UC San Diego
UC San Diego Electronic Theses and Dissertations

Title

Tackling computation uncertainty through fine-grained and predictable execution adaptivity in multicore systems

Permalink
https://escholarship.org/uc/item/22w4j8gt

Author
Yang, Chengmo

Publication Date
2010

Peer reviewed|Thesis/dissertation

eScholarship.org
Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Tackling Computation Uncertainty through Fine-grained and Predictable Execution Adaptivity in Multicore Systems

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

in

Computer Engineering

by

Chengmo Yang

Committee in charge:

Professor Alex Orailoglu, Chair
Professor Fan Chung Graham
Professor Keith Marzullo
Professor Michael Taylor
Professor Charles W. Tu

2010

Copyright
Chengmo Yang, 2010
All rights reserved.

The dissertation of Chengmo Yang is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

____

____

____

____

____

Chair

University of California, San Diego

2010

DEDICATION

To my grandmother.

TABLE OF CONTENTS

Signature Page iii

Dedication iv

Table of Contents v

List of Figures viii

List of Tables x

Acknowledgements xi

Vita xiii

Abstract of the Dissertation xv

Chapter 1 Introduction 1
1.1 Need for Execution Adaptivity 3
1.2 Challenges to be Addressed 5
1.3 Contributions of This Thesis 9
1.4 Roadmap 11

Chapter 2 Related Work 12
2.1 Resource Renegotiation 13
2.2 Reliability Enhancement 16
2.3 Heat Reduction 20

Chapter 3 Adaptive System Overview 23
3.1 Compiler-directed Runtime Optimization 23
3.2 Hybrid Scheduling for Resource Management 26
3.3 Possible Scenarios for Adapting the Computation 29

Chapter 4 Core-level Reconfiguration 32
4.1 Adaptive Static Schedules 33
4.1.1 Band Partitions of Execution Schedules 33
4.1.2 Inter-task Dependence Variations 35
4.2 Performance-oriented Core Reordering 37
4.2.1 PE Reordering: Problem Formulation 39
4.2.2 PE Reordering: L2R-free Mapping Identification 41
4.3 Performance Enhancement 44
4.3.1 Applied to Arbitrary Task Graphs 45
4.3.2 Overcoming Variations in Inter-core Communication 47

4.4 Tolerating Multiple Resource Variations 49
4.4.1 Band Partition Extension 49
4.4.2 Inter-task Dependence Constraints 51
4.4.3 Core Binding Permutation 53
4.4.4 Shiftable Core Identification 54
4.5 Algorithmic Implementation 56
4.5.1 Initial Schedule Generation 57
4.5.2 Adaptive Schedule Optimization 61
4.6 Experimental Results 62
4.6.1 Performance of single-core adaptive schedules 63
4.6.2 Performance of multi-core adaptive schedules 67
4.7 Conclusions 69

Chapter 5 Adaptivity-aware System Topology 71
5.1 Reconfiguration-induced Sharing Requirement 72
5.2 Locally Shareable Storage Organization 73
5.3 Physical Topology and Application Mapping 76
5.3.1 Topology instances and the associated properties . . . 76
5.3.2 Topology instance selection 78
5.3.3 Task Placement Requirements 81
5.4 Communication Overhead Minimization 82
5.4.1 Encoding-based Synchronization 83
5.5 Experimental Evaluation 90
5.5.1 Impact of Topology on Task Scheduling 90
5.5.2 Efficiency of Encoding-based Synchronization 93
5.6 Conclusions 95

Chapter 6 Architectural-level Fault Resilience 98
6.1 Full Resilience within Low Overhead 99
6.2 Cache-based Fault Tolerance 101
6.2.1 Run-ahead Property for Workload Balance 101
6.2.2 Fault Detection 102
6.2.3 Execution Checkpointing 103
6.2.4 Execution Recovery 106
6.2.5 Cache State Extension 106
6.2.6 Requirements on Memory Access Order 107
6.3 Execution Asynchronicity Enhancement 108
6.3.1 Relaxed Thread Synchronization at Checkpoints . . . 108
6.3.2 Selective Split Capability of Cache Blocks 109
6.3.3 Synchronization Condition Analysis 113
6.4 Fault Tolerant MPSoC Organization 114
6.4.1 Checkpointing Tradeoffs in Multi-level Cache Design 114
6.4.2 Checkpoint Coordination for Inter-thread Communications 115

6.4.3 Throughput Enhancement through Multi-threading . . 117
6.5 Cache Access Control Implementation 118
6.5.1 Cache Access Control 118
6.5.2 Implementation Efficiency 121
6.6 Simulation Results 122
6.6.1 Checkpointing and Writeback Frequencies 123
6.6.2 Thread performance 125
6.6.3 Impact of Fault Rate on Thread Performance 129
6.6.4 Checkpointing Tradeoffs for Memory Hierarchy . . . 130
6.7 Conclusions 130

Chapter 7 Compiler-Directed Heat Reduction 132
7.1 Challenges in Register Access Balance 133
7.2 Deterministic Register Shuffling 135
7.2.1 An illustrative example 136
7.2.2 Destination register name adjustment 137
7.2.3 Loop-carried dependence preservation 139
7.2.4 Shiftable logical register identification 141
7.2.5 Physical register reallocability analysis 142
7.2.6 Functional Evaluation 144
7.3 Implementation 146
7.3.1 Static register name adjustment 146
7.3.2 Dynamic register name shuffling 147
7.4 Simulation Results 149
7.4.1 Register Access Results 150
7.4.2 Temperature Results 152
7.5 Conclusions 154

Chapter 8 Conclusions 155
8.1 Future Work Directions 157

Bibliography 160

LIST OF FIGURES

Figure 3.1: Collaborative Optimization Framwork 24
Figure 3.2: Various scenarios for utilizing adaptive execution 30

Figure 4.1: Reconfigurable static schedules: band structure 34
Figure 4.2: Regularity in task reassignment 35
Figure 4.3: Timing variations of inter-task dependences 36
Figure 4.4: Impact of PE reordering on dependence directions 38
Figure 4.5: PE reordering formulated as graph embedding 39
Figure 4.6: L2R-free mapping of DAG, basic loop and nested loops 41
Figure 4.7: Mapping constraints of intersecting loops 43
Figure 4.8: An adaptive schedule for an arbitrary task graph 45
Figure 4.9: Band structure extension: the head and tail regions 46
Figure 4.10: Impact of PE rotation on inter-PE communications 48
Figure 4.11: Multi-band partitioning for increased amount of adaptivity 50
Figure 4.12: Inter-task dependence timing in a multi-band schedule 52
Figure 4.13: PE reordering in a multi-band schedule 54
Figure 4.14: PE shiftability constraints and a indirectly shiftable case 55
Figure 4.15: Impact of PE-distance and communication latency on the earliest start time of a sink task 59
Figure 4.16: Integrated task scheduling and core reordering flow 60
Figure 4.17: The benchmark task graphs 63
Figure 4.18: Impact of adaptivity degree and core reordering on **pre-**reconfiguration schedule length 68
Figure 4.19: Impact of adaptivity degree and core reordering on the amount of **L2R** communications 68
Figure 4.20: Impact of adaptivity degree and core reordering on the amount of **critical L2R** communications 69
Figure 4.21: Impact of adaptivity degree and core reordering on **post-**reconfiguration schedule length 69

Figure 5.1: Reconfiguration-induced sharing requirements 72
Figure 5.2: Bipartite graph representation of various topologies with distinct values of sharing degree and merging degree 74
Figure 5.3: Various 2-dimensional locally shareable MPSoC topologies 77
Figure 5.4: Finer-grained cluster partitions and communication link utilization . . 79
Figure 5.5: Reconfiguration-induced memory sharing and data placement 82
Figure 5.6: Encoding of point-to-point inter-PE communications 85
Figure 5.7: Implementation of the encoding-based synchronization scheme . . . 88
Figure 5.8: Total communication latency, assuming the average number of extra cycles spent in waiting for the consumer thread of 50 95
Figure 5.9: Total communication latency, assuming an average memory access latency of 10 cycles 96

Figure 6.1: Differences between lockstep CMP, redundant multi-threading, and the proposed cache-based detection/checkpointing scheme 100
Figure 6.2: Inconsistent access pattern caused by faults in store addresses 103
Figure 6.3: Loop with cache block dependences 105
Figure 6.4: Cache states extended for fault detection and checkpointing 107
Figure 6.5: Strictly vs. loosely- synchronized checkpointing 109
Figure 6.6: Adding a *split* state to cache state diagram 111
Figure 6.7: Hybrid detection and checkpointing policy in multi-level caches . . . 115
Figure 6.8: Applying the proposed shared cache organization to multi-core SoCs 116
Figure 6.9: Hardware extension to traditional cache 118

Figure 7.1: Cumulative register access ratio 134
Figure 7.2: A loop example obtained from *bzip2* 135
Figure 7.3: Register name adjustment in two consecutive iterations 140
Figure 7.4: Shiftability analysis of register *R1* 141
Figure 7.5: Building a shuffle window through swapping register values at loop entry and exit 144
Figure 7.6: Gate-level logic for translating register names 148
Figure 7.7: Reduction in peak temperature of the entire chip 152
Figure 7.8: The processor floorplan used in simulation 153
Figure 7.9: Reduction in peak temperature of the entire chip 154

LIST OF TABLES

Table 4.1: **Pre-reconfiguration** schedule length 64
Table 4.2: Impact of adaptivity on inter-PE communications 65
Table 4.3: Impact of PE reordering on L2R communications 66
Table 4.4: **Post-reconfiguration** schedule length 67

Table 5.1: The dynamic check/set of the *R-colors* for communication synchronization 90
Table 5.2: Impact of MPSoC topology on schedule length 91
Table 5.3: Impact of MPSoC topology on task mapping 92

Table 6.1: Impact of cache configuration on miss rate 122
Table 6.2: Impact of cache configuration on checkpointing frequency 123
Table 6.3: Overall writeback rate 124
Table 6.4: Checkpointing-induced writeback increase 125
Table 6.5: Impact of thread synchronization and block split on CPI increase (%): 16K-2way 126
Table 6.6: Impact of thread synchronization and block split on CPI increase (%): 32K-4way 127
Table 6.7: Cache Block Split Efficiency 127
Table 6.8: Impact of counter and victim cache sizes on CPI increase (%) 128
Table 6.9: Impact of fault rate on CPI increase (%) 129
Table 6.10: Memory hierarchy induced checkpointing tradeoffs 130

Table 7.1: The use of the two shuffle functions to shift register names in the *bzip2* example 139
Table 7.2: Access pattern-based register classification 143
Table 7.3: The design complexity of GF multipliers 149
Table 7.4: The number of hot loops, their occupancy in execution time, and register usage information 151

ACKNOWLEDGEMENTS

I would like to express my deepest thanks to my advisor, Professor Alex Orailoglu, for the great guidance he provided during my PhD years. Through multiple drafts, many debates and many long nights, his guidance has proved to be invaluable. His enthusiasm and intelligence in research is always a model for me to follow in my future research work.

I would also express my special thanks to Professor Keith Marzullo, for his effort in teaching me ukulele. The days we spent together playing and singing have brought a lot of happiness to my PhD life.

Many thanks go to the other professors I have met during the seven years in UCSD, as well as my academic siblings in the *Architecture, Reliability, and Testing* (ART) group, including Baris Arslan, Garo Bournoutian, Mingjing Chen, Kwangyoon Lee, Wenjing Rao, Peter Petrov, and Ozgur Sinanoglu.

I also want to thank my friends, Zheng Wu, Shan Yan, Haichang Sui, Dayou Zhou, Yuzhe Jin, Min Li, Junwen Wu, and many others, for sharing with me the unforgettable life in San Diego.

In the end, I want to thank my parents and my grandparents, for always being there. Without their love, support, tolerance and advice, I would never have gone so far.

The text of Chapter 4, is in part a reprint of the material as it appears in *C. Yang and A. Orailoglu, "Predictable Execution Adaptivity through Embedding Dynamic Reconfigurability into Static MPSoC Schedules," International Conference on Hardware/Software Codesign and System Synthesis (CODES-ISSS), October 2007*; in *C. Yang and A. Orailoglu, "Towards No-cost Adaptive MPSoC Static Schedules through Exploitation of Logical-to-physical Core Mapping Latitude," IEEE Design, Automation and Test in Europe (DATE), April 2009*; and in *C. Yang and A. Orailoglu, "Fully Adaptive Multicore Architectures through Statically-directed Dynamic Execution Reconfigurations," International Conference on VLSI and System-on-Chip (VLSI-SoC), September 2010*. The dissertation author was the primary researcher and author of the publications [94], [97], and [98].

The text of Chapter 5, is in part a reprint of the material as it appears in *C. Yang and A. Orailoglu, "Light-weight Synchronization for Inter-processor Communication Accel-*

eration on Embedded MPSoCs," International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), October 2007. The dissertation author was the primary researcher and author of the publication [93].

The text of Chapter 6, is in part a reprint of the material as it appears in *C. Yang and A. Orailoglu, "A Light-weight Cache-based Fault Detection and Checkpointing Scheme for MPSoCs Enabling Relaxed Execution Synchronization," International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), October 2008*. The dissertation author was the primary researcher and author of the publication [95].

The text of Chapter 7, is in part a reprint of the material as it appears in *C. Yang and A. Orailoglu, "Processor Reliability Enhancement through Compiler-Directed Register File Peak Temperature Reduction," International Conference on Dependable Systems and Networks (DSN), June 2009*. The dissertation author was the primary researcher and author of the publication [96].

VITA

2003	B. S. in Microelectronics, Peking University, China
2005	M. S. in Computer Science, University of California, San Diego
2004 – 2010	Teaching Assistant. Department of Computer Science and Engineering, University of California, San Diego
2004 – 2010	Research Assistant. Department of Computer Science and Engineering, University of California, San Diego
2010	Ph. D. in Computer Engineering, University of California, San Diego

PUBLICATIONS

Journal papers

C. Yang and A. Orailoglu, "Full Fault Resilience and Relaxed Synchronization Requirements at the Cache-Memory Interface," to appear in *IEEE Trans. on Very Large Scale Integration Systems (TVLSI)*

C. Yang, M. Chen, and A. Orailoglu, "Minimizing On-chip Code Storage in Microcoded IPs while Delivering High Decompression Speed," to appear in *the special issue of the Journal on Design Automation for Embedded Systems*

C. Yang and A. Orailoglu, "Tackling Resource Variations through Adaptive MPSoC Execution Frameworks," submitted to *IEEE Trans. on Computers*

Conference papers

C. Yang and A. Orailoglu, "Fully Adaptive Multicore Architectures through Statically-directed Dynamic Execution Reconfigurations," to appear in *International Conference on VLSI and System-on-Chip (VLSI-SoC)*, September 2010

C. Yang, C. J. Xue, and A. Orailoglu, "Fine-grained Adaptive CMP Cache Sharing through Access History Exploitation" to appear in *International Conference on VLSI and System-on-Chip (VLSI-SoC)*, September 2010

C. Yang, M. Chen, and A. Orailoglu, "Squashing Microcode Stores to Size in Embedded Systems while Delivering Rapid Microcode Accesses," in *International Conference on Hardware/Software Codesign and System Synthesis (CODES-ISSS)*, pp. 249-256, October 2009 (best paper nomination)

C. Yang and A. Orailoglu, "Processor Reliability Enhancement through Compiler-Directed Register File Peak Temperature Reduction," in *International Conference on Dependable Systems and Networks (DSN)*, pp. 468-477, June 2009

C. Yang and A. Orailoglu, “Towards No-cost Adaptive MPSoC Static Schedules through Exploitation of Logical-to-physical Core Mapping Latitude,” in *IEEE Design, Automation and Test in Europe (DATE)*, pp. 63-68, April 2009

C. Yang and A. Orailoglu, “A Light-weight Cache-based Fault Detection and Checkpointing Scheme for MPSoCs Enabling Relaxed Execution Synchronization,” in *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pp. 11- 20, October 2008

C. Yang and A. Orailoglu, “Predictable Execution Adaptivity through Embedding Dynamic Reconfigurability into Static MPSoC Schedules,” in *International Conference on Hardware/Software Codesign and System Synthesis (CODES-ISSS)*, pp. 15-20, October 2007

C. Yang and A. Orailoglu, “Light-weight Synchronization for Inter-processor Communication Acceleration on Embedded MPSoCs,” in *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pp. 150-154, October 2007

C. Yang and A. Orailoglu, “Power-efficient Branch Prediction through Early Identification of Branch Addresses,” in *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pp. 169-178, October 2006

C. Yang and A. Orailoglu, “Power-efficient Instruction Delivery through Trace Reuse,” in *the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 192-201, September 2006

Workshop paper

C. Yang and A. Orailoglu, “Accelerating Coupled Applications through Register Level Communication between Processing Elements,” in *the 4th Workshop on Application Specific Processors (WASP)*, pp. 51-59, September 2005

ABSTRACT OF THE DISSERTATION

Tackling Computation Uncertainty through Fine-grained and Predictable Execution Adaptivity in Multicore Systems

by

Chengmo Yang

Doctor of Philosophy in Computer Engineering

University of California, San Diego, 2010

Professor Alex Orailoglu, Chair

The continued scaling of silicon fabrication technologies has enabled the integration of dozens of processing cores on a single chip in the next computer generation. Our ability to exploit such computational power, however, is checkmated not only by limitations of parallelism extraction techniques, but furthermore by increasing levels of *execution uncertainty* within the system. As device feature sizes scale below 45nm, reliability has rapidly moved to the forefront of concerns for leading semiconductor companies, with the main challenge being the scaling of system performance while meeting power and reliability budgets. To make things worse, such an unreliable computational fabric is used to concurrently execute an increasing number of applications that constantly vie for execution resources, thus furthermore making the execution environment more dynamic and unpredictable.

The unreliability in the electronic fabric, in conjunction with the unpredictability in the execution process, has motivated the incorporation of **execution adaptivity** in future multicore systems, so that computational resources can be frequently renegotiated at run-time. The challenge, however, is to attain adaptivity in conjunction with the goals that designers already face, such as computation efficiency, power and thermal management, and predictability of worst-case performance. The traditional approaches of providing adaptivity at runtime dynamically will fail to scale as we move to systems of dozens of cores. Neither do static techniques that rely solely on compiler analysis deliver efficient adaptivity though. Instead, I have proposed a set of *compiler-directed run-time optimization* techniques that can combine the advantages of both, capable of reacting to unpredictable events while at the same time exploiting intensive program information to guide runtime decisions.

Technically, this thesis addresses the increasing levels of execution uncertainty in future multicore systems induced by device failures, heat buildups, or resource competitions from three aspects. It presents several tightly-coupled techniques to either 1) maximally mitigate a source of uncertainty, such as thermal stress, or 2) precisely detect resource variations, especially the ones induced by device failures, and then 3) quickly reconfigure the execution in a predictable manner with no reliance on spare units. These techniques are developed with the considerations of minimizing power and performance impact, localizing communication and migration so as to satisfy interconnect constraints, and ensuring high predictability so as to meet worst-case performance constraints of mission-critical applications. The successful incorporation of these techniques in future multicore systems, I believe, will engender adaptive, scalable architectures that can seamlessly reshape execution paths and schedules in an amortizable, high-volume, fixed-silicon fabric.

Chapter 1

Introduction

Over the past several decades, advances in silicon fabrication technologies have enabled dramatic advances in computer systems. Geometrical scaling of device sizes and die sizes has enabled the number of transistors per chip to double roughly every 24 months as stated in the well known Moore's law. The tremendous number of transistors imposes numerous technology challenges at the same time. Efficient utilization of the existing and ever increasing computational power necessitates the development of well organized systems that can efficiently extract the potential parallelism of applications. Meanwhile, technology scaling also makes devices more vulnerable to various failure mechanisms, imposing severe reliability challenges.

As single-processor systems built upon superscalar or very long instruction word (VLIW) architectures fail to respond to the well-known parallelism challenge, single-chip implementations of multiprocessor SoCs including CPUs, memories and communication architectures have become more and more popular. Examples include Stanford's Hydra [36], MIT's Raw [84], IBM's Cell [48], Sun's Niagara [50], and AMD's Opteron [52]. With the number of cores per die projected to double every two technology generations according to ITRS reports [2], chips of 100s to 1000s of cores are expected to be used even in common consumer applications of the next decade [2].

While currently multicore platforms are preferentially used for server and desktop systems, in the near future, they are expected [2] to be widely employed by various types of systems, including defense, consumer, medical, and networking/communication. The drastic increase in the number of available cores tempts designers to construct systems

that can run a large number of applications in parallel, while a dramatic increase in the diversity and the complexity of these applications is also expected. Efficient utilization of the ample hardware resources requires these applications to be decomposed into fine-grained concurrent tasks. Yet at different program phases, these applications typically exhibit diverse amounts of parallelism [92]. As a result, high levels of variability are expected both in the number of applications and in the number of cores an application needs. In such a dynamic execution environment, computation needs to be organized in an adaptive manner so that resource demands of various applications can be constantly renegotiated at run-time.

Yet the issue of application diversity is not the only aspect that is dictating the need for an adaptive organization of computations. The aforementioned degradation in device reliability may end up creating large variations in run-time resource availability, thus reinforcing the need for execution adaptivity. With the device shrinking projected to reach beyond 18nm in scale by 2015 [1], issues that were considered as second-order effects in the past, such as Soft-breakdowns (SBD) in device gate oxide, Negative Bias Temperature Instability (NBTI) in PMOS threshold voltage, Electro-Migration (EM) in copper interconnects, and dielectric breakdown in low-k materials [11], become clear threats for systems in near future technologies. These accentuated electronic effects may cause irreversible damage to a device, leading to *permanent faults*. Meanwhile, *transient faults*, which can be caused by alpha-particle strikes, cosmic rays, or radiation from radioactive atoms [47], are also expected to increase by orders of magnitude due to the reduced voltage and the resultant tighter noise margins. In fact, single event upset caused by cosmic particles has already been observed in large amounts in memory systems and sequential logic state elements. Similar transient faults have started to be observed in combinational logic as well [47, 78].

Not only is the fault rate projected to be high, but also a high variance in the duration of fault manifestation is to be expected in future computer systems. In particular, technology experts warn about an increase in *intermittent faults* – faults which occur frequently and irregularly for a period of time, commonly due to process variation or in-progress wear-out, combined with voltage and temperature fluctuations among other factors [12, 22]. These factors together cause the duration of fault manifestation to vary across a wide range of timescales. For instance, voltage fluctuations are typically

short-lived, on the order of several to hundreds of nanoseconds [12, 43]. Temperature fluctuations alter a device's timing characteristics over millisecond to second time scales [69]. Finally, as wear-out progresses over the course of days, it may even cause intermittent faults to become frequent enough to be classified as permanent [22]. Such a diverse behavior of fault manifestation brings further challenges to system designers. Fault tolerant solutions proposed solely for permanent or for transient faults become insufficient, as they rely on a fault to consistently manifest or never re-manifest itself. Instead, cost-effective solutions capable of uniformly detecting all faults, identifying the fault type, and then adaptively recovering the execution are necessitated.

The aforementioned reliability issues become even more severe when the impact of temperature is considered. While thermal buildup [55] even now is a significant concern, it will exacerbate as a result of the continuous scaling of circuit current, clock speed and device density. As higher temperature accelerates the chemical processes taking place inside the chip, the system will become more vulnerable to failure mechanisms such as electromigration and dielectric breakdown [11]. It has been reported that a mere 10 – $15°C$ rise in the operating temperature could halve the life span of the circuit [87]. Higher temperature also reduces the mobility of the charge carriers, thus diminishing the switching speed of the transistors. The amount of delay faults is expected to double [34] for every $10°C$ increase in temperature. Moreover, as every $20°C$ increase in temperature causes a 5-6% increase in Elmore delay in interconnects [5], clock skew problems also become noticeable for temperature spatial variations of around $20°C$ and above. Finally, these reliability and performance issues are worsened by the positive feedback loop between temperature and leakage power; leakage current is exponentially related to temperature, exacerbating further the effects of the positive feedback.

1.1 Need for Execution Adaptivity

The projected degradation in device reliability, in conjunction with the high variability in application resource utilization, imposes stringent requirements for future multi-core and many-core systems to display *execution adaptivity*. In the face of failing cores, it will be out of necessity to suspend the failing computation. In the face of thermal stress, it is preferable to shut or cool down the cores approaching thermal buildup thresholds, thus

in turn diminishing the extent of the factors that contribute to fault occurrences. Finally, in the face of resource competitions, the reallocation of one application's resource in favor of another application will boost overall throughput of the system. Not only will future multicore platforms be adaptive in the sense of deallocating cores, but conversely they will be adaptive in the sense of pulling cores back into operation once the fault durations have elapsed, heat driven throttling needs have abated, or resource competition pressures have diminished.

Frequent renegotiation of resources is not the only capability an adaptive system needs to have, though. While renegotiation addresses runtime resource variations from a "*recovery*" perspective, techniques capable of "*detecting*" resource variations or even "*precluding*" their occurrences are also indispensable. Prevention and detection may be easily achievable for certain causes of variations, such as resource competitions, as they can be directly monitored by the operating system (OS). If a global view of the application resource requirements is available, the OS may even be able to pre-adjust resource footprints to prevent a potential competition. In an analogous manner, heat-induced resource variations can also be quickly detected, as long as on-chip temperature sensors have been pre-fabricated. Moreover, as temperature is a function of power density and floorplan characteristics [41], a potential thermal stress can be prevented through either reducing power consumption in the most-overheated components, or deterministically controlling their access activity to balance power density.

Compared to resource competitions and thermal stress, the occurrence of execution faults is completely unpredictable and cannot be directly prevented.[1] Instead, a highly efficient detection mechanism, capable of scaling to the projected high fault rate is necessitated. Given the highly variable fault duration, techniques proposed solely for permanent faults [7, 79] or for transient faults [66, 72, 73] are insufficient. Circuit level replication techniques, such as Razor [28], fail to respond to the challenges imposed by the highly variable manifestations of the faults given their high cost and inflexibility. Instead, an architectural-level technique, capable of delivering full fault detection capability within minimum performance and heat overhead is necessitated.

Finally, providing execution adaptivity, while a highly desirable goal, needs to

[1] An indirect prevention of execution faults is possible, though. For instance, temperature-induced failures can be diminished through temporarily suspending the computation on a core that is sustaining thermal stress.

pay utmost attention to questions of performance, power, and system organization, if it is to be industrially relevant. Performance, which has always been of great importance, is becoming even more crucial given the projected high resource variations. Upon the detection of a variation, it is essential to minimize both the overhead in making reconfiguration decisions and the overhead in migrating computation, so as to reconfigure the execution before the next resource variation occurs. Power consumption also needs to be strictly controlled, since the solutions of variation detection and resource renegotiation should not create significant power overhead that may end up intensifying local heat buildup. Clearly, as performance, power and heat characteristics are largely determined by system organization, especially the underlying fabric topology, efficient adaptivity solutions also need to be topology-aware.

1.2 Challenges to be Addressed

Delivering the aforementioned execution adaptivity raises numerous technical challenges. In particular, the needs for resource renegotiation, reliability enhancement, and heat reduction should be attained in conjunction with the goals that designers already face, such as computation efficiency, power and thermal budget, and predictability of worst-case performance.

Fast, predictable, and localized execution reconfiguration

An adaptive multicore system should be able to reconfigure its execution to either withstand a core unavailability, or make use of a previously deallocated core once the cause of unavailability has been cleared. As variations of resource availability are expected to be frequent, the reconfiguration process should be as quick as possible. Predictability of worst-case performance also needs to be guaranteed, as multicore platforms are expected to be commonly used by deadline-driven realtime applications [2]. Meanwhile, as the interconnect cost becomes increasingly expensive in terms of both power and performance, workload migration should be confined within a neighborhood as well.

The strict requirement of fast and predictable reconfiguration cannot be straightforwardly attained through adopting pure run-time techniques [18, 26, 89]. Although these techniques naturally deliver adaptivity, the dynamic reactions waste significant com-

putation power and, due to their sub-optimal nature, unpredictably impact each application. Not only do these techniques need to collect workload information from every corner of the chip, but furthermore, the quality of reconfiguration decisions is determined by the complexity of the scheduling algorithms employed. The resultant communication and computation overhead drastically increases as the number of cores in the system grows, thus limiting the applicability of these techniques to systems of 100s of cores.

Neither do static techniques [21, 33, 44] that rely solely on compiler analysis deliver efficient adaptivity though. It is true that compared to pure run-time techniques, compiler-directed scheduling is more predictable and cost effective. Sophisticated application information can be extracted, and aggressive heuristics can be employed to globally balance the workload. However, the quality of static schedules degrades significantly in a dynamic environment. While it is possible for the compiler to generate multiple schedules that match diverse resource availability constraints, the numerous adaptivity needs are difficult to plan and compile for. The overall impact of a resource variation on a statically generated schedule is determined by the exact time at which a variation in resource availability occurs, which is essentially infeasible to predict statically.

Given the inability of pure run-time and pure compile-time techniques to deliver execution adaptivity efficiently, it is desirable to develop a *hybrid* approach that can combine the advantages of both, capable of reacting to unpredictable events while at the same time exploiting intensive program information to guide runtime decisions. Yet delivering this hybrid scheme also raises numerous technical challenges. What is the form of compiler analysis that embeds numerous reconfiguration possibilities in static schedules in a compact manner? How to localize task migration in the reconfiguration process? How is the reconfiguration process controlled by the runtime system? How to organize the underlying multicore fabric to minimize workload migration overhead? Addressing these questions requires the development of a collaborative framework between the OS, the compiler, and the architecture, with the constraint of adaptivity taking center stage in the design process.

Full variation detection capability within minimum overhead

To sense that an execution rearrangement is to be effected, an adaptive system also needs frugal technical support for detecting resource variations induced by faults, thermal

stress, and resource competitions. Among these various issues, the detection of execution faults is most challenging. On one hand, the projected high fault rate [22, 34, 47] argues for solutions of maximal efficiency. As full fault coverage is still necessitated, however, techniques [32, 71, 88] that reduce fault detection overhead at the cost of significantly increased rates of undetectable faults are not applicable. On the other hand, the diverse behavior of fault manifestation argues for solutions capable of uniformly detecting all faults and then identifying the fault type. As the duration of fault manifestation varies over nanosecond to second time scales, detection mechanisms that rely on a fault to never re-manifest [66, 72, 73] or consistently manifest [7, 79] itself become insufficient.

Unlike regular storage structures that can be efficiently protected using Error Correcting Codes (ECC), computation at various pipeline stages typically exhibits irregular patterns, thus requiring the entire execution to be *duplicated* in order to detect arbitrary faults. Unfortunately, traditional duplication-based fault detection approaches impose significant overhead either in checking execution results, or in constantly synchronizing two computation copies for value checking [91]. While extra buffers [72, 62] can be inserted into the architecture to relax synchronization conditions, these centralized hardware structures need to be constantly accessed, thus possibly ending up becoming thermal hotspots.

The projected high fault rate additionally imposes strict requirements on the development of a light-weight checkpointing scheme. Whenever a fault is detected, the computation needs to be restarted from a previously saved clean state, i.e., a *checkpoint*. In order for the computation to progress, a new checkpoint should be established before the next fault occurs. The higher the fault rate is, the more frequently the computation needs to be checkpointed. Yet in order to checkpoint the computation more frequently, the associated computational overhead needs to be strictly controlled. Typically a checkpoint is imposed on the processor state and the corresponding memory footprint. Yet checkpointing the memory, as OS needs to be involved in, is usually a quite expensive process [13, 57] that induces significant context switch overhead. Accordingly, the development of a light-weight checkpointing scheme requires the memory to be strictly protected from being polluted by execution faults.

In summary, the development of an efficient fault detection scheme for future multicore systems of elevated rates and diverse types of faults imposes *three* requirements,

namely, attaining full detection capability within a minimum level of result comparison and hardware duplication, maximally relaxing checking-induced synchronization conditions with no reliance on any centralized hardware buffer, and minimizing checkpointing overhead through strictly protecting memory against execution faults.

Maximum mitigation of thermal stress

As the system fault rates exponentially increase as peak temperature rises, mitigation of thermal stress can in turn reduce resource unavailability induced by both heat buildup and execution faults. However, as overall system performance is still of great importance, temperature reduction should not be attained through globally stalling [15] or slowing down [80, 81] the computation of an overheated core.

As temperature is determined by power density as well as the heat dissipation speed, the temperature distribution on a chip is typically asymmetric. This asymmetry can be observed at both the core-level and the microarchitectural-level; not only may the tasks assigned to different cores exhibit diverse power behavior, but furthermore the various components of a core exhibit distinct size and access characteristics. This observation implies that peak temperature of the entire chip can be reduced through shifting computation from a hot resource to a relatively cool resource. In particular, in an adaptive system, execution schedules should be generated in such a way that the "hot" tasks are distributed across various cores at different time. Meanwhile, for each individual core, the peak temperature of the most overheated module should be strictly controlled as well.

To perform thermal-aware task scheduling, the scheduler needs to obtain extra information regarding system topology and power consumption characteristics of tasks [23]. The challenge, however, is for the scheduler to consider temperature constraints simultaneously with the other scheduling constraints, such as communication minimization, workload balance, and execution adaptivity. The resultant drastically increased scheduling complexity would impose significant computation overhead, if the scheduling process is to be performed at runtime. This observation in turn argues for a *hybrid* scheduling approach, wherein the compiler can exploit intensive program information to guide runtime decisions.

Controlling the peak temperature of each individual core requires the identification of the most overheated module. Previous research [81] indicates that the register file,

due to its high utilization and relatively small area, is one of the hardware units most likely to overheat. Moreover, due to the fact that 90% of the execution time is spent on loops where only a small subset of registers is repetitively accessed, register file accesses also exhibit high asymmetry during program execution. This asymmetric register utilization may lead to considerable temperature differentials. Yet pure static register reassignment techniques [40, 100] cannot completely eliminate the access asymmetry to each individual register, since such asymmetry directly derives from the asymmetric variable utilization of the program.

Given the inability of the compiler to completely balance register accesses, a dynamic mechanism needs to be developed to physically remap heavily accessed logical registers prior to local heat buildup. Yet this task cannot be accomplished through the use of a hardware renaming table [77]. Not only does such a mapping table impose notable levels of of energy consumption, but more crucially, the table needs to be accessed using the skewed register names, at a frequency no lower than that of register file accesses, thus ending up itself becoming a temperature "hotspot". This observation indicates that the challenge for temperature-aware register remapping is to achieve this goal in a *table-free* manner; the system needs to deterministically keep track of run-time register usage and register mapping information. It is therefore necessary to establish an agreement between the compiler and the runtime remapping hardware, so that regularity can be embedded within consecutive register accesses.

1.3 Contributions of This Thesis

The development of an adaptive execution framework for the next generation of computer systems is a challenging task. In particular, to address the technical challenges identified in the last section, the system needs aggressive technical support from the OS, the compiler, and the architecture. A collaborative optimization framework needs to be established.

In light of this observation, we establish in this thesis a *compiler-directed runtime optimization* framework, capable of efficiently coupling static program information with runtime optimizations. Under this optimization framework, we furthermore introduce several tightly-coupled techniques that contribute to the development of reliable and

adaptive multicore systems from the perspectives of *variation mitigation*, *variation detection*, *execution recovery*, as well as *architectural reorganization* for a cost-effective implementation of all these functions. The contributions of these techniques are summarized as follows:

- A compiler-directed task scheduling framework, capable of spawning regular, transformable, and high-quality execution schedules in the face of unpredictable runtime resource variations. The pre-optimized schedules can be adaptively applied upon runtime resource variations, thus delivering high-speed and low-cost reconfiguration without any rescheduling decisions needing to be made on the fly.
- A scalable and shareable storage organization, capable of delivering high-speed communications within a neighborhood. This organization enables the simultaneous migration of multiple tasks between distinct cores without inducing any interferences or network congestion. The inherent redundancy furthermore assures the connectivity of the entire platform in the case of core or interconnect failures.
- An efficient fault detection mechanism, capable of minimizing fault detection overhead while at the same time delivering full fault detection capability. Through performing fault detection and checkpointing at the cache-memory interface, two threads are able to run independently without constantly synchronizing for value checking, while the memory is strictly protected against execution faults.
- An approach to reduce system peak temperature, through exploiting useful application information to fine-tune a microarchitectural component intelligently. This technique balances power density in the most overheated components, to wit, the register file in each individual core, thus attaining temperature reduction and hence enhancing overall system reliability at almost no cost.

To ensure that adaptivity can be attained in conjunction with the goals that designers already face, all the aforementioned techniques are developed with the considerations of minimizing power and performance impact, ensuring high predictability of worst-case performance, and localizing communication and migration so as to fulfill interconnect constraints. The successful incorporation of these techniques in future multicore systems, I believe, will engender adaptive, scalable architectures that can seamlessly reshape execution paths and schedules in an amortizable, high-volume, fixed-silicon fabric.

1.4 Roadmap

The rest of the thesis is organized as follows. Chapter 2 reviews the current state-of-art and analyzes the limitations of existing solutions of resource management, fault tolerance, and heat reduction. Chapter 3 presents a system overview of the envisioned adaptive multicore framework, focusing on the collaboration between the OS, the compiler, and the architecture. Chapter 4 introduces a compiler-directed task scheduling framework, wherein adaptivity is directly embedded into static schedules and task migration is localized to satisfy interconnect constraints. The corresponding storage organization for minimizing migration overhead and accelerating neighborhood-centered communications is presented in Chapter 5. Chapter 6 presents an architectural fault detection and checkpointing scheme, wherein the cache design is extended to implement fault detection, checkpointing and recovery. Chapter 7 presents a compiler-directed register shuffling technique that effectively diminishes register access asymmetry with no reliance on any hardware renaming table, thus preventing local heat buildup. Finally, Chapter 8 summarizes the adaptive multicore framework and subsequently outlines a set of possible future research directions.

Chapter 2

Related Work

While the underlying computational fabrics become increasingly dynamic and unreliable, the applications to be held by these fabrics impose stricter requirements of durability and safety. Researchers therefore have started to address the increasing levels of computation uncertainty from various perspectives.

First of all, given the diversity of the possible causes of execution uncertainty, researchers have focused on the *characterization* of the various causes of uncertainty, including device failures and thermal stress, as well as the *modeling* of their effects on system execution. Various circuit-level fault models have been built, architectural thermal models have been constructed, and the system level fault manifestation behavior has been studied. Meanwhile, researchers have also developed various techniques to overcome execution uncertainty through either *preventing* a cause of uncertainty from occurring, or *masking* its effect, or *detecting* a cause and then *recovering* the system. The *prevention* strategy has been adopted for the control of chip-wide temperature, through the adjustment of the floorplan and the layout of various hardware modules, or through the adjustment of execution schedules to distribute "hot" tasks across diverse computational resources. The *masking* strategy has been adopted for the toleration of device failures at the circuit-level, while the *detection-recovery* strategy has mainly been adopted for the toleration of device failures at the architectural-level. Yet compared to thermal stress, the toleration of device failures is more challenging, particularly due to the diverse behavior in fault manifestation. A large number of fault detection mechanisms therefore have been developed. These techniques target either faults in specific components, such as storage

units or the control flow [65], or specific types of faults, such as transient faults [72] or permanent faults [79]. Additionally, once a fault has been precisely identified, it is necessary to isolate the corresponding components through a dynamic *reconfiguration* process. Researchers have therefore examined the various possibilities for reconfiguring the system at various levels, for instance, at the component level to mask permanent faults [3], or at the core level to migrate the workload [89].

Despite the existence of various types of solutions for overcoming execution uncertainty, these solutions fall short of addressing the challenges induced by not only an elevated rate, but furthermore a diverse behavior of execution uncertainty in future multi-core and many-core systems. Due to the associated high overhead and the lack of predictability, these solutions fail to deliver the envisioned execution adaptivity in conjunction with the goals that designers already face, such as computation efficiency, power and thermal budget, and predictability of worst-case performance. A detailed review of the current state-of-art, as presented in the remaining parts of this chapter, clearly illustrates the limitations of existing resource renegotiation, reliability enhancement, and heat reduction techniques.

2.1 Resource Renegotiation

The increasing possibility of resource variations requires a reconsideration of the critical issue of scheduling the tasks of an application onto the cores of the target system. Traditionally task scheduling can be performed either dynamically at run time, or statically during compilation. The former approach delivers adaptivity straightforwardly, yet the associated high overhead challenges its scalability as we move to systems of 100s of cores and similar magnitude of concurrent tasks. The latter approach is more cost-effective and delivers worst-case predictability. Yet the numerous adaptivity needs are difficult to plan and compile for.

Run-time solutions

In dynamic scheduling, the OS is typically employed to monitor resource availability and schedule tasks (that are ready to be executed) only to available cores, thus naturally delivering resource reallocation upon runtime variations. In [18], upon a pro-

cessor failure, a *dynamic* rescheduling approach is employed to reassign its workload to the remaining available processors. A similar approach is employed in [89]. The OS is configured to use more virtual processors than the number of physical cores, thus tolerating variations in the availability of physical cores. The approach proposed in [26] adapts application execution to the varying CPU availability for the purpose of minimizing the energy-delay product (EDP). It relies on a helper thread, running in parallel with the application, to determine the ideal number of cores and the system configuration at any given point in execution.

While pure run-time techniques naturally deliver *execution adaptivity*, the dynamic reactions waste significant computation power and, due to their sub-optimal nature, unpredictably impact each application. More specifically, if a core becomes unavailable, its workload can be migrated either straightforwardly to its neighbor(s), or to the cores with minimum workload. The former ad-hoc decision incurs negligible overhead yet typically induces workload imbalance, since the adjacent cores may have already been assigned a significant amount of workload. In comparison, the latter decision displays more intelligence, yet imposes significant communication and computation overhead. First of all, to globally balance the workload, the dynamic scheduler needs to collect workload information from every corner of the chip. Clearly, the associated overhead drastically increases as the number of cores grows. Meanwhile, as the quality of migration decision is determined by the complexity of the scheduling algorithm employed, the generation of high quality decisions also introduces significant runtime scheduling overhead, at an amount superlinearly proportional to the number of concurrent tasks in the system. For example, the helper thread proposed in [26] needs to monitor the applications' EDP values through collecting performance counter information, and to determine the next system configuration through curve fitting methods. The resultant appreciable communication and computation overhead limits the applicability of these functions to future multi-core and many-core systems wherein the interconnect cost is projected to be high.

Compile-time solutions

Compared to pure run-time techniques, static scheduling is more cost effective, as it imposes neither run-time scheduling overhead nor communication overhead for collecting workload information. As scheduling is performed offline at compile time, not

only can sophisticated application information be extracted, but also aggressive heuristics can be employed to globally balance the workload. As a result, commercial embedded systems, such as MARS [51] and XBW [19], typically use static scheduling to ensure timing predictability and other safety-related properties, such as design simplicity and testability. However, the quality of traditional static schedules degrades significantly in a dynamic environment. In fact, when generated offline, static schedules are typically confined to the case of a fixed number of PEs, implying that a resource reduction usually dooms the entire schedule to uselessness.

To enable static schedules to tolerate resource variations, *redundancy* needs to be built within the system. As a result, traditional static scheduling techniques either keep spare processors [21] that can be used to replace failing ones, or back up each task [33, 44] so that upon the failure of the primary copy, the backup copy of the task can be invoked. The schedule should also have sufficient timing slack embedded [46, 61] so that upon a core variation, recovery and migration can be carried out before any of the tasks reaches its deadline.

Maintaining spare cores would be an efficient solution for fully connected systems, wherein one spare core is able to replace any of the remaining cores. Unfortunately, in future multicore systems of hundreds of cores, full connection is impossible, which in turn limits the replacement capability of spare cores. Computation on a failing core cannot be directly migrated to a spare core if the two are not physically adjacent. In such systems, it would be preferable to allocate more spare cores to regions of higher levels of resource variations. Unfortunately, such an allocation strategy is impossible to determine at compile time. As examined before, a core may become unavailable due to various reliability, thermal or resource competition reasons, with neither the occurrence nor the duration of unavailability being predictable *a priori*.

In the *primary-backup* approach, schedules should be generated in such a way that the primary and the backup copies are scheduled on distinct processors. To reduce the associated replication cost, backup overloading and backup deallocation are introduced in [75]. Backup copies of multiple independent tasks are allowed to be scheduled on the same or overlapping time intervals on a processor, and the resources reserved for a backup task are reclaimed when the corresponding primary tasks complete successfully. These techniques effectively improve resource utilization, while they are restricted to tolerating

only a single fault among the tasks with overlapped backup schedules.

In [58], a hierarchical scheduling approach is proposed. Task graphs are partitioned into disjoint regions, for which multiple schedules with diverse performance and power characteristics are generated. Such schedules are adaptive applied at runtime so as to explore energy-performance tradeoffs. At first sight, it seems that this approach can also be employed to overcome runtime resource variations, through the static generation of multiple schedules that match diverse resource availability constraints, followed up by a dynamic switch to a new schedule upon a variation in core availability. Unfortunately, the overhead for storing all these pre-optimized schedules in memory is quite high. More crucially, the numerous adaptivity needs are difficult to plan and compile for, since it is infeasible to predict *a priori* the exact time at which a variation in resource availability occurs. Due to this limitation, these pre-generated schedules exhibit no timing regularity; the execution order of tasks is not necessarily identical throughout the various schedules, implying that a switch between these schedules requires a search process for identifying the exact starting point. Neither do these schedules exhibit spatial regularity; a task may need to be shifted across multiple PEs during the reconfiguration process, thus inducing significant migration overhead and hence unpredictability of worst-case execution.

2.2 Reliability Enhancement

Increasing research attention has been paid to the incorporation of reliability enhancement solutions into computation systems, not only because of the elevation in fault rates, but also because durability and safety have been identified as an important design criterion for systems that hold server, defense, or medical applications. Semiconductor companies have started as well to incorporate reliability support into their newly-released designs, such as Intel *QuickPath*, and IBM *Power6* [59].

Given the severe power and cost constraints of modern multicore architectures, the need for maximally efficient fault tolerance methods becomes increasingly critical and urgent. It is thus essential to evaluate a technique not only by its *effectiveness* in detecting errors and recovering execution, but more importantly by its *efficiency* in terms of the associated performance, energy and hardware overhead.

Fault detection

In general, the detection of faults necessitates *redundancy*, at an amount inversely proportional to the regularity of the hardware components. Storage structures, such as caches and memory, have regular patterns, thus enabling the use of Error Correcting Codes (ECC) and parity bits. Faults in instructions or in control flow can also be effectively detected by *signature monitoring* techniques [65, 73] through exploiting internal redundancies. As a comparison, computation structures typically exhibit irregular patterns, thus requiring the entire execution to be *duplicated* in order to detect arbitrary faults at various pipeline stages.

Conventional duplication-based approaches employ either a *time redundancy* or a *spatial redundancy* strategy for fault detection. Time redundancy, achieved by executing a task on the *same* hardware multiple times, is only effective for transient fault detection. For instance, a number of software-based fault detection techniques replicate each instruction and add checking instructions to compare the results [73, 66]. These techniques offer the flexibility of turning redundancy on and off in the generated code, while at the same time imposing significant performance overhead due to the replication and checking of instructions. Some researchers have proposed a set of techniques [99] capable of reducing such overhead by 50%, yet at the cost of reduced fault coverage.

Space redundancy techniques, on the other hand, duplicate a single task on multiple processors. Not only transient, but furthermore intermittent and permanent faults can be detected, albeit at a cost of sizable hardware overhead. In the *Compaq NonStop Himalaya* system [91], each pair of redundant instructions is executed on two tightly-coupled cores on a cycle-by-cycle basis. An instruction cannot be committed until its correctness has been verified. The Dynamic Implementation Verification Architecture (DIVA) [7] employs k simple checker cores to detect errors in a k-wide superscalar processor. The BulletProof pipeline [79] uses built-in self-test to detect and precisely identify the faulty unit. As the technique relies on a fault to consistently manifest itself, it is only effective for permanent faults but not transient faults.

To attain full detection capability, previous techniques usually check all the store instructions, and hold an instruction off commitment until its correctness has been verified. This highly synchronized execution model significantly increases the latency of a

single instruction, thus delaying the release of hardware resources, such as physical registers and ROB entries in the architecture. Moreover, in the case when two redundant tasks are being executed on distinct cores [91], both copies incur mis-speculated branches and cache misses independently, leading to less efficient resource utilization and unnecessary power dissipation. To relax the lock-step execution model, extra buffers are necessitated so as to enable one thread to forward data to the other. For instance, the redundant multithreading approach (originally proposed in [72] for SMT cores and extended to CMPs in [62]) requires an *output comparator* to verify execution results, as well as an *input replicator* to ensure that both threads read identical input data. Typically these two components are implemented through two centralized shared structures, namely, a *Load Value Queue* and a *Store Queue*. However, even with these queues, the two threads still need to synchronize, as the leading thread needs to be stalled if either queue is full and the trailing thread needs to be stalled if either queue is empty.

To reduce duplication overhead, researchers have developed a set of *partial redundancy* techniques. Opportunistic Fault Tolerance [32] duplicates instructions only during periods of poor single-thread performance. ReStore [88] does not explicitly duplicate instructions yet considers mispredictions among highly confident branch predictions as symptoms of faults. Slipstream [71] combines partial duplication and confident predictions through creating a reduced alternate thread wherein many instructions are replaced with highly confident predictions. These techniques sizably reduce duplication overhead, however, at the cost of significantly increased rates of undetectable faults; faults in non-duplicated instructions cannot be detected, if they do not lead to branch mispredictions.

Execution recovery

In addition to fault detection, the achievement of fault resilience also necessitates *execution recovery* techniques, which should either preclude faults from modifying computation states, or roll the execution back to a previously saved clean state, i.e., a checkpoint upon a fault. The first strategy is typically employed together with highly-synchronized value checking. For instance, in both redundant multithreading cores [86] and lock-step multiprocessors [31], instruction results cannot be committed into registers or the cache until their correctness has been confirmed. In contrast, the *checkpointing and rollback* strategy allows results to be written into registers and the memory hierarchy

without being compared, yet needs to constantly check and save the computation state. Upon the detection of any fault, the system reloads the most recent checkpointed state to recover computation.

To establish a checkpoint, one set of techniques utilizes the virtual memory translation hardware [13], to create a backup copy before modifying any memory page. Another standard technique consists of the use of a recovery cache to record all the data written in memory that are part of a checkpoint state [57]. Every store to a memory location must be preceded by a load to maintain the data in the recovery cache. These hardware-oriented backup techniques impose not only storage but also performance overhead constantly on the system. To reduce such overhead, the CARER scheme [42] uses a normal cache with a writeback update policy to assist rapid rollback recovery. The work is subsequently extended to shared-memory multiprocessors through synchronizing the processors whenever one needs to take a checkpoint [45]. Software checkpointing [14] has also been proposed, yet at the cost of additional support required from the compiler and/or the OS.

While a single task can be checkpointed independently, a parallel application requires the coordination of dependent tasks in the checkpointing process. *Coordinated checkpointing* [17, 82] can be attained by stalling and validating all computations and communications in an ordered manner. *Uncoordinated checkpointing* [64], in contrast, is performed independently on each core. While this strategy eliminates the global synchronization requirement of coordinated checkpoints, multiple checkpoints need to be stored on each core, and the rollback process furthermore requires the identification of a checkpoint with a consistent state. To eliminate potential *domino effects* wherein no consistent checkpoint can be found, researchers have introduced extra constraints on checkpointing sequences based on, for example, the communication patterns [37] of applications.

Another set of techniques aim to continue to use a core despite permanent faults, through the use of finer-grained fault masking strategies. These techniques involve fine-grained testing, diagnosis [29], and recovery of core components. *Configurable isolation* [3] is a technique that performs reconfiguration at the micro-architectural level. When a component suffers a fault, processor resources are reallocated and partitioned dynamically so as to isolate the component and subsequently migrate its workload. *StageNetSlice* [35] is a processor core comprised of networked pipeline stages. It relies on a reconfig-

urable network of replicated processor pipeline stages to maximize the useful lifetime of the chip. In [68], the authors exploit cross-core redundancy, and use hardware to migrate offending threads to another core that can execute the operation. Clearly, the reconfigurability offered by these techniques is at the component level; the faulty functional unit, register, or cache block is isolated so that the core can still operate in a degraded performance mode.

2.3 Heat Reduction

Modeling temperature and the effects of temperature on reliability is essential for the simulation and analysis of heat reduction policies. The adverse impact of operating temperature on system reliability has been studied extensively. Researchers have built either analytical or experimental models for temperature-induced fault rate increases, such as delay violations [34], negative bias temperature instability [53], neutron-induced latchup [27], and on-chip interconnect [5]. In comparison, thermal modeling is typically accomplished by constructing an equivalent RC network of the given chip. Heat flow is analogous to the current passing through a thermal resistance in the RC network. The transient behavior of temperature is modeled by means of the thermal capacitance. Architectural thermal models of this type, such as the *HotSpot* [41], have been developed for calculating transient temperature response, for the given floorplan, package, and power consumption characteristics of various components.

In general, temperature is determined by power density as well as the heat dissipation speed. A traditional approach to accelerating the latter factor is through packaging and cooling solutions. Yet such solutions have been typically targeted for the worst case peak temperature, resulting in an extremely expensive packaging cost with the ever rising temperatures (approximately $10 per Watt above $65°C$). To keep the chip-wide temperature within the thermal capacity of the cooling package, researchers have proposed various architectural-level thermal management techniques to either reduce the access frequency to an overheated unit, or physically redistribute accesses before heat gets locally accumulated.

Access frequency reduction

System-level *Dynamic Thermal Management* (DTM) techniques control thermal hotspots by keeping the temperature below a critical threshold. Once a core reaches the thermal threshold, heat accumulation is slowed down either at fine granularity through clock gating [60], fetch toggling [80], decode throttling [9], frequency and voltage scaling [81], or at coarse granularity through periodically stopping the computation to induce cooling [15]. Obviously, slowing down or stopping the entire computation engenders significant performance degradation.

As different components exhibit diverse access and power consumption characteristics, local "hotspots" may reach critical temperature levels regardless of average or peak external package temperature of the entire chip. In particular, due to its high utilization (accessed 2–3 times per instruction) and relatively small area, the *register file* has been established as one of the most overheated hardware units in current processors [81]. Gate-level techniques, such as single- or multi-level banking [24] and bit-partitioning [49], can be employed to reduce the power consumed in each register file access and, hence, the peak temperature. Researchers have also proposed the incorporation of an extra register file [81] to increase the average idle time. Unfortunately, maintaining a duplicated register file requires the context of one register file to be periodically copied into the other. To preclude local heat buildup, the two register files also need to be physically distributed, thus engendering sizable increases in chip area and wiring complexity as well.

Although temperature increase is induced by power dissipation, it needs to be noted that power reduction techniques do not necessarily offer temperature reduction. For instance, the technique proposed in [8] switches the unused registers into hibernation to save leakage power. Yet peak temperature cannot be reduced since frequently accessed registers still consume power at levels identical to the original case. Another power-reduction technique [63] unevenly partitions the register file into two banks and maps the most frequently accessed registers into the smaller bank. However, as most accesses are directed to the smaller bank, the power density difference between the two register banks is enlarged, thus questioning its effectiveness in controlling peak temperature. The review of these techniques indicates that a power reduction technique can be employed for temperature control purposes if and only if the technique can reduce the peak power

consumption in one of the overheated components in a system.

Access redistribution

Another set of techniques aim to attain thermal management through shifting computation from a hot resource to a relatively cool resource. These techniques are effective because the temperature distribution on a chip is typically asymmetric; not only may the tasks assigned to different cores exhibit diverse power behavior, but furthermore the various components of a core exhibit distinct size and access characteristics. In light of this observation, researchers have proposed to either statically preadjust workload distribution [23], or dynamically migrate computation before heat gets locally accumulated, at the granularity of functional units [39], pipelines [39], execution clusters [20], or even cores on a single chip [69]. Overheated resources, such as the register file, can also be physically distributed into multiple clusters [20] to accelerate heat dissipation.

A detailed examination into program execution indicates that register accesses also exhibit high asymmetry. This is because 90% of the execution time is spent on loops where only a small subset of registers is repetitively accessed. This asymmetric register utilization furthermore leads to considerable temperature differentials, since most of the heat generated within a microarchitectural block is dissipated vertically to the heat sink rather than laterally to adjacent blocks [81]. To overcome this access asymmetry, thermal-aware register reassignment techniques [40, 100] have been proposed. Both techniques reduce the level of asymmetry in register accesses through mapping the most frequently accessed registers to distinct register banks. However, as both techniques need to revisit the NP-hard problem of register allocation, the quality of the solutions is determined by the quality of their heuristic algorithms. More crucially, no matter how good the heuristics are, such techniques cannot completely eliminate the access asymmetry to each register, since such asymmetry directly derives from the asymmetric variable utilization of the program. As a result, these techniques can only attain a coarse-grained access balance, at the granularity of register file banks instead of individual registers.

Chapter 3

Adaptive System Overview

The development of an efficient adaptive system addressing the technical challenges identified in Section 1.2 requires the construction of a collaborative optimization framework with aggressive technical support from the OS, the compiler, and the architecture. A *compiler-directed run-time optimization framework* is therefore proposed in this thesis. At compile time, program information can be extracted and synthesized statically. Such information is to be transferred to, and efficiently utilized by the runtime system, so that the OS can make superior dynamic decisions and architectural components can be fine tuned accordingly.

3.1 Compiler-directed Runtime Optimization

The proposed compiler-directed run-time optimization framework contributes to the development of an efficient adaptive system in various aspects, as shown in Figure 3.1. Perhaps most importantly, it can effectively reduce the overhead in adapting the computation upon unpredictable runtime variations. Such overhead consists of two parts, the overhead in making runtime re-scheduling decisions, and the overhead in migrating the code and the data sets of the tasks to be re-scheduled. The former type of overhead can be reduced through constructing a *hybrid scheduling framework* between the compiler and the OS. Through sophisticated static planning, the compiler can generate high-quality adaptive execution schedules, with pre-optimized reconfiguration decisions statically embedded. Upon resource variations, the OS can adaptively apply such schedules without

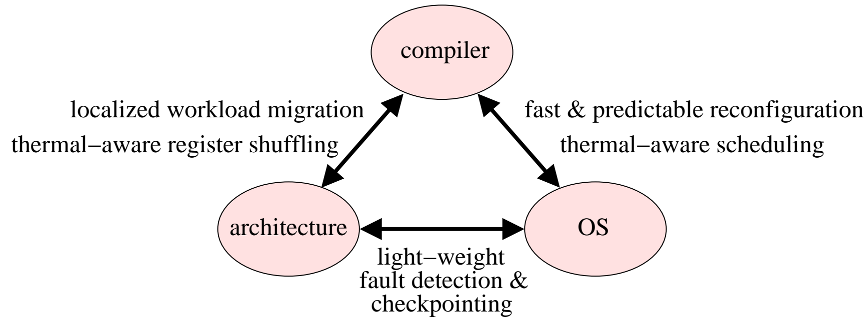


Figure 3.1: Collaborative Optimization Framwork

any rescheduling decisions needing to be made on the fly. In comparison, the latter type of overhead can be reduced through a collaboration between the compiler and the architecture. In the face of the increasing interconnect cost in future systems, the compiler can pre-optimize the adaptive schedules in such a way that task migration is localized yet still sufficient for workload balance. Meanwhile, the system architecture, specifically, the storage units (i.e., cache and memory) can be organized in a locally shareable manner, thus allowing tasks to be migrated between adjacent cores with no reliance on physically moving the code and data set around.

The exploitation of compiler-directed optimizations is additionally beneficial for reducing system peak temperature. At the core level, it is clear that temperature can be effectively balanced by the compiler; through performing aggressive thermal-aware task scheduling, the compiler can, without imposing any runtime overhead, effectively prevent any core from constantly being assigned "hot" tasks. Perhaps a more interesting observation is that within each individual core, the compiler can also diminish a local temperature hotspot induced by unbalanced component accesses. Taking the register file for example, the compiler can deterministically control the dynamic register remapping so as to balance the accesses to individual registers for heat reduction purposes. Through exploiting the fact that no fixed, preordained correspondence exists between program variables and register names, the compiler can establish a certain property between consecutive accesses to each register, thus enabling the hardware to redirect register accesses with no reliance on a mapping table.

Among the various issues that may induce variations in resource availability, execution faults exhibit the highest level of unpredictability. As a result, compared to work-

load migration and heat reduction, the utilization of compiler-extracted application information for fault detection purposes might be less effective. Although program information such as execution invariants and the range of execution results [76] can be utilized to quickly identify a certain set of faults, due to the diverse behavior in fault manifestation, such compiler-extracted information proves insufficient in providing full fault coverage. Instead, an efficient achievement of full detection capability should rely on a dynamic collaboration between the OS and the architecture, as shown in Figure 3.1. By providing architectural support for fault detection purposes, the OS can efficiently monitor core availability and coordinate the checkpointing process of dependent tasks. Specifically, an architectural examination indicates that caches, which serve as temporary storage for the main memory, can possibly be utilized to temporarily hold unconfirmed execution results for fault detection purposes. The architecture can be tuned in such a way that two redundant tasks share a single cache, with one task capable of directly checking the execution results of the other, thus efficiently delivering full fault detection capability. Meanwhile, as only confirmed results are allowed to be written to the lower level storage, the checkpointing overhead can be strictly controlled as well.

To provide a clearer picture of the proposed collaborative optimization framework, we herein summarize the tasks to be performed by the compiler, the OS, and the hardware.

The **compiler** plays a crucial role in guiding runtime reconfiguration decisions and preventing local heat buildups. In brief, it is responsible for:

- Generating task schedules with the consideration of both performance and reliability constraints. The goal is not only to embed reconfigurability into the schedules, but also to separate “hot” tasks into PEs that are not physically close to each other.
- Embedding regularity into static register names to enable a deterministic register access balance for heat reduction purposes.
- Extracting the characteristics of the reconfigurable schedule to guide dynamic workload balance. These characteristics include the minimum and the maximum number of cores needed to execute a schedule block, as well as the control and data dependences between schedule blocks.

With the statically extracted scheduling information at hand, the **OS support** needed in the proposed adaptive system is minimized. In brief, only a subset of standard

OS functionality is needed, including monitoring core availability and program resource demands, and signaling adaptivity needs, if necessary.

- Monitoring the status of a core, which can be either *busy*, or *idle*, or *unavailable* due to thermal stress or device failures. The OS therefore needs to keep track of the information regarding fault detection and the temperature of each PE.
- Dispatching pre-optimized schedule blocks to a set of idle cores. The statically extracted information regarding *inter-block dependences* will be utilized to determine if a schedule block is ready for execution, while the minimum and maximum number of cores needed for that schedule block will be utilized to select an appropriate set of cores.
- Adjusting application resource footprints to handle unexpected resource requests or unpredictable core failures. To effectively hide the decision-making latency, the OS adopts a 2-step approach. At the beginning, a reconfiguration step is invoked so that the corresponding application can isolate the problematic core. Then, the OS will check the availability of cores and the predicted resource requirements of various applications to make a globally optimized reallocation decision.

Finally, to support task migration, fault detection, and heat reduction, the underlying **architecture** needs to be extended in three directions.

- Reorganization of the system topology to locally share storage units among cores, so as to mitigate task migration overhead and accelerate neighborhood-centered communications.
- Extension to the cache design to implement a light-weight fault detection and checkpointing scheme.
- Extension to the register file design to implement a deterministic register shuffling mechanism.

3.2 Hybrid Scheduling for Resource Management

To concretely illustrate the advantages of the proposed optimization framework in efficiently delivering adaptivity support, we here provide a system overview of the

proposed resource management approach, focusing on the collaboration between the OS and the compiler.

To attain *execution adaptivity* in a fast and predictable manner, the proposed multi-core system employs a hybrid, hierarchical scheduling approach. At compile time, a static scheduler is responsible for generating reconfigurable schedules capable of tolerating core degradations for each application individually. Meanwhile, the information regarding the minimum and the maximum number of cores needed to execute these schedules is extracted to guide dynamic workload balance. At runtime, with the statically extracted scheduling information at hand, the OS only needs to perform scheduling at the application level. Specifically, the OS is responsible for dispatching the pre-optimized schedule blocks to a set of cores, monitoring core status, and adaptively adjusting application resource footprints upon variations in resource availability.

The collaboration between the static scheduler and the OS in adapting system execution can be illustrated more clearly by considering an example wherein an unexpected device failure occurs, resulting in one of the cores allocated to to the application, denoted as $App\ R$, to fail during execution. In general, the handling of this case can be partitioned into **three** major steps. Initially, upon a reduction in the number of available cores, $App\ R$ reconfigures its own execution to withstand this failure, with the reconfiguration decisions made based on the compiler-generated adaptive schedules. Meanwhile, $App\ R$ reports the failure to the OS and sends a request for an extra core. Once the OS receives this resource request, it determines whether or not to allocate an extra core to $App\ R$. Under the steady state assumption that all the cores are in use, the OS prioritizes through system topology considerations by examining the cores that are physically connected to the remaining cores of $App\ R$. By comparing the reconfiguration overhead against the attainable benefits of workload balance, the OS decides whether to reallocate a specific PE and furthermore what the appropriate time instance for reallocation is. Finally, if the OS has decided to reallocate the core currently used by $App\ S$ to $App\ R$, both applications reconfigure their execution in order to deactivate or utilize the core in negotiation.

The hybrid scheduling policy can effectively combine the advantages of static scheduling and dynamic scheduling. At compile time, the tasks of an application and the associated communication patterns can be largely determined. This information can be used for guiding resource reallocation decisions. The compiler can employ complex

algorithms, without imposing any runtime decision making overhead, to globally balance the workload in the reconfigurable schedules. Meanwhile, through exploiting the extra degree of freedom inherent in generating schedules, the compiler can embed regularity into the schedule at almost no cost. The schedule can be optimized in such a way that during the reconfiguration process, localized task migration is sufficient for workload balance, thus fulfilling the strict interconnect requirements. At runtime, in comparison, scheduling only needs to be performed at the *application level*, with context switches performed on all the tasks of an application in a synchronized manner. Upon a variation in resource availability, the OS only needs to signal one or more applications to perform the pre-optimized reconfiguration process, thus significantly reducing runtime scheduling overhead and hence improving system scalability.

As reconfigurable schedules are generated at compile time, they face the common challenge of static scheduling, namely, the precise delineation of the schedule in the face of dynamic variations in *task execution time*. In general, variations in task execution are typically induced by three issues: *unpredictable architectural events, resource competitions*, and *input-dependent computation variation*. Yet a careful examination shows that in the proposed hybrid scheduling scheme, these three sources of execution uncertainty can be maximally diminished.

- Unpredictable architectural events, such as *cache misses* or *branch mispredictions*, only result in variations of tens of cycles. Such small variations can be further reduced through architectural techniques such as data prefetching [101] or predicated execution [67].
- Resource competition among applications may cause a task in execution to be suspended, thus inducing unpredictability in its execution time. However, as in the proposed adaptive system, context switches are performed on all the tasks of an application in a synchronized manner, this type of interference can also be completely eliminated.
- As traditional parallelization techniques typically partition loops into a fixed number of tasks, the number of iterations in each task may be input-dependent, resulting in sizable variations in execution latency. Yet in the proposed system, this source of variation can be minimized through *fixing task granularity* during loop paralleliza-

tion. The execution time of these tasks therefore becomes largely identical, while the number of tasks is input dependent. Yet the schedule is still highly predictable at compile time, as such highly similar tasks can be captured in repetitive blocks in the schedule.

Utilizing the aforementioned techniques, the variations in task execution time can be strictly controlled. During static scheduling, we can therefore assume a worst-case execution time (WCET) [90] for each task to attain predictability, without resulting in sizable amount of unnecessary performance degradation.

3.3 Possible Scenarios for Adapting the Computation

The aforementioned resource management scheme effectively delivers fine-grained and predictable execution adaptivity in future multicore systems. The compiler-directed reconfiguration process can be invoked for various purposes, such as fault tolerance, thermal management, and workload balance purposes. To concretely illustrate the envisioned adaptive system, we herein present a number of possible scenarios wherein the system has neither idle nor spare cores, yet the proposed scheme can still deliver superior performance and predictability in the reconfiguration process. To perform worst case analysis, we only assume minimum reconfiguration capability of each schedule, that is, the schedule generated for each application is only capable of tolerating single resource variation.

Tolerating single core degradation: This is the most straightforward case, as the schedule of each application is capable of tolerating single core degradation. With no need for the OS to make any decision, only the application to which the unavailable core belongs needs to go through a reconfiguration process. As an example, a comparison between Figures 3.2a and 3.2b shows that P_2 that used to belong to Application **A** has been isolated with no influence on the resource footprints of Applications **B** and **C**. Clearly, if such a degradation is not permanent, the core can be re-utilized once the cause of unavailability (e.g. a thermal stress) is cleared.

Tolerating multiple core degradation: If each unavailable core belongs to distinct applications, it can be handled individually using the aforementioned scheme. If, on the other hand, more than one core of an application become unavailable, the OS needs to ne-

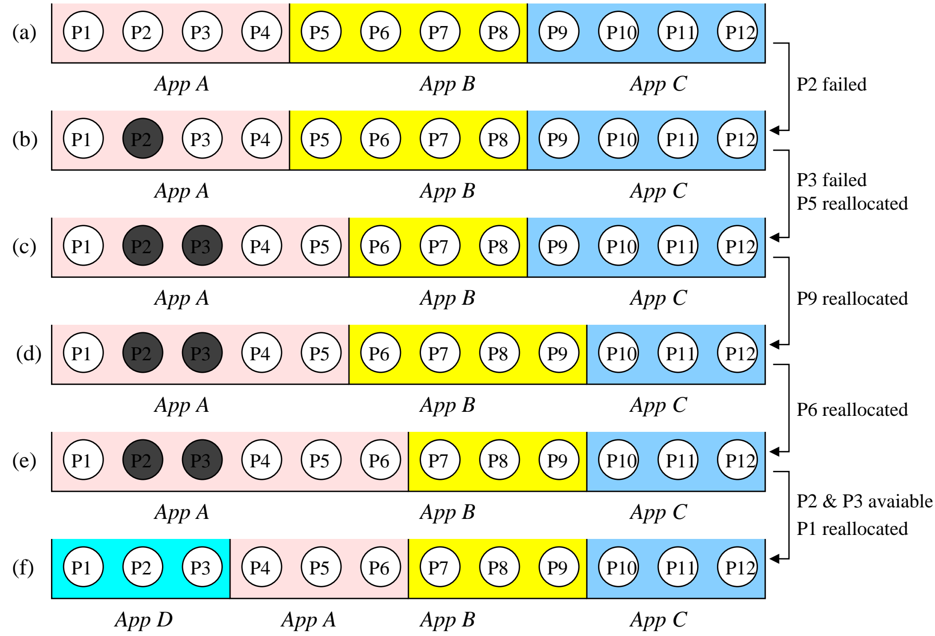


Figure 3.2: Various scenarios for utilizing adaptive execution

gotiate resources among applications that are physically adjacent. A concrete example is shown in Figure 3.2c, wherein P_2 and P_3 that used to belong to Application **A** have been isolated. Assuming that the pre-optimized schedule of Application **A** can only tolerate one core degradation, the OS needs to reallocate one more core to **A**. Therefore, not only **A** but also one of its neighbors, namely, Application **B**, needs to go through a reconfiguration process. More generally, if each application is capable of tolerating a reduction of up to m cores, the toleration of a reduction of n cores requires $\lceil n/m \rceil$ applications to go through a reconfiguration process.

Globally balancing workload: Once the system has encountered unpredictable resource reductions, the application(s) with reduced resources may end up dominating the execution time of the entire system. To re-balance the workload, the OS can periodically invoke reconfiguration processes, so that a limited number of cores can be shared among multiple adjacent applications using a time-multiplexing strategy. This scenario is concretely shown in Figures 3.2c, 3.2d, and 3.2e, wherein P_9 is shared between Applications **B** and **C**, while P_6 is shared between Applications **A** and **B**. It needs to be noted that unlike traditional context switches, the overhead of this time-multiplexing is very limited, as sharing is performed in a coarse-grained manner with only a highly limited number of

cores involved.

Accommodating more application(s): When one more application is invoked, the number of free cores in the system may fall short. Traditional resource management techniques would require cores to be time-multiplexed among multiple applications, engendering in turn significant context switch overhead. In contrast, in the proposed adaptive system, the OS can force each of the existing applications to go through a single reconfiguration process and give up one core, until there are sufficient resources for the new application to be executed. This scenario is concretely presented in Figure 3.2f. A comparison between Figures 3.2a and 3.2f shows that each of the three existing applications yields one core, thus delivering a total number of 3 cores to Application **D**. Clearly, this solution delivers superior performance as compared to the traditional time-multiplexing solution, as the distinct applications perform reconfiguration *simultaneously*.

Prevent potential thermal stress: One standard approach for the OS to mitigate thermal stress is to force the corresponding application to perform a reconfiguration process in order to move the "hot" tasks scheduled on an over-heated core to other cooler cores. However, because the heat of the hot core will be dissipated to its neighbors, in an extreme case all the cores that are currently allocated to that specific application may also suffer from a potentially high temperature. In this situation, the OS could benefit from the allocation of an extra core to that application, with "hot" tasks thus being distributed into more cores so as to increase the average idle time of each core, thus reducing the rate of heat accumulation.

Chapter 4

Core-level Reconfiguration

After introducing the adaptive multicore platform and optimization framework, we proceed to look into the various techniques needed in an adaptive multicore platform. In the face of the highly unpredictable yet frequent occurrence of runtime resource variations, the fundamental requirement in an adaptive system is to rapidly make intelligent execution reconfiguration decisions, with highly predictable impact on each individual application. As mentioned before, this can be attained through a *compiler-directed orderly reconfiguration* technique.

The main advantage of the proposed technique is its ability to compactly capture in readiness a set of possible execution schedules that match diverse resource availability constraints. Such schedules can be adaptively applied upon run-time resource variations, thus delivering regular and predictable reconfiguration responses without any rescheduling decisions being made on the fly. More importantly, the *regularity* inherent in the proposed reconfiguration process ensures that most of the inter-task dependences can be naturally preserved. For the remaining dependences, the compiler can also preserve them at almost no cost, through the exploitation of the *flexibility* in scheduling tasks on non-critical paths. Further schedule length reduction can be attained through a *core reordering* technique that exploits, during the scheduling process, the extra degrees of freedom in assigning tasks to cores. As traditional schedules are generated with no regard to specific core positions, they offer the opportunity of manipulating core positions during the scheduling process, which we exploit to maximally diminish the adaptivity-induced performance impact while retaining all the concomitant benefits.

In this chapter, we first illustrate the proposed approach for partitioning a schedule into regular yet shiftable *bands* to tolerate single core deallocations. Subsequently, we present a core reordering technique for mitigating the impact of adaptivity on schedule length, as well as various techniques for improving the performance of the proposed scheduling approach when it is applied to arbitrary task graphs. In Section 4.4, the band partition technique is extended and generalized to develop graded reconfigurability for the toleration of multiple resource variations.

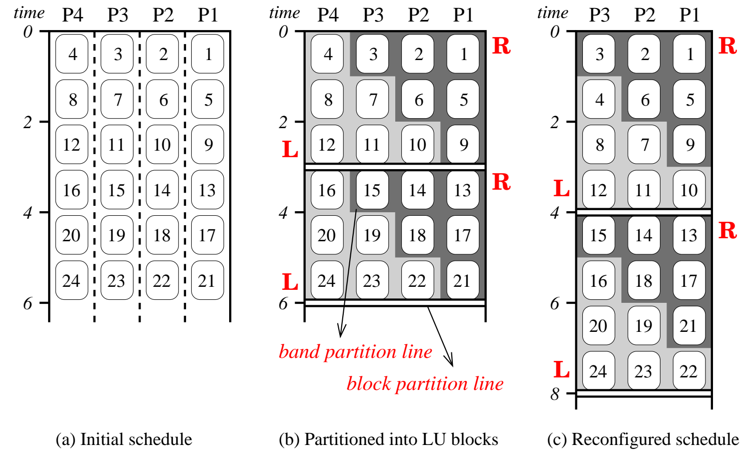
4.1 Adaptive Static Schedules

The success of the proposed adaptive execution framework hinges on the generation of adaptive static schedules, capable of tolerating core degradations. In this section we illustrate the proposed scheduling ideas through the examination of a *canonical schedule example*, wherein a parallel loop is decomposed into multiple tasks with largely identical execution time and inter-task communication latency. To simplify our analysis, we herein focus on the toleration of *single core deallocations*, in the context of a multicore system with homogeneous processing elements (PEs). Yet it needs to be noted that by individually applying the technique to various classes of processing units, the proposed technique can be easily extended to heterogeneous MPSoCs.

4.1.1 Band Partitions of Execution Schedules

To incorporate dynamic reconfigurability directly into MPSoC schedules, we propose to statically partition a schedule into a set of regular yet shiftable *bands*. Essentially, a schedule with n cores and m timing steps can be viewed as a $n \times m$ rectangle. Upon a resource reduction, *schedule reconfiguration* consists of cutting the initial schedule into multiple pieces, and reorganizing these pieces to form a $(n-1) \times (m+1)$ rectangle. The challenge is to cut the initial schedule in an intelligent manner such that the shapes of the pieces enable the development of a rapid reconfiguration process. This is attained by adopting a ***Band & Block (BB) partition*** approach.

To concretely illustrate the proposed band-partition approach, a canonical schedule example is presented in Figure 4.1, wherein 24 tasks of identical execution time are



(a) Initial schedule (b) Partitioned into LU blocks (c) Reconfigured schedule

Figure 4.1: Reconfigurable static schedules: band structure

statically scheduled onto a multicore system consisting of 4 *processing elements* (PEs). Figure 4.1a presents the initial scheduling results, with each rectangle labeled with a number denoting a task and each column representing one PE. The band-level partition of the initial schedule is presented in Figure 4.1b. As can be seen, the whole schedule is *horizontally* divided into two ***Basic Reconfiguration (BR) blocks***, with each BR block furthermore partitioned into two bands, a ***Left (L) band*** and a ***Right (R) band***. To form these partitions, two distinct types of lines are conceptually imposed on the original schedule:

Block partition line: the straight horizontal line between two sequential BR blocks.

Band partition line: the staircase line between the L and the R band within the same BR block.

The outlined shape of the L and the R band enables a highly regular task reassignment capability upon a dynamic resource variation. By comparing Figures 4.1b and 4.1c, it can be clearly observed that in both the *pre-* and the *post-*reconfiguration schedules, *BR blocks*, the minimal reconfiguration units, are executed **sequentially** in the same order. On the other hand, in each *BR block* the whole **L** band is shifted in a regular manner relative to the **R** band, that is, one timing step down and one PE to the right, as shown in Figure 4.2a. This allows all the tasks within each *BR block* to be completed with one less PE, albeit with an additional timing step after reconfiguration. More crucially, this reassignment process displays high regularity, achievable **independent** of the PE being removed. In comparison to Figure 4.2a, Figures 4.2b, 4.2c, and 4.2d respectively show the

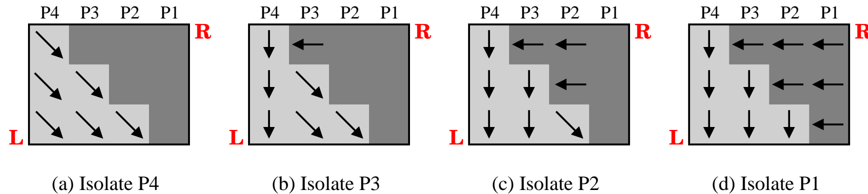


Figure 4.2: Regularity in task reassignment

required task shifting directions for tolerating the deallocation of P_3, P_2, and P_1. It can be seen that all the reassignment processes in Figure 4.2 are highly regular in that not only does the entire band of tasks share an identical timing offset (*temporal* reassignment), but also task transfers are only performed between adjacent PEs (*spatial* reassignment). Such regularity thus enables a predictable reconfiguration process to be attained without any rescheduling decisions being made on the fly, thus drastically reducing reconfiguration overhead.

Another important benefit of the proposed adaptive schedules is that both the *pre-* and the *post-*reconfiguration schedules are able to make **full utilization** of the available hardware resource. This benefit is directly derived from the size of each BR block. Each BR block in Figure 4.1 contains $4 * (4 - 1) = 12$ tasks, thus enabling it to be executed either by 4 PEs in 3 timing steps (as in Figure 4.1b), or by 3 PEs in 4 timing steps (as in Figure 4.1c). More generally, the attainment of full resource utilization requires the following block size constraint to be imposed:

> ***Block Size constraint:*** *A full utilization of PEs in both the pre- and the post-reconfiguration schedules requires each* ***BR block*** *to contain $n * (n - 1)$ tasks in order to tolerate a deallocation of one out of n PEs.*

4.1.2 Inter-task Dependence Variations

One fundamental requirement for any reconfiguration technique is to preserve the partial ordering imposed by inter-task data and control dependences. A significant benefit of the proposed orderly reconfiguration scheme is that the regularity inherent in band partitions enables most dependences to be naturally preserved. Specifically, as a result of the proposed block and band partitions, the dependences among tasks can be naturally

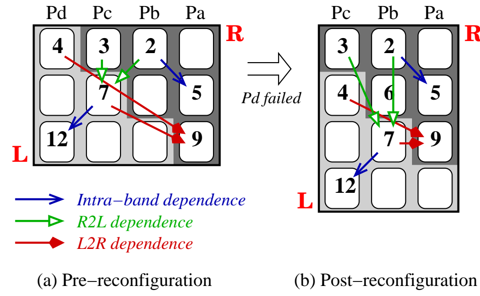


Figure 4.3: Timing variations of inter-task dependences

classified into four categories: *inter-block dependences, intra-band dependences, R2L dependences* and *L2R dependences*. Among these four categories, the first three can be naturally preserved. In this section, we will first illustrate this natural preservation property[1], and subsequently outline a scheduling constraint for the preservation of *L2R dependences*.

As BR blocks are executed sequentially in the same order both before and after reconfiguration, the timing slack of an *inter-block dependence* will never decrease. Meanwhile, dependent tasks lying within the same band retain their relative timing and spatial positions during reconfiguration, implying that *intra-band dependences* can also be naturally preserved. This property can be observed from the timing positions of tasks (2, 5) and (7, 12) in Figure 4.3b.

As the L bands are shifted downwards relative to the R bands, the timing slack of an *R2L dependence* will always **increase** after reconfiguration, while the timing slack of an *L2R dependence* will always **decrease**. Accordingly, *R2L dependences* can also be naturally preserved, as confirmed by the timing positions of tasks (2, 7) and (3, 7) in Figure 4.3b. On the other hand, for *L2R dependences*, a violation **may** occur if the original timing slack is insufficient. As an illustrative example, in Figure 4.3a task 9 is scheduled to be executed after the predecessor task 7, yet in Figure 4.3b these two tasks are scheduled at exactly the same timing slot, thus incurring a reconfiguration-induced semantic violation.

The analysis above clearly confirms that the regularity associated with the recon-

[1] To clearly illustrate the impact of reconfiguration, we decouple its *timing* and *spatial* impacts in our analysis. Here we analyze its *timing* impact, while the communication overhead is assumed to be negligible. The *spatial* impact of reconfiguration on communications will be analyzed in Section 4.3.2.

figuration process enables most inter-task dependences to be preserved naturally. As a result, during the scheduling process, the compiler only needs to pay attention to the *L2R dependences*. A potential dependence violation would only take place if two dependent tasks not only straddle the left to right direction divide between bands, but also are scheduled with no extra timing slack in between. This easily recognizable pattern can be avoided by the compiler through imposing a certain amount of timing slack between two tasks that form an *L2R dependence*, such as tasks 4 and 9 in Figure 4.3a. This property can be formalized as the following *spatial-temporal (S-T) constraint*:

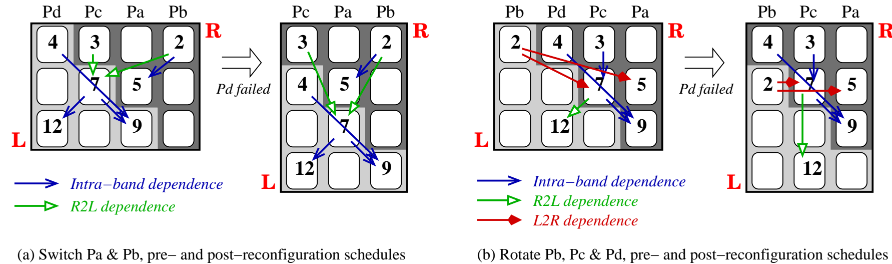
> ***S-T constraint:*** *to preserve the correct execution order after reconfiguration, two tasks with an L2R dependence need to have an intervening slack of a single timing step in the initial schedule.*

4.2 Performance-oriented Core Reordering

As the aforementioned S-T constraint may increase the timing slack between two dependent tasks forming L2R dependences, it may end up increasing the overall schedule length of the entire schedule. However, such performance impact can be maximally diminished by the compiler, through the exploitation of the following **two** types of extra flexibility inherent in task scheduling:

- As the tasks of a given application typically display varying amounts of criticality, tasks on non-critical paths can be scheduled, if necessary, to straddle the left-to-right direction partition while minimizing the performance impact on the overall schedule length.
- In traditional task scheduling, *the logical PE positions* can be adjusted with no impact on the scheduling decisions. This degree of freedom is quite useful to the technique we propose, since adjustments of PE positions can induce desired band relationships, thus varying the direction of a critical *L2R dependence* and hence eliminating the potential performance impact.

The first type of flexibility, provided by the application, is easily observable and utilizable by the compiler through the manipulation of scheduling priorities. Specifically,



(a) Switch Pa & Pb, pre– and post–reconfiguration schedules

(b) Rotate Pb, Pc & Pd, pre– and post–reconfiguration schedules

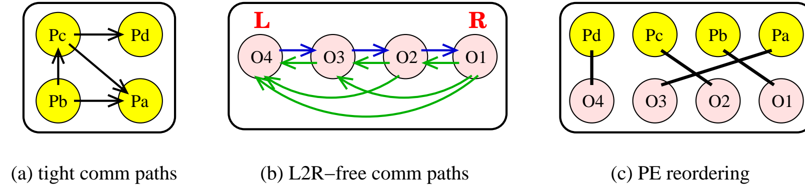
Figure 4.4: Impact of PE reordering on dependence directions

the proposed framework schedules tasks one by one in a priority-ranked manner, with higher priorities assigned to the more critical tasks which can in turn be scheduled earlier. In other words, the more critical tasks are automatically offered more scheduling flexibility, thus exhibiting a larger chance to be assigned to PEs that create no *L2R dependence*.

The second type of flexibility is derived from the fact that in traditional task scheduling, the adjustment of the logical PE positions has no impact on either the execution latency or the communication latency[2] of a task. In other words, the logical PE positions do not affect the scheduling decisions, and hence can be arbitrarily determined. In contrast, in the proposed adaptive schedule, the logical positions of PEs directly determine the band positions, which in turn induce the potential timing differences between various categories of inter-task dependences. Accordingly, by logically reordering the PEs, the compiler can effectively vary the direction of inter-task dependences. This potential benefit of *logical PE reordering* can be observed in Figure 4.4. The schedule in Figure 4.4a is generated through switching the positions of P_a and P_b in Figure 4.3a. A comparison between the two schedules indicates that during the PE reordering process the timing or the core assignments of each task are retained intact. Yet in Figure 4.4a the (4→9) and the (7→9) dependences are no longer *L2R* dependences, and hence no semantic violations would be incurred after reconfiguration.

To exploit the flexibility in manipulating logical PE positions, we propose herein a *PE reordering* technique, to be integrated into the scheduling flow. Initially, the logical PE positions are determined arbitrarily. During scheduling, if the starting time of the task currently under scheduling is constrained by a critical L2R dependence, the compiler

[2]For some systems, the affinity consideration results in the communication latency being a function of the physical PE distance. In such systems, the proposed communication graph representation can be extended to incorporate affinity constraints and capture only the extra amount of reordering flexibility.



(a) tight comm paths (b) L2R–free comm paths (c) PE reordering

Figure 4.5: PE reordering formulated as graph embedding

invokes the *reordering procedure* to quickly check whether this L2R dependence can be eliminated through adjusting the current logical PE positions.

4.2.1 PE Reordering: Problem Formulation

The most crucial challenge in developing an effective PE reordering scheme is to eliminate an L2R dependence while preserving the timing behavior of the remaining dependences. As the reordering effect is *globally* imposed on all the inter-PE task dependences, an inappropriate reordering may engender new *L2R dependences* in the post-reordering schedule. As an example, the schedule in Figure 4.4 is generated through rotating the positions of P_b, P_c and P_d in Figure 4.3a. While the (4→6) and the (7→9) dependences no longer constitute L2R dependences, the (2→5) and the (2→7) dependences emerge as new L2R dependences as a result of the reordering effect. To preclude the creation of *post*-reordering L2R dependences, the reordering process should be performed in a *global* manner with all the inter-PE communications being considered. Specifically, all the inter-PE communications that have no extra timing slack, denoted as *tight communications*, should be precluded from being mapped as L2R dependences.

The aforementioned problem of mapping all the *tight communications* to *L2R-free communications* can be formulated as embedding the *tight communication* graph G into the *L2R-free communication* graph F. The *tight communication* graph $G = (P, E_c)$ is a directed graph that captures all *tight communication paths*. Each node $p_i \in P$ represents a PE, while an edge $(p_i, p_j) \in E_c$ indicates the existence of at least one tight communication from node p_i to p_j. As an example, the graph representation of the inter-PE communications in Figure 4.3 is presented in Figure 4.5a. The graph contains four edges, as the (2→5), (2→7), (7→9), and (7→12) communications in Figure 4.3 exhibit no extra slack. In comparison, the *L2R-free communication* graph $F = (O, E_v)$ captures all

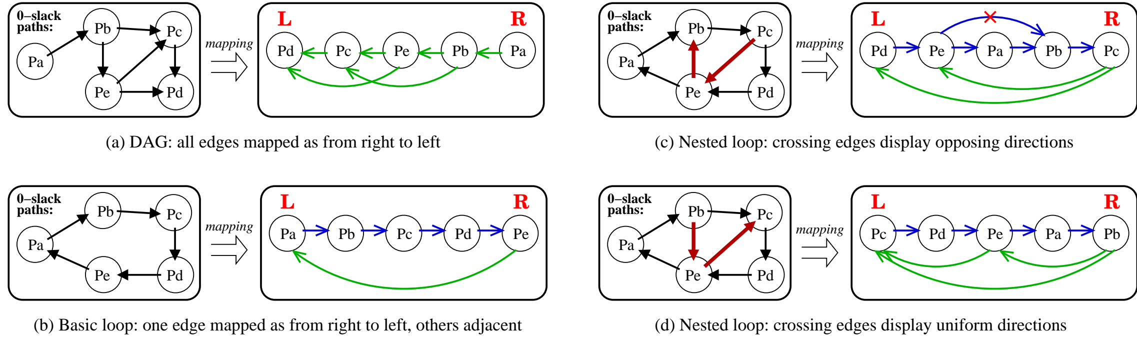
the possible *L2R-free* communication paths in an adaptive schedule. It can be observed in Figure 4.3 that in an *L2R* dependence, the source and the sink tasks not only straddle a left to right divide, but furthermore are scheduled on non-adjacent PEs. Accordingly, an *L2R-free communication* graph contains all the right-to-left edges, as well as all the left-to-right edges between adjacent nodes. As an example, Figure 4.5b shows all the L2R-free communication paths in a 4-PE adaptive schedule.

Utilizing these two directed graphs, the problem of embedding the *tight communication* graph G into the *L2R-free communication* graph F can be defined as follows:

> ***Directed graph embedding:*** *given two directed graphs* $G = (P, E_c)$ *and* $F = (O, E_v)$*,* G *can be embedded into* F ***iff*** *a node mapping* $(M : P \rightarrow O)$ *can be found such that* $\forall (p_i, p_j) \in E_c$*,* $(M(p_i), M(p_j)) \in E_v$ *holds.*

The amount of flexibility in PE reordering is strongly determined by the number of edges in the *tight communication* graph G. As each edge in G imposes an order requirement, the more edges the graph has, the less flexibility is left for the reordering process. To increase reordering flexibility, the graph G is considered when making the scheduling decisions. In the proposed scheduling framework, the compiler would reassign a task to a PE that can reuse an existing communication path (unless its starting time suffers a consequent delay), thus effectively reducing the number of edges in G.

The difficulty of the outlined graph embedding problem is determined by the characteristic of the target graph F. Embedding an arbitrary directed graph into a complete directed graph is straightforward because of the latter's full connectivity. On the other hand, if F is an arbitrary directed graph, the embedding problem is a hard problem for which a *branch-and-bound* approach is typically employed to search for a valid solution. However, regarding the proposed PE reordering issue, the target graph F, as shown in Figure 4.5b, contains all the right-to-left edges as well as the left-to-right edges between adjacent nodes. Such regularity allows us to develop a set of embedding criteria which, based on the connectivity characteristics of the tight communication graph G, directly determine whether a valid L2R-free mapping exists or not. In the following parts of this section, we outline a set of mapping criteria for various types of graphs, including *DAGs*, *basic loops*, *nested loops* and *intersecting loops*.



(a) DAG: all edges mapped as from right to left

(b) Basic loop: one edge mapped as from right to left, others adjacent

(c) Nested loop: crossing edges display opposing directions

(d) Nested loop: crossing edges display uniform directions

Figure 4.6: L2R-free mapping of DAG, basic loop and nested loops

4.2.2 PE Reordering: L2R-free Mapping Identification

Problem decomposition

The fundamental observation is that an *L2R-free* mapping can always be established for a tight communication graph (G), if G is a *directed acyclic graph* (DAG). This is because the outlined *L2R-free communication* graph (F) contains all the right-to-left edges, and the acyclic property of G enables all its communication edges to be naturally mapped from right to left, as shown in the schedule example presented in Figure 4.6a.

In contrast, if G is a cyclic directed graph, it can always be decomposed into a set of disjoint strongly connected components[3] (SCCs), with each SCC containing one or multiple loops. Moreover, these SCCs are connected in an acyclic form, implying that the tight communication paths corresponding to the *inter-SCC* edges can always be mapped from right to left. Accordingly, the following strategy can be employed to decompose the mapping problem:

> ***Problem decomposition:*** *An* L2R-free *mapping can be established for a cyclic directed graph (G),* ***iff*** *for each SCC of G an* L2R-free *mapping can be established.*

This decomposition policy indicates that each SCC of G can be considered *independently*. Mapping criteria thus only need to be developed for *strongly connected* directed graphs, which can further be classified as either a *basic loop*, or *nested loops*, or *intersecting loops*.

[3] An SCC consists of a maximal set of nodes such that for every pair of nodes p_i and p_j, there exists a path (p_i, p_j) and a path (p_j, p_i).

Mapping of basic loops

A *basic loop* that exhibits a single back edge and no crossing edge turns out to be the most straightforward mapping case. An L2R-free mapping can always be established, by placing one edge of the loop (e.g., (p_e, p_a) in Figure 4.6b) from right to left, and the remaining edges adjacently from left to right.

Mapping of nested loops

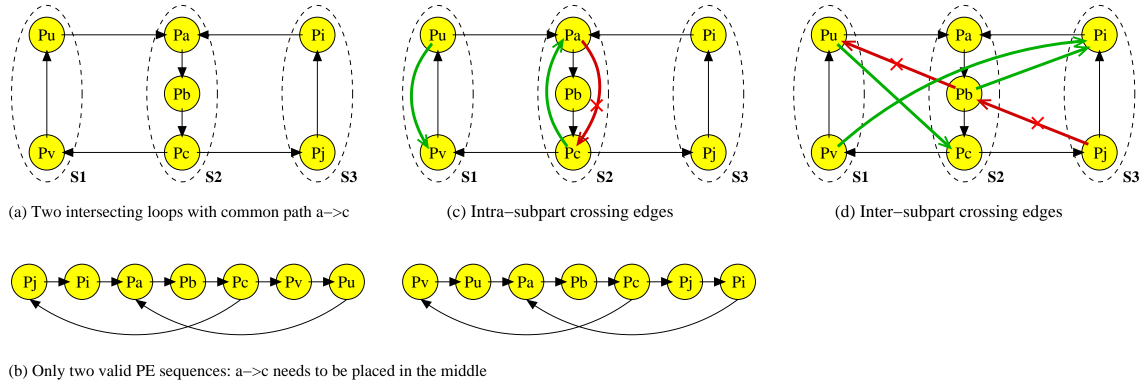
The mapping policy of *basic loops* displays a certain degree of flexibility; given an L2R-free mapping (e.g., the one shown in Figure 4.6b), a set of isomorphic mappings can be constructed through a clockwise rotation of all the nodes. This flexibility can be utilized to construct an L2R-free mapping for nested loops with a *single* crossing edge. The head PE of the crossing edge can be rotated to the rightmost position in the PE sequence, thus allowing the crossing edge to be mapped from right to left.

If an SCC happens to be a nested loop with multiple crossing edges, whether an L2R-free mapping can be established or not depends on the directions of the crossing edges. A nested loop is *unmappable* if it includes multiple crossing edges of *opposing* directions. As an example, in Figure 4.6c, the two crossing edges (p_c, p_e) and (p_e, p_b) exhibit *opposing* directions such that the shared node p_e cannot be placed in between p_c and p_b. If the first edge is mapped from *right* to *left*, the second edge has to be mapped from *left* to *right* and vice versa. In contrast, in Figure 4.6d, the two crossing edges (p_e, p_c) and (p_b, p_e) can be simultaneously mapped from right to left, as they exhibit a *congruent* direction such that the two head nodes p_b and p_e can be placed to the right of the corresponding tail PEs p_e and p_c simultaneously.

A detailed examination indicates that the question of whether two crossing edges exhibit opposing directions can be settled through a *depth-first search* of the tight communication graph G. Two crossing edges (p_i, p_j) and (p_u, p_v) exhibit opposing directions if both of the following conditions hold:

- All the backward paths[4] of edge (p_i, p_j) (i.e., the path from p_j to p_i) need to go through both nodes p_u and p_v.

[4] As the graph is an SCC, there exists at least one path from p_j to p_i.



(a) Two intersecting loops with common path a–>c

(c) Intra–subpart crossing edges

(d) Inter–subpart crossing edges

(b) Only two valid PE sequences: a–>c needs to be placed in the middle

Figure 4.7: Mapping constraints of intersecting loops

- All the backward paths of edge (p_u, p_v) need to go through p_i and p_j.

Mapping of intersecting loops

The most complicated mapping case is the situation when an SCC is composed of multiple intersecting loops. An illustrative example is presented in Figure 4.7a, wherein two loops, $(p_u, p_a, p_b, p_c, p_v)$ and $(p_i, p_a, p_b, p_c, p_j)$, share the nodes p_a, p_b, and p_c in common.

In order for an L2R-free mapping to be possible, each loop individually should refrain from containing crossing edges of opposing directions. Moreover, all the nodes involved in a single loop need to be placed in contiguous positions so as to satisfy the mapping rule of basic loops. This requirement, in conjunction with the intersection property, implies that the shared nodes (p_a, p_b, and p_c in Figure 4.7a) need to be placed in contiguous positions that separate the disjoint parts of the two loops. As a result, there exist only two possible node sequences corresponding to an L2R-free mapping of these two intersecting loops, shown in Figure 4.7b.

The limited number of valid node sequences strongly constrains the possible directions of crossing edges. To illustrate these constraints, we decompose the graph in Figure 4.7a into three disjoint subparts, S_1, S_2 (shared between the two loops), and S_3. The crossing edges, as a result, can be classified as either *intra-subpart* or *inter-subpart* edges. In order for an L2R-free mapping to be possible, all the *intra-subpart* crossing edges should display directions *in reverse* to the loop edges of that subpart. Since the

loop edges are mapped as left-to-right edges between adjacent nodes, this requirement ensures that these crossing edges can be mapped from right to left, such as the edges (p_v, p_u) and (p_c, p_a) in Figure 4.7c. As a counterexample, the edge (p_a, p_c) cannot be mapped as a right-to-left edge, since its direction is consistent with the loop edges.

As for the *inter-subpart* crossing edges, an L2R-free mapping requires them to exhibit *congruent* directions. For instance, given a crossing edge from subpart S_1 to S_2 (e.g., (p_u, p_c) in Figure 4.7d), only the first sequence shown in Figure 4.7b is valid. As a result, crossing edges with ascending indices (i.e., from S_i to S_j, $i < j$) can be mapped as right-to-left edges, such as the edges (p_v, p_i) and (p_b, p_i) in Figure 4.7d. In contrast, crossing edges in the other direction (i.e., from S_i to S_j, $i > j$) cannot be mapped from right to left.

In sum, an *L2R-free* mapping cannot be established for intersecting loops if any of the aforementioned requirements of *node sharing*, *intra-subpart* and *inter-subpart* crossing edges is violated. The reordering procedure thus checks the compatibility between these requirements to detect an unmappable case.

4.3 Performance Enhancement

The last two sections have discussed the proposed orderly reconfiguration schedule and the corresponding PE reordering for the *canonical case*, wherein a program is composed of multiple tasks with largely identical execution time and inter-task communication latency. While at first glance the canonical case would seem to be highly idealized, it actually turns out to be a representative model for the parallel sections of data-intensive applications. As a large fraction of these applications consists of regular data processing loops with limited or possibly even no loop-carried dependences whatsoever [70], they can be easily parallelized into a set of tasks with high regularity, as assumed in the canonical case.

To further enhance the applicability of the proposed BB reconfiguration scheme to various systems with diverse application sets, in this section we examine several issues encountered when it is applied to arbitrary programs with diverse task execution time and non-zero communication latency.

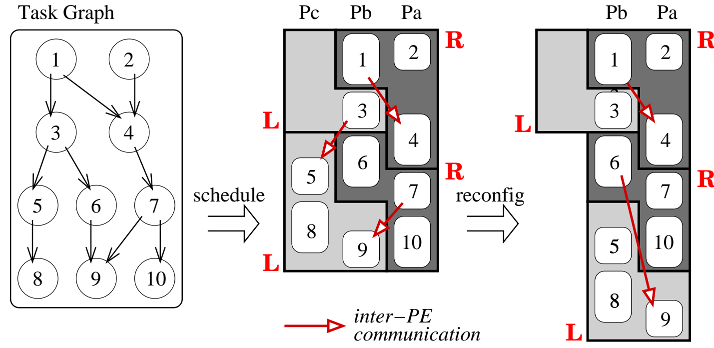


Figure 4.8: An adaptive schedule for an arbitrary task graph

4.3.1 Applied to Arbitrary Task Graphs

The application of the orderly BB reconfiguration scheme to an arbitrary task graph exposes several new characteristics that are not observed in the canonical case, outlined as follows:

Idle PE cycles in the initial schedule

As inter-task dependences and communication overhead may strictly constrain the earliest starting time of a task, the initial schedule may display significant under-utilization in certain portions. This can be observed in Figure 4.8, in which the PE P_c is left idle across the entire timing period of the first *BR block*.

The under-utilized portions of the initial schedule, as they have at least one idle PE, require no reconfiguration and are directly applicable in the case of single PE degradations. The exploitation of this property allows a sizable reduction in the length of the post-reconfiguration schedule. Specifically, the original *BR block* can be extended to contain a **head** and/or a **tail** region, as shown in Figure 4.9. During reconfiguration, both the head and the tail regions remain intact, while only the bands in the body regions need to be shifted. The under-utilized portions of the initial schedule thus can be mapped into the **head** or **tail** regions wherein the cost of reconfiguration is absolutely zero, while only the fully parallel portions need to be mapped into the **body** regions.

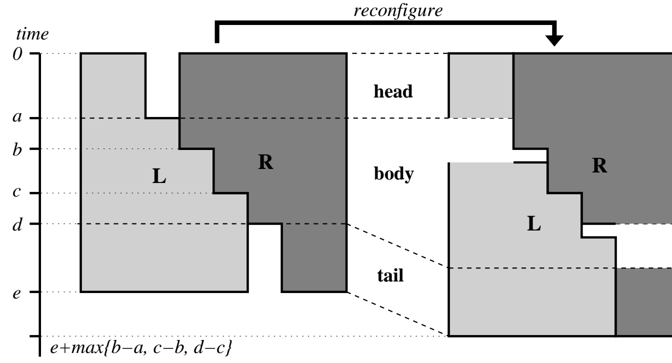


Figure 4.9: Band structure extension: the head and tail regions

Irregularity of partition lines

As the tasks in an arbitrary task graph typically exhibit diverse execution time, they create irregularity in partition lines. As can be observed from Figure 4.8, the heights of the steps on a *band partition line* are not necessarily identical, and the original horizontal *block partition lines* are not necessarily straight.

The aforementioned two types of irregularity may degrade performance by creating extra timing holes in the post-reconfiguration schedule. A detailed examination indicates that for each *BR block*, the reconfiguration penalty (in terms of schedule length increase) is constrained by the **maximum height** of the steps on the *band partition line*. This relationship is shown in the body region of the schedule in Figure 4.9. Assume the body region is originally scheduled to commence at time a and end at time d, and the positions of the intermediate two steps on the band partition line are at time b and time c, respectively. After reconfiguration, the length of the body region increases to

$$\begin{aligned} &\max\{(b-a)+(d-a),(c-a)+(d-b),(d-a)+(d-c)\} \\ &\qquad = (d-a)+\max\{b-a,c-b,d-c\} \end{aligned} \tag{4.1}$$

Because of this relationship, in the process of generating the adaptive schedule, the step height on the band partition lines should be maximally balanced. This can be attained through an adjustment of the task starting times. Moreover, to perform this optimization without increasing the length of the initial schedule, the compiler can exploit the flexibility inherent in an adaptive schedule, namely, the timing slacks of the tasks on the *non-critical*

paths. Such timing slacks can be easily calculated through the identification of the task starting time in both the *as-soon-as-possible* (ASAP) and the *as-late-as-possible* (ALAP) schedules.

4.3.2 Overcoming Variations in Inter-core Communication

Typically, a static scheduler prefers to assign dependent tasks on the same PE to hide communication latency. Yet one crucial impact of reconfiguration, as a result of band movement, is that two dependent tasks originally scheduled on the same PE may be separated onto two PEs or vice versa. These reconfiguration-induced variations in inter-core communications can be clearly observed in Figure 4.8, wherein the (3→5) and the (7→9) communications in the initial schedule disappear, and a new inter-PE communication (6→9) is created after reconfiguration.

While the communication disappearance can be exploited for reducing the length of the post-reconfiguration schedule, it is conversely highly crucial to provide mechanisms for compensating for the extra communications created after reconfiguration, especially when an extra communication is created between two *tightly scheduled* tasks. Figure 4.10a presents 8 pairs of tightly scheduled tasks that are separated into distinct bands. Depending on the positions of the source and the sink tasks, a new inter-PE communication may be created in only **two** ways:

- The source task is in an **R** band, while the sink is in the **L** band of the same BR block, such as the task pairs $(3, 7)$, $(6, 10)$, $(15, 19)$ and $(18, 22)$ in Figure 4.10.
- The source task is in an **L** band, while the sink is in the **R** band of the subsequent BR block, such as the task pairs $(10, 14)$ and $(11, 15)$ in Figure 4.10.

A comparison between Figures 4.10a and 4.10b indicates that these two cases display diverse timing characteristics. In the former case, an additional 1-step timing slack is implicitly inserted between the two tasks (e.g., 3 and 7 in Figure 4.10b) after reconfiguration, thus automatically compensating for the latency of the created communication. In contrast, in the latter case the relative timing positions of the L band and the subsequent R band remain intact. The created communication (e.g., 10→14 in Figure 4.10b) therefore may result in insufficient time for the sink task to receive its input. However, as all the

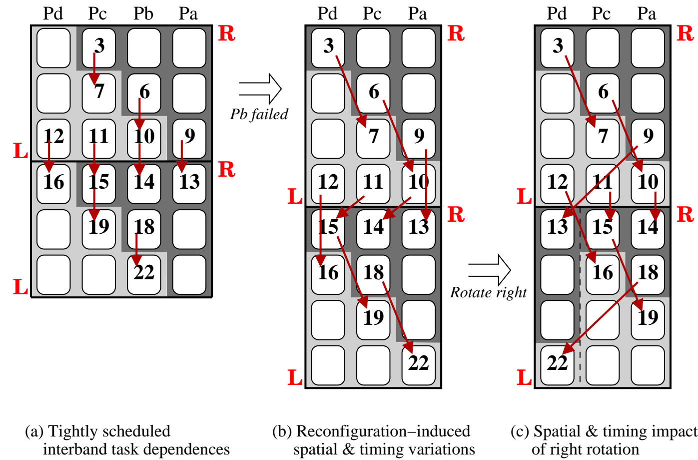


Figure 4.10: Impact of PE rotation on inter-PE communications

communications of this type are created by shifting the source task right relative to the sink task, they all display an identical offset in that the sink tasks (tasks 14 and 15 in Figure 4.10b) are located exactly one PE to the *left* of the corresponding sources (tasks 10 or 11). This observation implies that this class of communications can be simply eliminated, if the entire subsequent BR block can be dynamically rotated one PE to the right in the post-reconfiguration schedule, as shown in Figure 4.10c.

A comparison between Figures 4.10b and 4.10c confirms that each task still retains its band partition, implying that the right rotation process does not impact the band partition. Instead, only the logical-to-physical core binding is varied in a highly regular manner. As this regular transformation of the schedule requires no global program information, it can be performed dynamically during execution. The pre-generated adaptive schedule can be loaded into the OS at the granularity of BR blocks; the schedule of the subsequent BR block can be loaded during the execution of the current BR block. In this process, right rotation can be straightforwardly implemented through globally manipulating the core binding of the subsequent BR block. In a system with distributed memory structure, the code and the data set of the tasks to be executed can be loaded into local memory units in parallel with the schedule of the BR block.

By rotating the entire subsequent BR block one PE to the right, the spatial locality of the tasks $(10, 14)$ and $(11, 15)$ can be naturally preserved. Yet two tasks that are scheduled in either two **L** bands or two **R** bands of consecutive BR blocks (e.g., tasks $(9, 13)$

and $(12, 16)$ in Figure 4.10c) are to be separated, thus creating another class of inter-PE communications in the post-reconfiguration schedule. However, as each BR block takes one extra timing step for execution in the post-reconfiguration schedule, for this class of communications an implicit timing slack will be automatically inserted between the source and the sink tasks, thus compensating for the latency of the created communications.

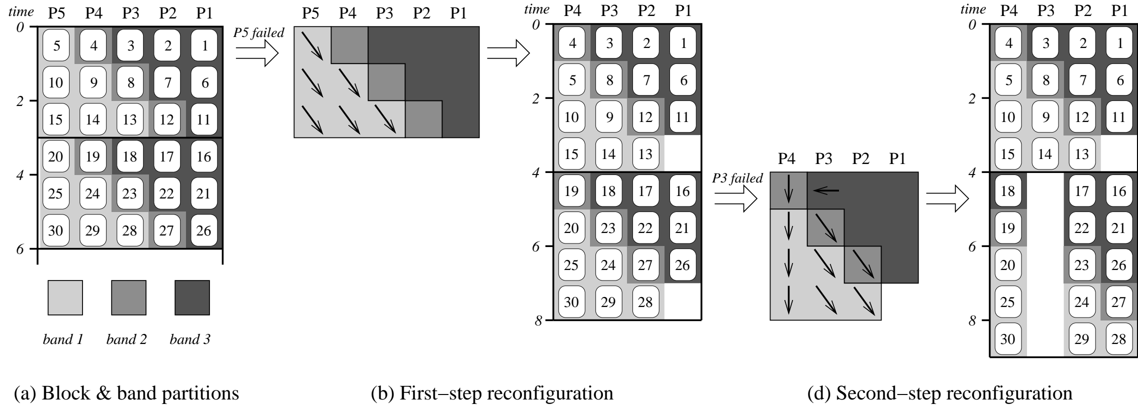
4.4 Tolerating Multiple Resource Variations

By now it should be clear that by partitioning each BR block into two bands, the proposed adaptive schedule is capable of tolerating single core degradations. Yet given the elevated rates of device faults, thermal stress, and resource competitions, numerous cores may become unexpectedly unavailable during execution. An adaptive multicore system therefore needs ***m-adaptive schedules***, i.e., schedules that are capable of withstanding a reduction of m cores or making use of m extra cores.

To generate adaptive schedules capable of tolerating multiple core variations, the *band partitioning* technique needs to be extended so that numerous schedules can be compactly engendered in readiness. More precisely, an m*-adaptive schedule* needs to compactly engender $m + 1$ schedules in readiness so as to deterministically respond to the unpredictable resource variations. The partial execution order imposed by inter-task dependences needs to be preserved in all these $m + 1$ schedules, which in turns imposes stricter constraints to be fulfilled during task scheduling.

4.4.1 Band Partition Extension

As each band individually offers the ability of tolerating the variation of a single core, to tolerate a variation of up to m cores, each *Basic Reconfiguration* (BR) block in an m-adaptive schedule needs to be partitioned into $(m+1)$ bands. A representative partition is shown in Figure 4.11a, wherein each BR block is partitioned into *three* bands to achieve a 2-step adaptivity. As can be seen, the whole schedule is *horizontally* divided into *two* **Basic Reconfiguration (BR) blocks**, each of which is furthermore partitioned into *three* **bands** in order to achieve an adaptivity degree of 2. While the lowest and highest bands



(a) Block & band partitions (b) First−step reconfiguration (d) Second−step reconfiguration

Figure 4.11: Multi-band partitioning for increased amount of adaptivity

should still be in triangle form, each of the middle $(m-1)$ bands should contain tasks in a diagonal row of width **1**. Finally, two sequential reconfigured scheduling results are respectively presented in Figures 4.11b and 4.11c.

The distinct schedules presented in Figure 4.11 clearly confirm that by dividing each BR block into $(m+1)$ bands, a total number of m distinct schedules (in addition to the initial schedule) can be adaptively applied, thus enabling a toleration of up to m core variations during execution. As can be seen in Figure 4.11b, in each *BR block* the whole leftmost band (*Band 1*) is shifted in a regular manner relative to the right two bands, that is, one timing step down and one PE to the right, to accomplish the first reconfiguration step. Similarly, Figure 4.11c shows that if another PE, P_3, becomes unavailable, further reconfiguration would be accomplished through shifting the two left bands (*Bands 1&2*) one timing step down relative to the rightmost band, while transferring tasks on P_3 to either of the adjacent PEs P_2 or P_4. This regular task reassignment capability can be achieved **independent** of the PE being removed.

The band partitioning approach introduced in Section 4.1 attains *full* resource utilization in both schedules through forcing each BR block to contain $n*(n-1)$ tasks that can be executed either by n PEs in $(n-1)$ timing steps or by $(n-1)$ PEs in n timing steps. Similarly, in a generalized m-core adaptive schedule, each BR block needs to contain $n*(n-m)$ tasks so as to attain *maximum* resource utilization. In this way, the schedules with n and $(n-m)$ cores can attain *full* resource utilization, while the other intermediate schedules would exhibit a small number of "individual PE stalls" in each

BR block. As an example, the two BR blocks in Figures 4.11a and the second BR block in Figure 4.11c deliver full utilization of the available cores, yet the schedule in Figure 4.11b exhibits a single stall on P_1 in each BR block. In sum, the generalized block size requirement and the number of stalls in the various schedules are formally specified as follows:

- ***Block size:*** *A maximum utilization of PEs in all the* $m+1$ *distinct schedules requires each BR block to contain* $n * (n - m)$ *tasks.*
- ***Resource usage:*** *in the schedule with* $(n - k)$ *cores* $(0 \leq k \leq m)$*, a BR block containing* $n * (n - m)$ *tasks can display up to* $k * (m - k)$ *"individual PE stalls".*

A crucial aspect of the proposed multi-step orderly reconfiguration technique is that although the technique is presented in the context of core degradation toleration, it is capable of conversely incorporating the use of additional cores allocated during execution as well. This "m-core augmentable" property can be easily observed if we assume the 3-core schedule in Figure 4.11c to be the initial one and the schedules in Figures 4.11b and 4.11a to be the two subsequent *post*-reconfiguration schedules. More generally, the proposed m-adaptive schedule can be invoked as long as the MPSoC contains k cores with $(n - m \leq k \leq n)$. In other words, any of the $(m + 1)$ compactly captured schedules can be considered as the initial one, while at runtime the other m schedules can be selectively applied according to the varying resource availability in the target platform.

4.4.2 Inter-task Dependence Constraints

The capability of tolerating a variation of m cores makes the satisfaction of this requirement increasingly challenging, as the ordering semantics need to be preserved in all the $m + 1$ distinct execution schedules that can be spawned. However, the regularity inherent in band partitions significantly simplifies this strict requirement.

In a manner analogous to the single-core degradable schedules, the inter-task dependences in an m-adaptive schedule can be naturally classified into four categories: *inter-block dependences, intra-band dependences, right-to-left (R2L) dependences* and *left-to-right (L2R) dependences*. Among these four types, the first two exhibit exactly an identical property for both 1-core and m-core adaptive schedules. Specifically, as BR

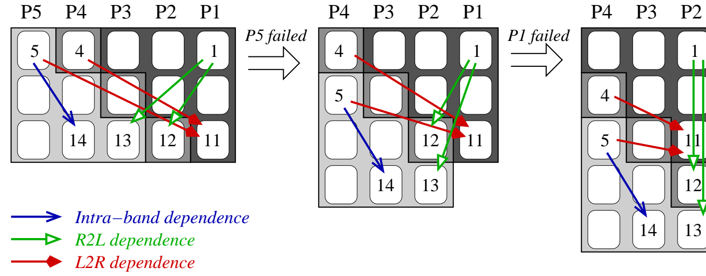


Figure 4.12: Inter-task dependence timing in a multi-band schedule

blocks are executed sequentially in the same order and each band retains its shape in all the $m+1$ distinct schedules, both *inter-block dependences* and *intra-band dependences* (e.g., task pairs (5, 14) in Figure 4.12a) can be naturally preserved.

Unlike a single-core degradable schedule, an m-adaptive schedule exhibits m levels of R2L and L2R dependences, as the schedule is partitioned into $(m+1)$ bands. As bands of smaller indices shift down relative to bands of larger indices, the *R2L* dependences (e.g., task pairs (1, 12) and (1, 14) in Figure 4.12) can always be preserved in all $m+1$ distinct schedules, while an *L2R* dependence (e.g., task pairs (4, 11) and (5, 11) in Figure 4.12) may be violated if the original timing slack is insufficient. However, unlike single-core degradable schedules, in an m-adaptive schedule, diverse L2R dependences exhibit distinct levels of timing slacks, depending on the band positions of the source and the sink tasks. For instance, in Figure 4.12, after the two reconfiguration steps, the timing slack of dependence (5, 11) is reduced by 2, while the slack of dependence (4, 11) is only reduced by 1. To compensate for such a diverse slack reduction and hence preclude potential dependence violations in all the $m+1$ distinct schedules, the extra slack amount of an L2R dependence should be set according to the *band distance* between the source and the sink tasks, formally specified as follows:

- ***Minimum L2R timing slack:*** *An L2R dependence between bands i and $(i+k)$ needs a minimum slack of k timing steps in the pre-reconfiguration schedule in order to preserve the correct execution order in all the post-reconfiguration schedules.*

4.4.3 Core Binding Permutation

The aforementioned minimum L2R timing slack drastically increases as the degradation tolerance capability grows. The more bands a BR block contains, the larger slack an L2R dependence may need, and the larger the amount by which it may end up increasing the entire schedule length of an m-adaptive schedule. To minimize this overhead, the **critical *L2R*** dependences, that is, the ones that delay the corresponding sink tasks, should be obviated as much as possible.

A strength of the proposed diagonal partitioning axis is that permutations of binding decisions have a strong and material impact on the direction of critical inter-task dependences. This property has been successfully exploited through the PE mapping technique shown in Section 4.2 for single core deallocations. Yet the technique can only make binary decisions regarding whether or not the extra timing slack of a critical dependence can be completely eliminated, thus falling short of addressing the needs of m-adaptive schedules wherein such dependences exhibit graded levels of timing slacks. In comparison, in this section we propose a novel *core binding permutation* technique to directly exploit the graded levels of dependence slacks to maximally diminish the schedule length overhead.

A noteworthy aspect is that L2R dependences in an m-adaptive schedule exhibit graded levels of timing slacks. This property is clearly shown in Figure 4.13. The schedule in Figure 4.13a contains two *L2R* dependences between task pairs (4, 11) and (5, 11). Yet Figure 4.13b shows that by shifting P_1 to the left of P_2, the first *L2R* dependence can be completely eliminated, and the *band distance* of the second *L2R* dependence can be reduced. Even if the $(5 \rightarrow 11)$ dependence cannot be eliminated, the required timing slack can still be reduced as long as the *PE position distance* between the corresponding tasks can be reduced. This can be attained through either shifting the PE of the source task (e.g., P_5) *right*, or shifting the PE of the sink task (e.g., P_1) *left*. Yet it needs to be noted that the flexibility of PE shifting is constrained by the remaining inter-PE task dependences; an inappropriate permutation may create an additional L2R dependence in the post-reordering schedule. As shown in Figure 4.13c, if P_1 is further shifted to the left of P_3, the L2R dependence between (5, 11) can be eliminated, yet a new L2R dependence between (5, 13) is created.

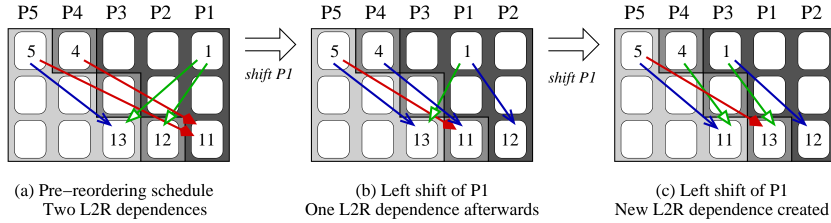


Figure 4.13: PE reordering in a multi-band schedule

4.4.4 Shiftable Core Identification

In this section, we analyze the various conditions for reducing the band distance between two tasks, through shifting the PE of the source task *right* and shifting the PE of the sink task *left*. Essentially, each communication path in an m-adaptive schedule imposes an order requirement for the two PEs involved. According to the aforementioned *L2R timing constraint*, if two dependent tasks display an extra timing slack k in between, the PE corresponding to the source task cannot be placed more than $k + 2$ PEs to the *left* of the sink. This constraint can be used to derive the left shift conditions for the sink PE of a critical L2R dependence, denoted as P_t. Whether P_t can be shifted to a position to the *left* depends on two sets of communication paths, the ones emanating from P_t, and the ones across the target position.

Communication paths emanating from P_t

Each of these paths imposes a constraint on the maximum left-shift distance of P_t. If such a distance is less than the distance between P_t and the target position, shifting P_t to the target position will create an L2R dependence. As an example, in Figure 4.14a, there exists a communication path from P_1 to P_2 with P_2 just to the left of P_1. This path constrains the maximum left shift distance of P_1 to be 2, implying that P_1 cannot be shifted to the left of P_4 or P_5, as confirmed by Figure 4.14c.

Communication paths across the target position

If any of these communication paths displays **no** extra spatial slack, P_t cannot be shifted to the target position. In Figure 4.14a, the communication path ($P_4 \rightarrow P_2$) displays no spatial slack, implying that P_1 cannot be placed into any position between P_2

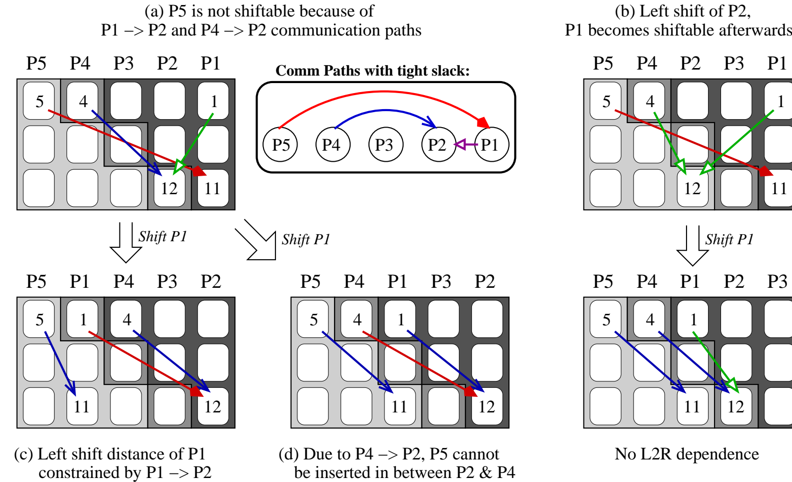


Figure 4.14: PE shiftability constraints and a indirectly shiftable case

and P_4. As shown in Figure 4.14d, shifting P_1 to the left of P_3 forces the dependence between Tasks 4 and 12 to become an L2R dependence with insufficient timing slack.

Indirect Shifting Possibilities

While the two left-shift constraints may preclude a core P_t from being *directly* shifted to a left position, through exploiting the shiftable possibilities of the PEs that impose these constraints, P_t may become *indirectly* shiftable.

The *indirectly* shiftable condition is concretely illustrated in Figure 4.14b. The analysis outlined above confirms that in Figure 4.14a, P_1 is not directly shiftable due to the aforementioned two constraints. Yet both constraints can be relaxed by shifting P_2 one position to the left. On one hand, as P_2 constrains the maximum left-shift distance of P_1, shifting it one position to the left relaxes the maximum left-shift distance of P_1 by 2. On the other hand, the left shift of P_2 also engenders one extra spatial slack in the communication $P_4 \rightarrow P_2$, thus enabling P_1 to be inserted in between. The relaxation of these two conditions thus makes P_1 shiftable, as shown in Figure 4.14b. The post-reordering schedule, generated by shifting P_1 to the left of P_2, exhibits no L2R dependences, thus eliminating any possible reconfiguration-induced timing violations.

Right-shift conditions of source PE P_s

So far we have examined the *direct* and *indirect* conditions for shifting the sink PE of a critical L2R dependence *left*. It turns out the conditions for shifting the source PE of a critical L2R dependence *right* are highly symmetric to the conditions for shifting the sink PE *left*. In sum, the source PE, denoted as P_s, cannot be directly shifted to a position to the *right* if either of the following two conditions holds:

- A communication path *terminating* at P_s imposes a maximum right-shift distance less than the distance between P_s and the target position.
- There exists a communication path across the target position from left to right with no extra spatial slack in between.

4.5 Algorithmic Implementation

Up until now we have presented the conceptual mechanisms underpinning the compiler-directed dynamic reconfiguration scheme. It needs to be noted that the effectiveness of these techniques does hinge on the class of the underlying static scheduling algorithm. In this section we implement the proposed orderly reconfiguration scheme through applying the outlined scheduling constraints and reordering conditions to one of the representative classes of scheduling heuristics, namely, *list scheduling*.

Given a parallel application represented as a weighted *directed acyclic graph* (DAG), the scheduling problem can be formalized as the association of a start time and a core with each node of the DAG. A list scheduling algorithm is typically composed of two phases, namely, a *task prioritization* phase, wherein the scheduling order of each node is determined, and a *processor assignment* phase, wherein each node is assigned to a PE that minimizes its start time. The main difference of the various list scheduling heuristics (e.g., DCP [54], CPND [4], etc) is the determination of the scheduling order. In our implementation, the Dynamic Critical Path (DCP) algorithm [54] is selected as the baseline algorithm.

4.5.1 Initial Schedule Generation

The concrete implementation of the integrated static task scheduling and core reordering procedure is shown in Algorithm 1. To implement *task prioritization*, the algorithm maintains an ordered list of all the ready tasks, from which the task with the highest priority, denoted as V_s, is selected for scheduling. The content of the ready list and furthermore the scheduling priorities of all the ready tasks are updated dynamically upon the scheduling of the current task, shown from Line 11 to 16 in Algorithm 1. Meanwhile, a communication path graph, employed to record the minimum amount of timing slack of all existing inter-PE communication paths, is updated (Line 10 in Algorithm 1) if the scheduling of V_s either creates a new edge in the graph, or reduces the *weight* of an existing edge.

Algorithm 1 Task Scheduling with PE Reordering

1: *Readylist* = $\{$all root tasks$\}$;
2: **while** *Readylist* $\neq \phi$ **do**
3: $V_s = v_i \in$ *Readylist* with highest priority;
4: *Readylist* = *Readylist* $-\{V_s\}$;
5: $\{V_s.PE, V_s.startTime\}$ = ScheduleT (V_s, *PEOrder*);
6: **if** extra timing slack has delayed $V_s.startTime$ **then**
7: $save =$ ReorderP ($V_s.PE$, *PEOrder*, *CommPath*);
8: $V_s.startTime = V_s.startTime - save$;
9: **end if**
10: UpdateComm(V_s, *CommPath*);
11: AdjustPriority(*Readylist*);
12: **for** $v_i \in$ *Child*(V_s) **do**
13: **if** all parents of v_i has scheduled **then**
14: *Readylist* = *Readylist* $+\{v_i\}$;
15: **end if**
16: **end for**
17: **end while**

Scheduling process

To schedule the selected task V_s, the procedure ScheduleT (line 5 in Algorithm 1), corresponding to the *processor assignment* phase of the baseline algorithm, iteratively places V_s on every PE to calculate the earliest starting time. This scheduling process of task V_s, with the timing constraints of *L2R dependences* incorporated, is formally represented in the following equations:

$$ST_{min}(V_s) = \min_k \{ST_{min}(V_s, P_k)\} \tag{4.2a}$$

$$ST_{min}(V_s, P_k) = \max\{FT(P_k), RT(V_s, P_k)\} \tag{4.2b}$$

$$RT(V_s, P_k) = \max_{v_j \in pre(V_s)} \{FT(v_j, PE(v_j)) + c(e_{js}) + T_{ex}\} \tag{4.2c}$$

As shown in Equation (4.2a), the earliest start times of V_s on each PE, denoted as $ST_{min}(V_s, P_k)$, are compared during the scheduling of V_s. Equation (4.2b) shows that the start time of V_s on PE P_k is furthermore constrained by either the current available time of the PE, denoted as $FT(P_k)$, or the ready time of incoming data, denoted as $RT(V_s, P_k)$. The latter factor is constrained by the last incoming communication, calculated by adding $c(e_{js})$, the communication latency between a predecessor task v_j and V_s, to the finish time of v_j, as shown in Equation (4.2c). The value of $c(e_{js})$ is set to zero if V_s and v_j are scheduled on the same PE, that is, if $PE(v_j) = P_k$.

L2R timing constraint incorporation

The most critical modification to the baseline algorithm is the incorporation of the timing constraint of *L2R* dependences outlined in Section 4.1.2. As shown in Equation (4.2c), this is implemented through adding an extra timing slack, denoted as T_{ex}, to the *data ready time* of V_s if it is placed on P_k that is at least two PEs to the right of the predecessor v_j.

The value of T_{ex} is a function of the *PE-distance* between P_k and $PE(v_j)$ as well as the *communication latency* $c(e_{js})$. To concretely illustrate this relationship, Figure 4.15 shows the earliest positions of a sink task on the various PEs under three distinct values of communication latency. As can be seen, a larger value of $c(e_{js})$ constrains the earliest starting time of V_s on each P_k, which may in turn preclude it from being placed

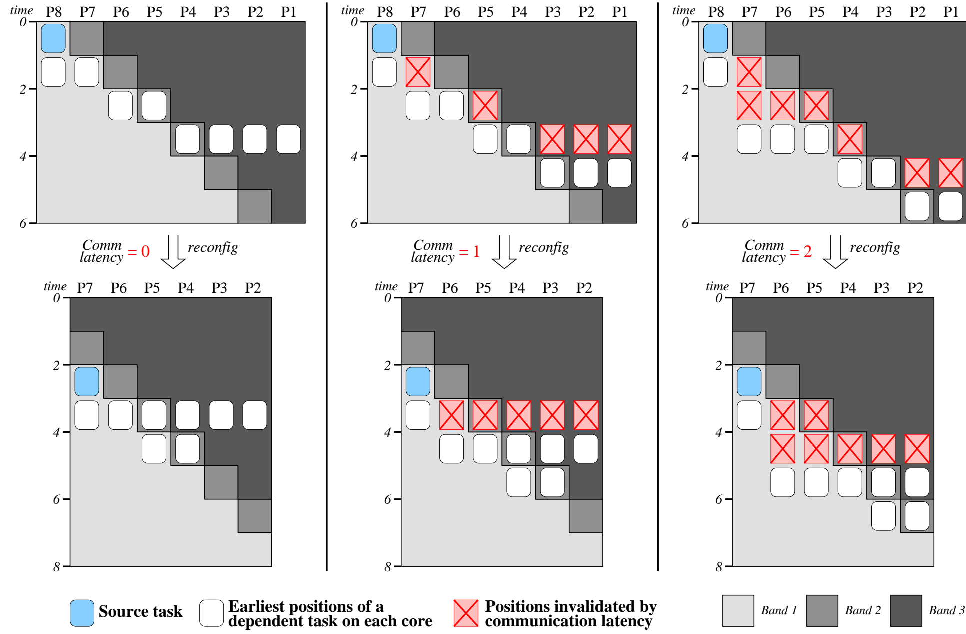


Figure 4.15: Impact of PE-distance and communication latency on the earliest start time of a sink task

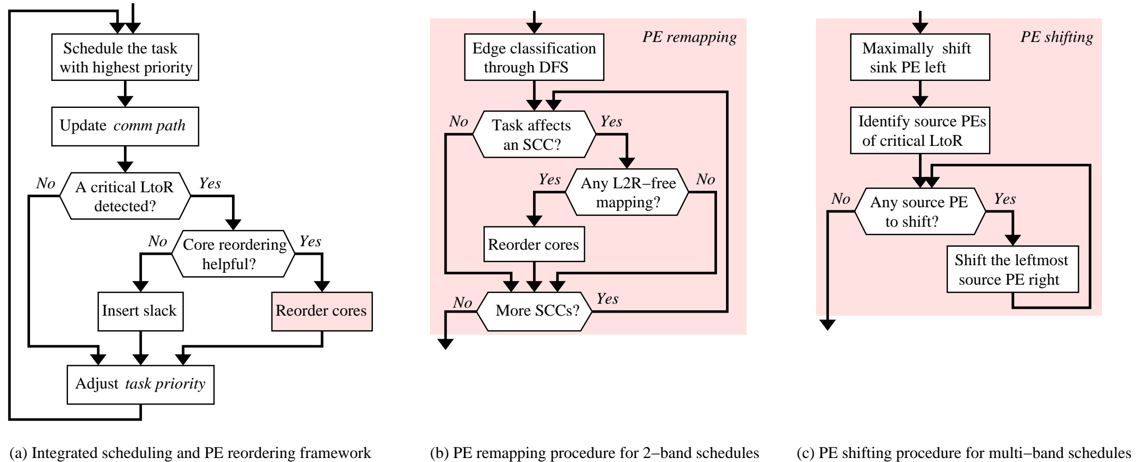
at a band with a larger index, thus incurring less amount of additional slack. Specifically, assuming a single-step timing slack of T_b, the T_{ex} value, as a function of the PE-distance $d = PE(v_j) - P_k$ and the communication latency c, can be determined by Equation (4.3a).

$$T_{ex}(c,d) = \begin{cases} 0 & \text{if } d \leq \lceil \frac{c}{T_b} \rceil \\ m * T_b & \text{if } d \geq \lceil \frac{c}{T_b} \rceil + 2m \\ \lfloor \frac{d - \lceil c/T_b \rceil}{2} \rfloor * T_b & \text{otherwise.} \end{cases} \tag{4.3a}$$

$$T_b = T_\mu + T_\sigma \tag{4.3b}$$

Equation (4.3a) indicates that T_{ex} is only imposed if $d \geq 2$, that is, the sink P_k is at least two PEs to the right of the source $PE(v_j)$. For a given value of $c(e_{js})$, the value of T_{ex} is proportional to the PE distance d, with the upper bound constrained by the maximum number of reconfiguration steps m.

As for the value of a single-step timing slack, i.e., T_b, ideally it should equal the maximum height of the steps on the *band partition line*. Yet at this point of the scheduling process, the initial schedule has not been fully generated, and neither have the *BR blocks*



(a) Integrated scheduling and PE reordering framework (b) PE remapping procedure for 2−band schedules (c) PE shifting procedure for multi−band schedules

Figure 4.16: Integrated task scheduling and core reordering flow

been formed. The value of T_b therefore needs to be estimated. We therefore make the assumption that the task execution time forms a *normal distribution* with mean T_μ and variance T_σ^2. As shown in Equation (4.3b), we set T_b to $T_\mu + T_\sigma$ to approximate the relative delay of the L band after reconfiguration.

As Equation (4.2a) selects the minimal start time of V_s among all the PEs, in most cases the reconfiguration-induced increase (T_{ex}) in data ready time would not delay the start time of Task V_s, but instead results in V_s being scheduled onto a PE that creates no *L2R dependence*. However, if the start time of V_s is unfortunately delayed by T_{ex}, the scheduler invokes the reordering procedure ReorderP (Line 7 in Algorithm 1) to determine whether the current PE order can be adjusted to eliminate this slack.

PE reordering flexibility exploitation

The PE reordering flow, integrated into the scheduling process, is shown in Figure 4.16a. Essentially, the reordering procedure identifies PE reordering possibilities according to the connectivity characteristics of the tight communication graph. Depending on the amount of adaptivity degree, the procedure either employs the *PE remapping* algorithm for 2-band schedules, or the *PE shifting* algorithm for multi-band schedules. As shown in Figure 4.16b, if the former is used, the reordering procedure first performs a depth-first search to classify the edges, and then iteratively checks all the strongly connected components whose connectivity is affected by the task just being scheduled. If the

checking results indicate the existence of an *L2R-free mapping* for all the tight communication paths in that SCC, the current PE binding order will be updated subsequently.

Figure 4.16c shows that if the *PE shifting* algorithm is used to exploit the PE reordering possibility in m-core adaptive schedules, the procedure first checks whether the sink PE, i.e., the one that V_s is scheduled on, can be shifted left. This is because if the start time of V_s is constrained by multiple critical L2R dependences, the left shift of the sink PE can *simultaneously* reduce the band distances and hence the timing slacks of these dependences. Once the sink PE has been maximally shifted left through the exploitation of the *direct* and *indirect* shift conditions, a number of critical *L2R* dependences may have already been eliminated. Subsequently, the reordering procedure tries to shift all the source PEs (the ones onto which the predecessors of V_s are scheduled) of the remaining critical L2R dependences right. In this process, the source PE at the leftmost position is shifted first, as this shifting may in turn eliminate or relax the shifting constraints among the multiple source PEs. Specifically, an L2R communication path ($i \rightarrow j$) between two source PEs i and j may preclude j from being shifted right. Therefore, by performing the right shift of PE i earlier than the right shift of j, the right-shift distance of PE j can be maximized.

4.5.2 Adaptive Schedule Optimization

The outlined task scheduling and core reordering process precludes potential violations of *L2R dependences* in the initial schedule. Yet the implementation of the proposed orderly reconfiguration scheme still requires the following functions to be incorporated into the static task scheduling process:

- Exploiting under-utilized portions of the initial schedule.
- Balancing step heights on *band partition lines*.
- Compensating extra inter-PE communication overhead.

To incorporate these three optimizations, a ***schedule partition*** phase is appended to the baseline algorithm to perform the block and band partitioning in **three steps**. The fully parallel regions of the initial schedule are first identified. Subsequently, the band and block *partition lines* are determined, and the step heights on each *band partition line*

are maximally balanced. Finally, the *PE rotation* approach is used to compensate for the extra inter-PE communication overhead, if any.

The positions of the fully parallel regions in the initial schedule determine the total number of BR blocks as well as the position of the *body* region of each BR block, thus strongly impacting the length of the post-reconfiguration schedule. For instance, if the initial schedule has no fully parallel regions, no change of initial schedule is needed in the case of a single processor deallocation.

More generally, in an m-adaptive schedule that originally utilizes n cores, if a search of the initial schedule identifies at least one region that utilizes more than $n - m$ cores, the *band and block partition lines* of each BR block will subsequently be determined. The goal of this step is to minimize the occurrence possibility of extra inter-PE communications. Consequently, each pair of dependent tasks tightly scheduled on the same PE is placed into the same band as much as possible.

Once the band partition lines have been determined, the next step is to balance the step height. To retain the length of the initial schedule intact, only the tasks on *non-critical paths* can be delayed. Here, the timing slack of each task v_j is determined through calculating its starting time in both the *as-soon-as-possible* (ASAP) and the *as-late-as-possible* (ALAP) schedules, as shown in the following equation.

$$Slack(v_j) = ALAP(v_j, PE(V_j)) - ASAP(v_j, PE(V_j))\} \tag{4.4}$$

At this point in the scheduling process, the shape of each band has been determined. If, as a result of band partitioning, two dependent tasks tightly scheduled on the same PE are separated into an **L** and a subsequent **R** band, the proposed PE rotation technique is applied. A *right rotation hint* is inserted between any such two BR blocks, so that the entire subsequent BR block can be rotated right at run-time in forming the *post*-reconfiguration schedule.

4.6 Experimental Results

In this section, the proposed adaptive scheduling and core reordering scheme is evaluated. The scheduling algorithm outlined in the last section is implemented in C,

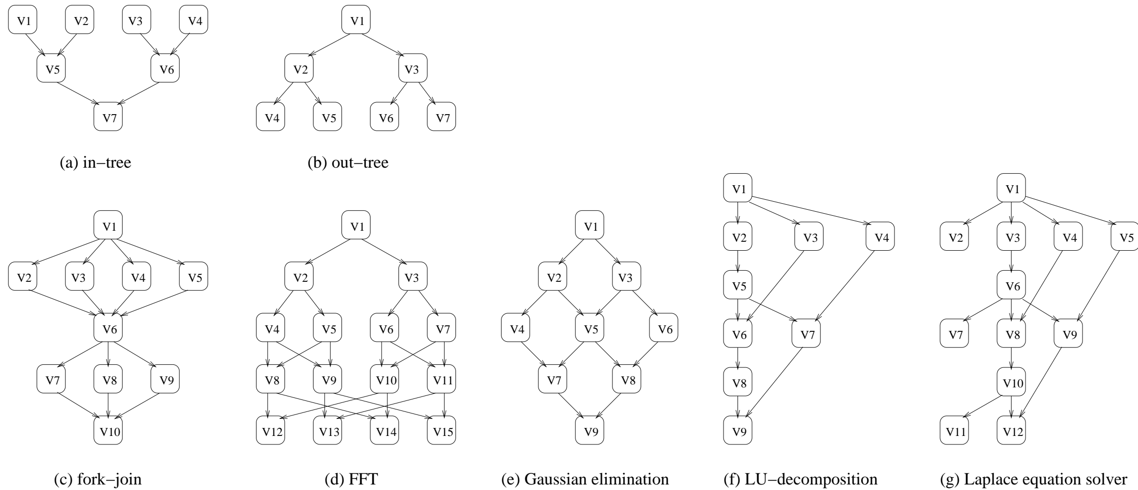


Figure 4.17: The benchmark task graphs

while the *Dynamic Critical Path* (DCP) algorithm [54] is selected as the baseline scheduling policy.

The application set under test is composed of standard parallel task graphs, including *fork-join*, *LU decomposition*, *Laplace equation solver*, *Gaussian elimination*, and *FFT*. DAG representations of these task graphs are shown in Figure 4.17. Meanwhile, to evaluate the effectiveness of the proposed technique when applied to various non-standard parallel applications, we use TGFF [25] to further generate a number of random task graphs representative of a large spectrum of possible parallel applications. The number of tasks varies from 40 up to 160. The *variation* in task execution time is controlled by setting the ratio of the upper to lower bound of task execution time to 2. Additionally, the frequency of inter-task dependences is controlled through varying the value of the average *out-degree* (the number of communications per task), while the communication overhead is controlled through varying the average *communication-to-computation ratio*.

4.6.1 Performance of single-core adaptive schedules

As the primary goal of the proposed reconfiguration scheme is to retain the performance of the initial schedule intact, we first evaluate the pre-reconfiguration schedule lengths. To evaluate the impact of both adaptivity and core reordering, we have experimentally compared the length of three schedules, the baseline (non-adaptive) schedule,

Table 4.1: **Pre-reconfiguration** schedule length

	***Pre-reconfiguration* schedule length overhead**		
cores	standard graph w/o → w/ reorder	random, 1-outdegree w/o → w/ reorder	random, 3-outdegree w/o → w/ reorder
3	3.14% → 1.90%	0.51% → 0.19%	1.12% → 0.48%
4	2.70% → 1.94%	2.00% → 1.00%	6.01% → 1.54%
5	-0.63% → -1.81%	0.22% → 0.15%	2.74% → 1.93%
6	1.41% → 0.67%	3.50% → 1.25%	5.37% → 2.28%
7	3.53% → 2.15%	0.73% → 0.40%	3.76% → 2.26%
8	5.32% → 2.31%	3.96% → 1.23%	4.55% → 2.51%
average	2.58% → 1.19%	1.82% → 0.70%	3.93% → 1.83%

the adaptive schedule without core reordering, and the adaptive schedule with core reordering. Meanwhile, to evaluate the effectiveness of the proposed technique for various workloads, we compare the schedule length results of standard parallel applications and the results of randomly generated task graphs with two distinct values of *out-degree* (i.e., the number of out-going communications per task). The results in pre-reconfiguration schedule length overhead achieved with and without PE reordering are reported in Table 4.1, as the number of cores considered in our experiments is varied from 3 to 8.

The results in Table 4.1 confirm that the overhead in pre-reconfiguration schedule length is insignificant. More precisely, without core reordering, the incorporation of adaptivity introduces roughly a 1.8–3.9% overhead on the schedule length, while the core reordering technique can further reduce such overhead to 0.7–1.8%. A more detailed examination shows that as the number of PEs increases, the amount of L2R dependences increases, which in turn causes the schedule length overhead to grow slightly.

A comparison between the two random workloads shows that the PE reordering technique delivers a relatively lower reduction in schedule length overhead for the high out-degree case. This is because an application with strong inter-task dependences exhibits a large number of edges in the communication path graph, thus limiting the possible reordering choices. A comparison between the standard and the random workloads shows that the two sets of results are highly similar, implying that the proposed adaptive scheduling and core reordering techniques are effective for a large spectrum of standard and non-standard parallel applications. Another noteworthy aspect is that during the scheduling process, a task is reassigned, if its starting time is not delayed thereafter, to a

Table 4.2: Impact of adaptivity on inter-PE communications

	Non-adaptive		Adaptive w/o PE reorder	
Cores	Total	Adjacent / R2L / L2R	Total	Adjacent / R2L / L2R
3	56.3%	0.503 / 0.237 / 0.260	54.1%	0.505 / 0.251 / 0.244
4	57.9%	0.405 / 0.275 / 0.320	57.9%	0.413 / 0.300 / 0.287
5	60.1%	0.340 / 0.304 / 0.356	59.3%	0.354 / 0.344 / 0.302
6	60.6%	0.293 / 0.316 / 0.391	61.1%	0.316 / 0.373 / 0.311
7	61.4%	0.260 / 0.329 / 0.411	62.1%	0.290 / 0.401 / 0.309
8	61.5%	0.235 / 0.336 / 0.429	62.6%	0.271 / 0.423 / 0.306
average	59.6%	0.339 / 0.361 / 0.300	59.5%	0.358 / 0.293 / 0.349

PE that can reuse an existing communication path. As a result, in some cases (such as the "5-core" case of the standard task graphs, for example), the adaptive schedules may even display a shorter length than the non-adaptive schedule.

As the quality of an adaptive schedule is largely determined by the amount of L2R communications, we additionally report the ratio of total inter-PE communications, as well as the communication breakdown information (i.e., the percentages of *adjacent, R2L*, and *L2R* communications) in Tables 4.2 and 4.3. Similarly, the number of cores varies from 3 to 8, while for each case three schedules are reported, namely, the *non-adaptive*, the *adaptive without reordering*, and the *adaptive with reordering* schedules.

The results in Tables 4.2 and 4.3 confirm that as the number of PEs increases, both adaptive schedules display a slightly increased amount of inter-PE communications as compared to the non-adaptive schedule. Regarding the amount of L2R communications, in the non-adaptive schedule, the value increases linearly as the number of PEs grows. Yet in both adaptive schedules, the amount of L2R communications is less sensitive to the number of cores. Compared to the non-adaptive schedule, a significant reduction (6-31%) in L2R communications is attained even without core reordering. This is because during task scheduling, as shown in Equation (4.2a), the scheduler selects the minimal start time of a task V_s among all the PEs, thus resulting in V_s being scheduled onto a PE that creates no L2R dependence. In comparison, the PE reordering technique reduces the amount of L2R communications by an additional amount of 5–7%.

At first sight the results in Tables 4.2 and 4.3 seem to indicate that the core reordering technique makes an insignificant contribution to reducing the amount of L2R communications. However, it needs to be noted that the core reordering procedure is in-

Table 4.3: Impact of PE reordering on L2R communications

	Communication breakdown		Critical L2R
Cores	Total	Adjacent / R2L / L2R	w/o → w/ reorder
3	55.2%	0.510 / 0.261 / 0.229	0.87% → 0.18%
4	57.7%	0.417 / 0.313 / 0.270	1.54% → 0.41%
5	60.0%	0.357 / 0.367 / 0.277	1.96% → 0.66%
6	61.0%	0.318 / 0.396 / 0.286	2.34% → 0.80%
7	62.0%	0.288 / 0.426 / 0.286	2.59% → 0.89%
8	62.4%	0.269 / 0.446 / 0.285	2.71% → 0.99%
average	59.7%	0.360 / 0.272 / 0.368	2.00% → 0.65%

voked only upon *critical L2R dependences* that delay the earliest start time of the task under scheduling. The ratios of critical L2R communications (to the total amount of inter-PE communications) in both adaptive schedules are reported in Table 4.3 as well. As can be seen, the core reordering technique delivers a 63–79% reduction in the critical L2R communications. Such a significant reduction in turn leads to a sizable improvement in schedule quality, as confirmed by the results of schedule length in Table 4.1. These results thus clearly confirm the effectiveness of the proposed core reordering technique in mitigating L2R dependences and in minimizing schedule length overhead.

Subsequently, we report the post-reconfiguration schedule lengths in Table 4.4. It needs to be noted that the adaptive schedules reported in the "k-core" row, as they utilize k-1 cores after reconfiguration, are actually compared to the baseline schedule of k-1 cores to ensure fairness in schedule quality evaluation. The results in Figure 4.4 show that the overhead in post-reconfiguration schedule length sizably decreases as the number of PEs increases. This is because the use of more PEs results in a larger part of under-utilized portions in the initial schedule that needs no reconfiguration in the case of a single PE deallocation. However, without PE rotation, the post-reconfiguration schedule length overhead is still significant, ranging from 12% to 23%. In comparison, the PE rotation technique, applied on the adaptive schedules optimized through PE remapping, can effectively reduce such overhead to the range of 4.5–9.6%. These results thus clearly confirm the criticality and the attainable benefit of the proposed PE rotation technique in helping generate adaptive schedules.

Table 4.4: **Post-reconfiguration** schedule length

	***Post-reconfiguration* schedule length overhead**		
cores	standard graph w/o → w/ reorder	random, 1-outdegree w/o → w/ reorder	random, 3-outdegree w/o → w/ reorder
3	23.2% → 11.4%	37.3% → 9.80%	21.0% → 10.9%
4	17.3% → 11.7%	23.5% → 2.36%	14.0% → 5.69%
5	8.25% → 4.22%	23.1% → 7.21%	13.1% → 8.85%
6	3.62% → 0.45%	17.0% → 4.00%	12.1% → 7.00%
7	9.19% → 6.02%	15.6% → 8.49%	12.4% → 8.93%
8	10.4% → 6.02%	12.8% → 5.72%	11.9% → 8.09%

4.6.2 Performance of multi-core adaptive schedules

To evaluate the impact of increased reconfiguration steps, we have experimentally compared the performance of m-adaptive schedules against the baseline schedule (without adaptivity), with values of m ranging from 1 to 3. For each adaptivity degree, we furthermore evaluate the impact of PE reordering. A total number of 7 schedules is therefore reported in Figure 4.18. The number of cores considered in these experiments is varied from 5 to 8, while the schedule length values are normalized to the schedule length of the baseline algorithm generated for 5 PEs.

The results in Figure 4.18 confirm that the toleration of more core variations degrades the performance of the pre-reconfiguration schedules. Without core reordering, the overhead in schedule length increases linearly as the amount of adaptivity degree increases. Yet the core reordering technique can significantly reduce the schedule length overhead, especially in the high adaptivity-degree case. More precisely, when the adaptivity degree increases from 1 to 3, the schedule length overhead increases from 1.7% to 4.9% **without** core reordering, and from 0.8% to 1.6% **with** core reordering. These results clearly confirm the necessity of core reordering, as well as the effectiveness of the proposed reordering technique in minimizing the overhead of m-adaptive schedules.

Figures 4.19 and 4.20 respectively report the ratios of L2R communications and *critical* L2R communications (to the total amount of inter-PE communications) in all the m-adaptive schedules. It can be observed that the schedules with a larger value of m display less amount of L2R communications. Same as in the 1-core adaptive schedules, a significant reduction in L2R communications is attained even without core reordering.

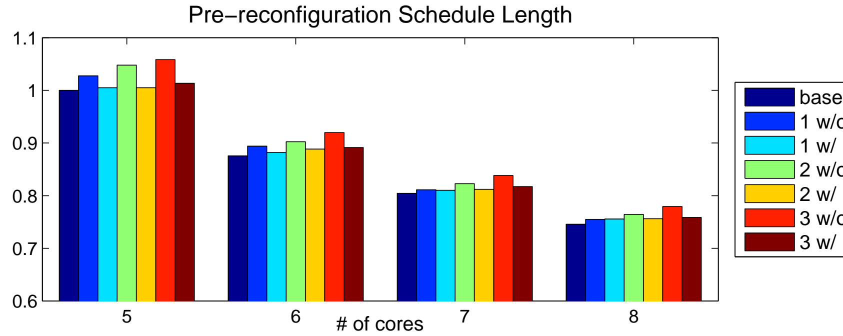


Figure 4.18: Impact of adaptivity degree and core reordering on **pre-**reconfiguration schedule length

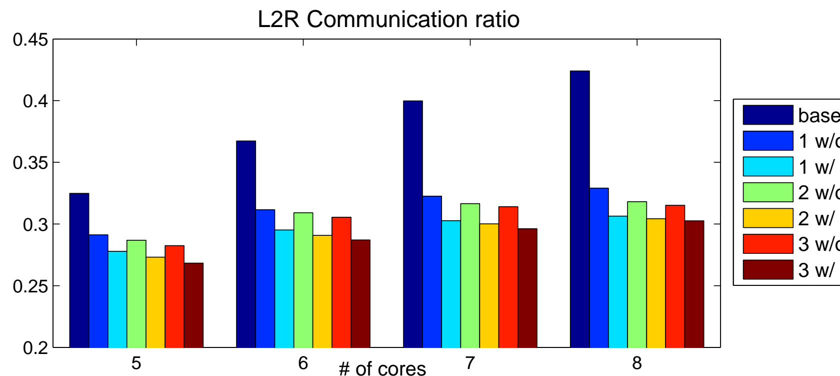


Figure 4.19: Impact of adaptivity degree and core reordering on the amount of **L2R** communications

On average, a 18% and a 22% reduction in L2R communications (over baseline schedules) is attained without and with core reordering, respectively. Yet the results regarding the *critical* L2R communications confirm that the core reordering technique delivers a 61% reduction in the critical L2R communications for 1-adaptive schedules, and a 55% reduction for 3-adaptive schedules. Such a significant reduction in turn leads to a sizable improvement in schedule quality, as confirmed by the results in Figure 4.18.

Finally, the post-reconfiguration schedule lengths are reported in Figure 4.21. To ensure fairness in comparison, an m-adaptive schedule of n cores needs to be compared to the baseline schedule of $n - m$ cores. Accordingly, in Figure 4.21, the results listed in the "k-core" column with an adaptivity degree of m actually use $k + m$ cores in the pre-reconfiguration schedule. As can be seen, the overhead in schedule length increases

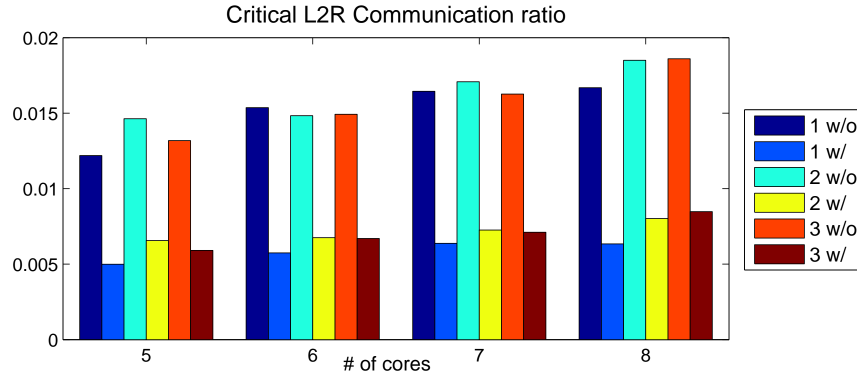


Figure 4.20: Impact of adaptivity degree and core reordering on the amount of **critical L2R** communications

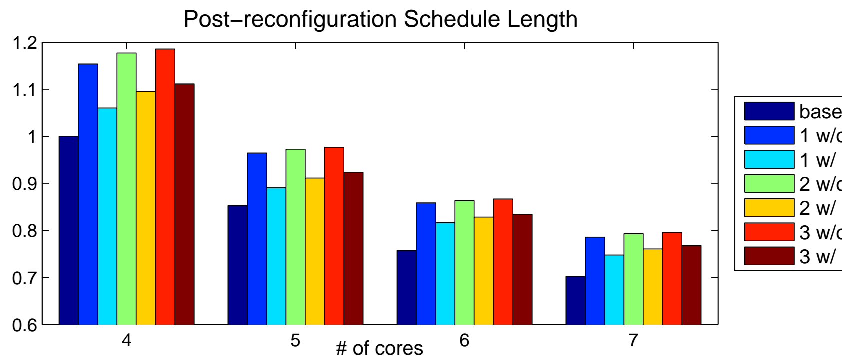


Figure 4.21: Impact of adaptivity degree and core reordering on **post-**reconfiguration schedule length

as the amount of adaptivity degree increases, yet at a much slower slope as compared to pre-reconfiguration schedules. Using the core reordering technique, the overhead in schedule length is reduced from 13% to 6% for 1-adaptive schedules, and from 15% to 10% for 3-adaptive schedules. These results therefore clearly confirm the criticality and the attainable benefit of the proposed reordering technique in helping generate m-adaptive schedules.

4.7 Conclusions

In this chapter, we have presented an effective technique that allows reconfigurability to be incorporated into static schedules to withstand resource variations at runtime

due to unpredictable device failure, thermal stress, resource competitions or preemptions. Such adaptivity, in a nutshell, is delivered by expanding the conventional static scheduling techniques to embed the concept of partitioned bands, which enable the compiler to compactly engender in readiness numerous execution schedules. Such schedules can be adaptively applied upon run-time resource variations, with no reliance on any run-time rescheduling decision. Moreover, through novel permutation approaches on the task allocation space regarding the logical core positions, a set of *core reordering* techniques can effectively mitigate the reconfiguration-induced performance overhead. This advantage is confirmed by the experimental results, which show that an overhead strictly less than 2.5% is imposed on the length of the initial, pre-reconfiguration schedule to accomplish the highly regular and predictable reconfiguration scheme for the toleration of three core degradations. The confluence of these approaches thus delivers a fixed-silicon architecture capable of extracting intensive static analysis that complements dynamic reconfigurations in the face of arbitrarily large resource variations. The emerging resource variation-based unpredictability in future multicore systems is thus tamed to deliver an orderly and deterministic adaptivity to address the future needs of both general-purpose and embedded system architectures alike.

The text of Chapter 4, is in part a reprint of the material as it appears in *C. Yang and A. Orailoglu, "Predictable Execution Adaptivity through Embedding Dynamic Reconfigurability into Static MPSoC Schedules," International Conference on Hardware/Software Codesign and System Synthesis (CODES-ISSS), October 2007*; in *C. Yang and A. Orailoglu, "Towards No-cost Adaptive MPSoC Static Schedules through Exploitation of Logical-to-physical Core Mapping Latitude," IEEE Design, Automation and Test in Europe (DATE), April 2009*; and in *C. Yang and A. Orailoglu, "Fully Adaptive Multicore Architectures through Statically-directed Dynamic Execution Reconfigurations," International Conference on VLSI and System-on-Chip (VLSI-SoC), September 2010.* The dissertation author was the primary researcher and author of the publications [94], [97], and [98].

Chapter 5

Adaptivity-aware System Topology

The compiler-directed reconfiguration scheme introduced in Chapter 4 allows tasks to be transferred among a set of PEs in a regular manner with no reliance on any dynamic rescheduling decisions, thus completely eliminating the runtime decision making overhead. However, to minimize the overall reconfiguration overhead, such a predictable reconfiguration process still needs to be supported by a flexible customization of the underlying system topology so as to effectively hide the overhead in transferring tasks between PEs.

One of the main benefits of the proposed compiler-directed reconfiguration process is that execution migration only involves a limited set of adjacent PEs, regardless of the PE being isolated. This property holds for both single-core and multi-core adaptive schedules, as can be clearly observed from the reassignment directions shown in Figures 4.2 and 4.11. This high locality indicates that reconfiguration-induced code/data transfer can be eliminated even in a distributed architecture that is envisioned as the dominant architecture for future multi-core and many-core systems. By providing neighborhood-centered, dedicated communication links, PEs that are physically adjacent can access and share a single storage unit in common, thus enabling tasks to be directly migrated among these PEs without any physical movement of the code/data set. Such a *locally shareable* storage organization in turn enables the development of a light-weight, neighborhood-centered communication scheme to accelerate task execution as well.

In this chapter, we first analyze the reconfiguration-induced sharing requirement on storage units. Subsequently, we present a version of a locally shareable storage orga-

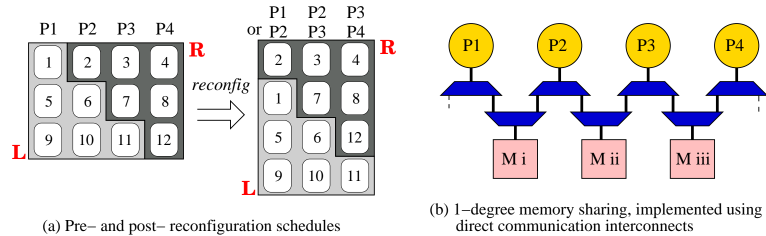


Figure 5.1: Reconfiguration-induced sharing requirements

nization model, an appealing compromise capable of responding to the twin requirements of scalability and shareability for future multi-core systems. At the system level, a set of 2-dimensional physical topologies can be developed, with diverse sharing degree embedded that matches different levels of reconfiguration and communication needs. We furthermore outline a set of topology selection criteria as well as the associated task placement decisions. Finally, we propose a static-encoding based distributed synchronization mechanism which, through the utilization of the dedicated communication links, effectively accelerates inter-task communications.

5.1 Reconfiguration-induced Sharing Requirement

To illustrate the sharing requirement imposed by the BB reconfiguration, we first consider the single-core adaptive schedule shown in Figure 5.1a, wherein 12 tasks are partitioned into two bands. Each column of the post-reconfiguration schedule can be executed either on core P_i or P_{i+1}, depending on the position of the deallocated PE. A detailed comparison of the pre- and post-reconfiguration schedules indicates that a task initially executed on PE P_i may need to be migrated *right* to PE P_{i+1}, if the task is in the *L* band. Similarly, if the task is in the *R* band, it may need to be migrated *left* to PE P_{i-1}. As an example, if P_3 fails, among the tasks that were initially scheduled on it, Task 7 needs to be migrated to P_2 , while Task 11 needs to be migrated to P_4.

The examination above of *single*-core adaptive schedules indicates that task migration is only performed between **two** adjacent PEs, implying that the sharing of a single storage unit between two PEs suffices for eliminating the reconfiguration-induced task movements. A more general examination into the *multi*-core adaptive schedules indicates that the maximum task migration distance is linearly proportional to the amount of recon-

figuration steps embedded within the schedule. More precisely, for a schedule capable of tolerating a variation of m cores, reconfiguration-induced task movement can be completely eliminated as long as a single storage unit is shared between every $m + 1$ adjacent PEs. This sharing requirement can be formally specified through a parameter of **sharing degree**, defined as follows:

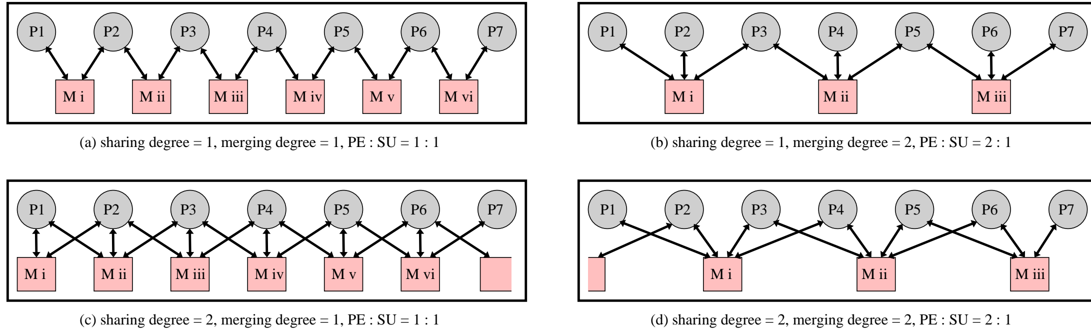
> *Sharing degree: the extra number of cores with which a core P_i shares a single storage unit.*

Under this definition, the *minimum* amount of sharing degree required in a m-core adaptive schedule is m. In comparison, assuming that the multicore platform contains a total number of N PEs, the traditional distributed and centralized memory organizations can be viewed as two extreme cases with sharing degrees of 0 and $N - 1$, respectively. Clearly, the distributed organization falls short of fulfilling the reconfiguration-induced sharing requirements. The centralized organization, on the other hand, offers overmuch sharing capability and hence suffers from the crucial limitation of the lack of *scalability*. It is therefore necessary to develop a *locally shareable* storage organization, wherein dedicated communication links are provided within a neighborhood to attain the right amount of sharing degree.

5.2 Locally Shareable Storage Organization

In the proposed locally shareable organization, storage units (SUs) are still organized in a distributed form across the entire platform, while sharing is achieved through enabling each PE to directly access multiple SUs. These SUs, which can be either L2 caches or on-chip memory units, are connected through a conventional **on-chip network** for the support of long-distance communications, while the PEs and SUs are directly connected through **point-to-point communication links**. These communication links, enable multiple tasks to be *simultaneously* migrated between distinct PEs without inducing any interferences or network congestion. As these links are highly localized, they impose negligible hardware cost and routing overhead.

Clearly, the number of communication links is directly determined by the amount of *sharing degree*. By definition, a sharing degree of s directly implies that each single



(a) sharing degree = 1, merging degree = 1, PE : SU = 1 : 1

(b) sharing degree = 1, merging degree = 2, PE : SU = 2 : 1

(c) sharing degree = 2, merging degree = 1, PE : SU = 1 : 1

(d) sharing degree = 2, merging degree = 2, PE : SU = 2 : 1

Figure 5.2: Bipartite graph representation of various topologies with distinct values of sharing degree and merging degree

SU is accessible to $s+1$ PEs. Given an MPSoC platform with a total number of N PEs and N SUs, the interconnect network thus exhibits the following characteristics:

$$\text{Average interconnects per PE} = s+1 \quad (5.1a)$$

$$\text{Average interconnects per SU} = s+1 \quad (5.1b)$$

$$\text{Total interconnects} = (s+1)\cdot N \quad (5.1c)$$

Equations (5.1a) and (5.1b) indicate that each PE and each SU are connected to multiple direct communication links. Yet this many-to-many interconnect network can still be attained without increasing the number of read/write ports for each PE. As a PE does not access distinct SUs simultaneously, a single read/write port, together with a set of decoders and multiplexers, suffices for the accesses from a single PE to multiple SUs. This type of organization is concretely illustrated in Figure 5.1b that shows 4 PEs with a sharing degree of 1.

As each direct communication link connects a PE with an SU, the entire interconnect network can be modeled as a *bipartite graph*[1] between two disjoint sets of nodes, the PEs and the SUs. The bipartite graph representations of MPSoC platforms with sharing degree values of 1 and 2 are respectively shown in Figures 5.2a and 5.2c. As can be seen, in Figure 5.2a, any two PEs with consecutive indices (P_i and P_{i+1}) share a single SU in common, while in Figure 5.2c, any three PEs with consecutive indices (P_i, P_{i+1}, and P_{i+2}) share a single SU in common. On the other hand, this increased sharing capacity of

[1]For simplification purposes, the direct communication links in the subsequent parts of this paper are shown in the bipartite graph format instead of the format presented in Figure 5.1b.

the storage units is attained at a cost of increased interconnects, as Figure 5.2b displays more communication links than 5.2a. This property is also confirmed by Equation (5.1c), which indicates that the total number of point-to-point communication links in this locally shareable organization is linearly proportional to the amount of sharing degree s.

The larger the sharing capability is, the more the communication links needed in the system and the higher the consequent cost would be. The consideration of scalability requires the reduction of the total number of communication links, however, without sacrificing the sharing capacity. To attain this goal, we exploit the flexibility of merging a number of adjacent SUs together, which in turn enables a combination of the communication links emanating from a single PE to these SUs. Here, we formally define the parameter *merging degree* as follows:

> *Merging degree: the number of SUs that have been merged to form a single SU, which is equal to the ratio of the number of PEs over the current number of SUs.*

Figures 5.2b and 5.2d present the bipartite graphs generated through merging every two contiguous SUs in Figures 5.2a and 5.2c, respectively. The comparisons between Figures 5.2a and 5.2b, and between Figures 5.2c and 5.2d confirm that link merging reduces the total number of SUs and the total number of interconnects, while increasing the number of PEs that share an SU in common. More formally, by merging k adjacent SUs into a single one, every set of $s + k$ PEs shares an SU in common, while the total number of SUs is reduced from N to N/k. The interconnect network thus exhibits the following properties as a result of the link merging:

$$\text{Average interconnects per PE} = s/k + 1 \quad (5.2a)$$

$$\text{Average interconnects per SU} = s + k \quad (5.2b)$$

$$\text{Ports per SU} = k \quad (5.2c)$$

$$\text{Total interconnects} = (s/k + 1) \cdot N \quad (5.2d)$$

A comparison between Equations (5.1c) and (5.2d) clearly confirms that link merging can effectively reduce the total number of direct communication links by k times. In an extreme case, if the values of the *sharing degree* and the *merging degree* are set to be equal ($s = k$), the total number of communication links will always equal $2N$, independent of the s and k values. As a concrete example, in Figure 5.2a the bipartite graph

($s = k = 1$) contains 12 links, while in Figure 5.2d the bipartite graph ($s = k = 2$) also contains 12 links.

The sharing of an SU among multiple PEs offers the extra benefit of *adaptive resource allocation*. Each PE does not necessarily to have the same amount of storage. Instead, the allocation of storage cells to different PEs can be determined according to the total amount of storage required by each PE. Clearly, the amount of flexibility in storage allocation is proportional to the *merging degree*. However, it needs to be noted that a merged SU needs to serve accesses from more PEs simultaneously, implying that the number of ports per SU needs to increase linearly as a function of the merging degree. Such an increased complexity in turn imposes upper bounds on the value of the *merging degree*.

5.3 Physical Topology and Application Mapping

After introducing the properties of the proposed *locally shareable* storage model, we subsequently examine the various topology instances that fulfill the *locally shareable* property. As such a property is independent of a particular topological structure, distinct 2-dimensional topologies that exhibit a varying amount of *sharing degree* and the *merging degree* can be developed.

5.3.1 Topology instances and the associated properties

Figure 5.3 presents two representative 2-dimensional physical topologies corresponding to the bipartite graph representations shown in Figures 5.2a and 5.2d. PEs and SUs are connected through the direct inteconnect links, while all the SUs are connected through an underlying on-chip network for the support of long-distance communication and data transfer. It can be observed that these topology instances exhibit the following properties:

- In a topology with a sharing degree of s, every $s + 1$ cores with consecutive indices (P_i, P_{i+1}, ..., and P_{i+s}) consistently hold in common an SU.
- In both topologies, each PE has at least **two** direct communication links, thus retaining its connectivity in the case of single failures of communication links.

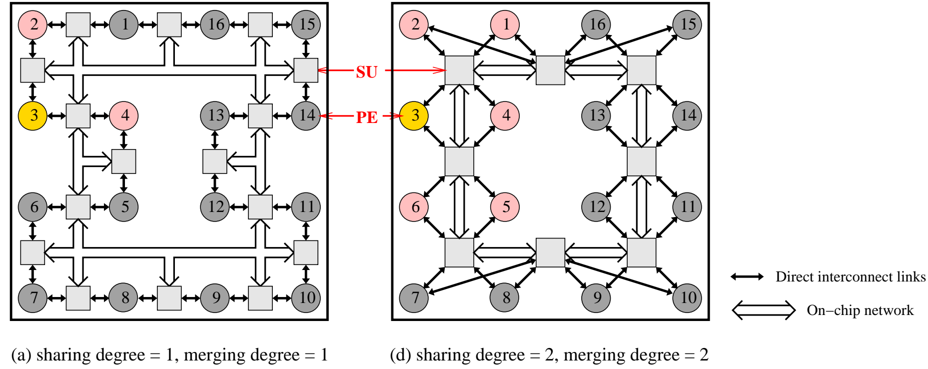


Figure 5.3: Various 2-dimensional locally shareable MPSoC topologies

The first property indicates that the proposed topologies are not only able to tolerate core failures but furthermore able to minimize the execution reconfiguration overhead. As every set of $s+1$ adjacent cores consistently hold in common an SU, the platform is able to obviate any transfer of code or data set among any $s+1$ adjacent PEs. In other words, for all the possible reconfiguration processes embedded in an m-core adaptive schedule, the platform is able to obviate any transfer of code or data set as long as $m \leq s$. If, on the other hand, $m > s$, the platform is able to tolerate a reduction of s cores with no reliance on any data movements during task migration, while the toleration of the remaining $m-s$ cores requires data to be transferred through the underlying on-chip network that connects all the SUs.

The ability of any $s+1$ cores with consecutive indices to hold in common an SU furthermore simplifies resource allocation in the proposed adaptive system. By definition, an m-core adaptive schedule can be executed by n, $n-1$, ..., $n-m$ cores. To execute this pre-optimized schedule, the OS only needs to find a consecutive "window" of i idle cores such that $n-m \leq i \leq n$. Such flexibility in window selection furthermore precludes any possible fragmentation in resource allocation. Even in the case when the total number of consecutive idle cores is insufficient (i.e., $i < n-m$), the OS can force the applications that is using the adjacent cores to go through a reconfiguration process and give up a limited number of resources. This scenario of execution adaption for resource allocation has already been examined in Section 3.

The second property guarantees that the topologies shown in Figures 5.3 can tolerate single failures of direct communication links, as single failures can block at most

one of the communication paths. A detailed examination indicates that this fault tolerant capability is directly determined by the sharing degree s and the merging degree k. More precisely, a merging of k out of $s + 1$ links may reduce the minimum number of interconnects per PE to $s - k + 1$. Accordingly, as long as $s - k \geq 0$ each PE is guaranteed to have at least 2 direct communication links.

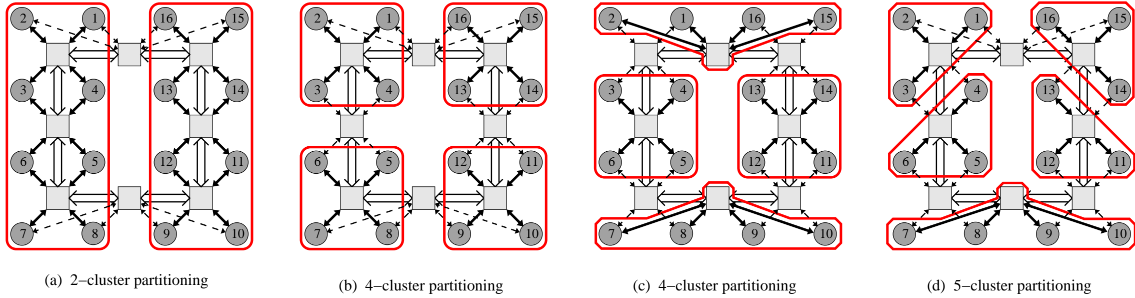
Finally, in contrast to PE and communication links, SUs are composed of multiple storage elements of high regularity. The failure of a set of storage cells would hardly have any impact on the connectivity of the entire multicore platform. Standard error detection and correction techniques can be furthermore employed to attain fault tolerance at a low cost.

The examination above confirms that all the MPSoC topologies shown in Figure 5.3 deliver the capability of tolerating device failures in the PEs, the SUs, and the communication links, thus enabling them to be employed by various application sets to attain fault tolerance and adaptive execution.

5.3.2 Topology instance selection

While the topology instances presented in Figure 5.3 provide a varying amount of *sharing capability*, they all exhibit a regular structure; the sharing and merging conditions of each PU are identical such that every set of $s+k$ cores with consecutive indices consistently hold in common a single PU. Such regularity enables these topological structures to be adopted as fixed-silicon MPSoC platforms, thus providing the benefits of high-volume amortization. Meanwhile, the regular topological structures also deliver flexible redefinitions of the platform to match parallelism characteristics and resilience needs of the application.

The selection of the most suitable topologies for diverse application sets can be based on two considerations. On one hand, the values of *sharing degree* and the *merging degree* should be determined according to the required reconfiguration needs as well as the inter-PE communication patterns. On the other hand, consideration of design complexity constrains the maximum number of interconnects as well as the number of interconnects per SU, which in turn imposes upper bounds on the values of the *sharing degree* and the *merging degree*.



(a) 2–cluster partitioning (b) 4–cluster partitioning (c) 4–cluster partitioning (d) 5–cluster partitioning

Figure 5.4: Finer-grained cluster partitions and communication link utilization

Parallelism consideration

While the number of PEs in each of the four topologies in Figure 5.3 is fixed to 16, this parameter can be easily customized to other values without impacting the structures of the bipartite graphs. Topologies with a larger number of PEs can be used to hold applications with large amount of thread-level parallelism. On the other hand, if the application cannot be parallelized into dozens of independent threads, the PEs can be partitioned into a number of *finer-grained* clusters so as to concurrently hold several applications.

Upon the scheduling of an application, the corresponding maximum resource utilization can be straightforwardly obtained. In the case of an underutilization of the resources, the target MPSoC can be partitioned into *finer-grained* clusters. Figure 5.4 presents four distinct ways for partitioning the 16 PEs into 2, 4 and 5 clusters. More formally, the 16 PEs in the proposed topology instances can be partitioned into 2 to 8 disjoint clusters, with each cluster containing no less than 2 cores so as to attain fault tolerance. During execution, communications are only performed within each cluster, implying that only the direct communication links within a cluster need to be activated (the bold lines in Figures 5.4) and the inter-cluster communication links can be deactivated (the dashed lines in Figures 5.4). Yet upon the failure of a communication link, the platform can be *repartitioned* so as to utilize a different set of communication links. This property can be observed through a comparison between Figures 5.4b and 5.4c. While in both figures the PEs are partitioned into 4 clusters, the sets of active communication links (highlighted in bold) used in these two partitions are disjoint.

Reconfiguration and communication consideration

To select an appropriate topology instance for an application, both the reconfiguration needs and the communication frequency of each application should be considered. First of all, to completely eliminate the reconfiguration-induced data transfer among SUs in an m-core adaptive schedule, the number of PEs that can directly access a single SU, i.e., the value of $s + k$, should be no less than the adaptivity degree m. Clearly, this topology selection criterion can be directly determined, once the adaptivity degree has been determined according to the occurrence frequency of core unavailability within the system.

In comparison, the second topology selection criterion that concerns the communication characteristics is application-specific. More precisely, to completely hide communication overhead, the number of *direct neighbors* of a PE should be no less than the maximum number of tasks involved in a single communication, that is, the *out-degree* of the corresponding task graph. Here, the *direct neighbors* of a PE P_i are defined as the PEs that share an SU with P_i.

Obviously, the value of the *direct neighbors* of a PE is determined by the *sharing degree* and the *merging degree*. By definition, a sharing degree of s directly implies that every $s + 1$ cores with consecutive indices consistently share a single SU in common. Accordingly, core P_i always has at least $2s$ direct neighbors with contiguous indices, ranging from P_{i-s} to P_{i+s}. This property can be observed in the topologies shown in Figure 5.3, wherein the direct neighbors of PE P_3 are shown in pink. In Figure 5.3a P_3 has P_2 and P_4 as its direct neighbors as the corresponding sharing degree is 1. In comparison, in Figures 5.3b P_3 has P_1, P_2, $P4$, and $P5$ as its direct neighbors as the corresponding sharing degree is 2. Meanwhile, as the link merging process increases the number of PEs that share a single SU, in Figure 5.3b P_3 additionally has P_6 as its direct neighbors. In sum, a detailed examination indicates that the average number of direct neighbors of a PE is $2s + k - 1$.

Design complexity consideration

Although the consideration of reconfiguration and communication needs argues for a topology with larger values of *sharing degree* and *merging degree*, the consideration

of design complexity, on the other hand, constrains the maximum number of interconnects as well as the number of interconnects per SU. As mentioned before, a larger value of the *sharing degree* increases the total number of interconnects, while a larger value of the *merging degree* requires both the size and the number of ports of the SU to scale proportionally. These considerations therefore impose upper bounds on the values of the *sharing degree* and the *merging degree*.

5.3.3 Task Placement Requirements

Upon the selection of the most suitable topology for an application, the associated task placement decisions can be made. As each PE can access multiple SUs in the proposed system organization, there exist multiple choices for placing the code and data set of each task. Given the pre-generated adaptive schedule, task placement decisions need to be made in such a way that the potential reconfiguration-induced data movement can be completely eliminated.

As we examined before, in a single-core adaptive schedule, a task initially scheduled on P_i may either need to be migrated *right* to PE P_{i+1} if it is in the L band, or need to be migrated *left* to PE P_{i-1} if it is in the R band. Task placement decisions can therefore be made accordingly. In brief, tasks in the L band should be placed in the SU shared between P_i and P_{i+1}, while tasks in the R band should be placed in the SU shared between P_i and P_{i-1}. These placement decisions can be observed in Figure 5.5b, which shows the overall data placement of the schedule presented in Figure 5.5a. Taking PE P_2 as an example, among the three tasks originally scheduled on it, Task 2, as it lies in the R band, is placed in the SU shared between P_1 and P_2. Tasks 6 and 10, as they lie in the L band, are placed in the SU shared between P_2 and P_3. Yet in both cases, task migration is only performed between **two** adjacent PEs, implying that the sharing of a single storage unit between two PEs suffices for hiding the reconfiguration latency completely.

If there exist multiple valid selections for the placement of a task, the decisions are made so as to balance the amount of storage. This case occurs if the sharing degree of the selected MPSoC topology exceeds the maximum reconfiguration step needed by the application. For instance, the bipartite graph shown in Figure 5.5c exhibits a sharing degree of 2. The tasks in Figure 5.5a that will never be executed by P_1 or P_4, namely,

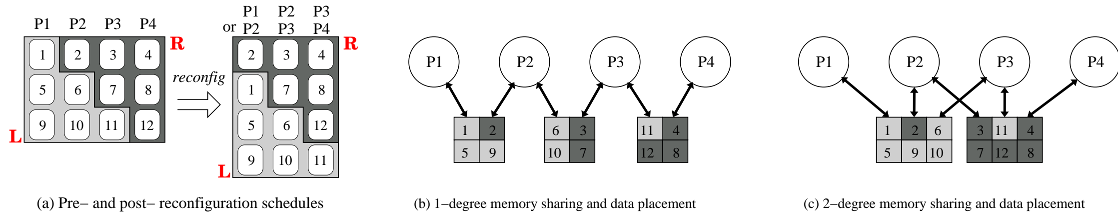


Figure 5.5: Reconfiguration-induced memory sharing and data placement

Tasks 3, 6, 7, and 10, can be placed in either SU. These tasks therefore can be evenly split into the two SUs in various ways, with one possible solution shown in Figure 5.5c. The only constraint here is to separate Tasks 6 and 7. This is because the placement of Tasks 6 and 7 in the same SU would require an increase in the number of access port, as the two tasks are executed at the same timing step in the pre-reconfiguration schedule.

The aforementioned data placement decisions are made under the assumption that the selected MPSoC topology exhibits a merging degree of 1. Clearly, these decisions display the finest granularity, thus resulting in their applicability to an MPSoC of merging degree $k > 1$, as the data placement can also be merged along with the SUs. In an extreme case, the PEs that can be utilized by an application share an SU in common (due to the lack of parallelism), implying that no data placement decisions need to be made.

5.4 Communication Overhead Minimization

Once the topology of the target MPSoC platform is determined, the static scheduler can utilize the direct communication links offered by the selected topology instance to minimize communication overhead.

Essentially, communications in the target MPSoC platform can be performed in two ways. Two PEs that share no SU in common need to communicate through a conventional on-chip network [10] that connects all the SUs. This type of communication, denoted as *remote* communication, requires data to be transferred between SUs through the underlying network. On the other hand, two PEs with a shared SU can directly communicate through one PE directly reading the data written by the other. This type of communication is denoted as *local* communication. Compared with *remote* communications, the overhead of *local* communications can be effectively hidden, if a cheap yet efficient

synchronization scheme is provided. Unfortunately, traditional synchronization mechanisms, such as *spin locks* and *barriers*, falls short of fulfilling this requirement. They ensure mutual exclusion through continuous polling of a shared variable, thus not only imposing large contention on the on-chip network, but also requiring memory accesses to be serialized.

To overcome this limitation, we additionally propose a light-weight distributed *synchronization mechanism* for the proposed locally shareable storage model. Rather than explicitly inserting synchronization variables to serialize the transmission of data through a shared memory location, we propose a mechanism to encode dependence information within each memory access, thus enabling synchronization to be combined together with data communication. Furthermore, by utilizing statically extracted application information, a sharp reduction in the number of code bits needed is attained through the proposed reference coloring algorithm, thus enabling an implementation within negligible hardware overhead.

5.4.1 Encoding-based Synchronization

As each communication is composed of a write operation followed by a read operation to the same memory location, the dependence information between these memory accesses can be statically extracted and explicitly encoded. These code words can be written/read together with the data in transfer. During execution, a dynamic checking of the encoded dependence information enables the identification of the status of the data in communication, based on which the read operation can be suspended to achieve semantically correct communication, thus completely eliminating the accesses to explicit synchronization variables.

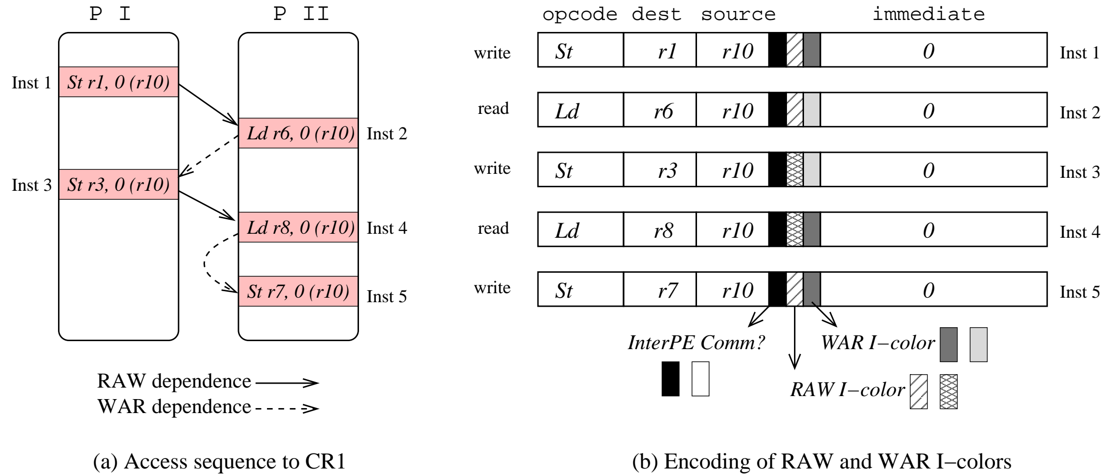
While statically encoding data dependences of each communication to reduce synchronization overhead is desirable, an effective encoding mechanism is still necessitated in order to capture the dependences within a highly constrained number of code bits, as otherwise the overhead of writing/reading the code words would be comparable to the overhead of writing/reading an explicit synchronization variable. Previous studies [85] have shown that most applications display highly similar and static communication patterns in that each thread regularly communicates with a small and fixed subset of the rest

of the threads. More crucially, a large portion of the communication is performed via point-to-point communication, that is, consistent communication between two processors. This property of restrictive communication patterns enables the design of a highly effective encoding mechanism. In light of this observation, we propose a novel ***reference coloring algorithm***, which can encode global dependence information in arbitrary access contexts within only a **2-bit** overhead for each memory access in a point-to-point communication.

When two PEs access the same location in the shared SU, due to unpredictable run-time events, such as cache misses in the data subsystem and branch execution in the control subsystem, they may access the shared data out of order. Semantically correct communication necessitates obviously in-order execution of all these *globally dependent* instructions. In general, a pair of globally dependent memory access instructions consists of either a store and a load *(RAW dependence)*, or a load and a store *(WAR dependence)*, or a store and a store *(WAW dependence)*. Considered from the aspect of inter-PE communications, a *RAW dependence* ensures that the read operation in each communication obtains the correct data, while a *WAR dependence* ensures that the data of an incomplete communication will not be overwritten by a write operation in a subsequent communication. A *WAW dependence* between consecutive memory accesses, on the other hand, implies a redundant usage of the shared memory location, as the value stored by the first write operation is not consumed by any read operation.

Redundant accesses to the same location in a shared SU may cause threads to unnecessarily wait on each other, thus necessitating their elimination to avoid performance degradation. The absence of redundant usage of globally shared memory locations is presumed in the proposed synchronization framework, a task easily achieved by standard compiler techniques. In sum, two types of access patterns can be classified as redundant usage of a global shared memory location:

- **Two consecutive *store* instructions**. As the value stored by the first *store* is not consumed by any *load* instruction, the first *store* is redundant.
- **Multiple *load* instructions from the same PE that depend on the same *store*.** As the first *load* instruction will load the data in communication into a local register of that PE, subsequent *load* instructions emanating from the same PE are redundant.



(a) Access sequence to CR1

(b) Encoding of RAW and WAR I–colors

Figure 5.6: Encoding of point-to-point inter-PE communications

Static encoding of global access dependence

Once the two redundant cases have been eliminated, dependent tasks executed on different PEs will communicate by using the store/load instructions in an **alternating order** to access a shared storage location. This property can be observed clearly in Figure 5.6a, which presents an instruction sequence executed on PEs *P I* and *P II* to access the memory location $MEM[r10]$. More crucially, this highly regular access pattern enables a highly efficient encoding technique to preserve the data dependence information.

Figure 5.6b presents the incorporation of the proposed static encoding technique into standard data transfer instructions.[2] As can be seen, our static encoding technique uses one bit to distinguish global load/store instructions, together with two additional bits to encode global *RAW* and *WAR* dependences.

To preserve semantic correctness of a *RAW dependence*, a read operation (e.g. *Inst 4* in Figure 5.6a) should be blocked if it attempts to access the data earlier than the corresponding producer (e.g. *Inst 3* in Figure 5.6a). This can be achieved through forcing each producer to write a distinct “signature” together with the data in communication, and forcing each read to verify the proper signature before it obtains the data. One straightforward solution would consist of the explicit specification of the address of the corresponding producer in each read. However, the encoding overhead of this solution is

[2]The format of the data transfer instructions shown in the figure is used by a wide range of embedded architectures [38], such as *ARM*, *Hitachi SuperH*, and *Mitsubishi M32R*.

nontrivial, as the instruction address typically incurs at least a 32-bit overhead. Furthermore, writing/reading a 32-bit signature at run-time may impose an overhead comparable to the write/read of an explicit synchronization variable.

We propose instead a more efficient encoding solution by exploiting the regularity of access patterns for the shared data. More specifically, because only two PEs are involved in each point-to-point communication, as long as two adjacent write operations can be differentiated, RAW violations can be precluded. Accordingly, we propose a reference coloring algorithm which alternatingly uses two *RAW I-colors* during the static compilation process to make sure adjacent write operations have distinct *RAW I-colors*. This property can be observed by examining the behavior of *Inst 1*, *Inst 3* and *Inst 5* in Figure 5.6b. During execution, each write operation will write its *RAW I-color* together with the data in communication, enabling each read operation to check the RAW color to ensure the completion of the execution of its producer. Accordingly, each read operation is assigned the same *RAW I-color* as its producer, as can be observed from *Inst 2* and *Inst 4* in Figure 5.6b.

The preservation of *WAR dependences* ensures that the data of an incomplete communication will not be overwritten by a write operation in a subsequent communication. As traditionally each producer may have more than one consumer, in general the preservation of *WAR dependences* encounters additional challenges in that a write operation cannot be performed until all the previous read operations, that is, the consumers of a previous producer, have been executed. However, for point-to-point communication each producer has a single corresponding consumer only, thus enabling a further reduction in the number of code bits needed. More specifically, WAR violations can be prevented in the same way as RAW violations, through the usage of two *WAR* ***I-colors***. The encoding results can be observed in Figure 5.6b, wherein *Inst 2* and *Inst 3* share the same *WAR* ***I-color***, which differs from the *WAR* ***I-color*** shared by *Inst 4* and *Inst 5*.

A pseudocode for a slight extension to the compiler in order to incorporate the necessary updates for generating the suggested I-colors can be undertaken as described in Algorithm 2. It can be easily seen that no more than two RAW and two WAR I-colors are needed, implying that a total of **two** bits, one RAW and one WAR I-color bit, suffice to encode all the dependences. These two bits, together with the bit that is used to indicate whether a memory access is involved in inter-processor communication, constitute the

only static encoding overhead of the proposed synchronization mechanism.

Dynamic checking and access blocking

The aforementioned reference coloring scheme explicitly encodes the dependence information between memory accesses involved in inter-PE communications at compile time. At runtime when a memory access instruction is executed, if the static encoded "inter-PE comm" bit is on, the PE will check the status of the data in communication using either the *RAW I-color* or the *WAR I-color*, based on which the execution flow is blocked if necessary to preclude a potential semantic violation.

Algorithm 2 Reference Coloring

1: **for** each memory location $M \in$ point-to-point communication **do**
2: order all the inter-PE communication accesses to M;
3: **for** $i = 1$ to $n - 1$ [$n =$ total inter-PE communication accesses to M] **do**
4: **if** two consecutive accesses i and $i + 1$ are reads, writes, or a write followed by a read emanating from the same PE **then**
5: prune them
6: **end if**
7: **end for**
8: **for all** the remaining write accesses to M **do**
9: assign one RAW I-color and one WAR I-color to each write, with the property that two adjacent writes have distinct RAW and WAR I-colors;
10: **end for**
11: **for all** the remaining read accesses to M **do**
12: color each read using the same RAW I-color as the write immediately *preceding* it, and the same WAR I-color as the write just *following* it.
13: **end for**
14: **end for**

The hardware implementation of the dynamic checking mechanism is presented in Figure 5.7, with Figure 5.7a presenting the extra hardware added to record the status of the data in communication, and Figure 5.7b presenting the corresponding control state diagram. As can be seen, two extra bits, denoted as the *RAW **R-color*** and the *WAR **R-***

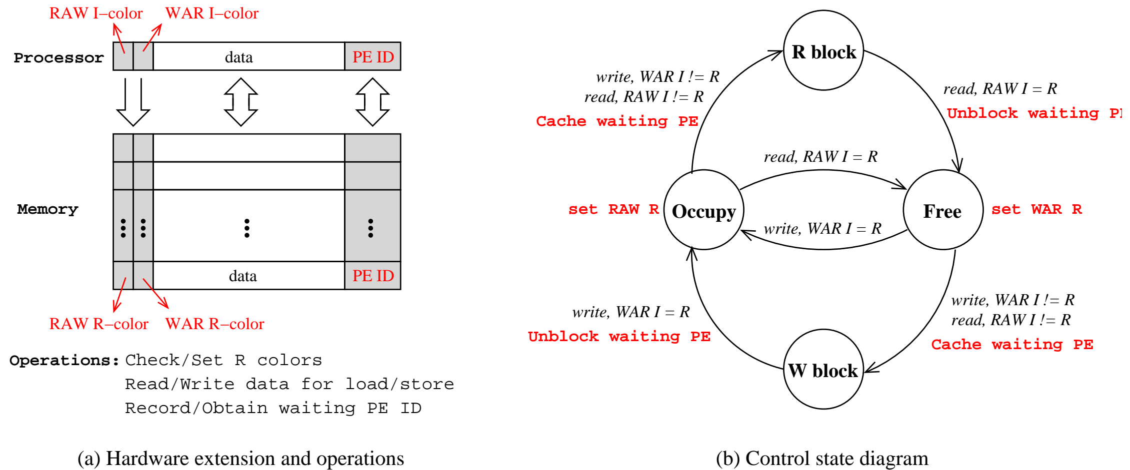


Figure 5.7: Implementation of the encoding-based synchronization scheme

color bits, are added to record the status of the data in communication. This can be clearly seen in Figure 5.7. Moreover, in order to eliminate a continuous polling of the **R-color** bits, a *PEID* field is also added to record whether a PE is waiting to access the data in communication, thus enabling a light-weight mechanism to await a blocking PE. As only two PEs are involved in each point-to-point communication, at most one PE needs to be blocked, implying that one *PEID* field suffices.

In the process of executing a global load/store instruction, two synchronization functions need to be performed in order to record the status of the data in communication: the **checking** and the **setting** of the *R-colors*. Furthermore, if an instruction attempts to access the data in a semantically incorrect order, two extra synchronization functions need to be performed: the **blocking** and the **unblocking** of the specific instruction. The following two cases delineate the detailed functions performed when executing load and store instructions, respectively.

- **Load instruction:** Before reading the data, the PE checks if the *RAW* ***R-color*** bit has the same color as the *RAW* ***I-color*** statically encoded in the load instruction. If so, the instruction can proceed to execution. Otherwise the instruction needs to be stalled, and the PE's ID will be recorded in the *PEID* field. The blocking of the load continues until a subsequent store instruction has updated the *RAW* ***R-color***.

  Once the execution of the load instruction has been completed, the PE sets the *WAR* ***R-color*** bit of the memory location to the *WAR* ***I-color*** encoded in the load

instruction. Furthermore, if the *PEID* field shows that a store instruction emanating from the alternative PE is waiting to update the data, the store will be unblocked.

- **Store instruction:** Before reading the data, the PE checks if the *WAR* ***R-color*** bit has the same color as the *WAR* ***I-color*** statically encoded in the store instruction. If so, the instruction can proceed to execution. Otherwise the instruction needs to be stalled, and the PE's ID will be recorded in the *PEID* field. The blocking of the store continues until a subsequent load instruction has updated the *WAR* ***R-color***.

  Once the execution of the store instruction has been completed, the PE sets the *RAW* ***R-color*** bit of the memory location to the *RAW* ***I-color*** encoded in the store instruction. Furthermore, if the *PEID* field shows that a load instruction emanating from the alternative PE is waiting to obtain the data, the load will be unblocked.

The above analysis clearly indicates that the execution of load and store instructions is symmetric. This property can also be clearly observed in Figure 5.7b, which presents the control state diagram in implementing the dynamic checking mechanism. As can be seen, the control of synchronization is composed of two pairs of symmetric states: the **Occupy** and the **Free** stages, as well as the **R block** and the **W block** stages.

The **Occupy** stage indicates that the shared memory location has just been updated by a write operation, implying that the access expected next is a read. If the expected read (identified through comparing the encoded RAW *I-color* with the RAW *R-color* stored with the data) arrives next, the control state will advance to the **Free** state, implying that the current communication has terminated and the operation expected next is the write operation of a subsequent communication. On the other hand, if the current state is the **Occupy** while the next access is not the expected read operation, the control state will advance to the **R block** state, and the blocked PE will be recorded in the *PEID* field. The PE is unblocked only when the expected read arrives, resulting in the control state advancing from the **R block** to the **Free** state.

The analysis presented above examines three transitions presented in Figure 5.7b: transitions from the **Occupy** to the **Free**, from the **Occupy** to the **R block**, and from the **R block** to the **Free**. The remaining three transitions are performed in an analogous manner because of the symmetric property.

To illustrate the dynamic execution of these four functions more concretely, let

Table 5.1: The dynamic check/set of the *R-colors* for communication synchronization

Step	Instrction	Operation	Control state	R-color RAW	R-color WAR
1	*Inst 1* (W)	Check *WAR R-color* = blue?	Free	Purple	**Blue**
2	*Inst 1*	Set *RAW R-color* ⇐ red	Occupy	**Red**	Blue
3	*Inst 2* (R)	Check *RAW R-color* = red?	Occupy	Red	Blue
4	*Inst 2*	Set *WAR R-color* ⇐ green	Free	Red	**Green**
5	*Inst 4* (R)	Check *RAW R-color* = purple?	Free	Red	Green
6	*Inst 4*	Block, record PEID ⇐ *P II*	W block	Red	Green
7	*Inst 3* (W)	Check *WAR R-color* = green?	W block	Red	Green
8	*Inst 3*	Read PEID, unblock *P II*	W block	Red	Green
9	*Inst 3*	Set *RAW R-color* ⇐ purple	Occupy	**Purple**	Green
10	*Inst 4* (R)	Check *RAW R-color* = purple?	Occupy	Purple	Green
11	*Inst 4*	Set *WAR R-color* ⇐ blue	Occupy	Purple	**Blue**

us consider the example presented in Figure 5.6a once more. Assume that the two *RAW* ***I-colors*** used by the reference coloring algorithm are **red** and **purple**, while the two *WAR* ***I-colors*** are **green** and **blue**. Accordingly, the reference coloring algorithm executed during the compilation process will encode the *RAW* and *WAR* ***I-colors*** of the first four instructions as presented in Table 5.1. An illustrative case can be examined if we assume that an unpredictable cache miss in *P I* occurred after *Inst 1* has caused the first four instructions to have a dynamic access order of (1, 2, 4, 3) to the location $MEM(r10)$. The operation performed and the status of the *RAW* and *WAR* ***R-color*** bits are presented[3] in Table 5.1. As can be seen, although instructions try to access $MEM(r10)$ in the semantically incorrect order of (1, 2, 4, 3), they are in actuality forced to be executed in the semantically correct order of (1, 2, 3, 4) with the help of the statically encoded *RAW* and *WAR I-colors*.

5.5 Experimental Evaluation

5.5.1 Impact of Topology on Task Scheduling

To evaluate the efficacy of the proposed locally shareable model, the various topology instances presented in Figure 5.3 are modeled. The algorithm outlined in Section 4.5

[3]Please note that Step 10 is a replay of step 5 since the blocking condition has been cleared through the execution of step 8.

Table 5.2: Impact of MPSoC topology on schedule length

			Schedule length			
Benchmark	Task #	Baseline	TP1-1	TP1-2	TP2-1	TP2-2
In-tree	63	11	0.73	0.64	0.64	0.64
Out-tree	63	11	0.73	0.64	0.64	0.64
Gaussian	100	37	0.51	0.51	0.51	0.51
FFT	95	17	0.82	0.71	0.65	0.59
Fork-join	45	17	1.00	1.00	0.88	0.88
LU	77	27	0.89	0.81	0.81	0.81
Laplace	75	25	0.88	0.80	0.80	0.80

has been employed for generating adaptive static schedules. The application set under test is composed of typically parallel algorithms, such as *LU decomposition*, *Laplace equation solver*, and *Gaussian elimination*. DAG representations of these task graphs are shown in Figure 4.17.

To illustrate the impact of MPSoC topology on task scheduling, the selected benchmarks are scheduled onto each of the 4 topologies. An MPSoC with a traditional distributed memory model is considered as the *baseline*. The obtained results are presented in Table 5.2, wherein we report the number of tasks in each application, the length of the baseline schedule, as well as the length of the adaptive schedule (normalized to the schedule length of the baseline MPSoC). Herein, "TP s-k" denotes a topology with sharing degree s and merging degree k. For all the four topologies, the amount of reconfiguration-induced data movement is consistently 0.

The experimental results show that the top 4 benchmarks can be effectively accelerated by the proposed locally shareable memory model. The fundamental reason for this significant improvement is that all the four applications display a large amount of parallelism and a limited number of out-going communications per task, implying that most of the inter-task communications can be performed through the direct communication links. To illustrate this property, we additionally report, in Table 5.3, the total number of *active links* in the topology that each benchmark can utilize. The results show that the first four benchmarks can utilize more than 60% of the direct communication links, and hence the schedule lengths have been reduced by 37% on average. The *in-tree* and the *out-tree* exhibit identical results due to the high similarity in their task graphs. In comparison, in the bottom 3 benchmarks, a task may fork a large number of dependent tasks that can-

Table 5.3: Impact of MPSoC topology on task mapping

	Active links				*SU over-utilization*			
Benchmark	TP1-1	TP1-2	TP2-1	TP2-2	TP1-1	TP1-2	TP2-1	TP2-2
In-tree	14	23	23	23	5	3	0	0
Out-tree	14	23	23	23	5	3	0	0
Gaussian	18	14	14	13	9	9	0	0
FFT	16	23	23	23	11	5	0	0
Fork-join	0	0	0	0	1	1	0	0
LU	4	6	6	6	3	3	0	0
Laplace	4	6	6	6	2	2	0	0

not simultaneously be placed on direct neighbors of the corresponding PE. The schedule length is thus constrained by the longest remote communication. The results show these benchmarks can only utilize less than 10% of direct communication links, and the average reduction of schedule length is around 14%.

The values reported in Table 5.2 confirm that the proposed locally shareable topologies offer multiple design points for the designer to trade off between the schedule length and the number of direct communication links, as a reduction in the former can generally be attained through increasing the value of the latter. On the other hand, it is not cost-effective if a negligible reduction in the scheduling length requires a significant number of communication links. According to this criterion, TP 2-2 can be considered as the most appropriate topology for *FFT* and *Fork-join*, while TP 1-2 the most appropriate topology for the remaining 5 applications.

By default the number of ports per SU equals the merging degree of the topology, as shown in Equation (5.2c). However, the data placement decisions, made for eliminating reconfiguration-induced data movement, may result in over-utilization of the memory throughput. Several tasks that are executed at the same timing step may end up being placed into a single SU. This conflicting usage of the memory bandwidth can be observed in Figure 5.5a, wherein Tasks 1 and 2 are both placed in the same SU and executed at the same timing step in Figure 5.1a.

The total occurrences of SU over-utilizations for each benchmark have been reported in Table 5.3 in the last group of columns. A relatively larger amount of over-utilization can be observed in the top 4 benchmarks, as their schedules are more dense due to the relatively higher amount of parallelism. More importantly, the results show

that SU over-utilization can be completely eliminated by increasing the sharing degree, that is, inserting more communication links into the topology. This is because the additional communication links create flexibility in making data placement decisions, which in turn enables two concurrent tasks to be separated into distinct SUs.

5.5.2 Efficiency of Encoding-based Synchronization

We evaluate the proposed light-weight synchronization mechanism by theoretically comparing the number of memory accesses necessitated in the proposed synchronization to the number of memory accesses necessitated in conventional spin-lock and barrier synchronization schemes.

Theoretical comparison

For each of the three synchronization schemes two cases are examined: the case wherein the producer thread updates the data earlier than the consumer thread, and the case wherein the consumer thread needs to wait for the producer thread.

1. Spin-lock based point-to-point communication. In a point-to-point communication, one pair of memory accesses are performed to obtain the data in communication, while another pair of memory accesses are performed to set and read *flag* if the communication has been statically scheduled appropriately so that the read operation does not need to wait for the write of data. If the producer and the consumer threads compete to access *flag*, however, the analysis in [38] shows that a total of $2i + 1$ bus transactions are needed for the ith thread to set and read the *flag*. In this case, the total number of memory accesses (T_{all}) needed to perform a spin-lock based communication among n threads is:

$$T_{all} = 2 + \sum_{i=1}^{n}(2i + 1) = 2 + n^2 + 2n \tag{5.3}$$

This equation clearly shows that for a spin-lock based point-to-point communication between two PEs, the total number of memory accesses needed is 8.

2. Barrier based point-to-point communication. A typical implementation of a barrier can be done with two spin locks: one to protect a counter that tallies the processes arriving at the barrier and one to hold the processes until the last process arrives at the

barrier. According to the analysis in [38], the ith thread needs to perform $3i + 4$ bus transactions, while the last process to reach the barrier requires one less. Thus, for a communication involving n threads, the total number of memory accesses (T_{all}) is:

$$T_{all} = 2 + \sum_{i=1}^{n}(3i + 4) - 1 = 1 + \frac{3n^2 + 11n}{2} \tag{5.4}$$

This equation clearly shows that for a barrier based point-to-point communication between two PEs, the total number of memory accesses needed is 18. As can be seen, since the barrier synchronization scheme is developed for globally synchronizing multiple threads at a time, it is not efficient for a point-to-point synchronization.

3. Proposed light-weight synchronization. The discussion presented in the last section clearly shows that the memory accesses for synchronization are combined together with the memory accesses for data transmission. Accordingly, if the communication has been statically scheduled appropriately so that the consumer thread does not need to wait for the producer thread, only a total of 2 memory accesses need to be performed. If an unpredictable run-time event causes the producer thread to be delayed, the consumer will be blocked once and unblocked later, with no need to spin on a shared variable. Consequently, the total number of memory accesses (T_{all}) needed to perform an encoding-based point-to-point communication is:

$$T_{all} = \begin{cases} 2 & \text{if producer arrives earlier than consumer,} \\ 3 & \text{otherwise.} \end{cases} \tag{5.5}$$

Simulation results

To illustrate the performance improvement provided by the proposed synchronization method more clearly, we randomly generate a sequence of 1000 point-to-point communications, of which the average communication latency is computed for each synchronization scheme.

In general, the communication latency is a function of the memory access latency, the total number of memory accesses involved in communication, as well as the number of extra cycles spent in waiting for the consumer thread. In our experimental framework,

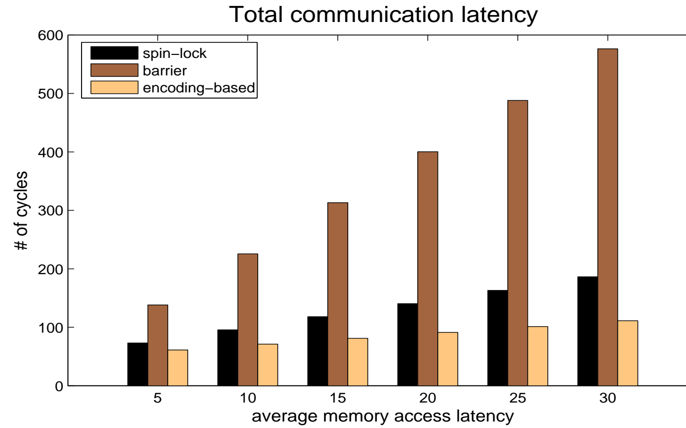


Figure 5.8: Total communication latency, assuming the average number of extra cycles spent in waiting for the consumer thread of 50

the memory access latency is varied from 5 to 30 cycles, while the average number of extra cycles spent in waiting for the consumer thread is varied from 0 to 50. The results are plotted in Figures 5.8 and 5.9. As can be seen, the proposed synchronization scheme outperforms both the spin-lock and the barrier synchronization schemes in reducing communication overhead. This is because the proposed encoding-based synchronization scheme significantly reduces the number of memory accesses needed in point-to-point communications. As all the memory accesses involved in synchronization and communication need to be serialized through sequential bus transactions which require tens of cycles, the significant reduction in the number of memory accesses directly implies a significant performance improvement enabled by the proposed encoding-based synchronization scheme.

5.6 Conclusions

As computational resources may increasingly become unavailable at runtime, a fast and predictable *execution reconfiguration* step is necessitated upon a resource variation, which in turn requires the development of advanced MPSoC topologies that can effectively hide task migration overhead. To attain this goal, we have proposed a locally shareable storage organization for adaptive multicore platforms. Through making each storage unit directly accessible to a set of adjacent PEs, tasks can be directly migrated among these PEs without data movement. As such a local sharing property is indepen-

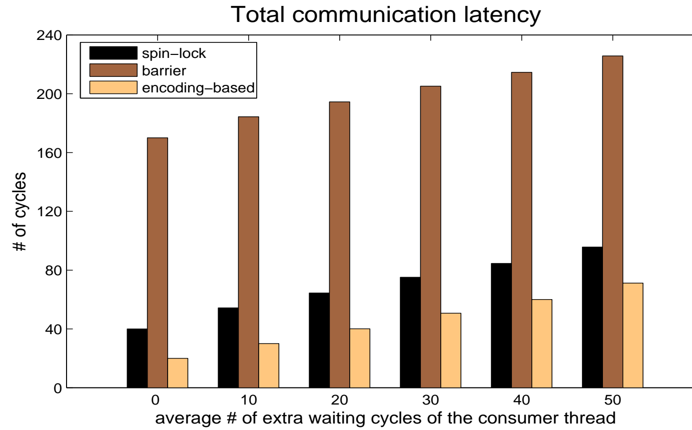


Figure 5.9: Total communication latency, assuming an average memory access latency of 10 cycles

dent of a particular topological structure, a set of fault tolerant MPSoC topologies have furthermore been proposed. Such topological structures can be adopted as fixed-silicon but dynamically reprogrammable MPSoC platforms, wherein decisions regarding topology selection and task placement can be made according to parallelism characteristics of the application and reconfiguration requirements of the system. The experimental results confirm that the proposed MPSoC topologies can even halve the execution time of parallel applications, while the reconfiguration-induced data movements between adjacent PEs can be completely eliminated.

The proposed MPSoC organization in turn enables the development of a light-weight distributed synchronization scheme to accelerate communications between adjacent PEs. Rather than employing a generic solution that allows any producer to send data to any consumer, we have developed a cost-efficient solution that differentiates neighborhood-centered communications from long-distance communications and accelerates the former. The synergistic collaboration between the compiler, responsible for statically identifying and encoding global data dependences between memory accesses involved in inter-PE communications, and the hardware extension of the conventional storage organization provide a novel synchronization framework. This light-weight synchronization mechanism allows dependent threads to frequently exchange data during execution, in turn enabling the exploration of fine-grained parallelism for applications with strong dependences.

The text of Chapter 5, is in part a reprint of the material as it appears in *C. Yang and A. Orailoglu, "Light-weight Synchronization for Inter-processor Communication Acceleration on Embedded MPSoCs," International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), October 2007*. The dissertation author was the primary researcher and author of the publication [93].

Chapter 6

Architectural-level Fault Resilience

In the face of the projected high elevation of fault rates, *fault resilience* needs to be considered as a primary design constraint, especially for systems dedicated to mission critical applications, such as server, defense, or medical applications. The adaptive static schedules discussed in Chapter 4 contribute to multicore reliability by delivering predictable execution reconfigurability upon core failures. Yet to attain full fault resilience, such schedules still need to be supported by an efficient fault detection mechanism.

As mentioned in Section 1.2, the development of an efficient fault detection scheme for future multicore systems imposes three *challenges*, namely, attaining full detection capability within a minimum level of result comparison and hardware duplication, maximally relaxing checking-induced synchronization conditions with no reliance on any centralized hardware buffer, and minimizing checkpointing overhead through strictly protecting memory against execution faults. Yet the review of the state-of-art in Section 2.2 indicates software-based techniques [76], although they can extract program information such as execution invariants and the range of execution results to quickly identify a certain set of faults, are insufficient for providing full fault coverage. On the other hand, traditional duplication-based fault detection and recovery approaches, although they provide high fault coverage, impose significant overhead either in checkpointing execution results, or in constantly synchronizing two threads for value checking. To make these approaches suitable for future multicore systems, further overhead reduction is still necessitated.

In this chapter, we first examine the fundamental causes of the limitations of traditional duplication-based solutions, and then present a technique that exploits the idea of

comparing and checkpointing at the *cache-memory* interface to attain light-weight fault detection and checkpointing. Subsequently, a set of performance optimizations, as well as the technical support for incorporating the proposed fault detection into a multicore platform are discussed in Sections 6.3, and 6.4, respectively.

6.1 Full Resilience within Low Overhead

Given the projected high fault rate, the fundamental challenge in developing a cost-effective fault tolerant multicore system is to minimize the overhead of fault detection and recovery simultaneously, as optimizing only one side of the problem may result in the complexity being shifted to the other. More crucially, due to the diverse behavior of fault manifestation, full fault coverage is still necessitated. Accordingly, overhead reduction should not be attained through partial redundancy techniques [32, 71, 88] that reduce duplication overhead at the cost of significantly increased rates of undetectable faults.

To attain full fault resilience, traditional approaches typically employ a *duplicate-compare* strategy to detect faults, as well as a *checkpoint-rollback* strategy to restore the computation to a previously saved clean *checkpoint* (composed of the processor state and the corresponding memory footprint). Yet one fundamental obstacle to the adoption of such techniques has been the associated high comparison/checkpointing cost. More specifically, traditional duplication-based fault tolerance techniques perform the comparison and checkpointing process either at the *task-level* or at the *instruction-level*. In the former case, a task needs to be duplicated on distinct cores, and each duplicated copy needs distinct memory regions. The operating system is typically involved in comparing and checkpointing all the modified memory pages, thus imposing significant overhead. In contrast, in the latter case, memory is prevented from being polluted by execution faults, yet instruction results cannot be committed until their correctness has been verified, as shown in Figure 6.1a. As discussed in Section 2.2, this highly synchronized execution model significantly increases the latency of a single instruction, thus delaying the release of hardware resources. The synchronization requirements can be relaxed, however, at the cost of duplicating each load/store value of the leading thread in extra centralized buffers [62, 72] for the trailing thread to access, as shown in Figure 6.1b.

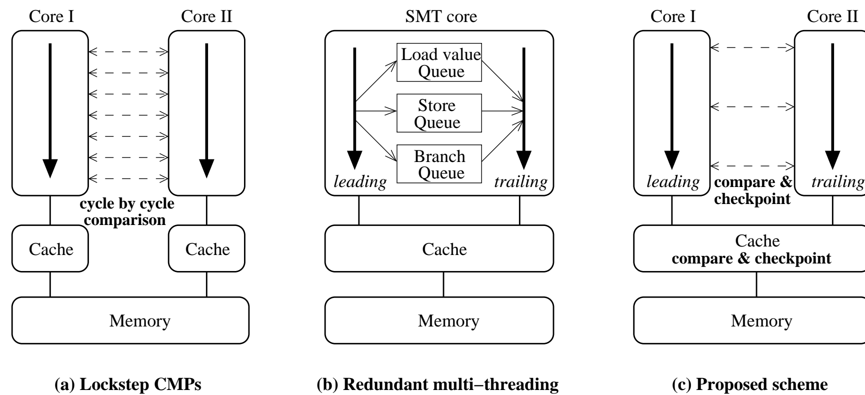


Figure 6.1: Differences between lockstep CMP, redundant multi-threading, and the proposed cache-based detection/checkpointing scheme

The examination presented above indicates that a cost-effective fault tolerance scheme needs to attain full detection capability in a *loosely-synchronized* manner wherein two redundant threads can be executed independently, however, with no reliance on sizable hardware buffers. An architectural examination indicates that *caches*, which serve as temporary storage for the main memory, can possibly be utilized to temporarily hold unconfirmed execution results for fault detection purposes. By sharing a single data cache between two redundant threads, one thread can directly check the execution results of the other, thus completely eliminating the necessity of dedicated hardware queues to capture load and store values. In this way, one thread can end up running ahead of the other in execution, in turn effectively relaxing the execution synchronization requirements.

In an execution environment displaying elevated fault rates, checkpoints need to be established more frequently so as to reduce the amount of computation to be rolled back upon a fault. Yet to prevent an application from spending most of its time and energy taking checkpoints, checkpointing overhead should be strictly controlled. As discussed before, significant complexity and overhead will be incurred in the checkpointing process, if unconfirmed data are allowed to be written into the main memory. Accordingly, a light-weight checkpointing scheme should strictly protect memory from being polluted by execution faults, thus motivating the proposal of checkpointing at the *cache-memory* interface. As shown in Figure 6.1c, while the cache holds unconfirmed results of the two threads, these results are written to the lower level storage in the memory hierarchy only when the two threads agree. In this way, a checkpoint only needs to be established when

a dirty cache line needs to be replaced, and only the *processor state*, i.e., the program counter and the register values,[1] needs to be checkpointed.

6.2 Cache-based Fault Tolerance

In the face of diverse behavior of fault manifestation, the proposed fault detection scheme duplicates a task into two thread copies that are simultaneously executed on **different** cores to detect not only transient, but also intermittent and permanent faults. Meanwhile, storage structures such as caches, register files, and the main memory are protected using ECC, while buses are presumed to be protected using parity. This safe storage is therefore utilized to store the checkpoints, so that execution can be recovered to a clean state upon the detection of any computation fault.

In the remaining parts of this section, we discuss various aspects of the proposed cache-based fault detection and recovery framework, including thread execution, fault detection, checkpointing, as well as execution recovery.

6.2.1 Run-ahead Property for Workload Balance

As two threads generate identical memory access patterns in the fault free case, a single data cache can be shared between them to achieve more efficient resource utilization. This organization furthermore enables the attainment of a more balanced workload, through ensuring that the ***L**eading* thread never falls behind the ***T**railing* thread. The L thread brings data into cache upon misses and initiates a checkpointing request upon the replacement of a dirty cache line, while the T thread manages fault detection through reading and comparing the values written by the L thread. Only the L thread encounters misses in the shared cache, while only the T thread needs to compare store values, thus effectively balancing the workload.

A noteworthy aspect of the outlined run-ahead requirement is that it is not imposed on a cycle-by-cycle basis, but only for memory access instructions. In other words, the two threads can execute non-memory access instructions *independently*, yet before executing a load/store, the T thread needs to ensure that the instruction has already been

[1] Some branch handling and exception taking techniques may necessitate a few special purpose registers to be additionally saved.

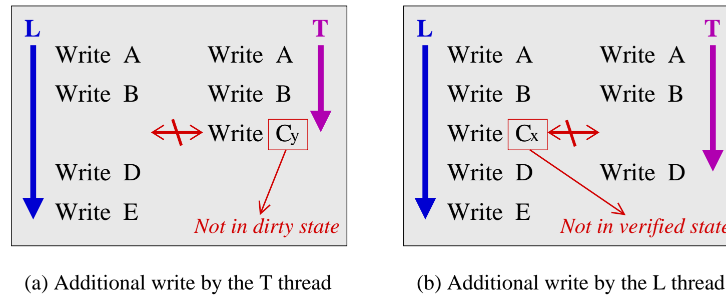
executed by the L thread. This requirement can be fulfilled through the use of an *access counter* to globally track the difference in the memory access counts; the counter value is incremented whenever the L thread executes a load/store, and decremented whenever the T thread executes a load/store. The execution of the T thread is stalled if the value of the counter is 0, thus fulfilling the run-ahead requirement.

6.2.2 Fault Detection

The proposed fault detection scheme consists of two parts: *store* verification and *register* verification. The sharing of a single cache enables the T thread to directly check the store values produced by the L thread with no need of any dedicated hardware queues. During execution, the L thread directly writes its results into cache, while each write initiated by the T thread is changed into a *read* of the corresponding cache block and a *comparison* of the two values. Meanwhile, the two threads also record their register values *individually* at each checkpoint, thus enabling a comparison of the two processor states to detect execution faults. The combination of the *store* verification and the *register* verification therefore ensures that any execution fault, if it has not been masked during computation, will be detected eventually.

Traditionally a valid cache block can either be '*clean*' or '*dirty*', depending on whether its value has been updated or not. To support fault detection, an extra '*verified*' state is maintained so as to differentiate whether or not the data in cache has been verified by the T thread. A store initiated by the L thread would therefore make a cache block *dirty*, while the same store later initiated by the T thread would make the cache block *verified*, if the two store values match.

With this extra state, any mismatch in a pair of store values can be directly detected through the aforementioned cache access strategy. Meanwhile, execution faults that propagate through dependence chains to store *addresses* can also be indirectly detected. If an execution fault causes a store address to change from C_x to C_y, both cache blocks (C_x and C_y) would exhibit a mismatch in the number of store instructions, as shown in Figure 6.2. This mismatch typically would cause either cache block to enter a state contradicting with the run-ahead property, which can be monitored by the cache controller. In sum, a "fault-detected" signal will be generated for any of the following



(a) Additional write by the T thread

(b) Additional write by the L thread

Figure 6.2: Inconsistent access pattern caused by faults in store addresses

three types of fault observations:

- **Disagreeing register values:** When the T thread reaches the checkpoint, its register values do not match the register values recorded by the L thread.
- **Disagreeing store values:** When the T thread is about to write a *dirty* cache block, the value to be written does not match the value in the block produced by the L thread.
- **Inconsistent store sequences:** As the L thread always runs ahead of the T thread, two cases will indicate the existence of a mismatch in store sequences: 1) when the T thread is about to execute a store, it misses in the cache, or the corresponding cache block is not in the '*dirty*' state, indicating that the L thread has not written to that block yet (Figure 6.2a). 2) When the T thread reaches the checkpoint, there exists a '*dirty*' block in the cache, indicating that the T thread has not verified the data written by the L thread (Figure 6.2b).

These three cases clearly confirm that the proposed technique can detect unmasked execution faults that propagate through a dependence chain to either store values, or store addresses, or register values at a checkpoint. Masked faults, on the other hand, would not affect the correctness of the computation. Such a full detection capability in turn enables the design of a light-weight checkpointing and rollback scheme.

6.2.3 Execution Checkpointing

Upon the detection of a fault, the computation needs to be restored to a previously saved clean state. A *checkpoint*, which records complete information about the computa-

tion state, typically consists of processor state and the corresponding memory footprint. To reduce checkpointing overhead, however, the proposed fault tolerance scheme maintains no extra copies for values written into memory. Instead, only the processor state is checkpointed whenever data needs to be written into memory, thus ensuring the consistency of the processor and the memory states.

In write-through caches, each store value needs to be written into memory, thus requiring the processor state to be recorded on every store instruction. This checkpointing frequency can become intolerably high. In contrast, the proposed fault tolerance scheme employs a write-back cache, implying that a checkpoint only needs to be established upon a write-back, that is, *whenever a dirty cache line is to be replaced*. Clearly, under this policy the checkpoint frequency is determined by the writeback frequency, which is in turn determined by the cache *size*, *associativity*, as well as *the replacement policy*. Yet given the low checkpointing overhead of the proposed scheme, the checkpointing frequency in high fault-rate systems can be adaptively scaled up so that upon a fault, less amount of computation needs to be rolled back. For instance, a checkpoint can be established upon the execution of K instructions, with the value of K determined by the projected fault rate.

Checkpointing request initiation

As the L thread always runs ahead of the T thread and as a cache block will not be replaced if it exhibits a pending access of the T thread, only the L thread will encounter cache misses. Checkpointing requests therefore will always be initiated by the L thread upon the replacement of a modified cache line. The processor state to be checkpointed is the *current* processor state of the L thread, that is, the computation point at which the checkpointing request is initiated. To establish a consistent checkpoint, not only the dirty cache line selected for replacement, but all the other dirty cache lines that have been updated since the last checkpoint need to be written into memory.

The aforementioned checkpointing initiation strategy can be illustrated more clearly by considering the loop example presented in Figure 6.3. In this loop, each array element $A[i]$ is first read at the $(i-1)^{th}$ iteration, and then written at the ith and $(i+1)^{th}$ iterations. To simplify the analysis, we assume that the cache is a directly mapped cache with 8 lines, with each line holding a single array element. As all the cache blocks are invalid at the

```
for (i=1; i<MAX; i++) {
   A[i] = A[i+1] + i;
   A[i-1] = 2*i;
}
```

Figure 6.3: Loop with cache block dependences

beginning of loop execution, the execution of the first six iterations ($i = 1$ to 6) causes the 8 cache lines to be filled with $A[0]$, ..., $A[7]$, respectively. At the 7th iteration, $A[8]$ needs to be brought into the cache and to replace $A[0]$. Since the value of $A[0]$ has been modified, this block needs to be written back into memory, engendering in turn a checkpointing request. The processor state to be checkpointed is the computation point when the L thread is about to replace $A[0]$ (at the 7th iteration). Accordingly, not only $A[0]$, but also the other dirty cache blocks, $A[1]$, ..., $A[6]$, should be written into the memory.

Checkpoint establishment

When the L thread reaches a checkpoint, the T thread is still in the process of verifying store values. To monitor whether the T thread has also reached that checkpoint, the proposed scheme tracks both the PC value and the difference between the number of memory accesses of the two threads. Specifically, if both threads have performed the same amount of memory accesses (i.e., the value of the *global access counter* is 0), the PC of the T thread is compared against the PC of the checkpoint. In the fault-free case, a match of the two PC values, indicating the arrival of the T thread at that checkpoint, can always be obtained. In contrast, if no match has been reported before the T thread issues a subsequent load/store, an execution fault will be reported.

Once the T thread also arrives at the checkpoint with no intervening error detection, a new checkpoint is established by saving the processor state into reliable storage (either a dedicated hardware buffer or a fixed location in main memory) and copying all *verified* cache lines into memory. However, this copying would generate a burst of memory requests that may appreciably degrade system performance. To overcome this issue, we employ the idea of making these lines *unchangeable*, originally proposed in [42] for single processors. Specifically, an extra '*retired*' state is maintained for each cache line, and all the '*verified*' cache lines are marked as '*retired*' once the T thread reaches the

checkpoint. The use of this extra state enables a distribution of the write back of *retired* blocks: during subsequent execution, a *retired* block is written back into memory upon a replacement of that block, or upon the first subsequent *write-hit* on that block.

6.2.4 Execution Recovery

As the memory is strictly protected against execution faults, the recovery process is highly efficient, attained through recovering only the processor state and invalidating only the cache lines that have been modified since the last checkpoint, namely, the *dirty* and *verified* blocks. In contrast, no invalidation is needed if a cache block is either in the *clean* state, indicating no update whatsoever, or in the *retired* state, indicating that the block was updated before the last checkpoint. Once this invalidation process is completed, the register values as well as the PC value saved at the last checkpoint can be reloaded so as to resume the execution of both threads.

The aforementioned strategy effectively rolls the execution back to a previously saved clean computation state, thus completely recovering a transient or intermittent fault if its fault duration has elapsed. On the other hand, if during the re-execution same fault occurs for a second time, an execution migration step is necessitated in the recovery process. The band-level reconfiguration approach outlined in Chapter 4, can be utilized to isolate both suspect cores. In this case, the proposed technique also effectively reduces the cost of task migration, which can be accomplished through migrating the processor state and committing *retired* cache blocks into the main memory. During subsequent execution, the two cores can be separately paired with healthy cores to achieve a complete fault identification. Upon a complete differentiation of the faulty and the fault-free core, the latter can be pulled back into execution through another band-level reconfiguration process.

6.2.5 Cache State Extension

The complete cache state diagram supporting fault detection, checkpointing and recovery is shown in Figure 6.4. Each cache block can be in any of five possible states: the three traditional states of *invalid*, *clean* and *dirty*, as well as the two extra states of *verified* and *retired*. The transitions among these five states accomplish **four** fundamental functions of the proposed fault tolerance scheme:

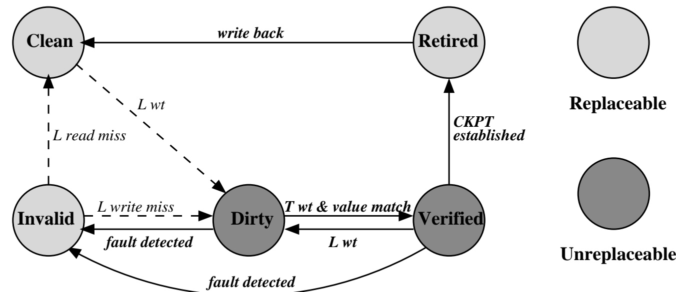


Figure 6.4: Cache states extended for fault detection and checkpointing

Fault detection: A store performed by the L thread will make the corresponding cache block *dirty*, while the same store performed by the T thread will make the cache block *verified* if the two store values match. A fault will be reported if the two store values differ.

Checkpoint initiation: The *invalid*, the *clean*, and the *retired* states are marked as **replaceable**, indicating a checkpoint-free replacement for any of these blocks. In contrast, if a *dirty* or a *verified* block is selected for replacement upon a cache miss, a new checkpointing request will be initiated by the L thread.

Checkpoint establishment: Once the T thread reaches a checkpoint and no fault has been reported, all the *verified* cache blocks will be marked as '*retired*'. During subsequent execution, a replacement or the first subsequent write-hit of a *retired* cache block requires the data to be written into memory, which in turn makes that block *clean*.

Execution rollback: Upon the detection of any fault, all *dirty* and *verified* cache blocks will be marked as *invalid*.

6.2.6 Requirements on Memory Access Order

Clearly, the aforementioned fault detection and checkpointing semantics can be naturally preserved if each core performs the memory accesses *non-speculatively* and *in-order*. Yet it turns out that these semantics can also be preserved in an execution environment with out-of-order memory accesses. In fact, the fault detection semantics only necessitate the stores to a single cache block to be performed in order. Clearly, this requirement is already obeyed by every processor, even if an out-of-order execution mechanism is employed. In comparison, as read operations do not pollute the cache

content, the proposed scheme forgoes the monitoring of the load values, and hence each core can perform loads *out-of-order*. Yet it is more desirable for each core to perform loads *non-speculatively* so that between two consecutive write operations to a single cache block, the total number of loads to the block remains identical.

6.3 Execution Asynchronicity Enhancement

While the proposed *dual-core-shared-cache* fault tolerance scheme effectively relaxes thread synchronization, the maximum run-ahead offset of the L thread and hence the attainable performance benefit is constrained by the synchronization requirement at each checkpoint as well as the cache-line dependences between the two threads. To relax these constraints, we outline in this section two performance enhancement techniques, namely, a *relaxed thread synchronization model*, as well as a cache block *selective split capability*.

6.3.1 Relaxed Thread Synchronization at Checkpoints

Because of the run-ahead execution strategy, once the L thread initiates a checkpointing request, it typically needs to await the T thread to establish the checkpoint. In a straightforward case, a new checkpoint can be established *synchronously* by forcing the L thread to await the completion of the fault detection process and the isochronism of the two threads. Both threads will simultaneously be allowed to proceed upon the completion of the checkpointing process.

The *strictly-synchronized* checkpointing strategy offers a benefit in that at any time only a single checkpoint needs to be maintained, however, at a cost of unnecessarily forcing the L thread to await the T thread to fully catch up. In contrast, if the T thread has verified the correctness of the data to be replaced and no more read accesses are pending, the L thread can proceed to replace the block. The data to be replaced, however, should be stored in a dedicated buffer rather than being written back into memory immediately. Meanwhile, the old checkpoint cannot be overwritten, since the new checkpoint has not been established yet.

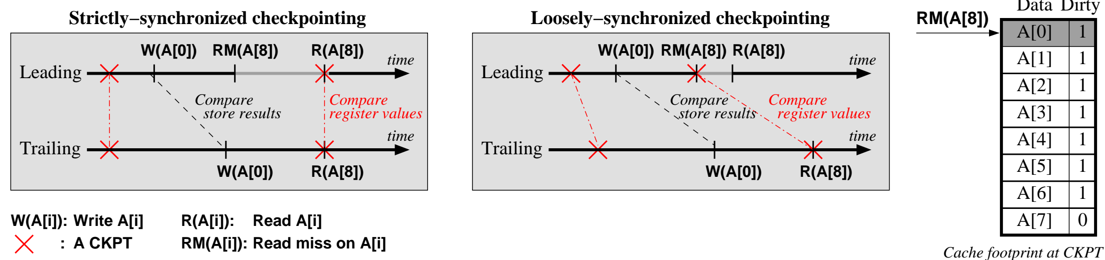


Figure 6.5: Strictly vs. loosely- synchronized checkpointing

The differences between the two synchronization schemes can be observed more clearly in Figure 6.5, wherein both strategies are applied to the loop example shown in Figure 6.3. A checkpointing request is initiated when the L thread encounters the read miss on $A[8]$ at the 7th iteration. In the strictly-synchronized checkpointing scheme, the L thread waits until the T thread also reaches this computation point, and then writes $A[0]$ into memory and brings $A[8]$ into the cache. In contrast, in the loosely-synchronized checkpointing scheme, the L thread only needs to await the T thread's verification of the correctness of $A[0]$, that is, the completion of the store instruction at the 1st iteration. Then, the value of $A[0]$ and the processor state of the L thread will be stored in safe storage, and the L thread will proceed to bring $A[8]$ into the cache.

In sum, the loosely-synchronized scheme reduces the waiting time of the L thread by allowing it to proceed beyond a pending checkpoint, yet necessitates extra storage to record the data to be replaced and the processor state of the L thread. More importantly, if the checkpoint is still pending to be established, the L thread should not be allowed to write to the cache, as doing so would either overwrite a *dirty* cache block or change a *clean/verified* block to *dirty*, both of which would cause a fault to be reported by the T thread. These two potential semantic violations constrain the L thread to proceed only in a restricted manner without performing any cache write operation, if the new checkpoint is still pending to be established.

6.3.2 Selective Split Capability of Cache Blocks

While sharing a single data cache between two duplicated threads delivers cost-effective fault detection and checkpointing, this sharing does create a critical issue of

pseudo cache block dependences, which may force the L thread to constantly await the T thread. Taking the loop presented in Figure 6.3 for example, as the L thread may run one iteration ahead of the T thread, two types of pseudo dependences may be created. The L thread writes $A[i]$ before the T thread reads the old value (a *WAR dependence*), thus causing the T thread to obtain an incorrect value, or overwrites $A[i-1]$ before the T thread checks the old value (a *WAW dependence*), thus causing the T thread to incorrectly report the detection of an error.

Although both types of *pseudo cache block dependences* can be preserved by forcing the L thread to await the T thread, this extra synchronization requirement may reduce the performance benefit that could be obtained by the *run-ahead* execution model. Another possible yet exceedingly inefficient solution (employed by redundant multithreading processors [72]) is to buffer each load/store value for the T thread to access. In contrast, our work aims at attaining execution asynchronicity within minimal hardware duplication. Instead of buffering all the load and store values, we propose to duplicate a cache block only upon the detection of a pseudo dependence, through incorporating a *selective split capability* into the cache design; whenever a block dependence is detected, the L thread is allowed to update the regular cache block, while the old value is placed in a victim cache for the T thread to read or to verify. The cache access controller is thus extended to incorporate **three** functions: *detecting* both types of cache block dependences, *splitting* a block into two versions, and *merging* the two versions if later the T thread catches up to the L thread in the execution progress.

Split Condition Detection

The fundamental issue encountered in implementing the *selective split capability* is the detection of a pending read or write of the T thread. The latter case can be easily detected through monitoring the block state: a *dirty* state indicates the existence of a pending write of the T thread. The detection of a pending read can be attained through a pure dynamic technique, through maintaining a *read counter* for each cache block. The counter value, initialized to **zero** upon a store by the L thread, is incremented upon a load by the L thread and decremented upon a load by the T thread. The L thread is only allowed to overwrite a cache block if the corresponding read counter is zero. If the counter is non-zero, however, a pending read of the T thread is detected.

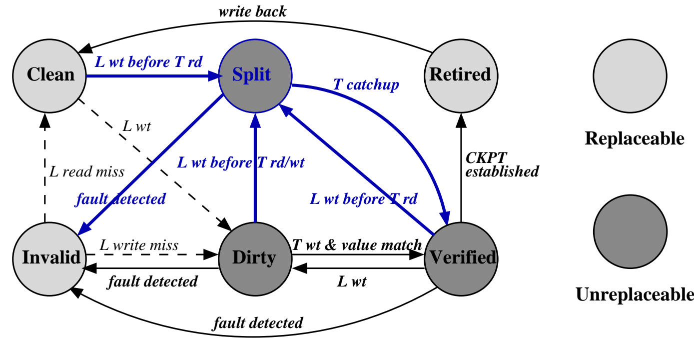


Figure 6.6: Adding a *split* state to cache state diagram

Cache Block Split and Accesses

Upon the detection of a pending read/write by the T thread, the cache block can be split into two if there exists a free entry in the victim cache: the old value will be saved in the victim cache for the T thread to access, while the L thread can proceed to overwrite the block in the regular cache. To differentiate such blocks, an extra *split* state is added to the cache state diagram, as shown in Figure 6.6. A write initiated by the L thread would cause a block to enter the *split* state if the current state is *dirty*, or if the current state is *clean* or *verified* yet the read count is nonzero. This state diagram exhibits no transition from the *retired* state to the *split* state, since a write-hit to a *retired* cache block requires the data to be first written into memory, thus making the block *clean*.

Once a cache block has been split into two, thread execution becomes *independent* in that each thread has its own place to write, and each thread would read the data written by itself. This independence furthermore implies that the L thread can issue **multiple** write operations ahead of the T thread to a single cache block, while only two versions of data need to be buffered. More precisely, if the number of outstanding write operations is k, the proposed technique only needs to buffer the 0th value for the T thread and the kth value for the L thread. The intermediate $k-1$ values, as they have been overwritten, are never written back into memory and hence need no comparisons for fault detection. Even if a fault in these $k-1$ values propagates through load instructions to subsequent computation, the fault will also propagate to subsequent store instructions if it is not masked during execution, and will eventually be detected once those store values are compared.

Merge Condition Detection

While the selective split capability can effectively enlarge the run-ahead offset of the L thread, it also increases the latency of the T thread, whose accesses to a split block need to be redirected to the victim cache. To reduce such overhead and hence reduce the required size of the victim cache, a victim cache block should be deallocated, if the T thread catches up to the L thread in execution.

Accomplishment of the merging capability requires the inter-thread execution offset to be monitored so as to determine whether two split data copies are produced by the same store of the two threads. A *version counter* is therefore added to each block in the *victim cache*. The counter value, initialized to **1** upon the split of a cache block, is incremented upon a store by the L thread and decremented upon a store by the T thread. If a store by the T thread changes the version counter value to **0**, the T thread will additionally trigger a comparison of the two split data copies. If the two values match, the entry in the victim cache will be deallocated, while the corresponding regular cache block will be set to the '*verified*' state, as shown in Figure 6.6. It needs to be noted that in this solution, two split blocks are only checked for merging possibility upon a *store* by the T thread, since read accesses would not alter the value of the version counter of a victim cache block.

Once two split blocks are merged together, the read counter value in the regular cache should be set to the difference between the read counts of the two threads. As merging is checked upon the completion of a store of the T thread, the read count of the corresponding victim cache block is always 0, implying that no *read counter* needs to be maintained for a victim cache block. Accordingly, for a *split* cache block, only the read count of the L thread needs to be maintained, and the read counter value remains constant during the merge process.

Block Split upon a Clean Replacement

So far, we have discussed the approach to split a cache block upon the detection of a *WAR* or *WAW cache block dependence*. Another situation that also necessitates the split of a cache block is when the L thread intends to replace a *clean* or *retired* block, yet the old value is still pending to be read by the T thread. Replacing a *clean* or *retired* block does not create a checkpoint request. However, if the L thread is allowed to replace the

data, a pending read by the T thread would encounter a cache miss, which is considered as an "inconsistent store sequence" case by the fault detection scheme discussed in Section 6.2.2.

To preclude this potential semantic violation, whenever the L thread selects a *clean* or *retired* cache block for replacement, the old value needs to be written into the victim cache if the read counter of that block is nonzero. The corresponding read count also needs to be written into the *version counter* field of the victim cache so that the T thread can decrement the read count upon subsequent read accesses, and the block can be freed once the counter becomes 0.

6.3.3 Synchronization Condition Analysis

The two execution asynchronicity enhancement techniques enable the two threads to be executed independently most of the time. In this process, the T thread is forced to await the L thread if and only if the value of the *access counter* is 0. The L thread, on the other hand, is forced to await the T thread in the following five cases:

1. The L thread tries to execute a load/store, while the *access counter* reaches its upper bound.
2. The L thread tries to read a cache block, while the corresponding *read counter* reaches its upper bound.
3. The L thread tries to write a cache block, while a checkpoint is pending to be established.
4. The L thread tries to split a cache block, while the victim cache is full.
5. The L thread tries to update a *split* cache block, while the corresponding *version counter* reaches its upper bound.

These five cases constrain the maximum run-ahead offset of the L thread. Nonetheless, except for the third one, the occurrence frequency of the remaining cases can be reduced by increasing the sizes of the counter and/or the victim cache. Yet this increase does not necessarily lead to an improvement in the average execution time of the two threads. While the conditions for blocking the L thread are relaxed, a larger counter and/or victim

cache also results in more cache blocks being split, thus causing the T thread to spend more cycles accessing the victim cache.

The five conditions for blocking the L thread contradict the condition for blocking the T thread, implying that **no** deadlock would occur in the fault-free case. However, in a faulty case a deadlock condition may occur due to a mismatch in memory access patterns. For instance, execution faults may cause the L thread to perform a number of extra read accesses and hence block the L thread upon a *read counter* reaching its upper bound. This blocking condition cannot be cleared by the T thread, as it would not perform these read accesses. Meanwhile, the T thread would also be blocked once the value of the global *access counter* reaches 0, thus creating a cyclic waiting condition. This condition, however, would never occur in the fault-free case, since the T thread is always able to unblock the L thread if the latter is blocked on any read counter. In sum, the proposed cache controller will report an error whenever it detects a cyclic waiting condition, in turn causing the execution to be rolled back to the most recent checkpoint.

6.4 Fault Tolerant MPSoC Organization

When applying the proposed cache-based fault tolerance scheme to a multicore platform with multiple applications executed concurrently, both the achievable performance and design complexity are highly influenced by the *multi-core, multi-thread* execution environment. In this section we specifically examine **three** issues, namely, the impact of memory hierarchy on the checkpointing strategy, the impact of inter-thread dependences on fault detection semantics, as well as the impact of multi-threading on the overall execution throughput.

6.4.1 Checkpointing Tradeoffs in Multi-level Cache Design

An important design decision for systems with a multi-level cache hierarchy is to determine the interface at which checkpoints should be established. If an MPSoC contains two levels of caches and both the L1 and the L2 caches can be shared between a pair of cores (such as in the Intel Hyperthreading architecture), checkpoints can be established at either the L1/L2 interface or the L2/memory interface. In the former case, only the L1

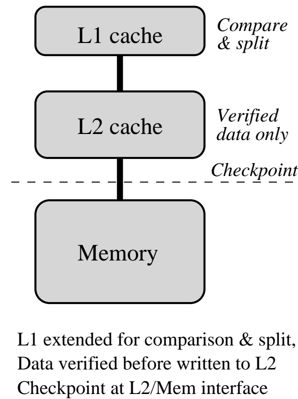


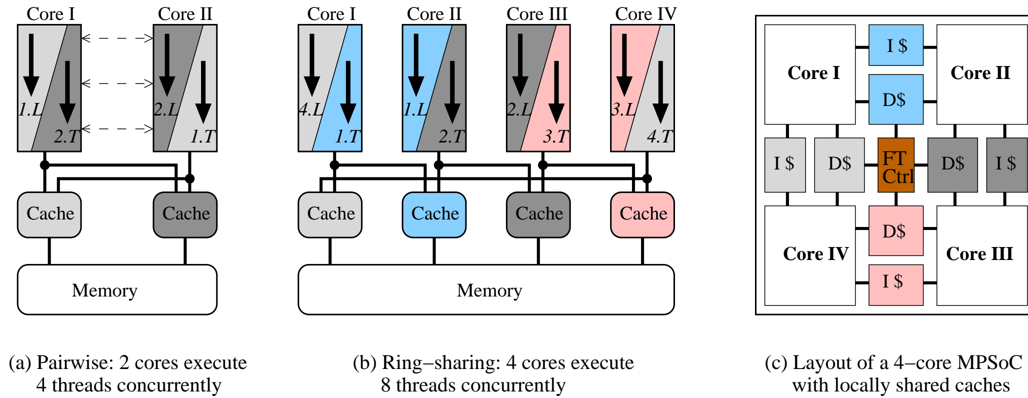
Figure 6.7: Hybrid detection and checkpointing policy in multi-level caches

cache design needs to be extended to include the extra states for fault detection, checkpointing, and block split. In contrast, if checkpointing is performed at the L2/memory interface, both the L1 and the L2 caches may contain dirty blocks when a checkpointing request is initiated. To verify all these blocks, result comparison and checkpointing need to be performed in both caches, which in turn requires an extension of both cache designs to include the extra states for fault detection, checkpointing, and block split.

While checkpointing at the L2/memory interface significantly reduces the checkpointing frequency and consequently the overall checkpointing overhead, it also increases the complexity of the L2 cache design. To reduce this overhead, we propose a *hybrid* fault detection and checkpointing solution, shown in Figure 6.7. Here, only *verified* data are allowed to be written into the L2 cache, while a checkpointing request is initiated when an L2 cache line is written to the memory. In this way, only the L1 cache needs to be extended to implement fault detection and block split. The L2 cache, which needs neither a victim cache nor any value comparison, only contains blocks of the **four** possible states, namely, the *invalid, clean, verified*, and *retired* states. This hybrid checkpointing strategy therefore can significantly reduce checkpointing frequency within a minimum hardware extension of the L2 cache design.

6.4.2 Checkpoint Coordination for Inter-thread Communications

When a set of dependent tasks are concurrently executed in a multicore platform, communication data need to be protected from being polluted by execution faults. If



(a) Pairwise: 2 cores execute 4 threads concurrently

(b) Ring–sharing: 4 cores execute 8 threads concurrently

(c) Layout of a 4–core MPSoC with locally shared caches

Figure 6.8: Applying the proposed shared cache organization to multi-core SoCs

inter-core communications are performed through an on-chip network, the checkpointing scheme outlined in Section 6.2.3 can be extended so that a checkpointing request will be initiated upon the communication of any modified data. In this way, the dependent tasks can be checkpointed and recovered *independently*. No *domino effect* would be induced in the recovery process, since execution faults produced in one task cannot propagate to dependent tasks through a communication chain.

In shared memory architectures, the protection of communication data against execution faults therefore requires the cooperation of the proposed fault tolerance scheme with cache coherency protocols. To preclude faulty data from being propagated among various cores, the ownership information maintained by a coherency protocol is updated only when the T thread verifies the correctness of a value, that is, when a write of the **T** thread causes a transition from the *dirty* or the *split* state to the *verified* state in Figure 6.6. Meanwhile, a third core that tries to modify a cache block through the cache coherency protocol should be prevented from polluting the cache states. Specifically, a third core should not be allowed to modify a cache block, if the block is pending to be verified by the T thread, or if the block is pending to be read by the T thread. Accordingly, in the proposed fault tolerance scheme, a write initiated by the cache coherency protocol is only performed if the corresponding cache block is at neither the *dirty* nor the *split* stage, and the block exhibits a balanced number of read accesses from the two threads, i.e., the read counter is 0. If any of these conditions fails to hold, the L thread is forced to await the T thread until all these conditions are fulfilled, and then the write operation is triggered.

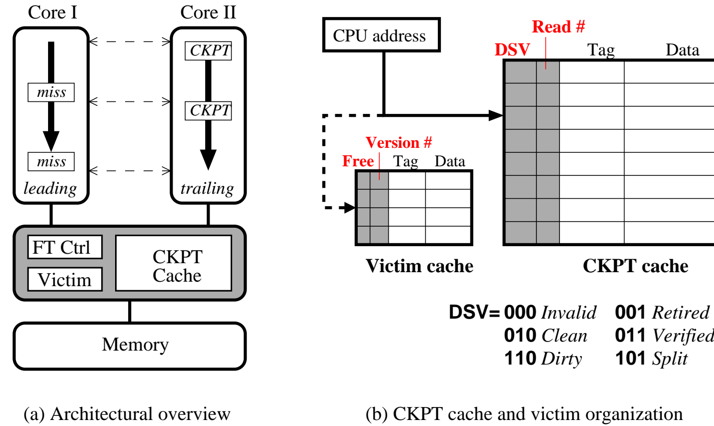
6.4.3 Throughput Enhancement through Multi-threading

The analysis in Section 6.3.3 indicates that the two performance enhancement strategies reduce the frequency of execution synchronization, but cannot fully eliminate it. Obviously, the time that the two threads spend on waiting can be utilized to execute another independent process/thread. In other words, as long as the corresponding cores offer a multithreading capability and a light-weight *context switch* between independent threads can be attained, the *throughput* of the target multicore system can be improved.

An important benefit of the *dual-core-shared-cache* fault tolerance scheme is its potential in minimizing the context switch overhead between processes/threads. Each core in a dual-core system can be employed to simultaneously execute two processes, specifically, the L thread of one and the T thread of the other. If the L thread needs to be blocked, the core can switch to execute the T thread of the other process, and vice versa. More importantly, the contexts of these threads can be captured separately in different caches, thus enabling context switches to be performed with no consequent interferences in the cache. This *dual-core-dual-cache* organization is concretely shown in Figure 6.8a.

The aforementioned 2-core-4-thread organization can be directly employed in a multi-core platform for throughput enhancement, through partitioning the cores into a set of disjoint clusters, with each cluster containing **two** cores. Meanwhile, this organization can also be extended into a *ring-sharing* manner such that the L and the T threads of the ith process are executed on the ith and the $(i+1)^{th}$ cores, respectively. In this way, an n-core MPSoC can be used to concurrently execute $2n$ redundant threads, as shown in Figure 6.8b. The cache that holds the context of the ith process is shared between the ith and the $(i+1)^{th}$ cores. As this sharing is only required between adjacent cores, the extra interconnects required on the chip can still be organized in a localized manner. As an illustrative example, a 4-core MPSoC with a *ring-sharing* cache organization is presented in Figure 6.8c.

Another noteworthy aspect of the aforementioned ring-sharing organization is its potential fault identification capability. Typically, upon a result mismatch, the exact faulty version of computation cannot be directly identified. However, in the ring-sharing organization, the ith core is employed to execute two different processes, whose results are respectively compared against the $(i-1)^{th}$ and the $(i+1)^{th}$ cores. If a result mismatch



(a) Architectural overview (b) CKPT cache and victim organization

Figure 6.9: Hardware extension to traditional cache

has been detected for each process, the ith core can be directly identified as faulty under a single fault assumption. This extra information inherent in fault detection can thus be exploited to differentiate transient and permanent faults and to further develop an adaptive fault recovery scheme accordingly.

6.5 Cache Access Control Implementation

6.5.1 Cache Access Control

To implement the proposed fault detection and checkpointing scheme, the traditional cache design is extended, as shown in Figure 6.9. A small fully associative victim cache is employed to implement the split capability. A small counter is added to each regular cache block to record the read count, and to each victim cache block to record the version count. Meanwhile, the *valid* and the *dirty* bits used in a traditional cache are replaced by a ***Dirty-Shared-Verified (DSV)*** vector that records block states. The encoding presented in Figure 6.9 is assigned in such a way that the ***D*** bit is on if and only if the block is at the *dirty* or the *split* state. As a result, when the T thread reaches a checkpoint, the cache controller can globally check whether any block is in the *dirty* or the *split* state for fault detection purposes. This encoding scheme furthermore enables the use of the following logic expressions to check whether a block has been **split** and whether a block is **replaceable**:

Algorithm 3 L thread: write hit

1: **if** a CKPT is pending **then**
2: Stall the thread
3: **else**
4: Write current value into memory if BLK_{hit} state = *retired*
5: **if** BLK_{hit} state = *split* **then**
6: *version counter* $\Leftarrow$ *version counter* + 1
7: **else if** *read counter* $\neq 0$ **then**
8: Write current value into victim cache
9: *version counter* $\Leftarrow 1$
10: BLK_{hit} state $\Leftarrow$ *split*
11: **else** $\{BLK_{hit}$ state $\neq$ *split* **and** *read counter* $= 0\}$
12: BLK_{hit} state $\Leftarrow$ *dirty*
13: **end if**
14: Perform regular cache write
15: *read counter* $\Leftarrow 0$
16: **end if**

$$\text{Split} = D \cdot V \tag{6.1a}$$

$$\text{Replaceable} = \overline{D + SV} \tag{6.1b}$$

In the proposed framework, the L thread is allowed to perform a read/write if the global *access counter* has not reached its upper bound, while the T thread is allowed to perform a read/write if the counter value is nonzero. Based on the values of the DSV vector and the counter, all the unblocked read/write accesses to each cache block can be controlled as follows:

- **Miss by the L thread:** If an *non-replaceable* block is selected for replacement, a checkpointing request is initiated; otherwise, a regular cache replacement is performed, while the old value needs to be first written into the victim cache if the read counter of that block is nonzero.
- **Read hit by the L thread:** A regular cache read is performed. The block state remains constant, while the read counter is incremented by 1.

- **Write hit by the L thread:** The control logic, shown in Algorithm 3, is extended to implement **three** functions, namely, the loosely-synchronized checkpointing scheme (lines 1–2), the additional writeback of retired blocks (line 4), the selective split capability (lines 5–10), as well as the block update for fault detection purposes (line 12).
- **Miss by the T thread:** A fault is reported upon a write miss, and upon a read miss if the corresponding data is not in the victim cache. It the read hits in the victim cache, the *version counter* (used to record the read count in this case) is decremented by 1, and the victim cache entry is released if the counter value changes to 0.
- **Read hit by the T thread:** If the block has been split, the read operation is redirected to the victim cache; otherwise, a regular cache read is performed, and the read counter of that block is decremented by 1.

Algorithm 4 T thread: write hit

```
1:  if BLK_hit state = split then
2:      Write to victim cache
3:      version counter ⇐ version counter −1
4:      if version counter = 0 then
5:          Perform a regular cache read
6:          if T value ≠ L value then
7:              Report a fault
8:          else
9:              Release victim cache entry
10:             BLK_hit state ⇐ verified
11:         end if
12:     end if
13: else {BLK_hit state ≠ split}
14:     Perform a regular cache read
15:     if BLK_hit state ≠ dirty or L value ≠ T value then
16:         Report a fault
17:     else
18:         BLK_hit state ⇐ verified
19:     end if
20: end if
```

- **Write hit by the T thread:** The control logic, shown in Algorithm 4, is extended to implement fault detection and the selective split capability. A write operation is changed to a read and a comparison, and a fault is reported either upon a value mismatch or upon an access sequence mismatch (lines 5–7, 14–16). Accesses to a *split* cache block are redirected to the victim cache (lines 1–3), while the block merge condition is checked once the version counter equals 0 (lines 4–12).

6.5.2 Implementation Efficiency

The aforementioned access strategy can be implemented as a small state machine. While a fair amount of complexity is added to the logic for controlling misses and write hits, these accesses are not on the critical path. In contrast, no extra condition is introduced for the performance-critical read hits of the L thread, and the read hits of the T thread can be directly controlled based on the DSV vector using the condition shown in Equation (6.1a). The performance overhead introduced by this state machine is practically nonexistent since the decoding of the DSV vector can be performed in parallel with the comparison of tag values for hit/miss checking. In fact, only the accesses to a *split* block performed by the T thread will be delayed by one clock cycle, since these reads need to be redirected to the victim cache.

The organization presented in Figure 6.9 indicates that the proposed cache design can be implemented within a limited amount of extra hardware. Typically the victim cache only needs to contain 16 blocks. A 2-bit counter would suffice to record either the read count or the version number, if the corresponding cache block is not continuously read/written within a short period. The 3-bit DSV vector replaces the *valid* and the *dirty* bits in the traditional cache. As a result, for an $8K$ byte L1 data cache with $2K$ blocks, the extra storage required by the proposed fault tolerance technique equals $(3 - 2 + 2) * 2K + 32 * 16 = 6.5K$ bits. Redundant multi-threading processors [31], in comparison, need to not only enlarge the reorder buffer, but also employ three centralized queues (so as to record load values, store values, and branch outcomes) with a total size of $(128 + 20 + 96) * 32 = 7.6K$ bits according to the queue sizes reported in [31].

The proposed fault tolerance technique is also more power- and heat-friendly than redundant multi-threading processors. The centralized queues [31] therein are constantly

Table 6.1: Impact of cache configuration on miss rate

	Miss rate %			
	16K-dm	16K-2w (L/C)	32K-2w (L/C)	32K-4w (L/C)
adpcm	0.203	0.203 / 0.203	0.198 / 0.198	0.198 / 0.198
epic	5.340	4.104 / 4.421	3.898 / 4.049	3.793 / 3.899
gsm	0.023	0.006 / 0.006	0.003 / 0.003	0.003 / 0.003
mpeg2	4.753	0.621 / 3.644	0.264 / 1.482	0.253 / 0.606
art	43.00	42.76 / 42.81	42.76 / 42.77	42.76 / 42.77
eon	2.962	1.004 / 1.129	0.334 / 0.405	0.113 / 0.162
facerec	13.61	8.051 / 8.119	7.586 / 7.651	7.235 / 7.278
gzip	4.782	4.437 / 4.459	3.930 / 3.938	3.936 / 3.897

accessed by both threads, thus not only consuming a large amount of energy, but also ending up becoming thermal hotspots which may degrade the reliability of the entire chip. In contrast, the proposed fault tolerance technique only employs a single centralized structure, i.e., the victim cache, which is accessed solely when the corresponding cache block has been split. The remaining extra storage, i.e., the counters and the DSV vectors, is distributed into every cache block. Each cache access only needs to read several extra bits, thus refraining from imposing significant power or heat overhead on the overall system.

6.6 Simulation Results

To evaluate the proposed fault detection and checkpointing scheme, we have performed a set of experimental studies on the *Mediabench* [56] benchmarks, as well as a number of graphics and compression programs selected from the *SPEC 2000* set, as for such application domains, reliability has been identified as a critical concern. The entire SPEC2000 programs are executed under *test* inputs so as to collect checkpointing information under different execution phases while saving simulation time. As these benchmarks display diverse cache access behaviors with miss rates ranging from 0.01% to 43%, they constitute a representative workload set for evaluating the proposed cache-based fault tolerance scheme.

The proposed fault tolerance scheme is evaluated in a dual-core execution environment. In this way, we can significantly reduce simulation complexity while thoroughly

Table 6.2: Impact of cache configuration on checkpointing frequency

	Checkpointing frequency (*K* insts/ckpt)			
	16K-dm	16K-2w (L/C)	32K-2w (L/C)	32K-4w (L/C)
adpcm	3346	3346 / 3346	- / -	- / -
epic	1.755	24.05 / 62.75	65.31 / 137.4	89.59 / 186.5
gsm	121.6	1229 / 1326	117327/117327	234653/234653
mpeg2	1.769	22.48 / 149.1	99.67 / 506.1	418.5 / 2140
art	1.905	3.184 / 3.938	6.238 / 7.992	6.810 / 8.769
eon	0.903	5.393 / 74.51	6.773 / 136.7	246.7 / 1693
facerec	0.963	10.62 / 16.52	22.21 / 37.50	31.40 / 81.46
gzip	1.839	9.791 / 13.10	17.49 / 23.72	40.62 / 72.98

evaluating the checkpointing frequency, the writeback overhead, and thread performance, with the only exception being the performance overhead that may be induced by the cache snooping protocol. The SimpleScalar toolset [6] has been extended to model two concurrently executed threads, with dedicated issue logic, register file, and functional units provided to each thread. Each core is a single-issue, in-order processor, while a 4K-entry BTB and a 4K *gshare* branch predictor are shared by the two cores. An 8-bit *global access counter* is used to keep track of the execution offset between the two threads so as to ensure that the L thread always runs ahead of the T thread. To model the proposed cache controller, the cache design is extended to incorporate the extra states shown in Figure 6.6, the checkpointing capability and the block split capability. In line with the fault tolerance literature [62, 31], we focus our experimental efforts on measuring the performance impact on the system.

6.6.1 Checkpointing and Writeback Frequencies

The checkpointing frequency of the proposed fault tolerance scheme is determined by the frequency at which a dirty cache line is replaced, which is in turn determined by the cache *miss rate* and the *replacement policy*. To evaluate such an impact, we retain the remaining architectural parameters, while simulating 4 distinct configurations for the L1 data cache: 16K directly mapped and 2-way associative, 32K 2-way and 4-way associative. For each set-associative cache, we furthermore simulate two distinct replacement algorithms: the standard *LRU*, and a *Clean-first* policy that selects clean cache lines over

Table 6.3: Overall writeback rate

	Writeback rate % $((WB_{ex} + WB_{ori})/REF_{total})$			
	16K-dm	16K-2w (L/C)	32K-2w (L/C)	32K-4w (L/C)
adpcm	0.023	0.024 / 0.024	- / -	- / -
epic	2.149	1.773 / 1.728	1.695 / 1.681	1.679 / 1.664
gsm	0.280	0.081 / 0.078	0.000 / 0.000	0.000 / 0.000
mpeg2	0.579	0.295 / 0.160	0.174 / 0.137	0.030 / 0.125
art	15.37	15.34 / 15.33	15.33 / 15.33	15.33 / 15.33
eon	8.309	4.857 / 0.553	4.351 / 0.386	0.288 / 0.175
facerec	9.315	6.447 / 6.171	4.834 / 4.767	4.317 / 4.115
gzip	4.048	2.996 / 2.872	2.746 / 2.631	2.432 / 2.235

dirty lines. The results on miss rate and checkpointing frequency are respectively listed in Tables 6.1 and 6.2, with a pair of values listed for each set-associative cache so as to clearly show the impact of replacement policies.

It can be easily seen from Table 6.2 that for most cases, the proposed scheme only imposes a checkpoint frequency of less than 1 per 10,000 instructions. *Mediabench* programs usually exhibit a smaller checkpointing frequency (for *adpcm* no checkpoints are ever taken for the latter two cache configurations) than *SPEC 2000* benchmarks, since the L1 cache is able to absorb most of the load/store requests during execution. The checkpointing frequency of directly-mapped caches is usually high due to the large amount of conflict misses. Increasing the associativity from direct-mapped to 2-way, even if it cannot appreciably reduce the miss rate for some applications (such as *adpcm*, *epic* and *gzip*), can still sizably reduce the number of checkpoints by approximately an order of magnitude. Moreover, for set-associative caches, the *Clean-first* replacement algorithm can significantly reduce the checkpointing frequency by 2 to 10 times, yet at the cost of an increased miss rate, especially for 2-way associative caches. This is because the selection of a clean line over a dirty line for replacement may overwrite the clean data that has just been brought into the cache. However, except for *mpeg2*, the miss rate increase of the remaining benchmarks is negligible.

Compared to a traditional write-back cache, the proposed cache needs to write a modified block back into memory not only upon a **replacement**, but additionally upon a subsequent **write-hit** if the block is at the *retired* state. To show this impact, Tables 6.3 and 6.4 respectively report the overall writeback rate as well as the ratio of the extra write-

Table 6.4: Checkpointing-induced writeback increase

	Writeback increase (WB_{ex}/WB_{ori})			
	16K-dm	16K-2w (L/C)	32K-2w (L/C)	32K-4w (L/C)
adpcm	0.017	0.016 / 0.016	- / -	- / -
epic	0.043	0.062 / 0.042	0.075 / 0.071	0.064 / 0.060
gsm	26.17	30.62 / 30.40	0.514 / 0.514	0.630 / 0.630
mpeg2	0.678	1.053 / 0.233	0.401 / 0.132	0.858 / 0.047
art	0.002	0.001 / 0.001	0.000 / 0.000	0.000 / 0.000
eon	7.400	16.53 / 2.080	38.47 / 7.063	3.944 / 2.144
facerec	0.322	0.508 / 0.448	0.196 / 0.181	0.127 / 0.075
gzip	0.837	0.477 / 0.420	0.464 / 0.406	0.309 / 0.212

back requests. As can be seen, the overall writeback rate decreases as the cache size or associativity increases. In most cases the extra writeback requests caused by checkpointing only creates a small increase ($\leq$ 50%) in the overall writeback rate. Significant increases ($\geq$5 times) are only reported for *gsm* and *eon* in the case of direct-mapped caches, and 2-way associative caches with an LRU policy. Even for these two benchmarks, the amount of extra writebacks can be sizably reduced if the *Clean-first* replacement algorithm is employed. This algorithm, which heavily reduces the checkpointing frequency, can in turn reduce the average writeback rate of set-associative caches to around 1%.

6.6.2 Thread performance

The proposed fault tolerance scheme affects the overall performance of the multicore system in that the L thread needs to be blocked under the conditions outlined in Section 6.3.3, while the T thread needs to spend extra time in accessing the victim cache and checkpointing the regular cache. Accordingly, the overall performance overhead is strongly affected by the *initial execution offset* between the two threads. Too small of an offset would result in the T thread quickly catching up to the L thread, thus causing the T thread to also wait for the missing data. In contrast, too large of an offset would result in the L thread splitting a large number of cache blocks, thus causing the T thread to constantly access the victim cache. Taking both effects into consideration, during simulation we initiate the execution of the T thread upon any of the following conditions: 1) The L thread splits a cache block; 2) The read count of a block is half full; 3) The global access counter is half full; 4) The L thread generates a checkpoint request.

Table 6.5: Impact of thread synchronization and block split on CPI increase (%): 16K-2way

	16K-2way-LRU				16K-2way-Clean first			
	None	Async	Split	Both	None	Async	Split	Both
adpcm	0.074	0.074	0.008	**0.008**	0.074	0.074	0.008	0.008
epic	3.100	2.882	**0.898**	1.075	3.316	3.249	1.088	1.022
gsm	19.04	19.04	2.920	2.919	19.04	19.04	2.929	**2.918**
mpeg2	9.282	5.091	5.054	**0.158**	1.378	12.98	12.97	9.285
art	0.397	0.363	0.245	0.160	0.325	0.296	0.173	**0.132**
eon	9.094	8.852	6.679	6.427	4.022	4.011	1.362	**1.347**
facerec	3.250	3.091	2.259	2.086	3.106	2.990	2.107	**1.985**
gzip	8.353	8.323	5.449	5.460	8.188	8.161	5.154	**5.008**
average	2.615	2.534	0.913	0.891	2.590	2.539	0.937	0.887

To evaluate the two optimization techniques outlined in Section 6.3, i.e., the *loosely-synchronized checkpointing model* and the *selective split capability*, the proposed fault tolerance technique is simulated in **four** different ways, with either of these optimizations, with both or with neither. Here, according to the results of checkpointing and writeback frequency, we select two representative cache configurations, 16K-2way and a 32K-4way, for performance simulation. The checkpointing latency is set to 80 and 120 cycles for the 16K and 32K caches, respectively, while the cache miss penalty is set to 30 cycles in both configurations. The baseline MPSoC employs the LRU replacement policy, while the fault tolerant MPSoC employs both the LRU and the *Clean-first* policies so as to clearly show the impact of the latter.

The obtained results of CPI increase are listed in Tables 6.5 and 6.6, with the minimum performance overhead for each cache configuration highlighted in bold. As can be seen, except for *epic* and *mpeg2*, the minimum performance overhead of all the other benchmarks is attained in the "*Both*" column, implying that both of the performance optimizations are quite effective. The average values (computed as geometric means) presented at the last row indicate that the *selective split capability* is more effective than the *loosely-synchronized checkpointing model*: the former can reduce the performance overhead by 64–75% while the latter only offers a reduction of 2–3%. Moreover, the performance benefit offered by the proposed *selectively split* technique is achieved at a minimum amount of duplication cost. To illustrate this property, we additionally report in Table 6.7 the ratio of store instructions that induce split requests, and the number of

Table 6.6: Impact of thread synchronization and block split on CPI increase (%): 32K-4way

	32K-4way-LRU				32K-4way-Clean first			
	None	Async	Split	Both	None	Async	Split	Both
adpcm	0.073	0.073	0.007	**0.007**	0.073	0.073	0.007	0.007
epic	2.984	2.910	0.727	**0.653**	3.066	3.044	0.774	0.751
gsm	19.01	19.01	2.884	**2.884**	19.01	19.01	2.884	2.884
mpeg2	4.962	4.960	0.878	**0.875**	5.864	5.863	1.792	1.792
art	0.309	0.285	0.155	0.113	0.209	0.185	0.056	**0.022**
eon	3.766	3.764	1.130	**1.127**	3.799	3.798	1.157	1.156
facerec	2.041	1.968	0.943	0.868	1.743	1.707	0.646	**0.608**
gzip	7.958	7.951	4.873	4.839	7.742	7.737	4.870	**4.498**
average	2.107	2.070	0.561	0.525	2.012	1.973	0.520	0.452

Table 6.7: Cache Block Split Efficiency

	Split Store (%)		Store/split block	
	16K	32K	16K	32K
adpcm	0.415	0.418	1.929	1.920
epic	11.94	15.96	1.003	1.229
gsm	18.33	18.32	2.243	2.244
mpeg2	5.830	11.82	2.158	2.123
art	0.449	0.446	1.001	1.001
eon	5.331	6.398	1.284	1.436
facerec	3.937	4.017	1.132	1.130
gzip	18.60	19.95	2.710	2.610
average	4.177	4.902	1.570	1.621

write operations mapped to each split block. As can be seen, in most benchmarks, only a small ratio of store instructions ($\leq$15%) induce split requests for the corresponding cache block. Sizable amounts of split requests are only reported for *gsm* and *gzip* that display intensive loop-carried dependences in frequently executed loops. Yet for these benchmarks, on average each split block in the victim cache is able to capture more than 2 store instructions. These results therefore confirm that the proposed *selectively split* technique imposes far less duplication cost than redundant multithreading processors [62] that typically duplicate each store value in hardware buffers.

When the *selective split capability* is incorporated into the cache design, the overall performance of the fault-tolerant MPSoC is also affected by the victim cache size and

Table 6.8: Impact of counter and victim cache sizes on CPI increase (%)

	16K-2way			32K-4way		
	2C-16V	3C-16V	3C-32V	2C-16V	3C-16V	3C-32V
adpcm	0.008	0.005	**0.005**	0.007	0.004	**0.004**
epic	0.898	0.746	**0.679**	0.653	0.515	**0.430**
gsm	2.918	2.435	**2.135**	2.884	2.400	**2.095**
mpeg2	1.158	**1.118**	1.119	**0.875**	0.933	0.956
art	0.132	0.132	**0.132**	0.022	0.081	**0.021**
eon	1.347	1.175	**1.175**	1.127	**0.810**	0.997
facerec	1.985	1.962	**1.962**	0.608	0.629	**0.571**
gzip	5.008	4.956	**4.955**	**4.498**	4.640	4.537
average	0.673	0.592	0.576	0.405	0.411	0.339

the counter size. To evaluate such an impact, we additionally simulate **three** distinct configurations, a 16-entry victim cache with 2-bit or 3-bit counters, and a 32-entry victim cache with 3-bit counters. The obtained performance overhead results (in terms of CPI increase) are presented in Table 6.8, generated under the *minimum performance overhead* configuration highlighted in Table 6.5. In general, as indicated by the average values shown in the last row, an increase in the counter size and/or the victim cache size can relax the conditions for blocking the L thread and hence reduce the performance overhead. Yet this property does not hold for each benchmark. In certain cases (e.g., *mpeg2, gzip*), a larger counter and victim cache would slightly degrade performance, as it ends up creating more split cache blocks. The T thread needs to spend more cycles accessing the victim cache, while the L thread needs to await the T thread for more cycles at each checkpoint.

In terms of the replacement policy, Table 6.2 shows that compared to *LRU*, the *Clean-first* policy sizably reduces the checkpointing frequency, yet slightly increases the miss rate. Accordingly, this policy induces a diverse impact on the overall thread performance, confirmed by the values reported in Table 6.5. Specifically, compared to *LRU*, the *Clean-first* policy imposes a negligible impact on *adpcm* and *gsm*, reduces the performance overhead of all the SPEC2000 benchmarks except for *eon* in the 32K-4way case, yet degrades the overall performance of *epic* and especially *mpeg2* of whose miss rate is increased significantly.

Table 6.9: Impact of fault rate on CPI increase (%)

	16K-2way-LRU			32K-4way-LRU		
	1E-5	3E-5	1E-4	1E-5	3E-5	1E-4
adpcm	0.008	0.008	43.96	0.007	0.007	33.68
epic	1.298	1.744	1.892	0.899	1.349	1.717
gsm	4.231	9.792	33.24	332.0	566.8	2205
mpeg2	2.129	4.042	9.888	12.35	28.28	82.84
art	0.348	1.467	3.696	0.425	0.778	3.405
eon	6.447	6.509	6.723	2.604	5.532	14.21
facerec	2.220	2.830	3.594	1.464	2.537	5.906
gzip	5.556	5.924	6.912	5.206	5.861	8.583

In sum, with the two performance optimizations applied, the proposed scheme causes an average increase of 0.45% to 0.9% to the non-fault-tolerant single thread performance. This technique outperforms the lock-step CMP that typically exhibits a performance overhead of 15% to 19% according to the data reported in [31], and incurs additional hardware costs, to boot.

6.6.3 Impact of Fault Rate on Thread Performance

The performance overhead of a checkpointing-based recovery scheme is determined not only by the checkpointing latency and frequency, but additionally by the fault rate in the system. To evaluate this impact, we simulate the proposed checkpointing scheme under distinct fault rates, ranging from 10^{-5} to 10^{-4}. We randomly insert faults in store instructions under the rate assumptions of 5% manifestation of device failures in the pipeline and 20% propagation of incorrect results through the dependence chain to store instructions. The results, obtained for two representative cache configurations, are presented in Table 6.9. As can be seen, the performance overhead linearly increases as the fault rate grows. For most benchmarks the CPI increase is insignificant ($\leq$10%), as the corresponding checkpoint frequency is 10 times higher than the fault rate. Yet significant overhead can be observed for *adpcm*, *gsm*, and *mpeg2* when the fault rate is high, especially in 32K caches. This is because for such cases, as shown in Table 6.2, the checkpointing frequency is comparable to the fault rate, thus resulting in a large amount of computation being rolled back upon failures.

Table 6.10: Memory hierarchy induced checkpointing tradeoffs

	M insts/ckpt		CPI increase (%)	
	L1	Hybrid	L1	Hybrid
adpcm	3.3455	-	0.008	0.007
epic	0.0241	6.5963	1.075	0.533
gsm	1.2285	-	2.919	2.917
mpeg2	0.0225	24.658	1.378	1.144
art	0.0032	0.0840	0.160	0.028
eon	0.0054	450.75	6.427	5.609
facerec	0.0106	0.0788	2.086	2.003
gzip	0.0098	5.7294	5.460	2.575

6.6.4 Checkpointing Tradeoffs for Memory Hierarchy

As outlined in Section 6.4.1, checkpointing at the L2/memory interface instead of the L1/L2 interface can significantly reduce the checkpointing frequency. To evaluate this impact, we additionally simulate the hybrid checkpointing strategy outlined in Section 6.4.1. The L1 cache is 16KB 2-way associative, while the L2 cache is 512KB 4-way associative, and both caches employ an LRU replacement algorithm. The obtained results in term of *checkpointing frequency* and *CPI increase (%)* are listed in Table 6.10. As can be seen, the hybrid checkpointing scheme significantly reduces the checkpointing frequencies for most programs, with a minimum reduction of 7.5 times reported for *facerec*. In terms of performance overhead, the hybrid method offers a sizable reduction (of more than 50%) for *epic*, *art*, and *gzip* that display relatively high checkpointing frequency in the L1 cache. In comparison, a negligible performance impact is observed in *adpcm* and *gsm* that already display a highly limited number of checkpoints in the L1 cache.

6.7 Conclusions

In this chapter, we have presented an integrated fault detection and checkpointing framework that simultaneously delivers full fault resilience and relaxed execution synchronization. Through sharing a single cache between two redundant threads, one thread can directly verify the execution results of the other, thus delivering light-weight fault detection while at the same time strictly protecting the memory against execution

faults. Meanwhile, detection-induced synchronization requirements are drastically relaxed through allowing unconfirmed results to be written into the cache, as well as selectively splitting a constantly updated cache block and skipping the comparison of the intermediate values. Further performance improvements in multicore systems can be attained through the utilization of multi-level caches to simultaneously minimize checkpointing and synchronization requirements, as well as the utilization of multithreading to increase overall computation throughput. The simulation results show that the average checkpointing frequency is as low as 1 per 30,000 instructions, with only a slight increase in the write-back rate and a less than 10% degradation in CPI under $\leq 10^{-4}$ fault rates. The diminution of checkpointing frequency, in conjunction with the negligible overhead in detection and checkpointing, introduces the possibility of efficient fault resilience insertion in various architectures.

The text of Chapter 6, is in part a reprint of the material as it appears in *C. Yang and A. Orailoglu, "A Light-weight Cache-based Fault Detection and Checkpointing Scheme for MPSoCs Enabling Relaxed Execution Synchronization," International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), October 2008.* The dissertation author was the primary researcher and author of the publication [95].

Chapter 7

Compiler-Directed Heat Reduction

As the system fault rates exponentially increase as peak temperature rises, mitigation of thermal stress can in turn reduces resource unavailability induced by both heat buildup and execution faults. Typically the chip-wide temperature exhibits a quite unbalanced distribution. Peak temperature of the entire chip therefore can be reduced through shifting computation from a hot component to a relatively cool component. Previous research has shown that at the core level, temperature can be effectively balanced by the compiler, through the generation of thermal-aware execution schedules [23]. Without imposing any runtime overhead, the compiler can generate execution schedules in such a way that the "hot" tasks are distributed across various cores at different time.

A more interesting observation is that the temperature within each individual core also exhibits a unbalanced distribution. In particular, due to its high utilization (accessed 2–3 times per instruction) and relatively small area, the *register file* has been established as one of the hardware units most likely to overheat in current processors [81]. This localized "hotspot" can reach critical temperature levels regardless of average or peak external package temperature, thus ending up constraining the overall performance and reliability of the whole chip. More crucially, due to the fact that 90% of the execution time is spent on loops where only a small subset of registers is repetitively accessed, register file accesses also exhibit high asymmetry during program execution. This asymmetric register utilization furthermore leads to considerable temperature differentials, since most of the heat generated within a microarchitectural block is dissipated vertically to the heat sink rather than laterally to adjacent blocks [81].

The aforementioned register file access characteristics indicate that the peak temperature within a register file, the hottest spot of a modern processor, can be effectively controlled through distributing the accesses uniformly throughout the register file. To achieve this goal, the register names obtained at the decode stage cannot be directly used to access the register file. Instead, a register shuffling technique that physically remaps heavily accessed logical registers prior to local heat buildup is necessitated. To attain this goal, in this chapter we introduce a *compiler-directed deterministic register shuffling* technique. A post-compilation adjustment of the register names allows *regularity* to be embedded within register accesses, so that accesses to each register can be evenly balanced across loop iterations with no need of any hardware mapping table to keep track of register usage or register mapping information. This extremely low hardware overhead therefore enables an easy incorporation of the proposed technique into low-power embedded processors to attain temperature control.

7.1 Challenges in Register Access Balance

The design of a dynamic register shuffling process to reduce the register file peak temperature and, hence, to improve chip reliability, is motivated by the observation that the code generated by the compiler exhibits highly asymmetric register access activity. A traditional register allocation scheme in a temperature-unaware compiler initially assumes an infinite set of *virtual registers* for representation, and subsequently maps these virtual registers into a fixed number of *architectural registers*. The decision regarding which physical registers in particular are to be allocated, however, does not take into consideration the access distribution, thus leading to a highly asymmetric register access activity. Figure 7.1, which presents the cumulative register access ratios of a set of *SPEC2000* (shown in the first 4 bars of Figure 7.1) and *mediabench* programs (the second 4 bars), provides experimental confirmation. As can be seen, both sets of benchmarks exhibit an appreciable amount of imbalance in that 48% to 71% of the total register accesses are to 4 registers. *Mediabench* programs display a higher amount of imbalance as compared to *SPEC2000* benchmarks. On the average, a set of 12 out of a total of 32 registers is able to capture more than 90% of the total register accesses.

The asymmetric register utilization shown in Figure 7.1 confirms that by evenly

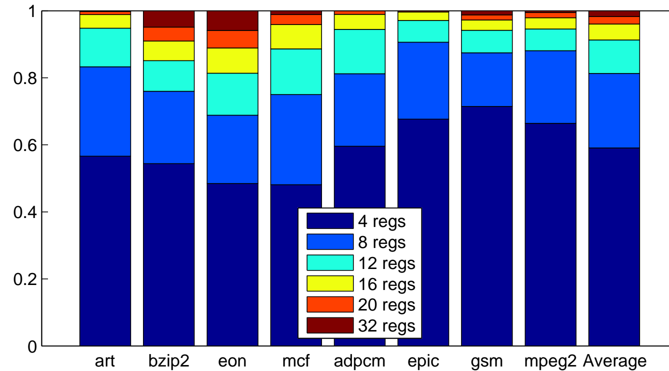
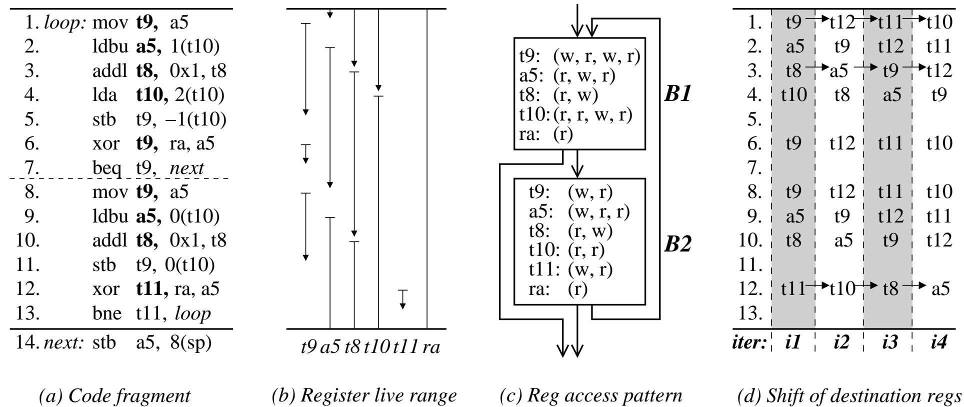


Figure 7.1: Cumulative register access ratio

distributing the accesses to each individual register within frequently executed loops, the peak power density and hence the peak temperature can be effectively reduced. However, such a balancing cannot be directly attained through an assignment of register names during code generation, as the access asymmetry directly derives from the asymmetric variable utilization of the program. This limitation can be more concretely illustrated by a representative code fragment presented in Figure 7.2a, an unrolled loop composed of 13 instructions that accounts for more than 25% of the total execution time of *bzip2*. The corresponding register live range and access pattern are respectively presented in Figures 7.2b and 7.2c. As can be seen, this loop exhibits a quite unbalanced register utilization as it only accesses 6 (a_5, t_8–t_{11}, ra) out of 32 general purpose registers. Among these 6 registers, a_5, t_9, and t_{10} are accessed most frequently. While the compiler may be able to use more registers by separating multiple definitions of a single register (a_5 and t_9) or further unrolling the loop, individual register accesses would still remain skewed due to the existence of singly assigned yet frequently referred registers, such as t_{10}.

Since the static register allocation process cannot completely balance the accesses to each individual register, a dynamic mapping needs to be established between the encoded register names, denoted as the *logical registers*, and the register instances in the register file, denoted as the *physical registers*, to physically remap heavily accessed logical registers prior to local heat buildup. At first sight, it seems that this task could be achieved through using a hardware *mapping table*, such as the one used in conventional superscalar processors. Unfortunately, such a mapping table imposes a notable amount of hardware complexity, energy consumption and performance overhead. More crucially,



(a) Code fragment *(b) Register live range* *(c) Reg access pattern* *(d) Shift of destination regs*

Figure 7.2: A loop example obtained from *bzip2*

as the table needs to be accessed using logical register names at a frequency no lower than that of register file accesses, this small hardware unit itself would hence become a temperature "hotspot" with skewed access distribution. This significant energy and heat overhead confirms that a temperature-aware register file should be able to evenly distribute accesses to each register with **no** reliance on a hardware mapping table. Moreover, as no hardware mapping table is used to keep track of run-time register usage or register mapping information, *regularity* needs to be embedded within register accesses so that the mapping between logical and physical register names can be controlled in a *deterministic* manner. Specifically, the following two tasks need to be accomplished through an easily computable dynamic mapping that deterministically controls the renaming process:

- Select a free physical register for each write access *with no dynamic register usage information*;
- Redirect each read access to the corresponding physical register *with no dynamic mapping information.*

7.2 Deterministic Register Shuffling

The fundamental goal of the proposed register shuffling technique is to evenly balance register accesses across loop iterations with no reliance on a hardware mapping table. Accordingly, not only a *dynamic* logical-to-physical register mapping needs to be

established, but furthermore *determinism* needs to be embedded within such a dynamic mapping.

To attain a deterministic register remapping, the proposed technique exploits the fact that no fixed, preordained correspondence exists between program variables and register names. The compiler can therefore establish a certain property between consecutive accesses to each register, thus enabling the hardware to redirect register accesses with no reliance on a mapping table.

7.2.1 An illustrative example

To deterministically select a free physical register for each write access register, the proposed technique exploits the fact that during loop execution, the physical register used at the *preceding iteration* becomes free whenever a new physical register is allocated as the destination of an instruction. Taking the code fragment presented in Figure 7.2a as an example, upon a new iteration, if a free register (t_{12} for example) is used as the destination of instruction 1, the old destination, t_9, becomes free thereafter. As a result, t_9 can be used as the new destination of instruction 2, which in turn frees up a_5, allowing it to be used for instruction 3, and so on and so forth. This shift in register assignments can be clearly seen in the ***i2*** column of Figure 7.2d. Finally, at the end of this iteration, t_{11} has been freed, thus allowing it to be used as the new destination of instruction 1 at the next loop iteration. The remapping of the destination registers during the first 4 consecutive iterations is summarized in Figure 7.2d.

It can be seen from the register names presented in Figure 7.2c that the proposed register remapping process exhibits the following two properties:

- Across loop iterations, a logical register is sequentially mapped to all the physical registers before it shuffles back to the initial mapping.
- Within a single iteration, all the assignments of a single logical register are mapped to the same physical register, thus establishing a one-to-one mapping between logical and physical register names.

The first property indicates that the proposed technique can effectively balance accesses to individual registers across loop iterations. Although this technique does not

reduce the energy consumed in each register access, it still effectively prevents local heat buildup since heavily accessed logical registers, such as $a5$, t_9 and t_{10} in the example, are mapped to distinct physical registers across loop iterations. As temperature takes at least 0.1 million cycles to rise by 0.1 $°C$ [81], this balanced access activity, achieved at the granularity of loop iterations, enables an effective reduction of the register file peak temperature.

The second property enables the proposed remapping scheme to attain access determinism. Specifically, by ensuring that a one-to-one mapping is established between logical and physical register names within a single iteration, the physical register names can completely determined according to the static name and the loop iteration count, thus eliminating the necessity of a hardware table to record dynamic register mapping.

7.2.2 Destination register name adjustment

At each iteration, the proposed scheme deterministically remaps the kth logical destination register to the physical register used as the $(k-1)^{th}$ destination in the last iteration. This recursive relationship can be formalized as follows, with $DN(R_k^i)$ denoting the dynamic name of the kth destination register at iteration i:

$$DN(R_k^i) = DN(R_{k-1}^{i-1}) = DN(R_{k-2}^{i-2}) = ... = DN(R_{k-i}^0) \tag{7.1}$$

Equation (7.1) illustrates a crucial property of the proposed register shuffling technique: during loop execution all the logical destination registers are iteratively mapped to the same set of physical registers in the same *shifting order*. The shifting order of the *bzip2* example, represented by the arrows in Figure 7.2d, is (t_9, t_{12}, t_{11}, t_{10}, t_8, a_5, t_9). Moreover, this shifting order happens to be the **reverse** of the order in which each logical register appears as a destination within the loop body.

The *reverse-order* property indicates that if a fixed offset has been imposed between any two *consecutive yet distinct destination register names*, any two *consecutive mappings of a logical register* would also exhibit a fixed offset. More formally, by imposing a fixed offset of O' between the static names of the kth and the $(k-1)^{th}$ destination registers, the dynamic name of register R_k at iteration i, denoted as $DN(R_k^i)$, can be generated through shuffling $DN(R_k^{i-1})$ by a fixed offset of O. Using V_O^α to denote the shuffle

of a value V by an offset O for α times, this fixed-offset relationship can be formalized into the following equations, with O and O' being complements in that $(V_O^\alpha)_{O'}^\alpha = V$ for any positive integers V and α.

$$DN(R_k^i) = DN(R_k^{i-\alpha})_O^\alpha = SN(R_k)_O^i \tag{7.2a}$$

$$SN(R_k) = SN(R_{k-\alpha})_{O'}^\alpha = SN(R_0)_{O'}^k \tag{7.2b}$$

According to Equation (7.2), at iteration i, the dynamic name of register R_k can be generated through shuffling the corresponding static register name $SN(R_k)$ by an offset of O for i times, while $SN(R_k)$ should be generated through shuffling the static name of the 0th destination register $SN(R_0)$ by an offset of O' for k times. These equations clearly confirm that at each iteration, the dynamic register names can be completely determined by the compiler.

To effectively balance register accesses while minimizing the hardware complexity, a light-weight shuffling function is furthermore necessitated. Given a *shuffle window* composed of a set of contiguous physical registers B, B+1, ..., B+$T-1$, an effective shuffling function needs to ensure that each logical destination register R_k will be sequentially mapped to each physical register within the window before it shuffles back to the initial mapping. In other words, the shuffling function needs to establish a one-to-one mapping from V_O^1, V_O^2, ..., V_O^T, the T consecutive dynamic names of register R_k, to the T distinct values B, B+1, ..., B+$T-1$, within the shuffle window. According to this requirement, **two** shuffle functions, namely, a modulo addition and a *Galois field* multiplication, can be employed to attain a deterministic register shifting, as summarized in the following equation.

$$SN(R_k) = B_{O'}^k = \begin{cases} B \otimes 2^{T-(k*O)\%T}, & T = 2^n - 1; \\ (B - k*O)\%T, & \text{GCD}(O,T) = 1. \end{cases} \tag{7.3}$$

In modulo T addition, the static register name of R_k is generated as $(B-k*O)\%T$. The values of the offset O and the window size T should be relatively *prime* so as to ensure the assignment of distinct physical names to different logic registers within a single iteration. In hardware, this addition function can be implemented using a j-bit modulo T adder for each register access port, assuming a total of 2^j registers provided in the

Table 7.1: The use of the two shuffle functions to shift register names in the *bzip2* example

	Modulo Addition							$GF(2^3)$ Multiplication						
	i_0	i_1	i_2	i_3	i_4	i_5	i_6	i_0	i_1	i_2	i_3	i_4	i_5	i_6
$t_9 \rightarrow B$	1	2	3	4	5	6	7	1	2	4	3	6	7	5
$a_5 \rightarrow B^1_{O'}$	7	1	2	3	4	5	6	5	1	2	4	3	6	7
$t_8 \rightarrow B^2_{O'}$	6	7	1	2	3	4	5	7	5	1	2	4	3	6
$t_{10} \rightarrow B^3_{O'}$	5	6	7	1	2	3	4	6	7	5	1	2	4	3
$t_{11} \rightarrow B^4_{O'}$	4	5	6	7	1	2	3	3	6	7	5	1	2	4
$DN(R^i_k)$	$= (SN(R_k) + i)\%7$							$= SN(R_k) \otimes 2^i$						
$SN(R_k)$	$= (SN(R_0) - k)\%7$							$= SN(R_0) \otimes 2^{7-k}$						

architecture. On the other hand, if the value of the window size T equals $2^n - 1$, a more efficient shuffle function can be implemented so that $B^k_{O'} = B \otimes 2^{T-(i*O)\%T}$, with $\otimes$ denoting the multiplication operators defined in the extension *Galois field* of $GF(2^n)$. The hardware implementation of this multiplication function is comparatively cheaper as no modulo adders but only a limited number of *xor* gates are required.

The differences between these two functions are concretely illustrated in Table 7.1, which shows the mapping of the five destination registers of the *bzip2* example in 7 consecutive iterations, with B, O and T respectively set to 1, 1, and 7, and the field generating polynomial of $GF(2^3)$ set to $x^3 + x + 1$.

7.2.3 Loop-carried dependence preservation

As the new name of each logical destination register can be determined using Equation (7.3), the names of source registers can be determined accordingly. Since the mapping of a logical register varies across iterations, read accesses before and after the first assignment within the loop body should be directed to distinct physical registers. Specifically, all the read accesses following the first write operation, as it remaps the logical register, should be directed to the new allocated physical register. In contrast, all the read accesses preceding the first write operation should obtain the value produced at the prior iteration, thus requiring the compiler to additionally shuffle the register name by O'.

The aforementioned name adjustment of source registers can be illustrated more clearly by considering the logical register a_5 in the *bzip2* loop presented in Figure 7.2a. As shown in the first column of Table 7.1, the name of a_5 is adjusted to $B^1_{O'}$ by the compiler. Since a_5 is remapped by instruction 2, all the subsequent read accesses to a_5 within the

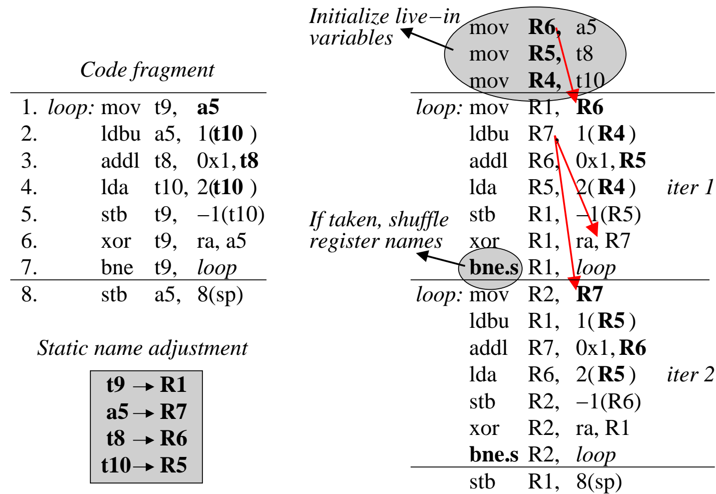


Figure 7.3: Register name adjustment in two consecutive iterations

loop body should be directed to $B^1_{O'}$. On the other hand, instruction 1, which reads a_5 before it is remapped, should obtain the value produced at the prior iteration wherein the name of a_5 is not $B^1_{O'}$ but $B^2_{O'}$. Accordingly, the compiler should adjust the name of a_5 appearing in instruction 1 by an additional amount of O' so as to preserve this loop-carried dependence.

An additional shuffle of O' to the names of the *live-in* variables allows register values to be effectively passed across loop boundaries during execution. Therefore, semantic correctness can be naturally guaranteed as long as *live-in* variables, such as a_5, t_8 and t_{10} in the *bzip2* loop, are correctly initialized prior to entering the loop. This task can be attained simply through the insertion of extra *move* instructions to transfer register values prior to entering the loop. These few register *move* instructions, as they are executed quite rarely outside the loop body, introduce no overhead in practice, neither in terms of performance nor in terms of energy.

To concretely illustrate the aforementioned name adjustment policy for destination and source registers, it has been applied to a non-unrolled version of the *bzip2* loop presented in Figure 7.3. Using the *modulo addition* in Table 7.1 as the shuffle function, Figure 7.3 presents the register names in the first two iterations of the transformed code. As can be seen, the compiler has globally adjusted register names according to the order in which they appear as destinations. The names of adjacent yet distinct register destinations differ by an offset of $O'{=}{-1}$, while an extra offset of O' is added to each live-in

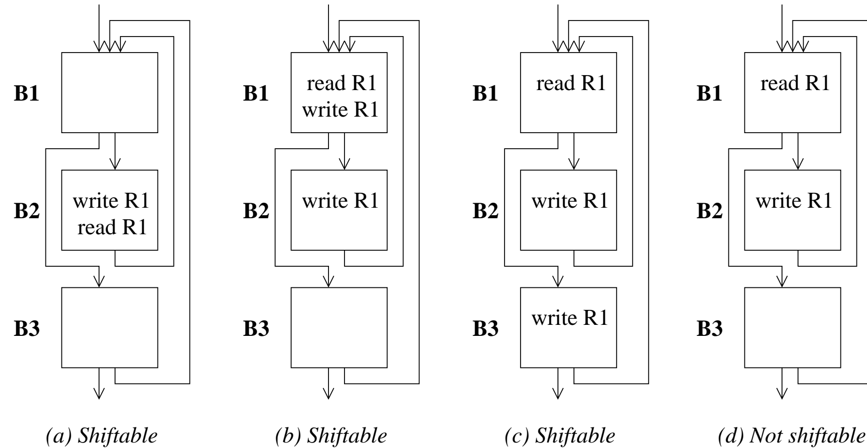


Figure 7.4: Shiftability analysis of register *R1*

read reference shown in instructions 1, 2, 3, and 4. Meanwhile, a hint is inserted into instruction 7, the loop branch, so that once the branch is taken, each register name, except for the read-only register *ra*, will be shifted by an additional offset of O=1. Finally, three register *mov* instructions have been inserted prior to entering the loop so as to initialize the live-in registers a_5, t_8, and t_{10}, respectively.

7.2.4 Shiftable logical register identification

The proposed register shuffling scheme requires a detailed examination of register access patterns so as to determine whether a logical register accessed within the loop body is *shiftable* or not. In general, the characteristics of the proposed register shuffling scheme preclude its application to **two** types of logical registers. Firstly, as a logical register is remapped upon the first assignment, *read-only* registers, such as *ra* in the *bzip2* example, become unshiftable. A more complex case is that of registers exhibiting conditional definitions within the loop body; as the compiler needs to identify for each read access the exact iteration at which the value is produced, a logical register cannot be shuffled if its value is not certifiably updated at each loop iteration, that is, if it exhibits *write* accesses only in *conditionally executed basic blocks* but *read* accesses outside those blocks.

Conditionally defined registers create an issue of **nondeterministic** loop-carried dependences, which can be illustrated more clearly through examining the four cases presented in Figure 7.4. These four cases share the same control flow yet exhibit a variety of access patterns to register *R1* within the loop body. In Figure 7.4a, the write access in basic block *B2* constitutes a conditional definition. However, *R1* is still *shiftable* since it is read within the same basic block following such a write access, thus allowing identical register names to be assigned to both accesses. In Figure 7.4b, *R1* is also shiftable as the write access in *B1* constitutes an unconditional definition, thus indicating that the read access in *B1* should always obtain the value defined in the preceding iteration. Similarly, in Figure 7.4c, while neither of the write accesses in *B2* and *B3* is guaranteed to be executed, the two accesses in conjunction constitute an unconditional definition, thus making *R1* shiftable. In comparison, in Figure 7.4d, *R1* is only written on the fall-through path of the branch, resulting in the read access in *B1* obtaining a value defined in either the preceding iteration or an even earlier iteration, depending on the branch outcome. As a result, for such a read access in *B1*, the compiler cannot statically determine the exact iteration at which the value is produced, resulting in *R1* being *unshiftable*.

An *unshiftable* logical register does not need to be remapped, if it is not accessed frequently within the loop body. However, in the extreme case of an unshiftable register being frequently accessed, **two** approaches can be adopted to prevent local heat buildup. In a hardware-oriented approach, the value of such a register can be duplicated into a dedicated buffer for access, instead of the power-hungry register file. In a software-oriented approach, an extra *move* instruction can be inserted within the loop body to make it *shiftable*. If this register happens to be a conditionally defined register (for example, *R1* in Figure 7.4d), such a *move* instruction can be inserted into the basic block executed on the other path of the branch (*B3* in Figure 7.4d). If, on the other hand, the frequently accessed yet unshiftable register happens to be a read-only register, the extra *move* instruction needs to be inserted into an unconditionally executed basic block.

7.2.5 Physical register reallocability analysis

The example presented in Section 7.2.1 indicates that the proposed deterministic shuffling approach requires the existence of at least **one** free extra register, such as t_{12} in

Table 7.2: Access pattern-based register classification

Shiftable	Unconditionally *written* Conditionally *written* and *read* on the same path
Unshiftable	*Read*-only Conditionally *written* yet unconditionally *read*
Free	*Not-accessed*, either *dead* or *live*

Figure 7.2c, for the shuffle of the first destination within the loop body. As most execution hotspots are composed of nested loops consisting of only a limited number of instructions, the requirement of one free register can be naturally satisfied since typically only a subset of registers is accessed during loop execution. The *bzip2* example presented in Figure 7.2a clearly confirms this property in that only 6 out of the total 32 registers are accessed within the loop body.

While theoretically the shuffle window only needs to include one extra free register in addition to the shiftable destination registers, the search for an increasingly balanced register access distribution motivates the maximization of the number of free registers within the shuffle window. A detailed examination indicates that according to the access pattern, all the logical registers and hence, the corresponding physical registers, can be classified into three categories: *shiftable*, *unshiftable*, and *not-accessed*. For the third type, a physical register not accessed within the loop body can be directly remapped, if it is not used to hold a live variable with infinite lifetime across the execution of the whole loop. As an example, in the *bzip2* loop all the *not-accessed* registers except for *sp* are free for remapping. Register *sp*, on the other hand, holds its lifetime across the whole loop as it is directly read after exiting the loop. However, even this type of *not-accessed* yet *live* registers can be freed up through employing extra store and load instructions to checkpoint and restore the original value at loop entries and exits, respectively. The introduced performance overhead is practically nonexistent since this checkpointing and restoration process is performed outside the loop execution.

By checkpointing and restoring *live* yet *not-accessed* register values, all the registers that are not accessed within the loop body become available for remapping. Accordingly, among the three classes of registers listed in Table 7.2, both the *shiftable* and the *free* registers are included in the shuffle window, while only the *unshiftable* registers need to be placed outside the shuffle window. The size of the shuffle window therefore

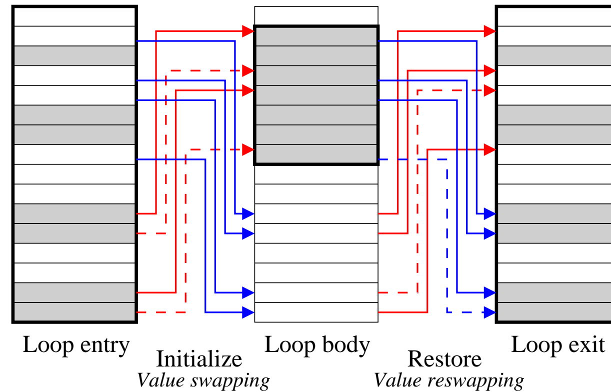


Figure 7.5: Building a shuffle window through swapping register values at loop entry and exit

can be maximized, thus enabling the achievement of a more balanced access distribution and, hence, a further reduction in peak temperature.

The identified shiftable and free registers may *scatter* across the entire register file. As the shuffle window should be composed of a set of *contiguous* registers, at the entry and the exit of each frequently executed loop, some register values need to be swapped so that the identified *shiftable* and *free* registers can be placed at contiguous positions. This process is concretely presented in Figure 7.5. At the loop entry the *live-in* register values need to be preserved, implying that *unshiftable* registers within the shuffle window need to be swapped out, while *shiftable* yet *live-in* registers outside the window need to be swapped in. On the other hand, at the loop exit, a register re-swapping process needs to be performed to preserve the *live-out* register values, both within and outside the shuffle window. Both the register swapping and reswapping processes are accomplished by the compiler through the insertion of extra *move* instructions which, as they are executed outside the loop body, introduce no overhead in practice.

7.2.6 Functional Evaluation

We have discussed the proposed deterministic register shuffling technique from three vantage points, namely, the dynamic shuffling functions, the adjustment of logic register names, as well as the identification of the shiftable and free registers. Since the proposed technique only remaps register names across loop iterations, it can be independently applied on each execution hotspot, i.e., a frequently executed loop. Due to the iter-

ative nature and the relatively short static code size of each loop, the proposed technique delivers maximum benefit at minimal cost, as only 10% of the code needs to be analyzed while balanced register accesses for 90% of execution time can be accomplished.

Compared to the thermal-aware register reassignment approaches [24, 100], the proposed deterministic register shuffling technique requires no revisitation of the NP-hard register allocation problem to perform live range reassignment. Therefore, the adjustment of logic register names can be implemented as a procedure to be performed subsequent to the conventional register allocation phase, thus retaining all the concomitant benefits of the latter. Moreover, a detailed examination indicates that neither of the two techniques can fully balance the accesses to each individual register at each loop iteration. Instead, both techniques attain a relatively *coarse-grained* access balance, yet one exploits the spatial domain while the other exploits the temporal domain. The thermal-aware register reassignment approaches attain a *spatial* balance at the granularity of register sub-banks, thus restricting their applicability solely to multi-bank register files. In contrast, the proposed technique aims to attain a *temporal* balance for each individual register at the granularity of loop iterations. As temperature takes at least 0.1 million cycles to rise by 0.1 $°C$ [81], this iteratively balanced access activity thus enables an effective reduction of peak temperature even for single-bank register files.

As the proposed technique deterministically shuffles register mapping across iterations, the attainable benefits in terms of reliability enhancement are maximized when it is applied to single processor architectures with no explicit register renaming support. For architectures with pure dynamic register renaming, such as conventional superscalar processors, a large hardware mapping table needs to be maintained so as to eliminate pseudo register name dependences. As this mapping table needs to be accessed using logical register names at a frequency no lower than that of register file accesses, it becomes a temperature “hotspot” with skewed access distribution. In this case, the proposed technique can be employed to evenly distribute the accesses to different entries within the mapping table.

Additionally, future computer systems are expected to intensively use multicore architectures, for which thermal induced reliability aspects have already been identified as a grand challenge. As such systems typically scale upwards in the number of cores but not necessarily in the complexity of each core, the proposed technique despite the possible

absence of the renaming logic, can be employed to effectively reduce the register file peak temperature for each core and hence, to improve the reliability of the entire system.

7.3 Implementation

The implementation of the proposed deterministic register shuffling technique consists of two collaborative parts, a compilation procedure that embeds regularity into static register names, as well as a hardware implementation of a shuffling function that dynamically determines the name of a register at each iteration.

7.3.1 Static register name adjustment

Algorithm 5 Register Name Adjustment

1: **for** each procedure **do**
2: **for** each frequently executed loop **do**
3: Differentiate *shiftable* and *unshiftable* registers;
4: Calculate AveAccessCnt;
5: **if** AccessCnt$(R_j) >$ AveAccessCnt for a *unshiftable* R_j **then**
6: Insert an extra *mov* to make R_j shiftable;
7: **end if**
8: Insert extra *store* and *load* to free up *not-accessed* yet *live* registers;
9: $T = N_{total} - N_{unshiftable}$, and select B and O thereafter;
10: Order the shiftable destination registers;
11: Globally adjust register names such that the static name of the kth register $SN(R_k) = B^k_{O'}$;
12: Shuffle the name of each *live-in* variable by an extra offset O';
13: Insert a hint in the loop branch;
14: Insert extra *mov* to initialize *live-in* variables at loop entry and restore *live-out* variables at loop exit;
15: **end for**
16: Globally perform register coalescing outside the renamed loops to eliminate redundant *mov* instructions;
17: **end for**

The pseudo-code for adjusting logic register names is outlined in Algorithm 5. This procedure only relies on the profiling information regarding the *execution counts* of each basic block, based on which a set of functions have been developed to accomplish static register name adjustment. Specifically, each frequently executed loop is transformed in the following 5 steps:

- Partition shiftable and unshiftable registers (lines 3-7);
- Free up *not-accessed yet live* registers (line 8);
- Determine shuffle functions (line 9);
- Sequentially adjust names of destination and source registers (lines 10-13);
- Initialize *live-in* variables and restore *live-out* variables at loop entry and loop exit, respectively (line 14);

As the goal of the register shuffling technique is to preclude local heat buildup through iterative mapping of a hot logical register to distinct physical registers, the algorithm inserts extra *move* instructions to shuffle a frequently accessed register (line 6), if it is detected to be unshiftable (line 5). These *move* instructions, together with the *store* and *load* instructions inserted for freeing up *not-accessed yet live* registers (line 8) and the *move* instructions inserted for *live-in* or *live-out* variables (line 14), constitute the overhead of the proposed technique. As most of these extra instructions are executed outside the loop body, the overhead in execution time is negligible. Such overhead can be further reduced through performing an extra step of register coalescing [30] on the transformed code (line 16) so as to eliminate redundant move instructions.

7.3.2 Dynamic register name shuffling

Using the code transformation support outlined in Algorithm 5, a deterministic register shuffling process can be accomplished during execution, as long as the hardware is informed by the compiler about the *shuffle vector*, <B, O, T>, prior to entering a frequently executed loop.

Using the $GF(2^3)$ *multiplication* in Table 7.1 as the shuffle function, the circuit presented in Figure 7.6 can be employed to convert logical register names to physical register indices for the *bzip2* example. As can be seen, during loop execution each logical

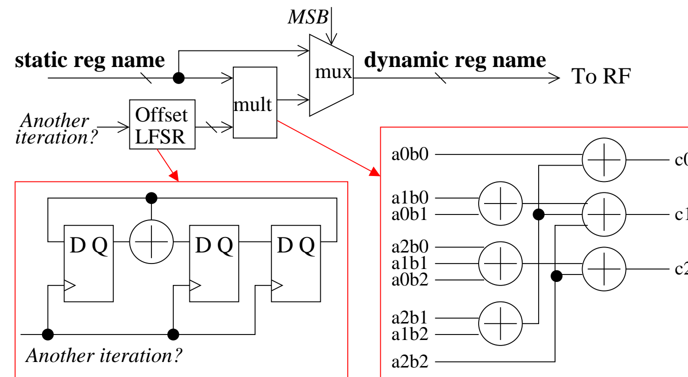


Figure 7.6: Gate-level logic for translating register names

register name is multiplied by the value of the offset register. The offset value is multiplied by 2 whenever a loop branch is encountered, implemented through shifting the 3-bit LFSR one bit to the right. Meanwhile, as in this example the shuffle window is composed of registers from R1 to R7, the most significant bits of the static encoded register name are used to differentiate whether the register falls within the shuffle window. If it is, the register is shiftable, resulting in the use of the multiplier's result as the physical register index. Otherwise, the logical name of the unshiftable register is directly used to access the register file.

It needs to be noted that the implementation shown in Figure 7.6 corresponds to the example shown in Figure 7.3. The implementation parameters are for illustrative purposes only, and can be customized according to the register utilization characteristic of the application. More concretely, it can be clearly seen from Figure 7.6 that the proposed register shuffling technique requires no hardware mapping table but only a n-bit *GF* multiplier and a n-bit 2-to-1 multiplexer for each register access port, together with a single n-bit LFSR to record the shuffling offset. Moreover, with the selection of an appropriate field generating polynomial, GF multipliers can be efficiently implemented using a small set of AND and XOR gates. For 3 to 7 bits parallel field multipliers, the cost-effective polynomial as well as the total gate count and longest path of the corresponding implementation have been listed in Table 7.3.

The *GF multipliers* shown in Table 7.3 require a size of $2^n - 1$ registers for the shuffle window. In contrast, the *modulo addition* can be employed more generally as the shuffling function for shuffle windows of other sizes. In this case, the proposed register

Table 7.3: The design complexity of GF multipliers

Window size	Field polynomial	Gates	Longest path
7 regs	$x^3 + x + 1$	15	1 AND + 2 XOR
15 regs	$x^4 + x + 1$	25	1 AND + 2 XOR
31 regs	$x^5 + x^2 + 1$	37	1 AND + 2 XOR
63 regs	$x^6 + x + 1$	54	1 AND + 3 XOR
127 regs	$x^7 + x + 1$	72	1 AND + 3 XOR

shuffling technique requires a n-bit modulo adder, a n-bit comparator, and a n-bit 2-to-1 multiplexer for each register access port, together with a single n-bit adder and a n-bit register to calculate and record the shuffling offset. Although this additional necessitated hardware is more complex than the *xor* gate based implementation of the *GF* multipliers, it is still negligibly small compared to the mapping tables used in conventional register renaming techniques.

As both the logic and the physical register names preserve all the true data dependences within the loop body, the behavior of the rest of the pipeline, such as the forwarding logic, would not be affected by the register shuffling process. Moreover, since register write accesses are typically performed at a later pipeline stage, the translation of register names can be performed in parallel with the calculation of the instruction result. Even for register read accesses, the access latency of the small translation logic can also be effectively hidden, since in the typical case cache accesses constitute the longest pipeline stage.

7.4 Simulation Results

In this section we experimentally evaluate the efficacy of the proposed register rotation technique in balancing register accesses, reducing the chip-wide peak temperature, and improving processor reliability. To evaluate the proposed technique for different types of applications, a set of experimental studies have been performed on both the Mediabench [56] and the SPECint 2000 benchmarks.

7.4.1 Register Access Results

The discussions presented in Sections 7.2.4 and 7.2.5 clearly show that the partition of shiftable/unshiftable registers and, hence, the effectiveness of the proposed register shuffling technique are strongly related to register access characteristics. As a result, the first step in our experimental evaluation is the examination, for each loop, of the numbers of *read-only* registers, *conditionally defined* yet *unconditionally referred* registers, and registers *not accessed* in the loop. This is achieved through using ATOM [83] to instrument the assembly code to identify execution hotspots (i.e., frequently executed loops) and to generate register usage profiles. The control flow and register usage information of each loop are analyzed thereafter.

The collected profiling results are presented in Table 7.4. Only the profiling results for the selected SPEC 2000 benchmarks are presented, since these benchmarks exhibit a more balanced register utilization than the Mediabench [56] programs due to their relatively larger working sets. For each benchmark, we report the number of hot loops that have been identified, their occupancy in the total execution time, as well as six sets of register usage data. Table 7.4 lists the maximal, the average, and the minimal number of *not-accessed* registers and *live not-accessed* registers, as well as the maximal and the average number of *read-only*, *hot read-only*, *cond-defined*, and *hot cond-defined* registers. The minimal values are not listed for the last four sets since these values are always 0.

The results regarding the minimal number of *not-accessed* registers indicate that **all** the hot loops identified by ATOM have at least 1 free register, thus clearly confirming the wide applicability of the proposed register shuffling technique. Due to the small code size, the average number of registers accessed within a loop body is less than 9. This highly skewed register utilization clearly confirms the necessity for register shuffling techniques, such as the one we herein propose, so as to deliver a more balanced access distribution. Meanwhile, Table 7.4 also shows that on average a loop contains only 2 *read-only* registers, and only 1 of them needs to be rotated to prevent local heat buildup. The number of *conditionally defined* yet *unconditionally referred* registers is even less, as most hot loops are composed of a limited number of basic blocks. These values clearly confirm that a highly limited number of extra *mov* instructions (less than two on average) would suffice to make this small set of *hot read-only* and *hot conditionally defined* registers shiftable.

Table 7.4: The number of hot loops, their occupancy in execution time, and register usage information

Benchmark		**art**	**bzip2**	**mcf**	**twolf**
Hot loop #		32	60	29	61
Execution time		79.8%	78.3%	64.8%	71.9%
Not-accessed #	max	28	29	29	29
	aver	25.34	22.33	23.66	22.56
	min	14	13	13	1
Live not-accessed #	max	20	13	13	19
	aver	10.47	5	3.76	5.98
	min	0	0	1	0
Read-only #	max	6	6	7	7
	aver	1.91	2.33	2.24	2.02
Hot read-only #	max	4	4	3	3
	aver	1	1.73	1.14	1.08
Cond-defined #	max	1	3	3	3
	aver	0.06	0.17	0.28	0.21
Hot cond-defined #	max	1	1	1	3
	aver	0.06	0.03	0.03	0.11

According to the register usage profiles generated by ATOM, the new register names are statically determined, based on which the SimpleScalar toolset [6] is modified to implement the proposed register shuffling technique on top of an in-order 2-way processor. We furthermore compare the proposed technique with the thermal-aware register reassignment technique [100]. Assuming that the register file is composed of 8 sub-banks, **two** sets of data are reported, namely, the access distribution to each *individual register* and the access distribution to each *sub-bank*. The cumulative ratios of the most frequently accessed registers and sub-banks are shown in Figure 7.7.

As can be seen, for the four SPEC2000 benchmarks, the proposed technique can achieve a more balanced access distribution to each individual register as compared to the thermal-aware register reassignment technique [100]. More concretely, initially 81% to 94% of all the register accesses are mapped to 12 registers which, in a completely balanced case, should only capture $12/32 = 37.5\%$ of total accesses. Using static register reassignment (the top-left quadrant in Figure 7.7), 64% to 80% of total accesses are mapped to 12 registers, while using the proposed register shuffling technique (the top-right quadrant), only 50% to 60% of total accesses are mapped to 12 registers.

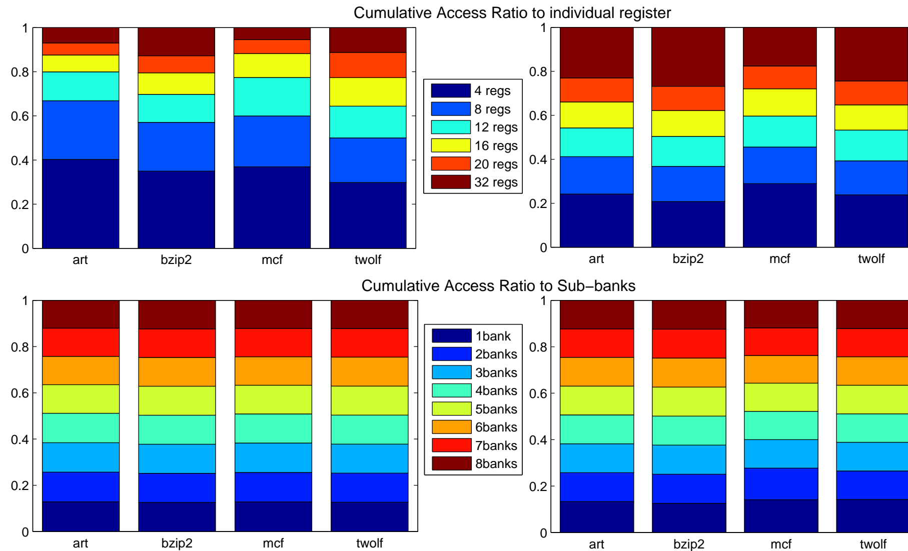


Figure 7.7: Reduction in peak temperature of the entire chip

If the access distribution is evaluated at the granularity of register sub-banks, both techniques can achieve a quite balanced access distribution in that only 38% to 40% of all the register accesses are mapped to 12 registers. Compared to the reassignment [100] technique, the proposed shuffling technique results, for *mcf* and *twolf*, in a slightly elevated amount of accesses (less than 2%) hitting the first subbank. This is because register R0, which cannot be rotated since its value corresponds strictly to 0, happens to be a frequently accessed register within several loop bodies. In the reassignment [100] technique, R0 can be placed into a subbank with a set of “cold” registers so as to balance the access counts to that subbank. However, in the proposed technique, these “cold” registers are iteratively mapped to “hot” logical registers. The increased amount of accesses thus results in the corresponding subbank being accessed slightly more frequently than the remaining subbanks.

7.4.2 Temperature Results

Our next step of evaluation focuses on the generation of temperature profiles. WATTCH [16] is modified to generate energy profiles of each hardware resource, especially each register within the register file. The power consumed by the small 5-bit adder and multiplexer is also included in each register file access. The aggressive clock gating

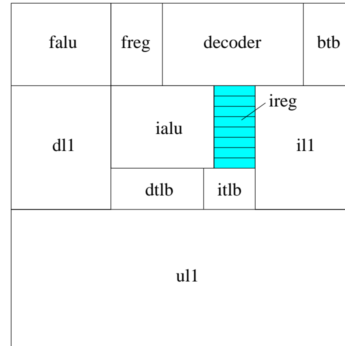


Figure 7.8: The processor floorplan used in simulation

provided by WATTCH is used to avoid unnecessary power consumption. Using these energy profiles, Hotspot [81] is employed to sample the transient temperature of each hardware resource. This sampling interval is set to 20,000 instructions, which is substantially less than the thermal time constant of any hardware resource. An Itanium-like [74] processor shown in Figure 7.8 is used as the floorplan input to Hotspot. The die size is set to $8mm \times 8mm$, and the initial temperature is set to 60$°C$.

The obtained reduction in chip-wide peak temperature is presented in Figure 7.9. As can be seen, the proposed register file access balancing technique can achieve a reduction of 3.1 to 7.4$°C$ in chip-wide peak temperature. The highest reduction is achieved in *art*, while the lowest reduction occurs in *mcf*. These temperature results are consistent with the register access results, since a more balanced access distribution is achieved in *art*, as compared to *mcf*.

The simulation results confirm that by targeting the register file, one of the most overheated hardware units in a processor, the proposed technique can effectively reduce the chip-wide peak temperature during program execution. While the amount of temperature reduction seems to be insignificant at first sight, it actually can effectively reduce the fault rate of the entire chip, since the fault rate doubles for every 10$°C$ increase in temperature [55]. Meanwhile, previous studies have shown that a large number of delay violations would occur if the peak temperature exceeds 85$°C$ [34, 81]. It can be seen from the results that for most benchmarks, the proposed algorithm can effectively reduce the peak temperature to below 84$°C$. On average, the proposed technique achieves a reduction from 88.8$°C$ to 83.5$°C$.

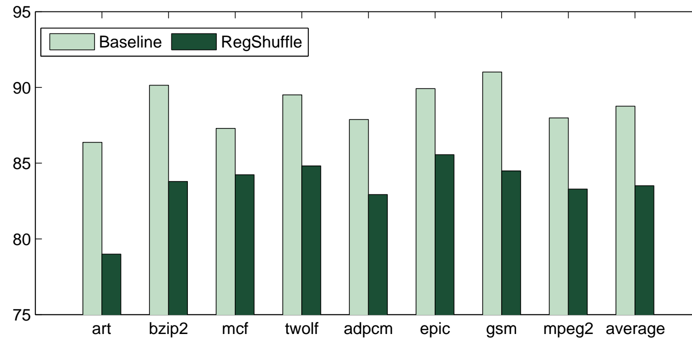


Figure 7.9: Reduction in peak temperature of the entire chip

7.5 Conclusions

In this chapter, we have presented a technique for improving the reliability of an entire chip, through reductions in the peak temperature of the register file, one of the most overheated modules in a core. Peak temperature can be effectively controlled through a register shuffling process that physically remaps the heavily accessed logical registers before heat gets locally accumulated. Furthermore, through the exploitation of application-specific access profiles, the compiler can deterministically control the register shuffling process, thus maximizing peak power reduction within a limited hardware budget and negligible performance degradation. This highly reduced hardware complexity enables the proposed technique to be easily incorporated into most embedded processors so as to effectively reduce peak temperature of the entire chip. Simulation results of SPEC2000 and mediabench programs furthermore confirm that the proposed register shuffling technique can achieve a 1.5 to 3 times more balanced access distribution and a reduction of 3.1 to 7.4°C in chip-wide peak temperature. Such a temperature reduction in turn effectively reduces the amount of run-time faults, thus improving the reliability of the entire chip.

The text of Chapter 7, is in part a reprint of the material as it appears in *C. Yang and A. Orailoglu, "Processor Reliability Enhancement through Compiler-Directed Register File Peak Temperature Reduction," International Conference on Dependable Systems and Networks (DSN), June 2009*. The dissertation author was the primary researcher and author of the publication [96].

Chapter 8

Conclusions

As devices scale beyond deep sub-micron while the number of cores on a single chip doubles every two generations, the capability of tolerating execution uncertainty induced by execution faults, thermal stress, and resource competitions is becoming a severe requirement for future multi-core and many-core systems. These sources of uncertainty demand flexible ways to reorganize the computation, which is addressed in this thesis through the introduction of a computational framework with fine-grained and predictable *execution adaptivity* support. In this framework, computational resources can be frequently *renegotiated* upon a dynamic, unpredictable event, with predictable execution migration attained through statically capturing a set of possible schedules in a compact form. Sources of execution uncertainty that display a certain amount of predictability, such as thermal stress, are maximally *mitigated*. Sources of execution uncertainty that are completely unpredictable, such as device failures, can be efficiently *detected* regardless of their diversity in manifestation. These techniques are furthermore integrated into a scalable, fixed-silicon, yet dynamically reprogrammable MPSoC platform, thus providing the benefit of high-volume amortization while at the same time delivering flexible redefinitions of the platform.

To attain execution adaptivity in conjunction with the goals that designers already face, the various techniques presented in this thesis are furthermore developed with the considerations of minimizing power and performance impact, ensuring high predictability of worst-case performance, and localizing communication and migration for the satisfaction of interconnect constraints. The employment of the compiler to compactly engender

in readiness numerous execution schedules enables the development of a fast, predictable, and highly regular reconfiguration process, without any runtime decision making overhead. The regularity inherent in the reconfiguration process, in conjunction with a flexible customization of the underlying system topology, furthermore enables multiple tasks to be simultaneously migrated between distinct PEs without inducing any interference or network congestion. Meanwhile, through the use of a shared, fault-tolerant cache, full fault detection capability is attained within a minimum level of hardware duplication, with no reliance on threads to constantly synchronize for value-checking. Additionally, the chip-wide peak temperature can be effectively reduced through a deterministic shuffling of the accesses to the most overheated module in each core, to wit, the register file. The employment of the compiler to embed regularity into register names enables the hardware to redirect register accesses with no reliance on a mapping table.

The fundamental intellectual merit of this thesis is a novel approach for coupling intensive static information extraction to dynamic system control and organization. The various techniques underpinning the proposed adaptive framework relies on the development of a collaborative optimization between the compiler, the OS, and the architecture. The compiler is responsible for embedding regularity into static register names, generating adaptive execution schedules with the consideration of temperature and workload balance, and extracting the characteristics of the reconfigurable schedule to guide dynamic workload balance. The OS is responsible for monitoring resource availability, dispatching pre-optimized application schedules to cores, and globally adjusting application resource footprints to prevent potential thermal stress and resource competitions. Finally, architectural support is needed for extending the cache design to perform light-weight fault detection and checkpointing, extending the register file design to perform deterministic register shuffling, and reorganizing the system topology to locally share storage units among cores, so as to mitigate task migration overhead and accelerate neighborhood-centered communications.

The thesis delivers not only theoretical breakthroughs but also practical solutions to current and future multicore systems. For instance, the compiler-directed execution reconfiguration framework is particularly suitable for adoption by the IBM CELL [48] architecture. This architecture contains a power processor element (PPE) that can be used to control the execution of a set of specialized coprocessors called synergistic pro-

cessor elements (SPEs). At runtime, the PPE can be employed as a hardware resource manager, to ascertain the operational health of each SPE, and signal a reconfiguration process to deactivate or reactivate an SPE during execution. These functions can be accomplished in a highly regular and extremely efficient manner, through the development of post-compilation scheduling tools for the CELL compiler to generate execution schedules for the SPEs, with various degrees of reconfiguration steps embedded. In addition, the outlined register shuffling technique is also particularly suitable for the SPEs. In the CELL architecture, each SPE contains a large register file, while the data-intensive nature of the workload induces highly asymmetric register utilization. Compiler optimizations can therefore be developed for the CELL architecture to embed regularity into the register names. As the execution in various SPEs is synchronized in a single-instruction-multiple-data manner, all the SPEs can benefit from a single shuffling of the register names, thus delivering maximum heat reduction benefits.

In sum, the successful completion of the fine-grained and predictable adaptive multicore platform that I have proposed herein, I believe, will engender *adaptive, scalable architectures that can seamlessly reshape execution paths and schedules in an amortizable, high-volume, fixed-silicon fabric*, thus providing avenues for effectively addressing thermal buildups, possible fault occurrences and even resource competition among multiple applications executing simultaneously.

8.1 Future Work Directions

As process technologies continue to evolve, the issue of execution uncertainty is exacerbated as we negotiate the end of the CMOS era and move onto the world of nanoelectronics. The thesis work opens up multiple directions for future explorations on the design of many-core systems with aggressive yet predictable execution adaptivity support.

The exploitation of *compiler* optimizations and *on-line testing* techniques enables further reductions in run-time decision-making overhead. In particular, the various types of device unreliability impose a crucial obstacle in multicore systems, namely, ambiguity in fault manifestation rates and in fault types. While the proposed fault detection technique can uniformly detect these various types of faults, precise identification of the faulty

component and the fault type still requires the use of on-line testing and diagnosis techniques. Additionally, application information regarding execution invariants can be used for property checking which may provide precise identification for system integrity in a cost-effective manner.

Interactions with VLSI design and OSs can be exploited to further improve the determinism of the overall system. VLSI design techniques can significantly impact various aspects of the proposed adaptive multicore platform, including cost-effective heat removal, design testability and reliability enhancement, and the achievement of execution predictability in the face of device variability. Real-time OSs determine the efficiency of design and maintenance of real-time systems for which determinism and responsiveness are important product requirements. Based on statically extracted information regarding variations in resource requirements, real-time OSs can attain resource reallocation within a small and predictable timing overhead. Additionally, hardware components that would facilitate the OS to efficiently dispatch the statically generated schedule blocks to a cluster of cores will be a valuable direction for future research.

Finally, as CMOS scaling is approaching its physical limits, nanotechnology has been widely acknowledged as the foundation for the next generation of computer systems. Yet the level of execution uncertainty in nanoelectronic systems is ever higher, since the fabrication process in nano environments is prone to defects due to the small scale of devices and the bottom-up self-assembly process. The issues of fault detection, execution reconfiguration and communication cost reduction should be addressed, however, with consideration of the particular characteristics of such systems. One characteristic to consider is the expected high variance in the fault rate, which in turn leads to significant differences in performance, robustness, as well as noise immunity among the devices. This clustering behavior should be considered in the development of fault tolerance approaches, together with other effects such as transient/permanent characteristics, temperature-induced fault rate increases, and testing-induced device damage. Another characteristic to consider is the strict interconnect constraint, which forces localized communication to become a critical criterion. Efficient topology and structure for such nanoelectronic systems, together with power-aware and reliable ways to communicate data across the chip, constitute significant obstacles that need to be overcome.

In sum, the design of an adaptive computational fabric capable of responding to the uncertainty challenges expected in the late CMOS, early nanoelectronic era creates a highly exciting research dimension. A complete realization of such a fully adaptive system requires the exploration of interactions between architectures and various other disciplines of computer science, computer engineering, and electrical engineering. I am eager to carry out my future research work in pushing the development of this new area, with extensive communication and collaboration with the research experts across multiple disciplines and universities.

Bibliography

[1] International Technology Roadmap for Semiconductors (ITRS) 2009 edition: Executive summary.

[2] International Technology Roadmap for Semiconductors (ITRS) 2009 edition: System drivers.

[3] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith. Isolation in commodity multicore processors. *IEEE Trans. Comput.*, 40(6):49–59, 2007.

[4] I. Ahmad and Y.-K. Kwok. On exploiting task duplication in parallel programming scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 9(7):872–892, Aug. 1998.

[5] A. H. Ajami, K. Banerjee, and M. Pedram. Modeling and analysis of nonuniform substrate temperature effects on global ULSI interconnects. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 24(6):849–861, June 2005.

[6] T. Austin, E. Larson, and D.Ernst. Simplescalar: an infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, Feb. 2002.

[7] T. M. Austin. DIVA: a reliable substrate for deep submicron microarchitecture design. In *32nd Intl. Symp. Microarchitecture (MICRO)*, pages 196–207, Nov. 1999.

[8] J. L. Ayala. *Power Estimation and Power Optimization Policies for Processor-Based Systems*. PhD thesis, Technical University of Madrid, 2005.

[9] A. Baniasadi and A. Moshovos. Instruction flow-based front-end throttling for power-aware high-performance processors. In *Proc. Intl. Symp. Low Power Electron. & Design (ISLPED)*, pages 16–21, Aug. 2001.

[10] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. D. Micheli. NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. *IEEE Trans. Parallel Distrib. Syst.*, 16(2):113–129, 2005.

[11] R. Blish and N. Durrant. Semiconductor device reliability failure models. Technical report, International SEMATECH, May 2000.

[12] S. Borkar, T. Karnik, J. Tschanz, A. Keshavarzi, , and V. De. Parameter variations and impact on circuits and microarchitecture. In *45th Design Autom. Conf. (DAC)*, pages 338–342, June 2003.

[13] N. S. Bowen and D. K. Pradhan. Virtual checkpoints: Architecture and performance. *IEEE Trans. Comput.*, 41:516–525, May 1992.

[14] G. Bronevetsky, D. Marques, K. Pingali, P. Szwed, and M. Schulz. Application-level checkpointing for shared memory programs. In *11th Intl. Conf. Archit. Support for Program. Lang. & OSs (ASPLOS)*, pages 235–247, Mar. 2004.

[15] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *7th Intl. Symp. High Perform. Comput. Archit. (HPCA)*, pages 171–182, Jan. 2001.

[16] D. Brooks, V. Tiwari, and M. Martonosi. WATTCH: A framework for architectural-level power analysis and optimizations. In *27th Intl. Symp. Comput. Archit. (ISCA)*, pages 83–94, May 2000.

[17] G. Cao and M. Singhal. On coordinated checkpointing in distributed systems. *IEEE Trans. Parallel Distrib. Syst.*, 9(12):1213–1225, Dec. 1998.

[18] S. Chabridon and E. Gelenbe. Failure detection algorithms for a reliable execution of parallel programs. In *14th Intl. Symp. Reliable Distrib. Syst. (SRDS)*, pages 229–238, Oct. 1995.

[19] V. Chaesson, S. Poledna, and J. Soderberg. The XBW model for dependable real-time systems. In *4thIntl. Conf. Parallel & Distrib. Syst. (ICPADS)*.

[20] P. Chaparro, J. Gonzalez, and A. Gonzalez. Thermal-aware clustered microarchitectures. In *Proc. Intl. Conf. Comput. Design (ICCD)*, pages 48–53, Oct. 2004.

[21] M. Chean and J. A. B. Fortes. The Full-Use-of-Suitable-Spares (FUSS) approach to hardware reconfiguration for fault-tolerant processor arrays. *IEEE Trans. Comput.*, 39(4):564–571, Apr. 1990.

[22] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, 23(4):14–19, July 2003.

[23] A. K. Coskun, T. S. Rosing, K. Whisnant, and K. Gross. Dynamic temperature-aware scheduling for multiprocessor SoCs. *IEEE Trans. VLSI Syst.*, 16(9):1127–1140, Sept. 2008.

[24] J.-L. Cruz, A. Gonzalez, M. Valero, and N. P. Topham. Multiple-banked register file architectures. In *27th Intl. Symp. Comput. Archit. (ISCA)*, pages 316–325, June 2000.

[25] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: Task graphs for free. In *Workshop on Hardware/Software Codesign*, pages 97–101, 1998.

[26] Y. Ding, M. Kandemir, P. Raghavan, and M. J. Irwin. A helper thread based EDP reduction scheme for adapting application execution in CMPs. In *22nd Intl. Conf. Parallel & Distrib. Process. Symp. (IPDPS)*, pages 1–14, Oct. 2008.

[27] P. E. Dodd, M. R. Shaneyfelt, J. R. Schwank, and G. Hash. Neutron-induced latchup in SRAMs at ground level. In *41st Intl. Reliability Physics Symp. Proc. (IRPS)*, pages 51–55, Apr. 2003.

[28] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: a low-power pipeline based on circuit-level timing speculation. In *36th Intl. Symp. Microarchitecture (MICRO)*, pages 7–18, Dec. 2003.

[29] D. J. S. F. A. Bower and S. Ozev. Online diagnosis of hard faults in microprocessors. *ACM Trans. Archit. Code Optim.*, 4(2), 2007.

[30] L. George and A. W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(13):300–324, May 1996.

[31] M. A. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. *IEEE Micro*, 23(6):76–83, Nov. 2003.

[32] M. A. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. In *32th Intl. Symp. Comput. Archit. (ISCA)*, pages 172–183, 2005.

[33] C. Gond, R. Melhem, and R. Gupta. Loop transformations for fault detection in regular loops on massively parallel systems. *IEEE Trans. Parallel Distrib. Syst.*, 7(12):1238–1249, Dec. 1996.

[34] H. Goto, S. Nakamura, and K. Iwasaki. Experimental fault analysis of 1Mb SRAM chips. In *15th Proc. VLSI Test Symp. (VTS)*, pages 31–36, Apr. 1997.

[35] S. Gupta, S. Feng, A. Ansari, J. A. Blome, and S. A. Mahlke. StageNetSlice: a reconfigurable microarchitecture building block for resilient CMP systemss. In *Intl. Conf. Compilers, Archit. & Synthesis for Embedded Syst. (CASES)*, pages 1–10, Oct. 2008.

[36] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, 2000.

[37] J.-M. Helary, A. Mostefaoui, R. H. B. Netzer, and M. Raynal. Preventing useless checkpoints in distributed computations. volume 0, pages 183–190, Oct. 1997.

[38] J. L. Hennessy and D. A. Patterson. *Computer architecture: A quantitative approach (4th edition)*. Morgan Kaufmann Publishers, Jan. 2007.

[39] S. Heo, K. Barr, and K. Asanovic. Reducing power density through activity migration. In *Proc. Intl. Symp. Low Power Electron. & Design (ISLPED)*, pages 217–222, Aug. 2003.

[40] W.-W. Hsieh and T.-T. Hwang. Thermal-aware post compilation for VLIW architectures. In *14th Asia & South Pacific Design Autom. Conf. (ASP-DAC)*, pages 606–611, Jan. 2009.

[41] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M. Stan. Hotspot: A compact thermal modeling methodology for early-stage VLSI design. *IEEE Trans. VLSI Syst.*, 14(5):501–513, May 2006.

[42] D. B. Hunt and P. N. Marinos. A general purpose cache-aided rollback error recovery (CARER) technique. In *Intl. Symp. Fault-Tolerant Computing (FTCS)*, pages 170–175, 1987.

[43] R. Joseph, D. Brooks, and M. Martonosi. Control techniques to eliminate voltage emergencies in high performance processors. In *9th Intl. Symp. High Perform. Comput. Archit. (HPCA)*, pages 79–90, Jan. 2003.

[44] T. T. K. Hashimoto and T. Kikuno. A multiprocessor scheduling algorithm for low overhead fault-tolerance. In *17th Intl. Symp. Reliable Distrib. Syst. (SRDS)*, pages 186–194, Oct. 1998.

[45] W. K. F. K.-L. Wu and J. H. Patel. Error recovery in shared memory multiprocessors using private caches. *IEEE Trans. Parallel Distrib. Syst.*, 1(2):231–240, Apr. 1990.

[46] N. Kandasamy, J. P. Hayes, and B. T. Murray. Transparent recovery from intermittent faults in time-triggered distributed systems. *IEEE Trans. Comput.*, 52(2):113–125, Feb. 2003.

[47] T. Karnik, P. Hazucha, and J. Patel. Characterization of soft errors caused by single event upsets in CMOS processes. *IEEE Trans. Dependable Secure Comput.*, 1(2):128–143, 2004.

[48] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, 2006.

[49] M. Kondo and H. Nakamura. A small, fast and low-power register file by bit-partitioning. In *11th Intl. Symp. High Perform. Comput. Archit. (HPCA)*, pages 40–49, Jan. 2005.

[50] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[51] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The MARS approach. *IEEE Micro*, 9(1):25–40, 1989.

[52] R. Kota and R. Oehler. Horus: large-scale symmetric multiprocessing for Opteron systems. *IEEE Micro*, 25(2):30–40, 2005.

[53] H. Kufluoglu and M. A. Alam. A computational model of NBTI and hot carrier injection time-exponents for MOSFET reliability. *J. Comput. Electron.*, 3(3):165–169, Oct. 2004.

[54] Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 7(6):506–521, June 1996.

[55] C. J. Lasance. Thermally driven reliability issues in microelectronic systems: Status-quo and challenges. *Microelectron. Reliab.*, 43(12):1969–1974, Dec. 2003.

[56] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *30th Intl. Symp. Microarchitecture (MICRO)*, pages 330–335, Dec. 1997.

[57] P. A. Lee, N. Ghani, and K. Heron. A recovery cache for the PDP-11. *IEEE Trans. Comput.*, 29:546–549, 1980.

[58] Z. Ma and F. Catthoor. Scalable performance-energy trade-off exploration of embedded real-time systems on multiprocessor platforms. In *Design Autom. & Test in Europe (DATE)*, pages 1073–1078, Apr. 2006.

[59] M. J. Mack, W. M. Sauer, S. B. Swaney, and B. G. Mealey. IBM Power6 reliability. 51:763–774, Nov. 2007.

[60] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *25th Intl. Symp. Comput. Archit. (ISCA)*, pages 132–141, June 1998.

[61] D. Mosse, R. Melhem, and S. Ghosh. A nonpreemptive real-time scheduler with recovery from transient faults and its implementation. *IEEE Trans. Softw. Eng.*, 29(8):752–767, Aug. 2003.

[62] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *29th Intl. Symp. Comput. Archit. (ISCA)*, pages 99–110, May 2002.

[63] R. Nalluri, R. Garg, and P. R. Panda. Customization of register file banking architecture for low power. In *Intl. Conf. VLSI Design*, pages 239–244, Jan. 2007.

[64] R. H. B. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Trans. Parallel Distrib. Syst.*, 6(2):165–169, Feb. 1995.

[65] N. Oh, P. P. Shirvani, and E. J. McCluskey. Control-flow checking by software signatures. *IEEE Trans. Rel.*, 51(1):111–122, Mar. 2002.

[66] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Trans. Rel.*, 51(1):63–75, Mar. 2002.

[67] C. Panis, U. Hirnschrott, and A. Krall. FSEL – selective predicated execution for a configurable DSP core. In *IEEE Symp. on VLSI*, pages 317–320, Feb. 2004.

[68] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee. Architectural core salvaging in a multi-core processor for hard-error tolerance. In *36th Intl. Symp. Comput. Archit. (ISCA)*, pages 93–104, Jule 2009.

[69] M. D. Powell, M. Gomaa, and T. N. Vijaykumar. Heat-and-run: Leveraging SMT and CMP to manage power density through the operating system. In *11th Intl. Conf. Archit. Support for Program. Lang. & OSs (ASPLOS)*, pages 260–270, Oct. 2004.

[70] P. Ranganathan, S. Adve, and N. P. Jouppi. Performance of image and video processing with general-purpose processors and media ISA extensions. In *26th Intl. Symp. Comput. Archit. (ISCA)*, pages 124–135, May 1999.

[71] V. K. Reddy, S. Parthasarathy, and E. Rotenberg. Understanding prediction-based partial redundant threading for low-overhead, high-coverage fault tolerance. In *12th Intl. Conf. Archit. Support for Program. Lang. & OSs (ASPLOS)*, pages 83–94, Mar. 2006.

[72] S. K. Reinhardt and S. S. Mukherjee. Transient-fault detection via simultaneous multithreading. In *27th Intl. Symp. Comput. Archit. (ISCA)*, pages 25–36, June 2000.

[73] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: software implemented fault tolerance. In *3rd Intl. Symp. Code Gener. & Optim. (CGO)*, pages 243–254, Mar. 2005.

[74] S. Rusu and G. Singer. The first IA-64 microprocessor. *IEEE J. Solid-State Circuits*, 35(11):1539–1544, Nov. 2000.

[75] R. M. S. Ghosh and D. Mosse. Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *IEEE Trans. Parallel Distrib. Syst.*, 8(3):272–284, Mar. 1997.

[76] S. K. Sahoo, M.-L. Li, P. Ramachandran, S. V. Adve, V. S. Adve, , and Y. Zhou. Using likely program invariants to detect hardware errors. In *Intl. Conf. Dependable Syst. & Netw. (DSN)*, pages 70–79, June 2008.

[77] R. Sangireddy. Reducing rename logic complexity for high-speed and low-power front-end architectures. *IEEE Trans. Comput.*, 55(6):672–685, June 2006.

[78] P. Shivakumar, S. W. Keckler, D. Burger, M. Kistler, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Intl. Conf. Dependable Syst. & Netw. (DSN)*, pages 389–398, June 2002.

[79] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin. Ultra low-cost defect protection for microprocessor pipelines. In *12th Intl. Conf. Archit. Support for Program. Lang. & OSs (ASPLOS)*, pages 73–82, Mar. 2006.

[80] K. Skadron, T. Abdelzaher, and M. Stan. Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management. In *8th Intl. Symp. High Perform. Comput. Archit. (HPCA)*, pages 17–28, Feb. 2002.

[81] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *30th Intl. Symp. Comput. Archit. (ISCA)*, pages 2–12, June 2003.

[82] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *29th Intl. Symp. Comput. Archit. (ISCA)*, pages 123–134, May 2002.

[83] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. Technical report, Western Research Lab, 1994.

[84] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmen, P. Johnson, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The RAW microprocessor: a computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[85] J. S. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architecture. In *16th Intl. Conf. Parallel & Distrib. Process. Symp. (IPDPS)*, pages 27–36, Apr. 2002.

[86] T. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *29th Intl. Symp. Comput. Archit. (ISCA)*, pages 87–98, May 2002.

[87] R. Viswanath, V. Wakharkar, A. Watwe, and V. Lebonheur. Thermal performance challenges from silicon to systems. Technical report, Technology and Manufacturing Group, Intel Corp., 2000.

[88] N. J. Wang and S. J. Patel. ReStore: Symptom-based soft error detection in microprocessors. *IEEE Trans. Dependable Secure Comput.*, 3(3):188–201.

[89] P. M. Wells, K. Chakraborty, and G. S. Sohi. Adapting to intermittent faults in multicore systems. In *13th Intl. Conf. Archit. Support for Program. Lang. & OSs (ASPLOS)*, pages 255–264, Mar. 2008.

[90] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom. The worst-case execution-time problem–overview

of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), Apr. 2008.

[91] A. Wood. Data integrity concepts, features, and technology. White Paper.

[92] C. Xue, Z. Shao, and E. H.-M. Sha. Maximize parallelism minimize overhead for nested loops via loop striping. *J. Sig. Process. Syst.*, 47:153–167, May 2007.

[93] C. Yang and A. Orailoglu. Light-weight synchronization for inter-processor communication acceleration on embedded MPSoCs. In *Intl. Conf. Compilers, Archit. & Synthesis for Embedded Syst. (CASES)*, pages 150–154, Sept. 2007.

[94] C. Yang and A. Orailoglu. Predictable execution adaptivity through embedding dynamic reconfigurability into static MPSoC schedules. In *5th Intl. Conf. HW/SW Codesign & Syst. Synthesis (CODES-ISSS)*, pages 15–20, Sept. 2007.

[95] C. Yang and A. Orailoglu. A light-weight cache-based fault detection and checkpointing scheme for MPSoCs enabling relaxed execution synchronization. In *Intl. Conf. Compilers, Archit. & Synthesis for Embedded Syst. (CASES)*, pages 11–20, Oct. 2008.

[96] C. Yang and A. Orailoglu. Processor reliability enhancement through compiler-directed register file peak temperature reduction. In *Intl. Conf. Dependable Syst. & Netw. (DSN)*, pages 468–477, June 2009.

[97] C. Yang and A. Orailoglu. Towards no-cost adaptive MPSoC static schedules through exploitation of logical-to-physical core mapping latitude. In *Design Autom. & Test in Europe (DATE)*, pages 63–68, Apr. 2009.

[98] C. Yang and A. Orailoglu. Fully adaptive multicore architectures through statically-directed dynamic execution reconfigurations. In *Intl. Conf. VLSI & System-on-Chip (VLSI-SoC)*, Sept. 2010.

[99] J. Yu, M. J. Garzaran, and M. Snir. Efficient software checking for fault tolerance. In *22nd Intl. Conf. Parallel & Distrib. Process. Symp. (IPDPS)*, pages 1–5, Apr. 2008.

[100] X. Zhou, C. Yu, and P. Petrov. Compiler-driven register reassignment for register file power-density and temperature reduction. In *45th Design Autom. Conf. (DAC)*, pages 750–753, June 2008.

[101] D. F. Zucker, R. B. Lee, and M. J. Flynn. Hardware and software cache prefetching techniques for MPEG benchmarks. *IEEE Trans. Circuits Syst. Video Technol.*, 10(5):782–796, Aug. 2000.