limited.

We also investigate the impact of partitioning the ICache vertically into two equal sized banks. A core can alternate accesses between the two banks. It can fetch 4 instructions every cycle but only if the desired bank is available. A core has access to bank 0 one cycle, bank 1 the next, etc., with the other core having the opposite allocation. This allows both threads to access the cache in some cycles. It is also possible for both threads to be blocked in some cycles.

However, bandwidth is guaranteed to exceed the previous case (ignoring cache miss effects) of one 4-instruction fetch every other cycle, because every cycle that both threads fail to get access will be immediately followed by a cycle in which they both can access the cache.

Figure VI.5 shows the results. Degradation goes down by 55% for integer workloads and 53% for FP workloads due to overall improvement in fetch bandwidth.

VI.E.2 DCache sharing

Similar to the ICache, performance degradation due to DCache sharing comes from: increased access latency, reduced cache bandwidth, and inter-thread conflicts. Unlike the ICache, the DCache latency has a direct effect on performance, as the latency of the load is effectively increased if it cannot issue on the first cycle it is ready.

Figure VI.6 shows the impact on performance due to DCache sharing for SPEC workloads. The results are shown for various threading levels. We observe a performance degradation of 4-10% for integer workloads, 1-9% for floating point workloads and 2-13% for mixed workloads. Degradation is higher for integer workloads than floating point workloads for small numbers of threads. This is because the typically higher ILP of the FP workloads allows them to hide a small increase in latency more effectively. Also, inter-thread conflicts are higher, resulting in increased performance degradation for higher numbers of threads.

We also studied the case where the shared DCache has the same access latency as the unshared DCache. Figure VI.7 shows the results for the 8-thread case. Degradation lessens for both integer workloads as well as floating-point workloads, but less so in the case of FP workloads as conflict misses and cache bandwidth pressure remain.

VI.E.3 FPU sharing

FPUs may be the most obvious candidates for sharing. For SPEC CINT2000 benchmarks only 0.1% of instructions are floating point while even for CFP2000 benchmarks, only 32.3% of instructions are floating-point instructions [20]. Also, FPU bandwidth is a performance bottleneck only for specialized applications.

We evaluated FPU sharing for integer workloads, FP workloads, and mixed workloads, but only present the FP and mixed results (Figure VI.8) here. The degradation is less than 0.5% for all levels of threading, even in these cases.

One reason for these results is that the competition for the non-pipelined units (divide and square root) is negligible in the SPEC benchmarks. To illustrate code where non-pipelined units are more heavily used, Figure VI.9 shows the performance of *water* (which has a non-trivial number of divides) running eight threads. It shows performance with a shared FP divide unit vs. unshared FP divide units. In this case, unless each core has its own copy of the FP divide unit, performance degradation can be significant.

VI.E.4 Crossbar sharing

We implement the L1-L2 interconnect as a blocking fully-connected matrix crossbar, based on the initial Piranha design. As the volume of traffic between L1 and L2 increases, the utilization of the crossbar goes up. Since there is a single path from a core to a bank, high utilization can result in contention and queuing delays.

As discussed in Section VI.C, the area of the crossbar can be reduced by decreasing the width of the crossbar links or by sharing the ports of the crossbar, thereby reducing the number of links. We examine both techniques. Crossbar sharing involves the conjoined cores sharing an input port of the crossbar. Fig-

ure VI.10 shows the results for eight copies of the STREAM benchmark. It must be noted that this is a component benchmark we have tuned for worst-case utilization of the crossbar. The results are shown in terms of performance degradation caused for achieving certain area savings. For example, for achieving crossbar area savings of 75% (*area/4*), we assume that the latency of every crossbar link has been doubled for the *crossbar sharing* case while the transfer latency has been quadrupled for the *crossbar width reduction* case.

We observe that crossbar sharing outperforms crossbar width reduction in all cases. Even though sharing results in increased contention at the input ports, it is the latency of the links that is primarily responsible for queuing of requests and hence overall performance degradation.

We also conducted crossbar exploration experiments using SPEC benchmarks. However, most of the benchmarks do not exercise L1-L2 bandwidth much, resulting in relatively low crossbar utilization rates. The performance degradation in the worst case was less than 5% for an area reduction factor of 2.

VI.E.5 Simple sharing summary

Note that for all the results in this section, we assume that the shared resource is accessible only every other cycle even if the other core on a conjoined-core pair is idle. This was done to expose the factors contributing to overall performance degradation. However, in a realistic case, if there is no program running on the other core, the shared resources can be made fully accessible to the core running the program and hence there would be no (or much smaller) degradation. Thus, for the above sharing cases, the degradation values for threading levels of four are overstated. In fact, the performance degradation due to conjoining will be minimal for light as well as medium loads.

This section indicates that, even in the absence of sophisticated sharing

techniques, conjoined-core multiprocessing is a reasonable approach. Optimizations in the next section make it even more attractive. It might be argued that this is simply evidence of the over-provisioning of our baseline design. There are two reasons why that is the wrong conclusion. First, our baseline is based on real designs, and is not at all aggressive compared to modern processor architectures. Second, real processors are over-provisioned – to some extent that is the point of this study. Designers provision the CPU for the few important applications that really stress a particular resource. What this research shows is that we can maintain that same level of provisioning for any single thread, without multiplying the cost of that provisioning by the number of cores.

VI.F Intelligent Sharing of Resources

The previous section assumed a very basic sharing policy and hence gave an upper bound on the degradation for each kind of sharing. In this section, we discuss more advanced techniques for minimizing performance degradation.

VI.F.1 ICache sharing

In this section, we will focus on that configuration that minimized area, but maximized slowdown — the four-wide fetch shared ICache, assuming an extra cycle of latency. In that case, both access latency and fetch bandwidth contribute to the overall degradation. We propose two techniques for minimizing degradation in that case. Most of these results would also apply to the other configurations of shared ICache, taking them even closer to zero degradation.

Assertive ICache Access

Chapter VI.E discussed sharing such that the shared resource gets accessed evenly irrespective of the access needs of the individual cores. Instead,

the control of a shared resource can be decided assertively based on the resource needs.

We explore *assertive ICache access* where, whenever there is an L1 miss, the other core can take control of the cache after miss detection. We assume that a miss can be detected and communicated to the other core in 3 cycles. The control would start getting shared again when the data returns. This does not incur any additional latency since the arrival cycle of the data is known well in advance of its return.

Figure VI.11 shows the results for *assertive icache access*. Like all graphs in this section, we show results for eight threads, where contention is highest. We observe a 13.7% improvement in the degradation of integer workloads and an improvement of 22.5% for floating point workloads. Performance improvement is because of improved effective fetch bandwidth. These results are for eight threads, so there is no contribution from threads that are not sharing an ICache. A minor tweak to *assertive access* (for ICache as well as DCache and FPU) can ensure that the shared resource becomes a private resource when the other core of the conjoined pair is idle.

Fetch combining

Most parallel code is composed of multiple threads, each executing code from the same or similar regions of the shared executable (possibly synchronizing occasionally to ensure they stay in the same region). Hence, it is not uncommon for two or more threads to be fetching from the same address in a particular cycle.

In a conjoined-core architecture with shared ICache, this property can be exploited for improving overall fetch bandwidth. We propose *fetch combining* – when two threads running on the same conjoined-core pair have the same

nextPC cache index, then they both can return data from the cache that cycle. The overhead for fetch combining is minimal, under the following assumptions. We assume that the fetch units are designed so that, in the absence of sharing (no thread assigned to the alternate core), one core can fetch every cycle. Thus, each core has the ability to generate a nextPC cache index every cycle, and to consume a fetch line every cycle. In sharing mode, however, only one request is filled. Thus, in sharing mode with fetch combining, both cores can present a PC to the ICache, but only the PC associated with the core with access rights that cycle is serviced. However, if the presented PCs are identical, the alternate core also reads the data presented on the (already shared) output port and bus. This is simplified if there is some decoupling of the branch predictor from the fetch unit. If a queue of nextPC cache indices is buffered close to the ICache, we can continue to present new PCs to the cache every cycle, even if it takes more than a cycle for the result of the PC comparison to get back to the core.

We find that the frequency of coincident indices is quite high – this is because once the addresses match, they tend to stay synched up until the control flow diverges.

Figure VI.12 shows the performance of fetch combining for *water* running eight threads. We observed a 25% reduction of performance degradation. Note that fetch combining is appropriate for other multithreading schemes like SMT, etc.

VI.F.2 DCache sharing

Performance loss due to DCache sharing is due to three factors – inter-thread conflict misses, reduced bandwidth and increased latency (if applicable). We propose two techniques for minimizing degradation due to DCache sharing.

Assertive DCache Access

Assertive access can also be used for the shared DCaches. Whenever there is an L1 miss on some data requested by a core, if the load is determined to be on the right path, the core relinquishes control over the shared DCache. There may be some delay between detection of L1 miss and the determination that the load is on the right path. Once the core relinquishes control, the other core takes over full control and can then access the DCache whenever it wants. The timings are the same as with the ICache assertive access. This policy is still somewhat naive, assuming that the processor will stall for this load (recall, these are in-order cores) before another load is ready to issue – more sophisticated policies are possible.

Figure VI.13 shows the results. Assertive access leads to 29.6% improvements in the degradation for integer workloads and 23.7% improvements for floating point workloads. Improvements are due to improved data bandwidth.

I/O partitioning

The Dcache interface consists of two R/W ports. In the basic DCache sharing case, the DCache (and hence both the ports) can be accessed only every other cycle. Instead, one port can be statically assigned to each of the cores and that will make the DCache accessible every cycle.

Figure VI.13 shows the results comparing the baseline sharing policy against static port-to-core assignment. We observed a 33.6% reduction in degradation for integer workloads while the difference for FP workloads was only 3%. This outperforms the cycle-slicing mechanism, particularly for integer benchmarks, for the following reason: when load port utilization is not high, the likelihood (with port partitioning) of a port being available when a load becomes ready is high. However, with cycle-by-cycle slicing, the likelihood of a port being

available that cycle is only 50%.

VI.F.3 Symbiotic assignment of threads

Previous techniques involved either using additional hardware for minimizing performance impact or scheduling accesses to the shared resources intelligently. Alternatively, high-level scheduling of applications can be done such that "friendly" threads run on conjoined cores. Symbiotic scheduling has been shown previously to result in significant benefits on an SMT architecture [120] and involves co-scheduling threads to minimize competition for shared resources.

Since conflict misses are a significant source of performance degradation for DCache sharing, we evaluated the impact of scheduling applications intelligently on the cores instead of random mapping. Intelligent mapping involved putting programs on a conjoined-core pair that would not cause as many conflict misses and hence lessen the degradation. For symbiotic scheduling with 8 threads, we found the degradation decreased by 20% for integer workloads and 25.5% for FP workloads.

VI.G A Unified Conjoined-Core Architecture

This section examines various combinations of FPU, crossbar, ICache, and DCache sharing. For all these experiments, we assume a shared doubly-banked ICache with a fetch-width of 16 bytes (similar to that used in Section VI.E.1), I/O partitioned shared DCache (similar to that used in Section VI.F.2), a fully-shared FPU and a shared crossbar input port for every conjoined-core pair. Sharing ICache, DCache, as well as the crossbar are each assumed to have one cycle extra overhead. We assume that each shared structure can be *assertively accessed*. Assertive access for the I/O partitioned dual-ported DCache involves accessing the other port (the one not assigned to the core) assertively. Table VI.G

Table VI.3: Results with multiple sharings.

Units Shared	Perf. Degradation		Core Area
	Int Aps	FP Aps	Savings
Crossbar+FPU	0.97%	1.2%	23.1%
Crossbar+FPU+ICache	4.7%	3.9%	33.0%
Crossbar+FPU+DCache	6.1%	6.8%	45.2%
ICache+DCache	11.4%	7.6%	32.0%
Crossbar+FPU+ICache+DCache	11.9%	8.5%	55.1%

shows the resulting area savings and performance for various sharing combinations. We map the applications to the cores such that "friendly" threads run on the conjoined cores where possible. All performance numbers are for the worst case when all cores are busy with threads.

The combination with all four types of sharing results in 38.1% core-area savings (excluding crossbar savings). In absolute terms, this is equivalent to the area occupied by 3.76 cores. *If crossbar savings are included, then the total area saved is equivalent to 5.14 times the area of a core.* We observed a 11.9% degradation for integer workloads and 8.5% degradation for floating-point workloads. Note that the degradation is significantly less than the sum of the individual degradation values that we observed for each kind of sharing. This is because a stall due to one bottleneck often either tolerates or obviates a stall due to some other bottleneck.

Another attractive configuration utilizes only FPU and crossbar sharing. This configuration provides a 23.1% reduction in core area while degrading performance by around 1% in the worst case with all cores busy, and provides the highest marginal utility for sharing. This configuration also has the advantage of

being simpler to implement than configurations that share caches.

The results in Table VI.G show that conjoined-core architectures can give superior computational efficiency over conventional non-conjoined cores. The area savings they give can be used to provide either reduced die area and hence increased yield, or can be leveraged to provide a significant increase in performance by implementing more cores in the same area.

Finally, besides providing an area efficiency advantage, conjoining can also result in more power-efficient computation. Since memory cells can be engineered to have low leakage, leakage power is primarily a function of the amount of high-performance logic. Thus by sharing FPUs and/or the peripheral logic of caches, the number of logic circuits and hence the leakage power of a multiprocessor can be significantly reduced. Moreover, dynamic power per instruction can also be reduced since the crossbar interconnect lengths that computation must traverse can be reduced by reducing the area of the cores.

VI.H Acknowledgment

The text of Chapter VI is in part a reprint of the material as it appears in the proceedings of the Thirty-seventh International Symposium on Microarchitecture (pp195-206, December 2004). The dissertation author was the primary researcher and author and the co-authors involved in the submission directed the research which forms the basis for Chapter VI.

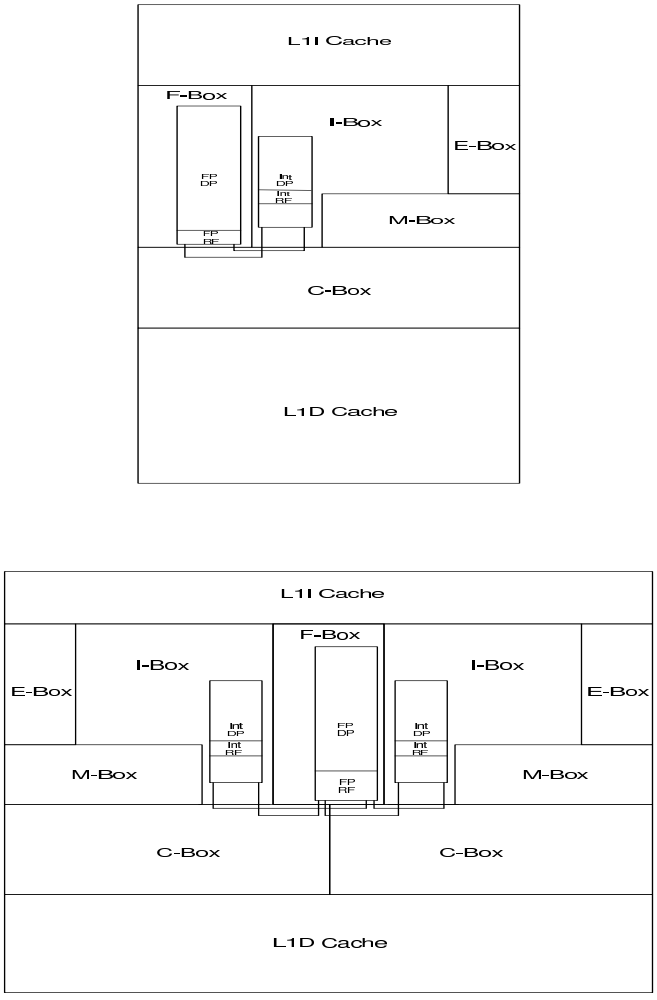

Figure VI.2: (a)Floorplan of the original core (b)Layout of a conjoined-core pair, both showing FPU routing. Routing and register files are schematic and not drawn to scale

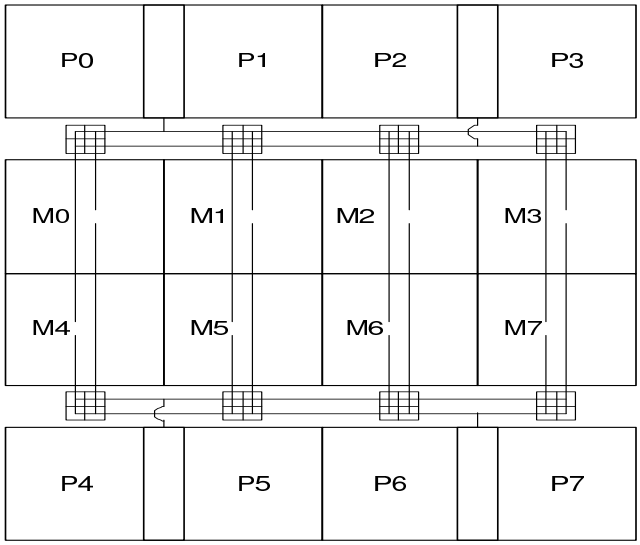

Figure VI.3: A die floorplan with crossbar sharing

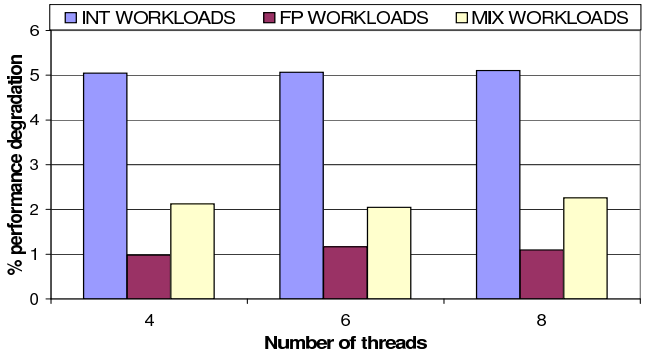

Figure VI.4: Impact of ICache sharing for various threading levels

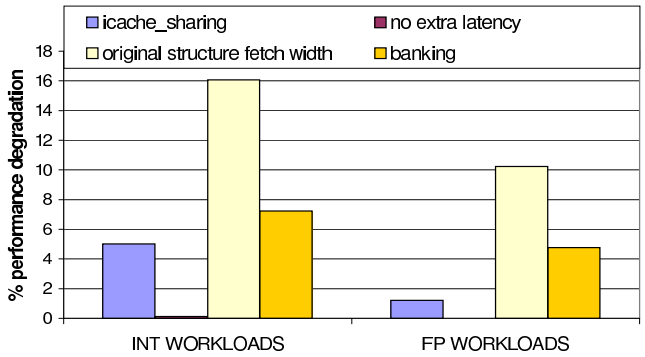

Figure VI.5: ICache sharing when no extra latency overhead is assumed, cache structure bandwidth is not doubled, and cache is doubly banked

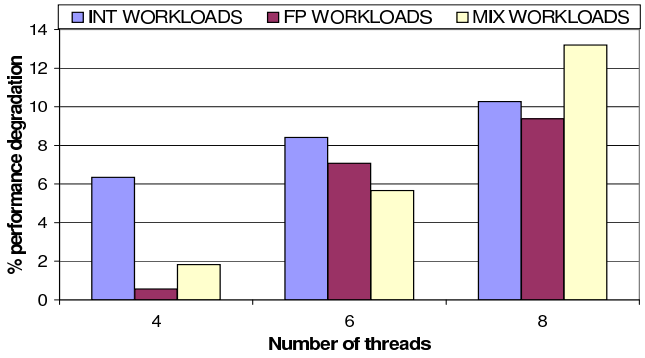

Figure VI.6: Impact of Dcache sharing for various threading levels

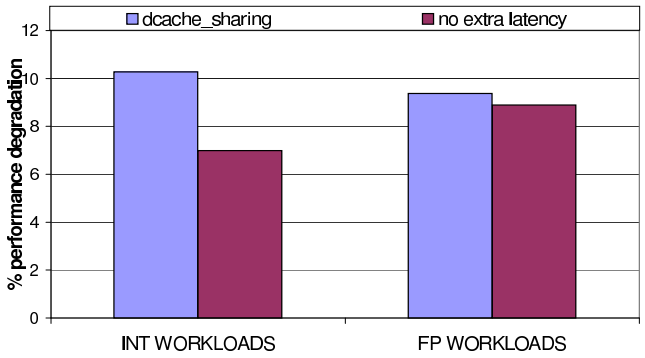

Figure VI.7: DCache sharing when no extra latency overhead is assumed

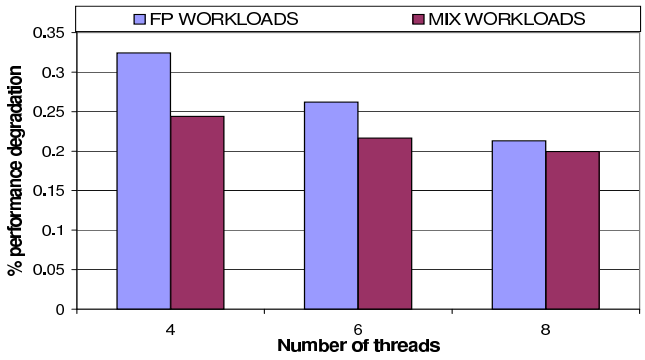

Figure VI.8: Impact of FPU sharing for various threading levels

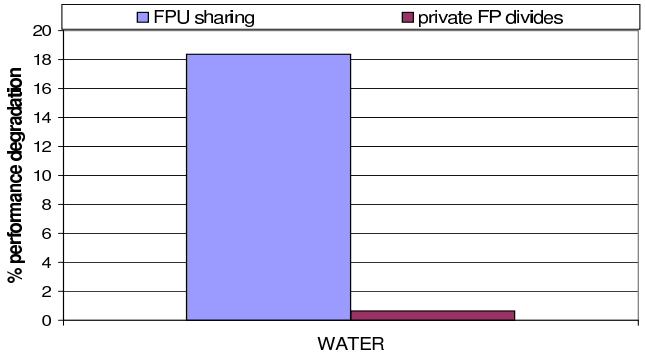

Figure VI.9: Impact of private FP divide sub-units

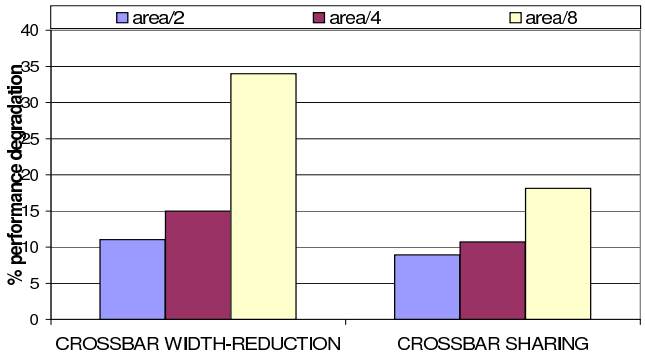

Figure VI.10: Reducing crossbar area through width reduction and port sharing

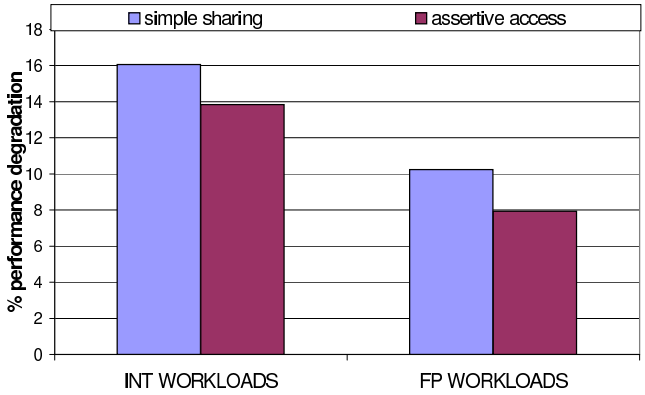

Figure VI.11: ICache assertive access results when the original structure bandwidth is not doubled

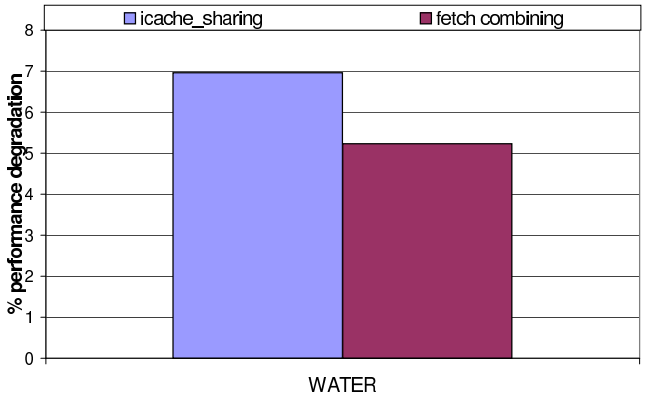

Figure VI.12: Fetch-combining results

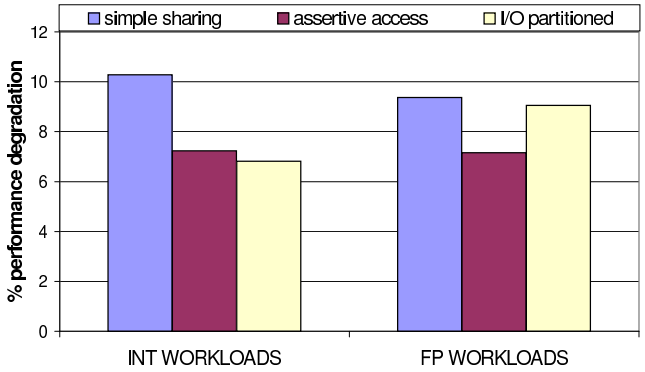

Figure VI.13: Effect of assertive access and static assignment

VII

The Interconnect Problem and the Need for Co-design

This chapter examines the area, power, performance, and design issues for the on-chip interconnects on a chip multiprocessor, attempting to present a comprehensive view of a class of interconnect architectures. It shows that the design choices for the interconnect have significant effect on the rest of the chip, potentially consuming a significant fraction of the real estate and power budget. This research shows that designs that treat interconnect as an entity that can be independently architected and optimized ("multi-core oblivious") would not arrive at the best multi-core design. Several examples are presented showing the need for a holistic approach to design (or careful co-design). For instance, increasing interconnect bandwidth requires area that then constrains the number of cores or cache sizes, and does not necessarily increase performance. Also, shared level-2 caches become significantly less attractive when the overhead of the resulting crossbar is accounted for. A hierarchical bus structure is examined which negates some of the performance costs of the assumed baseline architecture.

VII.A Related Work

There have been several proposals and implementations of high-performance chip multiprocessor architectures [26, 59, 68, 69]. The proposed interconnect for Piranha [26] was a fast, high-bandwidth switch. Cores in Hydra [59] are connected to the L2 cache through a crossbar. In both cases, the L2 cache is fully shared. IBM Power4 [68] has two cores sharing a triply-banked L2 cache. Connection is through a crossbar-like structure called the CIU (core-interface unit).

In this chapter, we consider bus-based and crossbar-based interconnections to illustrate the value of holistic design. There have been recent proposals for packet-based on-chip interconnection networks [61, 36, 107], *i.e.* networks where data is sent in form of packets that are reassembled at at the destination. Packet-based networks structure the top level wires on a chip and facilitate modular design. Modularity results in enhanced control over electrical parameters and hence can result in higher performance or reduced power consumption. These interconnections can be highly effective in particular environments where most communication is local, explicit core-to-core communication. However, the cost of distant communication is high. Due to their scalability, these architectures are attractive for a large number of cores. The crossover point where these architectures become superior to the more conventional interconnects studied in this chapter is not clear, and probably depends on implementation details.

There is a large body of related work evaluating tradeoffs between bus-based and scalable shared memory multiprocessors, in the context of conventional (multiple-chip) multiprocessors. Some earlier implementations of the interconnection networks for multiprocessors have been described in [47, 133, 96, 113, 109, 48, 16, 94, 21]. However, on-chip interconnects have different sets of tradeoffs and design issues. We will show that on-chip interconnects can often affect the number, size, and design of cores and memory, and vice versa. Thus, the

conclusions of those prior studies must be re-evaluated in the context of on-chip multiprocessors with on-chip interconnects.

VII.B Interconnection Mechanisms

In this section, we detail three interconnection mechanisms that may serve particular roles in on-chip interconnect hierarchy – a shared bus fabric (SBF) that provides a shared connection to various modules that can source and sink coherence traffic, a point-to-point link (P2P link) that connects two SBFs in a system with multiple SBFs, and a crossbar interconnection system. In the subsequent sections, we will demonstrate the need for co-design using these mechanisms as a baseline.

Many different modules may be connected to these fabrics, which use them in different ways. But from the perspective of the core, an L2 miss goes out over the SBF to be serviced by higher levels of the memory hierarchy, another L2 on the same SBF, or possibly an L2 on another SBF connected to this one by a P2P link. If the core shares L2 cache with another core, there is a crossbar between the cores/L1 caches and the shared L2 banks. Our initial discussion of the SBF in this section assumes private L2 caches.

The results in this chapter are derived from a detailed model of a complex system, which are described in the next few sections.

VII.B.1 Shared Bus Fabric

A Shared Bus Fabric is a high speed link needed to communicate data between processors, caches, IO, and memory within a CMP system in a coherent fashion. It is the on-chip equivalent of the system bus for snoop-based shared memory multiprocessors [47, 133, 96]. We model a MESI-like snoopy write-invalidate protocol with write-back L2s for this study [24, 68]. Therefore, the

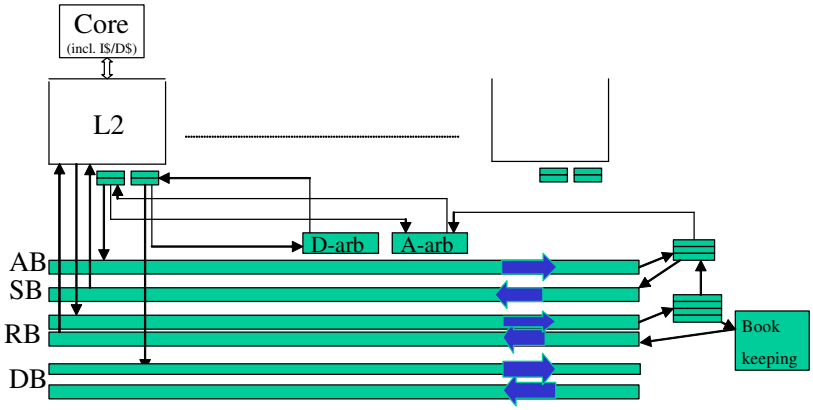

Figure VII.1: The assumed shared bus fabric for our interconnection study

SBF needs to support several coherence transactions (request, snoop, response, data transfer, invalidates, etc.) as well as arbitrate access to the corresponding buses. Due to large transfer distances on the chip and high wire delays, all buses must be pipelined, and therefore unidirectional. Thus, these buses appear in pairs; typically, a request traverses from the requester to the end of one bus, where it is queued up to be re-routed (possibly after some computation) across a broadcast bus that every node will eventually see, regardless of their position on the bus and distance from the origin. In the following discussion a bidirectional bus is really a combination of two unidirectional pipelined buses.

We are assuming, for this discussion, all cores have private L1 and L2 caches, and that the shared bus fabric connects the L2 caches (along with other units on the chip and off-chip links) to satisfy memory requests and maintain coherence. Below we describe a typical transaction on the fabric.

Typical transaction on the SBF

A load that misses in the L2 cache will enter the shared bus fabric to be serviced. First, the requester (in this case, one of the cores) will signal the *central address arbiter* that it has a request. Upon being granted access, it sends the request over an *address bus* (AB in Figure VII.1). Requests are taken off the end of the address bus and placed in a snoop queue, awaiting access to the *snoop bus* (SB). Transactions placed on the snoop bus cause each snooping node to place a response on the *response bus* (RB). Logic and queues at the end of the response bus collect these responses and generate a broadcast message that goes back over the response bus identifying the action each involved party should take (e.g., source the data, change coherence state). Finally, the data is sent over a bidirectional *data bus* (DB) to the original requester. If there are multiple SBFs (e.g., connected by a P2P link), the address request will be broadcast to the other SBFs via that link, and a combined response from the remote SBF returned to the local one, to be merged with the local responses.

Note that the above transactions are quite standard for any shared memory multiprocessor implementing a snoopy write-invalidate coherence protocol [24].

Elements of the SBF

The composition of the SBF allows it to support all the coherence transactions mentioned above. We now discuss the primary buses, queues and logic that would typically be required for supporting these transactions. Figure VII.1 illustrates a typical SBF. Details of the modeled design are based heavily on the shared bus fabric in the Power 5 multi-core architecture [69].

Each requester on the SBF interfaces with it via *request* and *data queues*. It takes at least one cycle to communicate information about the occupancy of

the request queue to the requester. The request queue must then have at least two entries to maintain the throughput of one request every cycle. Similarly, all the units that can source data need to have data queues of at least two entries. Requesters connected to the SBF include cores, L2 and L3 caches, IO devices, memory controllers, and non-cacheable instruction units.

All requesters interface to the fabric through an arbiter for the address bus. The minimum latency through the arbiter depends on (1) the physical distance from the central arbiter to the most distant unit, and (2) the levels of arbitration. Caches are typically given higher priority than other units, so arbitration can take multiple levels based on priority. Distance is determined by the actual floorplan. Since the address bus is pipelined, the arbiter must account for the location of a requester on the bus in determining what cycle access is granted. Overhead of the arbiter includes control signals to/from the requesters, arbitration logic and some latches.

After receiving a grant from the central arbiter, the requester unit puts the address on the *address bus*. Each address request goes over the address bus and is then copied into multiple queues, corresponding to outgoing P2P links (discussed later) and to off-chip links. There is also a local *snoop queue* that queues up the requests and participates in the arbitration for the local *snoop bus*. Every queue in the fabric incurs at least one bus cycle of delay. The minimum size of each queue in the interconnect (there are typically queues associated with each bus) depends on the delay required for the arbiter to stall further address requests if the corresponding bus gets stalled. Thus it depends on the distance and communication protocol to the device or queue responsible for generating requests that are sinked in the queue, and the latency of requests already in transit on the bus. We therefore compute queue size based on floorplan and distance.

The snoop bus can be shared, for example by off-chip links and other SBFs, so it also must be accessed via an arbiter, with associated delay and area overhead. Since the snoop queue is at one end of the address bus, the snoop bus must run in the opposite direction of the address bus, as shown in Figure VII.1. Each module connected to the snoop bus snoops the requests. Snooping involves comparing the request address with the address range allocated to that module (e.g., memory controllers) or checking the directory (tag array) for caches.

A response is generated after a predefined number of cycles by each snooper, and goes out over the *response bus*. The delay can be significant, because it can involve tag-array lookups by the caches, and we must account for possible conflicts with other accesses to the tag arrays. Logic at one end of the bidirectional response bus collects all responses and broadcasts a message to all nodes, directing their response to the access. This may involve sourcing the data, invalidating, changing coherence state, etc. Some responders can initiate a data transfer on a read request simultaneously with generating the snoop response, when the requested data is in appropriate coherence state. The responses are collected in queues. All units that can source data to the fabric need to be equipped with a data queue. A central arbiter interfacing with the data queues is needed to grant one of the sources access to the bus at a time.

Bidirectional data buses source data. They support two different data streams, one in either direction. Data bandwidth requirements are typically high.

It should be noted that designs are possible with fewer buses, and the various types of transactions multiplexed onto the same bus. However, that would require higher bandwidth (e.g., wider) buses to support the same level of traffic at the same performance, so the overheads are unlikely to change significantly. We assume for the purpose of this study that only the above queues, logic and buses form a part of the SBF and contribute to the interconnection latency, power, and

area overheads.

VII.B.2 P2P Links

If there are multiple SBFs in the system, the connection between the SBFs is accomplished using P2P links. Multiple SBFs might be required to increase bandwidth, decrease signal latencies, or to ease floorplanning (all connections to a single SBF must be on a line). For example, if a processor has 16 cores as shown in Figure VII.3, it becomes impossible to maintain die aspect ratio close to 1 unless there are two SBFs each supporting 8 cores.

Each P2P link should be capable of transferring all kinds of transactions (request/response/data) in both directions. Each P2P link is terminated with multiple queues at each end. There needs to be a queue and an arbiter for each kind of transaction described above.

VII.B.3 Crossbar Interconnection System

The previous section assumed private L2 caches, with communication and coherence only occurring on L2 misses. However, if our architecture allows two or more cores to share L2 cache banks, a high bandwidth connection is required between the cores and the cache banks. This is typically accomplished by using a crossbar. It allows multiple core ports to launch operations to the L2 subsystem in the same cycle. Likewise, multiple L2 banks are able to return data or send invalidates to the various core ports in the same cycle.

The crossbar interconnection system consists of crossbar links and crossbar interface logic. A crossbar consists of address lines going from each core to all the banks (required for loads, stores, prefetches, TLB misses), data lines going from each core to the banks (required for writebacks) and data lines going from every bank to the cores (required for data reload as well as invalidate addresses).

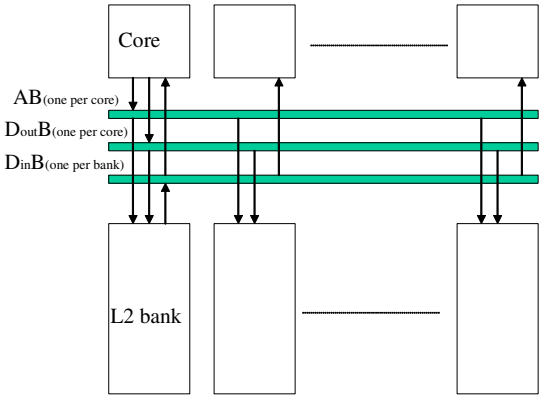

Figure VII.2: A typical crossbar

A typical implementation, shown in Figure VII.2, consists of one address bus per core from which all the banks feed. Each bank has one outgoing data bus from which all the cores feed. Similarly, corresponding to each write port of a core is an outgoing data bus that feeds all the banks.

Crossbar interface logic presents a simplified interface to the instruction fetch unit and the Load Store Unit in the cores. It typically consists of a load queue corresponding to each core sharing the L2. The load queue sends a request to the L2 bank appropriate to the request, where it is enqueued in a bank load queue (BLQ) (one per core for each bank to avoid conflict between cores accessing the same bank). The BLQs must arbitrate for the L2 tags and arrays, both among the BLQs, as well as with the snoop queue, the writeback queue, and the data reload queue — all of which may be trying to access the L2 at the same time. After L2 access (on a load request), the data goes through the reload queue, one per bank, and over the data bus back to the core. The above description of the crossbar interface logic is based on the crossbar implementation (also called core interface unit) in Power4 [68] and Power5 [69].

Note that even when the caches (or cache banks) are shared, an SBF is required to maintain coherence between various units in the CMP system.

Table VII.1: Design parameters for wires in different metal planes

Metal Plane Plane	Pitch (μm) (μm)	Signal Pitch (μm)	Repeater Spacing (mm)	Repeater Width (μm)	Latch Spacing (mm)	Latch Height (μm)	Channel Leakage per repeater (uA)	Gate Leakage per repeater (uA)
1X	0.2	0.5	0.4	0.4	1.5	120	10	2
2X	0.4	1.0	0.8	0.8	3.0	60	20	4
4X	0.8	2.0	1.6	1.6	5.0	30	40	8
8X	1.6	4.0	3.2	3.2	8.0	15	80	10

VII.C Modeling Area, Power, and Latency

Both wires and logic contribute to interconnect overhead. This section describes our methodology for computing various overheads for 65nm technology. The scaling of overheads with technology as well as other design parameters is discussed in Section VII.G.

VII.C.1 Wiring Area Overhead

We address the area overheads of wires and logic separately.

The latency, area, and power overhead of a metal wire depends on the metal layer used for that wire. Technology that we consider facilitates 10 layers of metal, 4 layers in 1X plane and 2 layers in the higher planes (2X, 4X and 8X) [74]. The 1X metal layers are typically used for macro-level wiring [74]. Wiring tracks in higher layers of metal are very expensive and only used for time-critical signals running over a considerable distance (several millimeters of wire).

We evaluate crossbar implementations for 1X, 2X and 4X metal planes where both data and address lines use the same metal plane. For our SBF evaluations, the address bus, snoop bus, and control signals always use the 8X plane. Response buses preferably use the 8X plane, but can use the 4X plane. Data buses can be placed in the 4X plane (as they have more relaxed latency considerations). All buses for P2P links are routed in the 8X plane.

The area occupied by a bus is determined by the number of wires times

the effective pitch of the wires times the length. We account for the case where the area of one bus (partially) subsumes the area of some other bus in a different plane. When buses are wired without logic underneath, repeaters and latches are placed under the buses without incurring any additional area overhead. However, when interconnection buses are routed over array structures (e.g. cache arrays, directories etc.), we account for the fact that the sub-arrays (or memory macros) have to be shifted to make space for the placement of wire repeaters and latches. In this case the minimal repeater and latch spacing is an essential parameter determining the area overhead. We believe that 4X and 8X wires can be routed over memory arrays. However, in Section VII.F, we also evaluate routing 1X and 2X over memory, even though we believe it to be technologically more difficult.

Table VII.C shows the signal wiring pitch for wires in different metal planes for 65nm. These pitch values are estimated by conforming to the considerations mentioned in [125].

The table also shows the minimum spacing for repeaters and latches as well as their heights for computing the corresponding area overheads. We model the height of the repeater macro to be 15 μm. The height of the latch macro given in the table includes the overhead of the local clock buffer and local clock wiring, but excludes the overhead of rebuffering the latch output which is counted separately. The values in Table VII.C are for a bus frequency of 2.5 GHz and a bus voltage of 1.1 V. Analysis for different bus frequencies can be found in Section VII.G.

VII.C.2 Logic Area Overhead

Area overhead due to interconnection-related logic comes primarily from queues. Queues are assumed to be implemented using latches. We estimate the area of a 1-bit latch used for implementing the queues to be 115 μm^2 for 65nm

technology [132]. This size includes the local clock driver and the area overhead of local clock distribution. We also estimated that there is a 30% overhead in area due to logic needed to maintain the queues (such as head, tail pointers, queue bypass, overflow signaling, request/grant logic, etc.) [33].

The interconnect architecture can typically be designed such that buses run over interconnection-related logic. The area taken up due to wiring is usually big enough that it (almost) subsumes the area taken up by the logic.

Because queues overwhelmingly dominate the logic area, we ignore the area (but not latency) of multiplexors and arbiters. It should be noted that the assumed overheads can be reduced by implementing queues using custom arrays instead of latches.

VII.C.3 Power

Power overhead comes from wires, repeaters, and latches. For calculating dynamic dissipation in the wires, we optimistically estimate the capacitance per unit length of wire (for all planes) to be 0.2 pF/mm [66]. Repeater capacitance is assumed to be 30% of the wire capacitance [15]. The dynamic power per latch is estimated to be 0.05 mW per latch for 2.5 GHz at 65 nm [132]. This includes the power of the local clock buffer and the local clock distribution, but does not include rebuffering that typically follows latches.

Total dynamic power of a bus would depend on utilization of the bus as well as the efficacy of clock gating. We assume that in 30% of the unused cycles the latches will be gated off, to be consistent with clock gating efficiencies typically quoted for high-end microprocessors [33]. Even though the cycles of inactivity are easier to predict in the fabric than in the core, the physical distance between latches of the neighboring clock stages is much larger in the fabric, which complicates the timing of the clock gate signals.

Repeater leakage is computed using the parameters given in Table VII.C. For latches, we estimate channel leakage to be 20uA per bit in all planes (again not counting the repeaters following a latch). Gate leakage for a latch is estimated to be 2uA per bit in all planes [15]. For computing dynamic and leakage power in the queues, we use the same assumptions as for the wiring latches.

VII.C.4 Latency

The latency of a signal traveling through the interconnect is primarily due to wire latencies, wait time in the queues for access to a bus, arbitration latencies, and latching that is required between stages of interconnection logic. Latency of wires is determined by the spacing of latches. Spacing between latches for wires is given in Table VII.C.

Arbitration can take place in multiple stages (where each stage involves arbitration among the same priority units) and latching needs to be done between every two stages. For 65 nm technology, we estimate that no more than four units can be arbitrated in a cycle. The latency of arbitration also comes from the travel of control between a central arbiter and the interfaces corresponding to request/data queues. Other than arbiters, every time a transaction has to be queued, there is at least a bus cycle of delay — additional delays depend on the utilization of the outbound bus.

VII.D Modeling Multi-core Architectures

For this study, we consider a stripped version of out-of-order Power4-like cores [68]. We determine the area taken up by such a core at 65nm to be $10mm^2$. The area and power determination methodology is similar to the one presented in Chapter IV. The power taken up by the core is determined to be 10W, including leakage.

For calculating on-chip memory sizes, we use the Power5 cache density, as measured from die photos [69], scaled to 65nm. We determine it to be 1 bit per square micron, or $0.125MB/mm^2$. For the purpose of this study, we consider L2 caches as the only type of on-chip memory. We do not assume off-chip L3 cache, but in 65nm systems, it is likely that L3 chips would be present as well (the number of L3 chips would be limited, however, due to the large number of pins that every L3 chip would require), but we account for that effect using somewhat optimistic estimates for effective bandwidth and memory latency. Off-chip bandwidth was modeled carefully based on pincount [15] and number of memory channels (Rambus RDRAM interface was assumed).

Our models include the effects of not only L2 banks and memory controllers, but also DMA controllers and Non-cacheable Instruction Units (NCUs) on the die that can generate transactions. NCUs handle instructions like syncs, eieios (enforce inorder execution of I/Os), TLB invalidates, partial read/writes, etc., that are not cached and are directly put on the fabric. Each core has a corresponding NCU. The assumptions about these units are taken from Power 4 and Power 5 [68, 69] designs. We simulate a MESI-like [106, 68] coherence protocol, and all transactions required by that protocol are faithfully modeled in our simulations. We also model weak consistency [42] for the multiprocessor, so there is no impact on CPI due to the latency of stores and writebacks.

For performance modeling, we use a combination of detailed functional simulation and queuing simulation tools [92]. The functional simulator is used for modeling the memory subsystem as well as the interconnection between modules. It takes instruction traces from an SMP system as input and generates coherence statistics for the modeled memory/interconnect sub-system. The queuing simulator takes as input the modeled subsystem, its latencies, coherence statistics and the inherent CPI of the modeled core assuming perfect L2. It then generates

the CPI of the entire system, accounting for real L2 miss rates and real interconnection latencies. Traffic due to syncs, speculation, and MPL (message passing library) effects is accounted for as well. The tools and our interconnection models have been validated against a real, implemented design.

The cache access times are calculated using assumptions similar to those made in CACTI [119]. Memory latency is set to 500 cycles. The average CPI of the modeled core over all the workloads that we use, assuming perfect L2, is measured to be 2.65. Core frequency is assumed to be 5GHz for the 65nm studies. Buses as well as the L2 are assumed to be clocked at half the CPU speed.

VII.D.1 Workload

All our performance evaluations have been done using commercial workloads, including TPC-C, TPC-W, TPC-H, Notesbench and others further described in [92]. The server workloads represent such market segments as on-line transaction processing (OLTP), business intelligence, enterprise resource planning, web serving, and collaborative groupware. These applications are large and function rich; they use a large number of operating system services and access large databases. These characteristics make the instruction and data working sets large. These workloads are also inherently multiuser and multitasking, with frequent read-write sharing.

We use PowerPC instruction and data reference traces of the workloads running under AIX. The traces are taken in a non-intrusive manner by attaching a hardware monitor to a processor [43, 92]. This enables the traces to be gathered while the system is fully loaded with the normal number of users, and captures the full effects of multitasking, data sharing, interrupts, etc. These traces contain even DMA instructions and non-cacheable accesses.

VII.E Shared Bus Fabric: Overheads and Design Issues

This section examines the various overheads of the shared bus fabric, and the implications this has for the entire multi-core architecture. We examine floorplans for several design points, and characterize the impact on the overall design and performance of the processor of the area, power, and latency overheads. This section demonstrates that the overheads of the SBF can be quite significant. It also illustrates the tension between the desire to have more cores, more cache, and more interconnect bandwidth, and how that plays out in total performance.

In this section, we assume private L2 caches and that all the L2s (along with NCUs, memory controllers, and IO Devices) are connected using a shared bus fabric. We consider architectures with 4, 8, and 16 cores. Total die area is assumed to be constant at $400mm^2$ due to yield considerations. Hence, the amount of L2 per core decreases with increasing number of cores. For 4, 8 and 16 cores, we evaluate multiple floorplans and choose those that maximized cache size per core while maintaining a die aspect ratio close to 1. In the default case, we consider the width of the address, snoop, response and data buses of the SBF to be 7, 12, 8, 38 (in each direction) bytes respectively — these widths are determined such that no more than 0.15 requests get queued up, on average, for the 8 core case. We also evaluate the effect of varying bandwidths. We can lay out 4 or 8 cores with a single SBF, but for 16 cores, we need two SBFs connected by a P2P link. In that case, we model two half-width SBFs and a 76 byte wide P2P link. Figure VII.3 shows the floorplans arrived at for the three cases. The amount of L2 cache per core is 8MB, 3MB and 0.5MB for 4, 8 and 16 core processors respectively. It must be mentioned that the 16-core configuration is somewhat unrealistic for this technology as it would result in inordinately high power consumption. However, we present the results here for completeness

reasons.

Wires are slow and hence cannot be clocked at very high speeds without inserting an inordinately large number of latches. For our evaluations, the SBF buses are cycled at half the core frequency.

VII.E.1 Area

The area consumed by the shared bus fabric comes from wiring and interconnection-related logic, as described in Section VII.C. Wiring overhead depends on the architected buses and the control wires that are required for flow control and arbitration. Control wires are needed for each multiplexor connected to the buses and signals to and from every arbiter. Flow control wires are needed from each queue to the units that control traffic to the queue.

Since data buses run in the 4X plane, the area taken up by (76-byte) data buses is calculated as $2\mu m \times 76 \times 8 \times length$, which results in $24.32mm^2$. Address (7 bytes), snoop (12 bytes), response buses (8 bytes) as well as control wires (at least 198 of them for the stated control architecture for 8 cores) that run in the 8X plane can be routed over the data buses. In this case, the area overhead of the data buses is subsumed by that of the remaining buses. For 16 cores, the P2P links do not result in direct area overhead because they can be routed over L2 caches. However, reduction in the cache density (due to bus latches and repeaters) does result in an area overhead.

Figure VII.4 shows the wiring area overhead for various processors. The graph shows the area overhead due to architected wires, control wires, and the total. We see that area overhead due to interconnections in a CMP environment can be significant. For the assumed die area of $400mm^2$, area overhead for the interconnect with 16 cores is 13%. Area overhead for 8 cores and 4 cores is 8.7% and 7.2% of the die area, respectively. Considering that each core is $10mm^2$, the

area taken up by the SBF is sufficient to place 3-5 extra cores or 4-6 MB of extra cache.

The graph also shows that area overhead increases quickly with the number of cores. This result assumes constant width architected buses, even when the number of cores is increased. If the effective bandwidth per core is kept constant, overhead would increase even faster.

The overhead due to control wires is high. Control takes up at least 37% of SBF area for 4 cores and at least 62.8% of the SBF area for 16 cores. This is because the number of control wires grows linearly with the number of connected units, in addition to the linear growth in the average length of the wires. Reducing SBF bandwidth does not reduce the control area overhead, thus it constrains how much area can be regained with narrower buses. Note that this argues against very lightweight (small, low performance) cores on this type of chip multiprocessor, because the lightweight core does not amortize the incremental cost to the interconnect of adding each core. Interconnect area due to logic is primarily due to the various queues, as described in Section VII.C. Table VII.E.1 shows the area overhead due to interconnect-related logic and the corresponding breakdown. The area taken up by interconnection-related logic increases superlinearly with the number of connected units (note that the number of connected units is 14, 22 and 38 respectively for 4, 8 and 16 core processors). When going from 8 to 16 cores, the logic-area overhead jumps because the queues are required to support two SBFs and a P2P link. Note, however, that the logic can typically be placed underneath the SBF wires. Thus, under these assumptions the SBF area is dominated by wires, but only by a small amount.

Table VII.2: Interconnection-related Logic overhead

Number of cores	4	8	16
Number of data queue latches	28672	45056	92160
Number of request queue latches	3136	4928	8512
Number of snoop queue latches	336	336	896
Number of latches for response bus queue	1680	1680	6720
Total number of latches	33824	52000	108288
Area(in mm^2)	5.6	8.6	17.94

VII.E.2 Power

The power dissipated by the interconnect is the sum of the power dissipated by wires and the logic. Figure VII.5 shows a breakdown of the total power dissipation by the interconnect.

The graph shows that total power due to the interconnect can be significant. The interconnect power overhead for the 16-core processor is more than the combined power of two cores. It is equal to the power dissipation of one full core even for the 8 core processor. Power increases superlinearly with the number of connected units. This is because of the (at least linear) increase in the number of control wires as well as the (at least linear) increase in the number of queuing latches. There is also a considerable increase in the bus traffic with the growing number of cores. Half the power due to wiring is leakage (mostly from repeaters).

Contrary to popular belief, interconnect power is not always dominated by the wires. The power due to logic can be, as in this case, more than the power due to wiring.

VII.E.3 Performance

For the chip multiprocessor consisting of four cores, the end to end latency of the architected buses, except the data buses, would be 6 processor

cycles (as three latches would be required for 20mm wires running in the 8X plane). Similarly, the latency of the data buses (routed in the 4X plane) would be 12 processor cycles.

Address arbitration would take 8 processor cycles, but the request transfer between the central address arbiter and the request queues would take 8 processor cycles (two latches each way – we assume central arbiters to be placed around the middle of the chip). Every queue would take at least one bus cycle. Snoop response generation will take at least 20 processor cycles (cache tag access and two latches). Generating a final response will take 4 cycles (two latches). We also estimate that an 8MB cache can be accessed in 46 cycles (includes array access time, time to go through queues, arbitration overhead for getting access to the array, etc.; note that L2 is assumed to be cycled at half the core frequency). Accounting for all arbitration, bus latencies, queues, and the cache access, the minimum (no contention) latency of a load is 124 cycles for a four core multi-core. Even under these optimistic assumptions, the interconnect accounts for over half the total latency to the L2 cache.

Figure VII.6 shows the per-core performance for 4, 8 and 16 core architectures, both assuming no interconnection overhead (zero latency interconnection) and with interconnection overheads modeled carefully. Single-thread performance (even assuming no interconnection overhead) goes down as the number of cores increases due to the reduced cache size per core. If interconnect overhead is considered, then the performance decreases much faster. In fact, performance overhead due to interconnection is more than 10% for 4 cores, more than 13% for 8 cores and more than 26% for 16 cores.

In results to this point, we keep bus bandwidth constant. In Figure VII.7, we show the single-thread performance of a core in the 8 core processor case, when the width of the architected buses is varied by factors of two. The

graph also shows the real estate saved compared to the baseline. We see that with wide buses, the area costs are significant, and the incremental performance is minimal. On the other hand, with narrow buses, the area saved by small changes in bandwidth is small, but the performance impact is significant.

Alternatively, we could put the area saved through bandwidth reduction to use. We ran simulations that assume that we put that area back into the caches. We find that over certain ranges, if the bandwidth is reduced by small factors, the performance degradation can be recovered using bigger caches. For example, decreasing the bandwidth by a factor of 2 decreases the performance by 0.57%. But it saves $8.64mm^2$. This can be used to increase the per-core cache size by 135KB. When we ran simulations using new cache sizes, we observed a performance improvement of 0.675%. Thus, *we can decrease bus bandwidth and improve performance* (if only by small margins in this example), because the resulting bigger caches protect the interconnect from a commensurate increase in utilization. On the other hand, when bandwidth is decreased by a factor of 8, performance decreases by 31%, while the area it saves is $15.12mm^2$. The area savings is sufficient to increase per core cache size by only 240KB. The increase in cache size was not sufficient to offset the performance loss in this case. Similarly, when doubling interconnect bandwidth over our baseline configuration, total performance decreased by 1.2% due to the reduced cache sizes.

This demonstrates the importance of co-designing the interconnect and memory hierarchy. It is neither true that the biggest caches nor the widest interconnect give the best performance; designing each of these subsystems independently is unlikely to result in the best design. Similarly, the core itself should be co-architected with the caches and interconnect, but for this study we treat the cores as a constant. Chapter V presented our explorations with core designs.

VII.F Shared Caches and the Crossbar

The previous section presented evaluations with private L1 and L2 caches for each core, but many proposed chip multiprocessors have featured shared L2 caches, connected with crossbars. Shared caches allow the cache space to be partitioned dynamically rather than statically, typically improving overall hit rates. Also, shared data does not have to be duplicated. To fully understand the tradeoffs between private and shared L2 caches, however, we find that it is absolutely critical that we account for the impact of the interconnect.

VII.F.1 Area and power overhead

The crossbar, shown in Figure VII.2, connects cores (with L1 caches) to the shared L2 banks. The data buses are 32 bytes while the address bus is 5 bytes. Lower bandwidth solutions were found to adversely affect performance and render sharing highly unfruitful. In this section we focus on an 8-core processor with 8 cache banks, giving us the options of 2-way, 4-way, and full (8-way) sharing of cache banks. Crossbar wires can be implemented in the 1X, 2X or 4X plane. For almost 2x reduction in the latency, the wire thickness doubles every time we go to a higher metal plane.

Figure VII.8 shows the area overhead for implementing different mechanisms of cache sharing. The area overhead is shown for two cases – one where the crossbar runs between cores and L2, and the other where the crossbar can be routed over L2. When the crossbar is placed between the L2 and the cores, interfacing is easy, but all wiring tracks result in area overhead. When the crossbar is routed over L2, area overhead is only due to reduced cache density to accommodate repeaters and latches. However, the implementation is relatively complex as vertical wires are needed to interface the core with the L2. We show the results assuming that the L2 density is kept uniform (i.e. even if repeaters/latches are

dropped only over the top region of the cache, sub-arrays are displaced even in the other regions to maintain uniform density).

Cache sharing carries a heavy area overhead. If the total die area is around 400 mm^2, then the area overhead for an acceptable latency (2X) is 11.4% for 2-way sharing, 22.8% for four-way sharing and 46.8% for full sharing (nearly half the chip!). Overhead increases as we go to higher metal layers due to increasing signal pitch values. When we assume that the crossbar can be routed over L2, area overhead is still substantial; however, in that case it improves as we move up in metal layers. At low levels the number of repeater/latches, which must displace cache, is highest.

The point of sharing caches is to get the effect of having more cache space. In this case, the cores can gain significant real cache space by *foregoing sharing*, raising doubts about whether sharing has any benefit. This is seen even more clearly in the next subsection.

The high area overhead again suggests that issues of interconnect/cache/core co-design must be considered. For crossbars sitting between cores and L2, just two-way sharing results in an area overhead equivalent to more than the area of two cores. Four-way sharing results in an area overhead of 4 cores. An 8-way sharing results in an area overhead of 9 cores. If the same area were devoted to caches, one could instead put 2.75 MB, 5.5 MB and 11.6 MB of extra caches, respectively.

Figure VII.9 shows the corresponding power overhead. A breakdown is also provided for various sources of power dissipation. The graph shows that power overhead due to crossbars is very significant. The overhead can be more than the power taken up by three full cores for a completely shared cache and more than the power of one full core for 4-way sharing. Even for 2-way sharing, power overhead is more than half the power dissipation of a single core. Hence,

even if power is the primary constraint, the benefits of the shared caches must be weighed against the possibility of more cores or significantly more cache.

Leakage is a smaller fraction of the total power for crossbars than for SBFs; however, it is still significant — 18-20% depending on the metal layer.

VII.F.2 Performance

Because of the high area overhead for cache sharing, the total amount of on-chip caches decreases with sharing. We performed our evaluations for the most tightly packed floorplans that we could find for 8-core processors with different levels of sharing. When the crossbar wires are assumed to be routed in the 2X plane between cores and L2, total cache size is 20MB, 14MB and 4MB respectively for 2-way, 4-way and full sharing. When crossbar is assumed to be routed over L2 (and assuming uniform cache density), the total cache size was 22MB for 4X and 18.2MB for 2X. We also conducted experiments assuming no crossbar area overhead to isolate the benefit of sharing. Figure VII.10 shows results for a fixed on-chip cache size (i.e. assuming no crossbar area overhead – crossbar latency overhead is assumed, however). Figure VII.11 presents the results for a fixed die area and cache sizes varied accordingly (i.e. taking into account crossbar area overhead).

Figure VII.10 shows that cache sharing, in general, results in higher performance than just having private caches if interconnection area overheads are not considered. It also shows that crossbar performance is very sensitive to the metal plane used to implement it.

Figure VII.11, assumes a constant die area and considers interconnection area overhead. It shows that performance, even without considering the interconnection latency overhead (and hence purely the effect of cache sharing), either does not improve or improves only by a slight margin. This is due to

reduced size of on-chip caches to accommodate the crossbar. If interconnect latencies are accounted for (higher sharing means longer crossbar latencies), sharing degrades performance even between two cores. Note that in this case, the conclusion reached ignoring interconnect area effects is opposite that reached when those effects are considered.

Note that performance loss due to increased L2 hit latency can be mitigated by using L2 latency hiding techniques, like overlapping of L2 accesses or prefetching. Also, crossbar area overhead can be reduced (and hence performance improved) by implementing caches with non-uniform density. In fact, we observed that two-way and four-way sharing improves performance for the 4X crossbar implementation if the crossbar is routed over memory and the L2 is allowed to have non-uniform density. Sharing might also result in benefit for other workloads with different working set and sharing behavior. Also, the smaller the relative frequency of the bus, the less prohibitive it is to implement L2 with large-scale sharing. For example, if the crossbars are routed over L2 and are clocked at one-fourth the core frequency, we observe that two-way and four-way sharing improves performance on the 4X metal plane.

However, our results definitely show that having shared caches becomes *significantly less desirable* than previously accepted if interconnection overheads are considered. We believe that the conclusion holds, in general, for uniform access time caches, and calls for evaluation of caching strategies with careful consideration of interconnect overheads. Further analysis needs to be done for intelligent NUCA (non-uniform cache access) caches [78].

The results also emphasize the need for holistic design as the best processor designs are not the ones that ignored interconnection costs.

VII.G Scaling with Technological Parameters

All results to this point in the chapter have been carefully parameterized to the 65 nm process. This section extends those results to future technologies, and also considers the effect of deeper pipelining (i.e., faster CPU clocks).

As technology shrinks, the repeater and latch spacings decrease, thereby increasing the latency overhead due to wires. Going to deeper pipelines/faster clocks also decreases repeater and latch spacings, as well as increased logic overhead (in terms of cycles). Technology scaling and deeper pipelines also increase perceived memory and cache access times. We used ITRS scaling trends to compute the latencies for 45nm and 32nm technologies. Figure VII.12 shows the results for a fixed die area (400 mm^2) for 65nm, 45nm and 32nm. The results are shown for the 8-core case. Each core has a private cache of 3MB, 6MB and 10MB respectively for the three technologies. We assume the base CPI of the core (apart from memory behavior) to remain the same for this study.

Figure VII.12 shows that with deeper pipelines for the same technology, the interconnection overhead on performance increases. For example at 65 nm, if the pipeline depth is changed from 26FO4 to 10FO4, the interconnection overhead increases by 30%. But for 45 nm, the corresponding change is 55%, and for 32nm, 44.4%. Also, for the same die area, and for the same pipeline depth, interconnection overhead increases with technology. Overhead increase is due to increased latencies, increased arbitration overhead, etc. Note that increased cache size for better technologies leads to higher hit rates as well as reduced traffic on the interconnect, but *it is not sufficient to hide the effect of the increased interconnect latencies*. It must be mentioned, however, that overall performance improves because of higher frequencies.

VII.H An Example Holistic Approach to Interconnection

The intent of this section is to apply one lesson learned from the high volume of data gathered in the research described in this chapter. Our interconnect architectures to this point were highly driven by layout. The SBF spans the width of the chip, allowing us to connect as many units as possible in a straight line across the chip. However, the latency overheads of a long SBF encourage us to consider alternatives. This section describes a more hierarchical approach to interconnects, which can exploit shorter buses with shorter latencies when traffic remains local. We will be considering the 8-core processor again.

The effectiveness of such an approach will depend on the probability that an L2 miss is serviced on a local cache (an L2 connected to the same SBF), rather than a cache on a remote SBF. We will refer to this probability as *"thread bias"*. A workload with high thread bias means that we can identify and map "clusters" of threads that principally communicate with each other on the same SBF.

In this section, we split the single SBF that spans the chip into two SBFs, with a P2P link between them. Local accesses benefit from decreased distances. Remote accesses suffer because they travel the same distances and see additional queuing and arbitration overheads between interconnects.

Figure VII.14 shows the performance of the split SBF for various thread bias levels. The SBF is split vertically into two, such that each SBF piece now supports 4 cores, 4 NCUs, 2 memory controllers and 1 IO Device. The X-axis shows the thread bias in terms of the fraction of misses satisfied by an L2 connected to the same SBF. A 25% thread bias means that one out of four L2 misses are satisfied by an L2 connected to the same SBF piece. These results are obtained through statistical simulation by synthetically skewing the traffic pattern.

The figure also shows the system performance for a a single monolithic SBF (the one used in previous sections). As can be seen, if thread bias is more than 17%, the performance of the split SBF can overtake performance of a monolithic SBF. Note that 17% is lower than the statistical probability of locally satisfying an L2 miss assuming uniform distribution (3/7). Hence, the split SBF, in this case, is clearly a good idea.

VII.I Acknowledgment

The text of Chapter VII is in part a reprint of the material as appears in the proceedings of the Thirty-second International Symposium on Computer Architecture (pp408-419, June 2005). The dissertation author was the primary researcher and author and the co-authors involved in the submission directed, supervised, and assisted the research which forms the basis for Chapter VII.

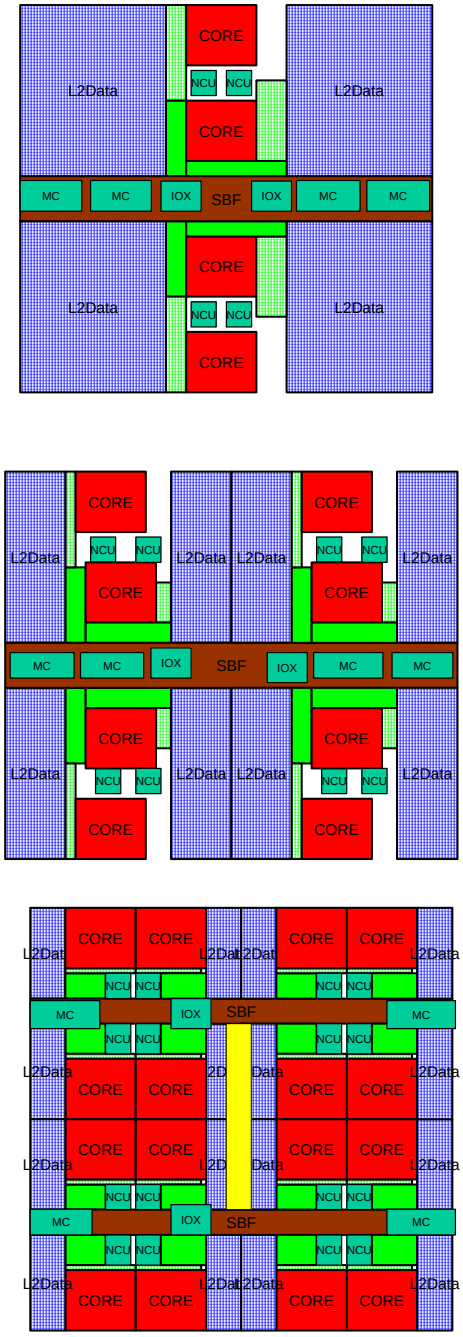

Figure VII.3: Floorplans for 4, 8 and 16 core processors

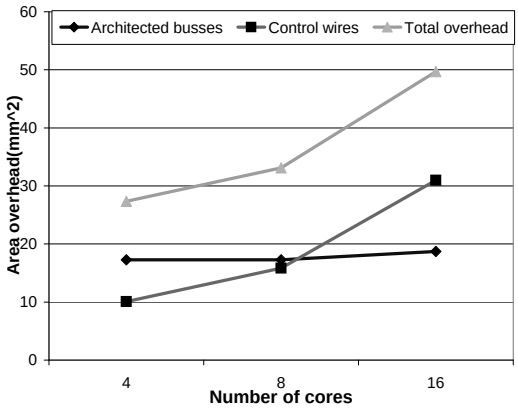

Figure VII.4: Area overhead for shared bus fabric.

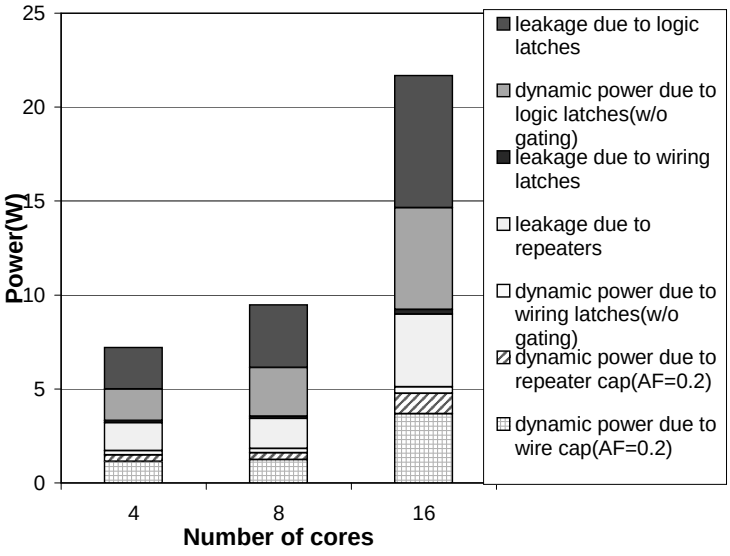

Figure VII.5: Power overhead for shared bus fabric

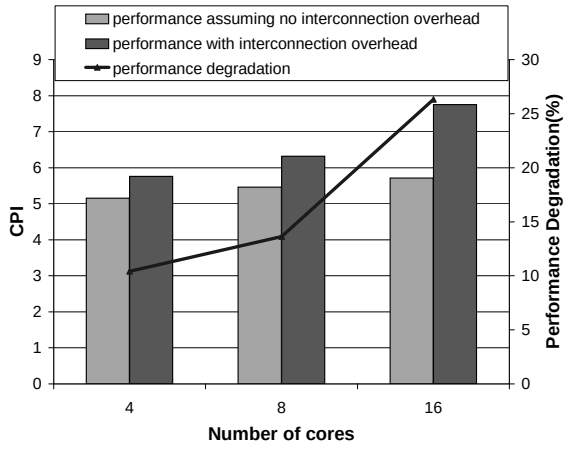

Figure VII.6: Performance overhead due to shared bus fabric.

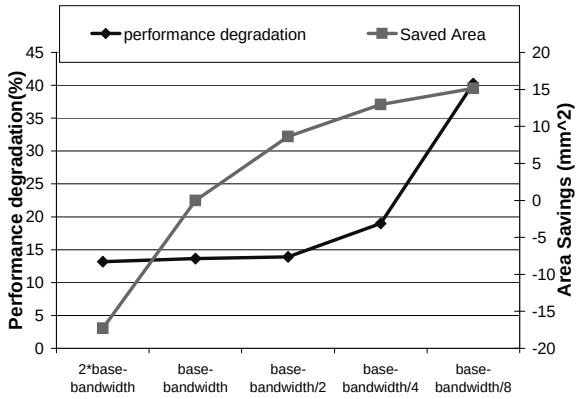

Figure VII.7: Trading off interconnection bandwidth with area.

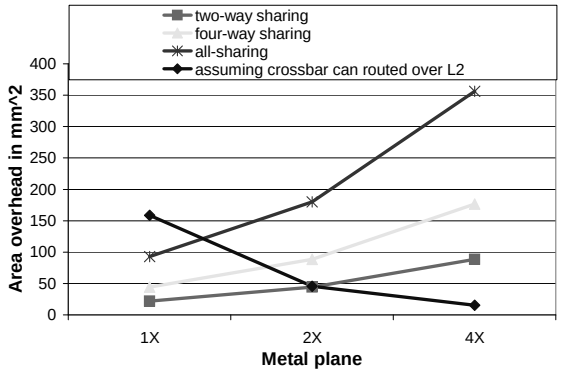

Figure VII.8: Area overhead for cache sharing – results for crossbar routed over L2 assume uniform cache density.

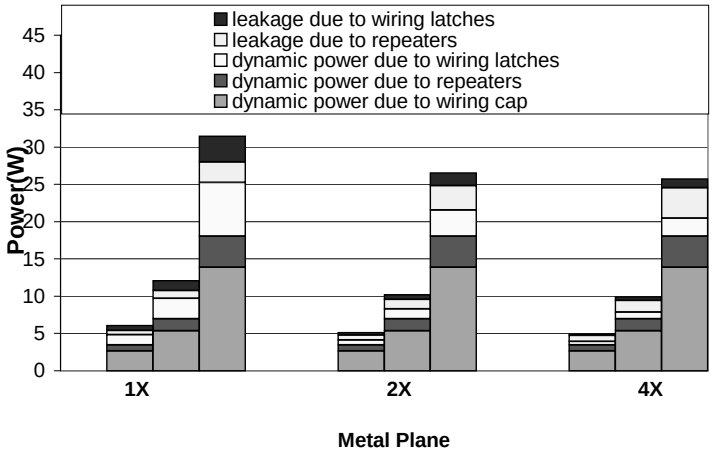

Figure VII.9: Power overhead for cache sharing (the three bars, left to right, correspond to 2-way, 4-way and full sharing).

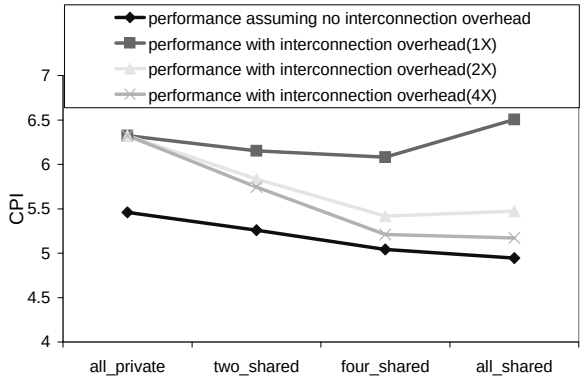

Figure VII.10: Evaluating cache sharing for a fixed cache size for different crossbar implementations – no area overhead is assumed

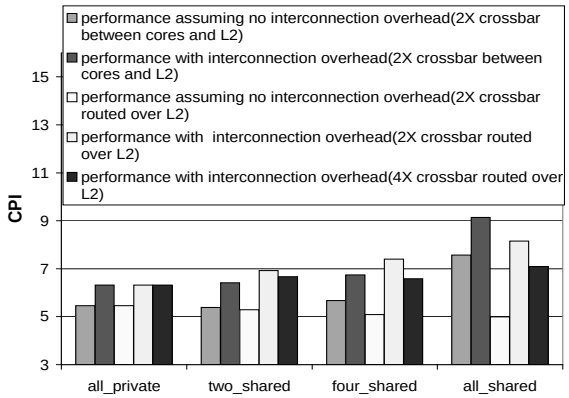

Figure VII.11: Evaluating cache sharing for a fixed die area – area overhead taken into account

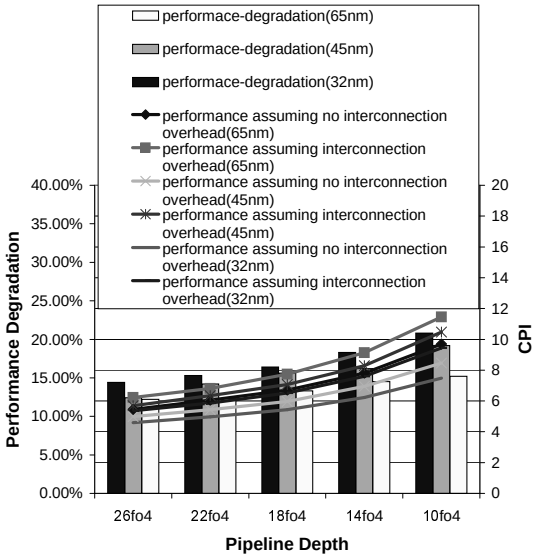

Figure VII.12: Scaling of interconnection overhead with pipelining and technology

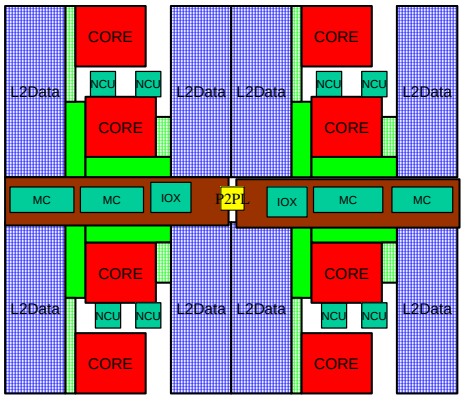

Figure VII.13: Hierarchical approach (splitting SBFs)

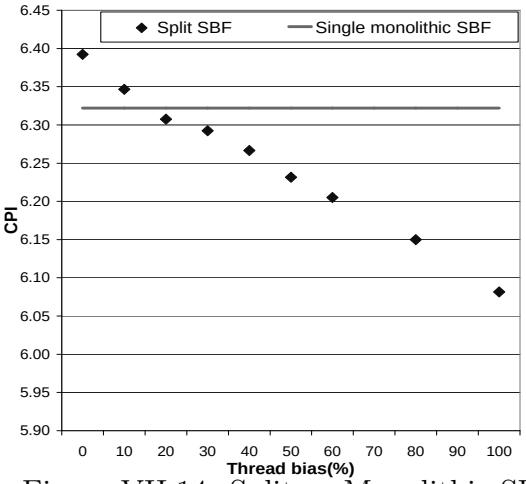

Figure VII.14: Split vs Monolithic SBF

VIII

Summary and Future Work

The decreasing marginal utility of transistors and increasing complexity of design has led to the advent of multi-core architectures. However, fundamental questions remain regarding the right form, methodology, and implementation for multi-core designs. This thesis seeks to address these questions. Specifically, the thesis shows that a "multi-core oblivious" design methodology, where the processor subsystems are designed and optimized without any cognizance of the overall chip multiprocessing systems that they would become parts of, results in processors that are inefficient in terms of area and power. The thesis shows that this inefficiency is due to the inability of such processors to react to (1) workload diversity, (2) processor overprovisioning, and (3) high cost of connecting cores and caches. This thesis recommends a holistic approach to multi-core design where the processor subsystems are designed from the ground up to be parts of chip multiprocessing system. Specifically, we propose *single-ISA heterogeneous multi-core architectures* for adapting to workload diversity. These architectures consist of multiple types of processing cores on the same die. These cores can all execute the same ISA, but represent different points in the power-performance continuum. Applications are mapped to cores in a way that the resources demands of an application match the resources provided by the assigned core. This

results in increased computational efficiency and hence higher throughput for a given area and/or power budget. We also proposed *conjoined-core chip multiprocessing architectures*. These architectures consist of multiple cores on the die where the adjacent cores share the large, overprovisioned structures. Sharing results in reduced area requirement for such processors at the cost of minimal loss in performance. Reduced area, in turn, results in higher yield, lower leakage, and potentially higher overall throughput per unit area. The thesis also studies the conventional interconnection mechanisms for multi-core architectures and demonstrates that the interconnection overheads are significant enough that they affect the number, size, and design of cores and caches. This thesis shows the need to co-design the cores, caches, and the interconnects, and also presents an example holistic approach to interconnection design for multi-core architectures.

VIII.A Holistic Design for Adaptability

We have demonstrated that heterogeneous multi-core architectures can provide significant throughput advantages over equivalent-area homogeneous architectures. This throughput advantage results from the ability of heterogeneous processors to better exploit both variations in thread-level parallelism as well as inter- and intra- thread diversity. We also propose and evaluate a set of thread scheduling mechanisms to best realize the potential performance gain available from heterogeneity.

Over a wide range of threading parallelism, the representative heterogeneous architecture we study perform 18% better on average than a homogeneous CMP architecture of the same area on SPEC workloads. For an open system with random task arrivals and exits, our results showed that heterogeneous architectures can have much lower response times than corresponding homogeneous configurations. Also, the heterogeneous systems were stable at job arrival rates

that were up to 43% higher.

Having a diversity of cores with varying resources and pipeline architectures enables the system to efficiently leverage application diversity both at the inter-thread and intra-thread level. Applications least able to derive significant benefits from large and complex cores can instead be run on smaller, less complex cores with much better area efficiencies.

This work demonstrates effective yet relatively simple task scheduling mechanisms to best match the applications to cores. Our best core assignment strategy achieves more than a 30% performance improvement over a naive heuristic, while still being straightforward to implement.

This thesis also introduces and seeks to gain some insights into the energy benefits available for the new architecture. The particular opportunity examined is a single application switching among cores to optimize some function of energy and performance.

We show that a sample heterogeneous multi-core design with four complexity-graded cores has the potential to increase energy efficiency (defined as energy-delay product, in this case) by a factor of three, in one experiment, without dramatic losses in performance. Energy efficiency improvements significantly outdistance chip-wide voltage/frequency scaling. It is shown that most of these gains are possible even by using as few as two cores.

This work demonstrates that there can be great power advantage to diversity within an on-chip multiprocessor, allowing that architecture to adapt to the workload in ways that a uniform CMP cannot. A multi-core heterogeneous architecture can support a range of execution characteristics not possible in an adaptable single-core processor, even one that employs aggressive 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 results indicate that not only is there significant potential for this style of architecture, but that reasonable runtime heuristics for switching cores, using limited runtime information, can achieve most of that potential.

The thesis also looks at the design of cores for a heterogeneous CMP. The goal is to determine how to design a heterogeneous CMP for a given set of workloads and given area and power budgets. We try to identify the characteristics of the cores of a heterogeneous multiprocessor for the highest area or power efficiency, and quantify the benefit that can be obtained by doing a design from the ground up. We also present a methodology for the design and optimization of the constituent processor cores given a set of target applications, as well as specific design budgets. We call these cores non-monotonic if their performance is not fully ordered over a range of different applications.

We show that the best way to design a heterogeneous CMP is not to find individual cores that are well suited for the entire universe of applications, but rather to tune the cores to different classes of applications. We find that customizing cores to subsets of the workload results in processors that have greater performance and power benefits than heterogeneous designs with an ordered set of cores. An example such design outperformed the best homogeneous CMP design by 15.4% and the best fully-customized monotonic design by 7.5%. There were performance and power improvements even for fully homogeneous workloads as well as for single-threaded workloads. Benefits are even greater when power and area budgets are increasingly constrained. Performance improvements of up to 40% are shown.

Given current trends in processor design, we expect dynamic power (and static power which is roughly proportional to area) to become increasingly con-

strained in the future, even in desktop and server markets. This means that the value of custom core architecture, as well as the benefits of aggressive heterogeneity will only increase with time.

VIII.B Obviating Overprovisioning in Multi-cores

This thesis also examines conjoined core multiprocessing, selectively targeting opportunities to share resources on an otherwise statically partitioned chip multiprocessor. In particular, we seek to achieve area savings, dynamic power reduction, and leakage reduction by sharing resources that have sufficient bandwidth and/or capacity to service multiple cores. We add the additional constraint that the sharing is topologically feasible with minimal impact to a conventional core layout.

This thesis examines sharing of the floating point units, the crossbar network ports, and the first-level ICache and DCache. We show that, given a set of novel optimizations that reduce the negative impacts of this sharing, we can reduce area requirements by more than 50%, while achieving performance within 9-12% of conventional cores without conjoining. Alternatively, by only sharing floating point units and crossbar ports, core area can be reduced by more than 23% while achieving performance within 2% of conventional cores without conjoining.

These gains are a combination of the inherent advantage of sharing resources provisioned for worst-case utilization, and the application of new sharing policies that allow high bandwidth access to these resources without additional complexity.

VIII.C Interconnection-aware Co-design

This thesis presents the results of a detailed modeling of the impact of the interconnection fabric on a hypothetical chip multiprocessor. These results show that the architecture of the interconnect interacts with the design and architecture of the cores and caches to a much greater degree than conventional off-chip interconnections. Thus, any design that hopes to achieve high performance or even energy efficiency needs to be the result of a careful co-design of all three elements.

This study shows several examples of this need for co-design. The interconnect fabric itself is large and power-hungry, consuming resources that would otherwise be available for more cores and caches. The interconnect, even without the sharing of L2 caches, can take the area of three cores and the power of one.

We show examples where decreasing interconnection bandwidth can improve performance, due to the constrained window on total resources. In the same way, large caches can also decrease performance when they constrain the interconnect to too small an area.

We also show that while it is generally believed that shared L2 caches improve cache hit rates, we show that the implications on the interconnect are extreme. For example, sharing four caches among four cores can require a quarter of the chip area just for the crossbar network. When accounting for the area overheads and the latency of the long interconnect, the desirability of shared L2 caches is significantly lower than is assumed if the interconnect is not accounted for.

VIII.D Future Work

While the processor industry has seemingly embraced the technology shift towards multi-core architectures, continued success along the multi-core path requires addressing several fundamental research issues.

- Most applications and tools are written for the uniprocessor world. With the advent of multi-core architectures, the entire computing stack – from the applications, to the compilers, to the operating system, to the hardware – needs to be re-examined and "fixed". For example, scheduling is a difficult task even for a uniprocessor OS. With the multiplicity of cores on the die, the task of scheduling becomes very complex, more so if the cores are not identical. Hence, new schedulers would be needed.

- Similarly, applications need to be parallelized in order to take advantage of multi-core architectures. However, programmers today are used to writing sequential programs. Writing parallel code would require changing the way programmers are trained. Even if a programmer is trained to write parallel code, not all programs are (easily) parallelizable. So, new models of programming or extracting parallelism might be needed.

- Alternatively, the complexity of parallelization can be pushed to the compiler. In that case, the compiler either needs to be able to automatically parallelize the code, or annotate the code sufficiently so that hardware can do the parallelization.

- Correspondingly, support needs to be present in hardware either to dynamically parallelize/compile the programs, or enough support needs to be provided at the hardware level so that the job of the programmers and compiler writers gets easier. For example, one major reason for the complexity of parallel code is due to the complexity of locking and synchronization. Hardware

support can be possibly be provided for lock-free synchronization or transactional style of programming.

- One question that needs to be addressed also is how to make effective use of multiple cores. While there are scenarios with abundant thread-level parallelism (e.g., web servers) or multiprogramming loads, other scenarios might make it difficult keep the cores busy, even after careful parallelization of applications. One possible approach might be to use the cores to enhance usability and provide additional functionalities like security, debuggability, reliability, etc. Creative new ways to provide these functionalities timely and efficiently are needed.

- Then there are the technological issues as well. The more the number of cores on the die, the more the bandwidth requirement. However, because of cost reasons, only a fixed number of pins (and hence bandwidth) can be supported for a given process technology. Current processors are already bandwidth-limited [67]. Breakthroughs would be required to enhance the bandwidth of processors. Novel techniques would also be required in software and hardware for effective utilization of bandwidth.

- Worst-case power of processors is another issue that multi-cores would have to face. The number of cores for a given core type will be limited less by the area budget, and more by the power budget. Effective worst-case power reduction techniques would be needed to be able to put an increasing number of cores on the die.

- Communication between cores will also have increasing overhead as the number of cores increase. In certain scenarios, they might even start dominating overall overheads. New interconnection architectures, mechanisms, and technologies would be required to manage the interconnection overhead.

Similarly, multi-core aware coherence policies are also needed in order to effectively make use of the underlying interconnection.

Subsequent chapters of this thesis address several of the above mentioned issues and advocate holistic design of multi-cores to push the yellow and the red walls further.

Bibliography

[1] http://www.amd.com/us-en/processors/productinformation/0_30_118_12651_00.html.

[2] http://www.amd.com/us-en/processors/productinformation/0_30_118_8825_00.html.

[3] http://www.amd.com/us-en/processors/productinformation/0_30_118_9484_00.html.

[4] http://www.arm.com/products/cpus/arm11mpcoremultiprocessor.html.

[5] http://www.broadcom.com/products/enterprise-small-office/communications-processors.

[6] http://www.cavium.com/octeon_mips64.html.

[7] http://www.geek.com/procspec/hp/pa8800.htm.

[8] http://www.intel.com/pressroom/kits/quickreffam.htm.

[9] http://www.intel.com/products/processor/coreduo/.

[10] http://www.intel.com/products/processor/pentium_d/index.htm.

[11] http://www.intel.com/products/processor/pentiumxe/index.htm.

[12] http://www.intel.com/products/processor/xeon/index.htm.

[13] http://www.razamicroelectronics.com/products/xlr.htm.

[14] http://www.xbox.com/en-us/hardware/xbox360/powerplay.htm.

[15] International Technology Roadmap for Semiconductors 2003, http://public.itrs.net.

[16] Butterfly parallel processor overview. In *BBN Report No 6148,*, March 1986.

[17] Alpha 21064 and Alpha 21064A: Hardware Reference Manual. Digital Equipment Corporation, 1992.

[18] Alpha 21164 Microprocessor:Hardware Reference Manual. Digital Equipment Corporation, 1998.

[19] Alpha 21264/EV6 Microprocessor:Hardware Reference Manual. Compaq Corporation, 1998.

[20] Measuring processor performance with SPEC2000- a white paper, Intel Corporation. 2002.

[21] A. Agarwal, J. Kubiatowicz, D. Kranz, B-H. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, June 1993.

[22] D. H. Albonesi. Selective cache-ways: On demand cache resource allocation. In *International Symposium on Microarchitecture*, November 1999.

[23] Murali Annavaram, Ed Grochowski, and John Shen. Mitigating Amdahl's Law Through EPI Throttling. In *Proceedings of International Symposium on Computer Architecture*, 2005.

[24] James Archibald and Jean-Loup Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, 4(4):273–298, 1986.

[25] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. The impact of performance asymmetry in emerging multicore architectures. In *Proceedings of International Symposium on Computer Architecture*, 2005.

[26] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[27] O. Beaumont, A. Legrand, and Y. Robert. The master-slave paradigm with heterogeneous processors. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'01)*, October 2001.

[28] William Bowhill. A 300-MHz 64-b quad-issue CMOS microprocessor. In *ISSCC Digest of Technical Papers*, February 1995.

[29] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *27th Annual International Symposium on Computer Architecture*, June 2000.

[30] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *International Symposium on Computer Architecture*, June 2000.

[31] James Burns and Jean-Luc Gaudiot. Area and system clock effects on smt/cmp processors. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, page 211. IEEE Computer Society, 2001.

[32] James Burns and Jean-Luc Gaudiot. SMT layout overhead and scalability. *IEEE Transactions on Parallel and Distributed Systems*, 13(2), February 2002.

[33] Joachim Clabes, Joshua Friedrich, Mark Sweet, Jack DiLullo, Sam Chu, Donald Plass, James Dawson, Paul Muench, Larry Powell, Michael Floyd, Balaram Sinharoy, Mike Lee, Michael Goulet, James Wagoner, Nicole Schwartz, Steve Runyon, Gary Gorman, Phillip Restle, Ronald Kalla, Joseph McGill, and Steve Dodson. Design and implementation of the power5 microprocessor. In *ISSCC*, 2004.

[34] Jamison Collins and Dean Tullsen. Clustered multithreaded architectures – pursuing both IPC and cycle time. In *Proceedings of IPDPS*, April 2004.

[35] J.D. Collins, H. Wang, D.M. Tullsen, C.J. Hughes, Y.-F. Lee, D. Lavery, and J.P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *28th Annual International Symposium on Computer Architecture*, July 2001.

[36] William J. Dally and Brian Towles. Route packets, not wires: On-chip interconnection networks. In *DAC-38*, pages 684–689, 2001.

[37] John D. Davis, James Laudon, and Kunle Olukotun. Maximizing cmp throughput with mediocre cores. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 51–62, Washington, DC, USA, 2005. IEEE Computer Society.

[38] R. H. Denard. Design of ion-implanted MOSFETs with very small physical dimensions. In *IEEE Journal of Solid-state Circuits*, volume 98, 1974.

[39] Keith Diefendorff. Compaq chooses SMT for Alpha. In *Microprocessor Report, Vol 13, No. 16*, December 1999.

[40] Daniel W Dobberpuhl. A 200-MHz 64-b dual-issue CMOS Microprocessor. In *IEEE Journal of Solid-State Circuits, Vol 27, No. 11*, November 1992.

[41] R. Dolbeau and A. Seznec. CASH: Revisiting hardware sharing in single-chip parallel processor. IRISA Report 1491, November 2002.

[42] M. Dubois, C. Scheurich, and F.A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 21(2), 1988.

[43] Richard J. Eickemeyer, Ross E. Johnson, Steven R. Kunkel, Mark S. Squillante, and Shiafun Liu. Evaluation of multithreaded uniprocessors for commercial application environments. In *ISCA-23*, 1996.

[44] Joel Emer. EV8:The Post-ultimate Alpha. In *PACT Keynote Address*, September 2001.

[45] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: A vector extension to the alpha architecture. In *International Symposium on Computer Architecture*, May 2002.

[46] D. Folegnani and A. Gonzalez. Reducing power consumption of the issue logic. In *Workshop on Complexity-Effective Design*, June 2000.

[47] S. J. Frank. Tightly coupled multiprocessor systems speed memory access times. In *Electron*, January 1984.

[48] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh. Cedar - a large scale multiprocessor. In *ICPP*, August 1983.

[49] S. Ghiasi, J. Casmira, and D. Grunwald. Using IPC variation in workloads with externally specified rates to reduce power consumption. In *Workshop on Complexity Effective Design.*, June 2000.

[50] Soraya Ghiasi and Dirk Grunwald. Aide de camp: Asymmetric dual core design for power and energy reduction. In *University of Colorado Technical Report CU-CS-964-03*, 2003.

[51] Soraya Ghiasi, Tom Keller, and Freeman Rawson. Scheduling for heterogeneous processors in server systems. In *Proceedings of Computing Frontiers*, 2005.

[52] B Gieseke. A 600-MHz Superscalar RISC Microprocessor with Out-of-Order Execution. In *ISSCC Digest of Technical Papers*, February 1997.

[53] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. In *IEEE International Symposium on Low Power Electronics 1995*, October 1995.

[54] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power cpu. In *International Conference on Mobile Computing and Networking*, November 1995.

[55] Ed Grochowski, Ronny Ronen, John Shen, and Hong Wang. Best of both latency and throughput. In *Proceedings of IEEE International Conference on Computer Design*, 2004.

[56] Dirk Grunwald, Artur Klauser, Srilatha Manne, and Andrew Pleskun. Confidence estimation for speculation control. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[57] Stephen H. Gunther, Frank Binns, Douglas Carmean, and Jonathan C. Hall. Managing the Impact of Increasing Microprocessor Power Consumption. In *Intel Technology Journal*, 1st Quarter 2001.

[58] S. Gupta, S.W. Keckler, and D.C. Burger. Technology independent area and delay estimates for microprocessor building blocks. In *University of Texas at Austin Technical Report TR-00-05*, 1998.

[59] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30(9), 1997.

[60] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, October 1998.

[61] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Oberg, M. Millberg, and D. Lindqvist. Network on chip: An architecture for billion transistor era. In *IEEE NorChip Conference*, November 2000.

[62] J. Hennessy and D. Patterson. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 2002.

[63] J. L. Hennessy and N. P. Jouppi. Computer technology and architecture: An evolving interaction. *Computer*, 24(9):18–29, September 1991.

[64] R. Ho, K.W. Mai, and M.A Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, 2001.

[65] M. Horowitz, E. Alon, D. Patil, S. Naffziger, R. Kumar, and K. Bernstein. Scaling, power, and the future of cmos. In *IEEE International Electron Devices Meeting*, December 2005.

[66] M. Horowitz, R. Ho, and K. Mai. The future of wires. 1999.

[67] Jaehyuk Huh, Stephen W. Keckler, and Doug Burger. Exploring the design space of future CMPs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2001.

[68] IBM. Power4:http://www.research.ibm.com/power4.

[69] IBM. Power5: Presentation at microprocessor forum. 2003.

[70] Intel Corp. *Intel Pentium 4 Processor in the 423-pin Package Thermal Design Guidelines*, November 2000.

[71] A. Iyer and D. Marculescu. Power aware microarchitecture resource scaling. In *IEEE Design, Automation and Test in Europe Conference*, 2001.

[72] A. Iyer and D. Marculescu. Power-performance exaluation of globally-asynchronous, locally-synchronous processors. In *International Symposium on Computer Architecture*, 2001.

[73] Russ Joseph and Margaret Martonosi. Run-time Power Estimation in High-Performance Microprocessors. In *The International Symposium on Low-Power Estimation and Design*, August 2001.

[74] C. Kaanta, W. Cote, J. Cronin, K. Holland, P.I. Lee, and T. Wright. Sub-micron wiring technology with tungsten and planarization. In *Fifth VLSI Multilevel Interconnection Conference*, 1988.

[75] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. In *IBM Journal of Research and Development*, September 2005.

[76] Faraydon Karim, Anh Nguyen, Sujit Dey, and Ramesh Rao. On-chip communication architecture for oc-768 network processors. In *Proceedings of the 2001 Design and Automation Conference*, 2001.

[77] R.E. Kessler, E.J. McLellan, and D.A. Webb. The alpha 21264 microprocessor architecture. In *International Conference on Computer Design*, December 1998.

[78] C. Kim, D. Burger, and S. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ASPLOS*, 2002.

[79] Arthur Klauser. Trends in high-performance microprocessor design. In *Telematik-2001*, 2001.

[80] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. In *IEEE MICRO Magazine*, March 2005.

[81] Ramakrishna Kotla, Anirudh Devgan, Soraya Ghiasi, Tom Keller, and Freeman Rawson. Characterizing the impact of different memory-intensity levels. In *Proceedings of IEEE 7th Annual Workshop on Workload Characterization (WWC-7)*, 2004.

[82] John Kowaleski. Implementation of an Alpha Microprocessor in SOI. In *ISSCC Digest of Tecnical Papers*, February 2003.

[83] V. Krishnan and J. Torrellas. A clustered approach to multithreaded processors. In *Proceedings of the International Parallel Processing Symposium*, pages 627–634, March 1998.

[84] Ashok Kumar. The HP PA-8000 RISC CPU. In *Hot Chips VIII*, August 1996.

[85] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. A multi-core approach to addressing the energy-complexity problem in microprocessors. In *Workshop on Complexity-Effective Design*, June 2003.

[86] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Processor power reduction via single-ISA heterogeneous multi-core architectures. In *Computer Architecture Letters, Vol 2*, April 2003.

[87] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA Heterogeneous Multi-core Architectures: The Potential for Processor Power Reduction. In *MICRO-36*, December 2003.

[88] Rakesh Kumar, Norman P. Jouppi, and Dean M. Tullsen. Conjoined-core chip multiprocessing. In *International Symposium on Microarchitecture*, December 2004.

[89] Rakesh Kumar, Dean M. Tullsen, and Norman Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *Proceedings of International Symposium on Parallel Architectures and Computing Technologies (PACT)*, 2006.

[90] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance. In *International Symposium on Computer Architecture*, June 2004.

[91] Rakesh Kumar, Victor Zyuban, and Dean M. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *Proceedings of International Symposium on Computer Architecture*, 2005.

[92] S.R. Kunkel, R.J. Eickemeyer, M.H. Lipasti, T.J. Mullins, B.O. Krafka, H. Rosenberg, S.P. VanderWiel, P.L. Vitale, and L.D. Whitley. A performance methodology for commercial servers. In *IBM Journal of R&D*, November 2000.

[93] Jim Laudon. Performance/watt the new server focus. In *Proceeedings of First Workshop on Design, Architecture, and Simulation of Chip-Multiprocessors (dasCMP)*, November 2005.

[94] D. Lenoski, J. Laudon, K. Gharachorloo, W.D. Weber, A. Gupta, J. Henessy, M. Horowitz, and M.S. Lam. The stanford DASH multiprocessor. In *IEEE Computer*, 1992.

[95] J. Li and J.F. Martinez. Power-performance implications of thread-level parallelism in chip multiprocessors. In *Proceedings of International Symposium on Performance Analysis of Systems and Software*, 2005.

[96] T. Lovett and S. Thakkar. The symmetry multiprocessor system. In *ICPP*, August 1988.

[97] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *International Symposium on Computer Architecture*, June 1998.

[98] Pedro Marcuello and Antonio Gonzalez. Clustered speculative multithreaded processors. In *International Conference on Supercomputing*, 1999.

[99] R. Maro, Y. Bai, and R.I. Bahar. Dynamically reconfiguring processor resources to reduce power consumption in high-performance processors. In *Workshop on Power-aware Computer Systems*, November 2000.

[100] Rick Merritt. Designers cut fresh paths to parallelism. In *EE Times.*, October 1999.

[101] Gordon Moore. Cramming more components onto integrated circuits. volume 38, 1965.

[102] Tomer Morad, Uri Weiser, and Avinoam Kolodny. ACCMP - assymetric cluster chip-multiprocessing. In *CCIT Technical Report 488*, 2004.

[103] Tomer Y. Morad, Uri C. Weiser, Avinoam Kolodny, Mateo Valero, and Eduard Ayguade. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. In *Computer Architecture Letters, Vol 4*, July 2005.

[104] J. M. Mulder, N. T. Quach, and M. J. Flynn. An area model for on-chip memories and its applications. In *IEEE Journal of Solid State Circuits, Vol 26, No. 2*, February 1991.

[105] Hyunok Oh and Soonhoi Ha. A static scheduling heuristic for heterogeneous processors. In *Euro-Par, Vol. II*, pages 573–577, 1996.

[106] M. Papamarcos and J. Patel. A low overhead coherence solution for multiprocessors with private cache memories. In *ISCA-15*, 1988.

[107] Li-Shiuan Peh. Flow control and microarchitectural mechanisms for extending the performance of interconnection networks. PhD Thesis, Stanford University, 2001.

[108] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *International Symposium on Low Power Electronics and Design*, August 1998.

[109] G.F Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, , and J. Weiss. The IBM Research Parallel Processor prototype (RP3): Introduction and Architecture. In *ICPP*, August 1985.

[110] Jan M. Rabaey. The quest for ultra-low energy computation opportunities for architectures exploiting low-current devices. April 2000.

[111] Elaine Rich and Kevin Knight. *Artificial Intelligence, 2nd Edition*. Morgan Kaufmann, 1991.

[112] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *International Symposium on Computer Architecture*, June 2003.

[113] Charles L. Seitz. The cosmic cube. In *Communications of ACM*, 1985.

[114] G Semeraro, G Maklis, R Balasubramonian, D H Albonesi, S Dwarkadas, and M L Scott. Energy efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *International Symposium on High-Performance Computer Architecture*, February 2002.

[115] T. Sherwood and B. Calder. Time varying behavior of programs. In *UC San Diego Technical Report UCSD-CS-99-630*, August 1999.

[116] T. Sherwood, E. Perelman, G. Hammerley, and B. Calder. Automatically characterizing large-scale program behavior. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[117] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Tenth International Comference on Architectural Support for Programming Languages and Operating Systems(ASPLOS 2002)*, October 2002.

[118] Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. Discovering and exploiting program phases. In *IEEE Micro: Micro's Top Picks from Computer Architecture Conferences*, December 2003.

[119] Premkishore Shivakumar and Norm Jouppi. CACTI 3.0: An integrated cache timing, power and area model. In *Technical Report 2001/2, Compaq Computer Corporation*, August 2001.

[120] A. Snavely and D.M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading architecture. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[121] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *25 Years ISCA: Retrospectives and Reprints*, pages 521–532, 1998.

[122] Gurinder S. Sohi and Amir Roth. Speculative multithreaded processors. volume 34, pages 66–33, 2001.

[123] Sun. UltrasparcIV: http://siliconvalley.internet.com/news/print.php/3090801.

[124] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 291, Washington, DC, USA, 2003. IEEE Computer Society.

[125] T. N. Theis. The future of interconnection technology. In *IBM Journal of R&D*, May 2000.

[126] Marc Tremblay. Majc-5200: A vliw convergent mpsoc. In *Microprocessor Forum*, October 1999.

[127] D.M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, December 1996.

[128] D.M. Tullsen and J.A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *34th International Symposium on Microarchitecture*, December 2001.

[129] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[130] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, 1997.

[131] 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-IV)*, pages 176–188, April 1991.

[132] J.D. Warnock, J.M. Keaty, J. Petrovick, J.G. Clabes, C.J. Kircher, B.L. Krauter, P.J. Restle, B.A. Zoric, and C.J. Anderson. The circuit and physical design of the Power4 microprocessor. In *IBM Journal of R&D*, January 2002.

[133] A. Wilson. Hierarchical cache/bus architecture for shared memory multiprocessors. In *ISCA-14*, June 1987.

[134] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *28th Annual International Symposium on Computer Architecture*, July 2001.

[135] Victor Zyuban. Unified architecture level energy-efficiency metric. In *2002 Great Lakes Symposium on VLSI*, April 2002.