ABSTRACT

Title of dissertation: CROSS-LAYER CUSTOMIZATION FOR LOW POWER AND HIGH PERFORMANCE EMBEDDED MULTI-CORE PROCESSORS

Chenjie Yu, Doctor of Philosophy, 2010

Dissertation directed by: Assistant Professor Peter Petrov
Electrical and Computer Engineering

Due to physical limitations and design difficulties, computer processor architecture has shifted to multi-core and even many-core based approaches in recent years. Such architectures provide potentials for sustainable performance scaling into future peta-scale/exa-scale computing platforms, at affordable power budget, design complexity, and verification efforts. To date, multi-core processor products have been replacing uni-core processors in almost every market segment, including embedded systems, general-purpose desktops and laptops, and super computers.

However, many issues still remain with multi-core processor architectures that need to be addressed before their potentials could be fully realized. People in both academia and industry research community are still seeking proper ways to make efficient and effective use of these processors. The issues involve hardware architecture trade-offs, the system software service, the run-time management, and user application design, which demand more research effort into this field.

Due to the architectural specialties with multi-core based computers, a Cross-

Layer Customization framework is proposed in this work, which combines application specific information and system platform features, along with necessary operating system service support, to achieve exceptional power and performance efficiency for targeted multi-core platforms. Several topics are covered with specific optimization goals, including snoop cache coherence protocol, inter-core communication for producer-consumer applications, synchronization mechanisms, and off-chip memory bandwidth limitations.

Analysis of benchmark program execution with conventional mechanisms is made to reveal the overheads in terms of power and performance. Specific customizations are proposed to eliminate such overheads with support from hardware, system software, compiler, and user applications. Experiments show significant improvement on system performance and power efficiency.

CROSS-LAYER CUSTOMIZATION FOR LOW POWER AND HIGH PERFORMANCE EMBEDDED MULTI-CORE PROCESSORS

by

Chenjie Yu

Dissertation to be submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2010

Advisory Committee:
Assistant Professor Peter Petrov, Chair/Advisor
Professor Shuvra Bhattacharyya
Associate Professor Gang Qu
Associate Professor Manoj Franklin
Associate Professor Chau-Wen Tseng

© Copyright by
Chenjie Yu
2010

Acknowledgments

I would like to give my gratitude to all the people who have made this thesis possible and to all the people who have helped me in my Ph.D program.

I would like to thank my advisor, Dr. Peter Petrov for teaching and directing on my study and research. He has lent me tremendous help through the entire processes of all my research projects, from idea forming, experiment implementation, all the way to publications and presentations.

I also want to thank Dr. Shuvra Bhattacharyya, Dr. Gang Qu, Dr. Manoj Franklin, and Dr. Chau-Wen Tseng for serving in my thesis defense committee and for sparing their invaluable time reviewing the manuscript of my dissertation.

I owe my thanks to my colleagues and fellow graduate students at Electrical and Computer Engineering Department. Among them, Dr. Xiangrong Zhou had participated in some of my projects and shared his experience with me.

Most of my research projects use the M5 simulator that originally came from Dr. Donald Yeung's group. Dr. Yeung has also been generous enough to allow me to use their computing facilities for simulations. Things will be far more different without their support.

I owe my deepest thanks to my family - my mother and father. Without their support I couldn't have gone this far.

It is impossible to remember all, and I apologize to those I've inadvertently left out.

Table of Contents

List of Tables vi

List of Figures viii

1 Introduction and Motivation 1
- 1.1 Embedded Systems . . . 1
- 1.2 Embedded system design . . . 2
- 1.3 Embedded Multi-Core Processors . . . 3
- 1.4 Embedded Multi-Core Platform Design Challenges . . . 5
  - 1.4.1 User Application Layer . . . 5
  - 1.4.2 System Software Layer . . . 6
  - 1.4.3 Hardware Layer . . . 6
- 1.5 System-Level Customization in Embedded Systems . . . 7
- 1.6 Dissertation Outline . . . 11

2 Background and Related Work 12
- 2.1 Embedded Systems . . . 12
- 2.2 Power-saving Techniques for Embedded Systems . . . 13
  - 2.2.1 DVFS and Clock-Gating . . . 13
  - 2.2.2 Architecture Level Power Reduction . . . 15
  - 2.2.3 System Software Techniques for Power Saving . . . 16
- 2.3 Cross-Layer Customization for Embedded Multi-Cores . . . 17
  - 2.3.1 Cross-Layer Customization Approach for Embedded Systems . 17
  - 2.3.2 Improving Snoop Protocol Power Efficiency . . . 19
  - 2.3.3 Inter-Core Communication based on Producer-Consumer Patterns . . . 20
  - 2.3.4 Hardware Based Synchronization Mechanisms . . . 20
  - 2.3.5 Cache Partitioning for Memory Bandwidth Minimization . . . 21

3 Low-Power Snoop Architecture for Synchronized Producer-Consumer Communication in Embedded Multiprocessors 22
- 3.1 Overview . . . 22
- 3.2 Related Work . . . 23
- 3.3 Functional Overview . . . 26
  - 3.3.1 Synchronized Producer-Consumer Communication . . . 28
  - 3.3.2 Snoop-Phases in Producer-Consumer Communication . . . 29
  - 3.3.3 Snoop-Phase Detection . . . 32
  - 3.3.4 Shared Buffer Identification . . . 34
- 3.4 Passive SPoT Detection . . . 36
- 3.5 Active SPoT Migration . . . 40
- 3.6 Experimental results . . . 43

4 Energy and Performance Efficient Communication Framework for Embedded MPSoCs through Application-Driven Release Consistency 57
4.1 Overview . . . 57
4.2 Related Work . . . 63
4.3 Motivation and Overview . . . 66
4.3.1 Inter-Core Data Sharing: To Invalidate or to Update? . . . 67
4.3.2 Cross-Layer Integration for Data Communication . . . 69
4.3.3 Cache Way Partitioning for Low-Power Data Sharing . . . 74
4.4 Compiler Support . . . 76
4.4.1 Shared Memory Identification . . . 77
4.4.2 Loop Transformations for Software-triggered Remote Updates 78
4.5 System Software Support . . . 84
4.5.1 Memory Reference Identification . . . 84
4.5.2 Multi-Tasking Support and False Sharing Avoidance . . . 86
4.6 Cache Partitioning for Low-Power Data Sharing . . . 88
4.6.1 Functional Overview . . . 89
4.6.2 Cache Way Partitioning: Advantages and Pitfalls . . . 90
4.7 Hardware Support . . . 92
4.7.1 Shared Data Communication Support . . . 93
4.7.2 Cache Way Partitioning Hardware . . . 96
4.8 Experimental Results . . . 97
4.9 Conclusion . . . 120

5 Low-Cost and Energy-Efficient Distributed Synchronization for Embedded Multiprocessors 131
5.1 Overview . . . 131
5.2 Related work . . . 135
5.3 Functional Overview . . . 139
5.3.1 Distributed Queue Abstraction Model . . . 142
5.3.2 Synchronization Efficiency with Distributed Queues . . . 144
5.4 System Architecture . . . 147
5.4.1 Synchronization Variable Identification . . . 147
5.4.2 Distributed Synchronization Controller . . . 150
5.4.3 Lock implementation . . . 153
5.4.4 Barrier implementation . . . 155
5.4.5 Power Management . . . 158
5.5 Compiler and OS Support . . . 160
5.5.1 OS Power Management Role . . . 162
5.5.2 Multi-tasking support per core . . . 165
5.6 Experimental results . . . 165
5.7 Conclusions . . . 181

6 Off-Chip Memory Bandwidth Minimization through Cache Partitioning for Multi-Core Processors 182
6.1 Overview . . . 182
6.2 Related work . . . 185
6.3 Memory Bandwidth and Last Level Caches in Multi-Core Systems . . 187
6.3.1 Bandwidth Demand and Cache Resources . . . 188
6.3.2 Cache Sharing in Multi-Core Processor Systems . . . 191
6.4 Partitioning Algorithm . . . 192
6.4.1 Cache Partitioning Basics . . . 192
6.4.2 Algorithm Overview . . . 193
6.4.3 Intuitive and Formal Description . . . 196
6.5 Experimental Results and Discussion . . . 199
6.5.1 Experimental Setup . . . 199
6.5.2 Benchmark Applications . . . 200
6.5.3 Results and Analysis . . . 201
6.5.4 Comparison Between Heuristic Algorithm and Exhaustive Search 203
6.6 Conclusions . . . 204

7 Conclusions 206
7.1 Embedded Multi-Core Architecture Challenges . . . 206
7.2 Cross-Layer Customization for Embedded Multi-Cores . . . 207

Bibliography 208

List of Tables

3.1 Snoop-induced cache lookups for 16K shared data buffers; Passive SPoT v.s. baseline snoop protocol. . . . 46
3.2 Snoop-induced cache lookups for 64K shared data buffers; Passive SPoT v.s. baseline snoop protocol. . . . 47
3.3 Energy consumption (μJ) for 16K shared data buffers; Passive SPoT v.s. Baseline 48
3.4 Energy consumption (μJ) for 64K shared data buffers; Passive SPoT v.s. Baseline 49
3.5 Snoop-induced cache lookups for 16K shared data buffers; Active SPoT v.s. baseline snoop protocol. . . . 53
3.6 Snoop-induced cache lookups for 64K shared data buffers; Active SPoT v.s. baseline snoop protocol. . . . 54
3.7 Energy consumption (μJ) for 16K shared data buffers; Active SPoT v.s. Baseline 55
3.8 Energy consumption (μJ) for 64K shared data buffers; Active SPoT v.s. Baseline 56

4.1 Cache Misses: Baseline vs. Achieved Reductions (%); 32K D-Caches . 98
4.2 Cache Misses: Baseline vs. Achieved Reductions (%); 32K D-Caches (continued) 99
4.3 Cache Misses: Baseline vs. Achieved Reductions (%); 32K D-Caches (continued) 100
4.4 Cache Misses: Baseline vs. Achieved Reductions (%); 64K D-Caches . 101
4.5 Cache Misses: Baseline vs. Achieved Reductions (%); 64K D-Caches (continued) 102
4.6 Cache Misses: Baseline vs. Achieved Reductions (%); 64K D-Caches (continued) 103
4.7 Bus Transactions: Baseline vs. Achieved Reductions (%); 32K D-Caches 104
4.8 Bus Transactions: Baseline vs. Achieved Reductions (%); 32K D-Caches (continued) 105
4.9 Bus Transactions: Baseline vs. Achieved Reductions (%); 32K D-Caches (continued) 106
4.10 Bus Transactions: Baseline vs. Achieved Reductions (%); 64K D-Caches 107
4.11 Bus Transactions: Baseline vs. Achieved Reductions (%); 64K D-Caches (continued) 108
4.12 Bus Transactions: Baseline vs. Achieved Reductions (%); 64K D-Caches (continued) 109
4.13 Cache way partitioning: Cache energy (mJ) and reductions 121
4.14 Cache way partitioning: Cache energy (mJ) and reductions (continued) 122
4.15 Cache way partitioning: Cache misses and impact on miss-rate 123

4.16 Cache way partitioning: Cache misses and impact on miss-rate (continued) . . . 124
4.17 Average memory access latency reduction (32K D-Cache) . . . 125
4.18 Average memory access latency reduction (32K D-Cache) (continued) 126
4.19 Average memory access latency reduction (64K D-Cache) . . . 127
4.20 Average memory access latency reduction (64K D-Cache) (continued) 128
4.21 Average memory access latency reduction with cache way allocation . 129
4.22 Average memory access latency reduction with cache way allocation (continued) . . . 130

5.1 Performance characteristics (in number of cycles) and DSC reductions - Increasing data set . . . 167
5.2 Performance characteristics (in number of cycles) and DSC reductions - Increasing data set (continued) . . . 167
5.3 Bus bandwidth characteristics (in number of bus transactions) and DSC reductions - Increasing data set . . . 168
5.4 Bus bandwidth characteristics (in number of bus transactions) and DSC reductions - Increasing data set (continued) . . . 168
5.5 Performance characteristics (in number of cycles) and DSC reductions - Fixed computational workload . . . 171
5.6 Performance characteristics (in number of cycles) and DSC reductions - Fixed computational workload (continued) . . . 172
5.7 Bus bandwidth characteristics (in number of bus transactions) and DSC reductions - Fixed computational workload . . . 173
5.8 Bus bandwidth characteristics (in number of bus transactions) and DSC reductions - Fixed computational workload (continued) . . . 174
5.9 Thread load imbalance: 4-processor system . . . 177
5.10 Energy characteristics: 4-processor system . . . 178
5.11 Thread load imbalance: 8-processor systems . . . 180
5.12 Energy characteristics: 8-processor system . . . 181

6.1 Benchmark Workloads . . . 199

List of Figures

1.1 General Purpose System . . . 3
1.2 Feed-Back and Customization Desig . . . 10

3.1 Synchronized producer-consumer communication with shared memory. At any time moment access to the shared buffer is exclusive required by the producer or the consumer only. . . . 26
3.2 Producer-Consumer cache snooping activities. . . . 29
3.3 Transferring to OS shared buffers information . . . 35
3.4 Hardware architecture for SPoT detection. . . . 36
3.5 Hardware architecture for Active SPoT migration. . . . 41
3.6 Application benchmarks organization. . . . 43
3.7 Energy reduction for 16K shared data buffers . . . 50
3.8 Energy reduction for 64K shared data buffers . . . 51
3.9 Active vs. Passive: Energy reductions for 16K shared data buffers . . . 55
3.10 Active vs Passive: Energy reduction for 64K shared data buffers . . . 56

4.1 Shared memory multiprocessor organization . . . 66
4.2 Bus transactions involved in communicating data . . . 68
4.3 Synchronized inter-task communication . . . 71
4.4 Propagating updates with explicit store.update . . . 73
4.5 MPSoC cache partitioning . . . 75
4.6 Transformations for row-wise array traversal with *st.update* support . . . 79
4.7 Transformations for row-wise traversal with “irregular” row sizes . . . 79
4.8 Transformations for column-wise traversal with *st.update* support . . . 81
4.9 Loop peeling for *st.update* support . . . 81
4.10 Transformation for *while* loops with unknown at compile-time upper bounds . . . 82
4.11 Cache controller support for bus-based systems . . . 93
4.12 Shared/Private-data cache way allocation architecture . . . 95
4.13 Application benchmarks organization. . . . 97

5.1 Distributed lock queue information . . . 142
5.2 Local lock queue management . . . 144
5.3 Distributed Synchronization Controller (DSC) organization . . . 152
5.4 Local barrier queue management . . . 155
5.5 Overall system organization . . . 157
5.6 Example parallel application . . . 161
5.7 Data-streaming benchmarks organization. . . . 166

6.1 Memory Bandwidth Requirement Curves . . . 188
6.2 Cache Misses Curves . . . 189
6.3 Cache Miss-rate Curves . . . 190
6.4 Partitioning Heuristic Pseudocode . . . 194

6.5 Algorithm Walkthrough on Example 198
6.6 Achieved bandwidth v.s. baseline: APP1, APP2 and APP3 201
6.7 Achieved bandwidth v.s. baseline: APP4, APP5 and APP6 201
6.8 Heuristic Algorithm Compared to Exhaustive Search 204

Chapter 1

Introduction and Motivation

1.1 Embedded Systems

Embedded systems have a wide range of applications. The products span from day-to-day household and consumer electronics, such as microwave ovens, digital TVs, set-top boxes, mobile phones, PDAs, vehicular devices, etc, to industry devices and equipments like wireless communication basestations, robots, aviation equipments, to the high end military and scientific devices like missles and devices used in space missions. It is estimated that over 10.76 billion embedded systems/devices were shipped worldwide in 2009[5].

Because of its wide application range, there is no dominating system architecture for embedded systems. The very simple microcontrollers, such as Atmel's 8051, AVR[56], and the advanced advanced massive parallel architecture, such as PC102 from picoChip[3] and Tile64 from Tilera[4] co-exist in today's design choices. About 50% of embedded system products shipped use no formal or in-house operating systems, whereas the other half use commercial or open-source operating systems. In general, however, embedded system designs are moving from board level to System-On-Chip (SOC), and from single processor to multiprocessors[49] with more powerful operating systems, which is driven by increasing demand for performance and power efficiency.

1.2 Embedded system design

In general, embedded system design features fast time-to-market, very low cost, strict performance specification and tight power budget. Embedded system designers have to make trade-offs between all those factors and fine-tune the system at hardware architecture, operating system, and application software layers.

System function partitioning is very important in embedded systems. Application functions can be implemented in either hardware or software. Using pure hardware design, such as custom ASIC (Application Specific IC), gives the best performance, hardware cost, and power efficiency, but suffers from long time to market and excessive design cost and little flexibility for future product upgrades. Using pure software running on a general-purpose computer system, however, exchanges performance, cost, and power efficiency to time to market and small design cost and more flexibility. A designer needs to balance between the two extremes.

Most micro-processor based designs follow the layered system model borrowed from the general-purpose domain[108], as show in Figure 1.1. The hardware layer sits at the very bottom, including the processor unit, the off-chip memory, and peripherals. The operating system layer lies in the middle and directly controls and manages the hardware resources and provides services for user applications. The application layer consists of user programs that achieve the desired functionalities. Off-the-shelf hardware components are used to shorten the time-to market. The operating system is also chosen from open-source or commercial available ones.

However, to meet design goals, an embedded system design needs to go through

Figure 1.1: General Purpose System

a series of customization processes to achieve the design specifications at the lowest cost. This is achieved by system level hardware-software co-design method, which tries to meet system level objectives by exploiting the synergism of hardware and software through the design process. Thus, the designers need to be knowledgeable in both hardware and software domains to make good trade-offs.

1.3 Embedded Multi-Core Processors

In recent years, the scaling of silicon fabrication technologies has enabled unprecedented level of chip integration, which provides rich on-chip resource for powerful processor designs. However, the traditional design methodology with monolithic single core architectures has met tremendous difficulties in frequency scaling, power consumption, design complexity, verification effort, and so on.

The frequency scaling stops around 3GHz and has not advanced ever since. Yet there is constant need for more powerful processors with affordable energy consumption and cost. Such conflicts have led to the proliferation of multi-core processor architectures, which naturally address many of these problems. With multiple but simpler cores, running at lower frequencies, the chip voltage and power consumption are well contained, while achieving very good performance scaling. As a result,

multi-core processor products have been widely adopted in today's computing systems – including the industrial embedded system applications, the general purpose desktops and servers, and the building blocks for supercomputers. In fact, embedded systems are among the first to benefit from multi-core based designs[1].

Although the market has seen flourish of multi-core processor products, the proper use of such architectures is still far from satisfactory and is thus under intensive study in both academia and industry research communities. The fundamentals of the hardware architectures are evolving into different directions, including the key topologies such as on-chip memory structures, coherence mechanisms, interconnect technologies, etc. The proper system software mechanisms that manage the hardware resrouce and task scheduling are being explored to meet the future many-core processor demands. Even the programming paradigms for multi-core/many core architectures are also evolving into varieties, including OpenMP, MPI, pthread, Thread Building Block(TBB), and many others.

Although embedded systems have an early start with multi-core processors, because of the wide spectrum of embedded system applications and areas, as well as vast variation of implementation methods, the problems are even more profound and complex with embedded multi-core systems.

The fact that so little has been settled on so many issues with multi-core architectures, which have already been in commercial use for about seven years, is urging much more involvement from researchers in essentially all sub-areas of computer engineering.

1.4 Embedded Multi-Core Platform Design Challenges

As discussed in the previous section, while multi-core architectures help keep system performance scaling at affordable hardware, power, and design cost, they also bring a large number of challenges and a whole new world of design trade-offs. The challenges span in about every sub-area of the systems and are shared by both embedded system and general purpose system designs.

1.4.1 User Application Layer

There are programming challenges at user application layer, which have attracted a large number of attention and research interest in both academia and industry. There has been significant investment in the topics of compiler auto-parallelization and code transformation techniques, as well as research works in how to do programming for these highly integrated paralle machines. There is also significant industry interest in techniques that could help legacy code benefit from future multi-core platforms. For embedded systems, an important topic is to maintain hard real-time constraints with significantly higher system complexity. Currently, there is still no sight of any solution in these areas that present itself general enough and effective enough.

The current de facto standard of parallel programming on mid/large scale multi-core processors is up to the programmers to parallelize the sequential solutions and perform specific tunings according to hardware and OS. Such effort requires heavy software customization and generally does not provide portable performance

across platforms. Furthermore, as will be shown, even perfectly parallelized applications may suffer execution inefficiencies from the underlying platforms. This is becoming more prominent as system integration scaling to larger core counts.

1.4.2 System Software Layer

There are also special requirements with system software for embedded multi-core architectures, including operating systems and hypervisor software. With increasing number of processor cores and application threads in the system, the role of operating system is becoming more important in task scheduling, resource allocation and partitioning, and architecture specific system services. It is extremely important for system software on a multi-core platform to provide system-level resource monitoring to prevent serialization on certain bottlenecks. There have already been attempts to address such need in academia research and commercial practices[2, 6]. However, such mechanisms are often bound with entirely different programming models and paradigms, which are yet to be accepted by the vast majority of programmers and market. Like user software applications, the system software also needs to be tailored to specific multi-core hardware architectures, with possibly different API interfaces to the programmers.

1.4.3 Hardware Layer

Most of the problems in the application and system software layer have their roots in the hardware architectures. With multi-core architectures, the processor

cores are getting cheaper and simpler, but the shared resources become expensive and the interaction between different cores becomes more frequent and complicated, which bring in a large number of design trade-offs. Current multi-core/many-core hardware architectures are evolving into all kinds of different directions. Vendors like Intel, IBM, Nvidia and Tilera all have drastically different multi-core processors. The differences are often in the fundamental aspects of architecture design, such as being heterogeneous or homogeneous, the interconnect technology, the cache structure, etc. Such differences make the program performance generally not portable across the platforms. Even with the same vendor, the new generations of architectures could also be very different in terms of topology organization, especially as the systems scale into large scale chip multi-processors (CMP). All these have significant impact to system performance and often require significant re-structuring to the existing software stack.

Such challenges must be addressed before the potentials of multi-core/many-core systems can be truely realized in embedded applications. On the other hand, this also means tremendous research opportunity in this area.

1.5 System-Level Customization in Embedded Systems

All embedded systems need to be optimized to specific application needs and purposes, such as power efficiency and performance requirements. For example, a smart phone design may have chosen processors and operating system that promise sufficient computing power as well as media and communication processing capa-

bilities. Yet, it can still fall short of the power budget, which is often times the upmost important factor for such devices to be successful in the market place. It is common practice for embedded system designers to fine-tune every design element and parameters to meet design specifications.

The complexities with embedded multi-core systems argue strongly for a such a cross-layer customization approach.

The customization methods span across different layers of the entire system. Various embedded multi-core processors can be selected for different computing and control needs. A chosen processor platform needs further customization. For example, the processor architecture parameters need to be carefully examined and adjusted to satisfy system demand. The speed of the processors, the register file sizes, the size and way associativity of cache subsystem or scratch pad memory, all have significant impact on system performance and run-time power consumption. Sometimes, such customization may go into micro-architecture level. The designer may need to implement certain critical systems in hardware blocks so as to optimize critical path behavior and add instructions to the ISA [1, 7].

The operating system (OS) also needs significant modification while it is being ported to the target multi-core system. Compared with hardware components which are more often off-the-shelf commercial products, operating systems in embedded systems traditionally have a much greater variety. It is reported that, of all the embedded system products shipped in 2007, more than half of them come with in-house-built operating systems[5], and the other smaller half come with open-source or commercial operating systems such as VxWorks[119], embedded Linux

[121], ThreadX[58] and WinCE[118]. The use of bigger operating systems is growing faster. More often, the layered system structure in Figure 1.1 is violated a little bit to give application software direct access to certain hardware resources, such as IO peripherals. Certain system services, such as scheduling, memory management, need to be changed for more efficient execution. The use of application specific instructions and certain micro-architecture facilities also need operating system support which is not included in standard releases. As embedded multi-core processors scale to large scale CMPs, there is also added risk of violation of traditional symmetrical multiprocessor (SMP) model, which needs special attention for future many-core platforms.

Likewise, user programs also need a large number of tuning to achieve the best performance and power efficiency. This usually comes in the form of compiler/user directed runtime support for specific optimization goals. The compiler or the programmer extracts application information and pass it down to OS and hardware layers, via speccial system APIs. At times, for certain critical data paths, the programmer may need to hand optimize the code at assembly language level. Such process would require knowledge of the underlying hardware and operating system platforms.

All above customization is achieved by a cross-layer system level customization framework in embedded systems. Application specific information is extracted by profiling and code examination. This information is fed to the operating system and hardware design. The outcome of the process can be profiled again, until all system specifications are met. The whole process is usually in a feed-back loop, as shown

in Figure 1.2.

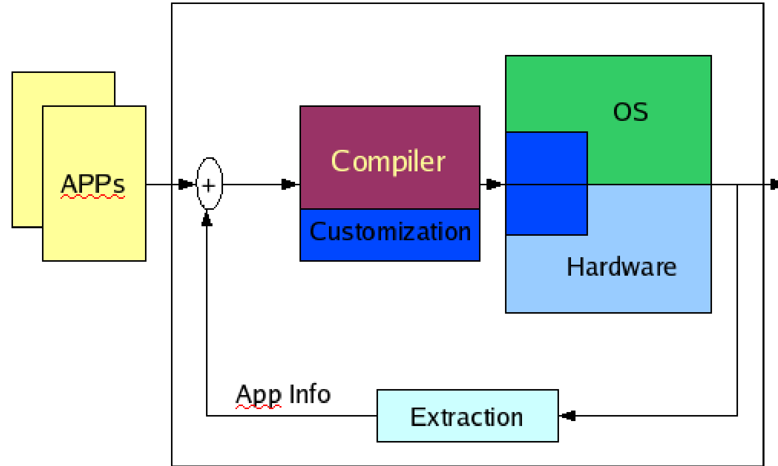

Figure 1.2: Feed-Back and Customization Desig

Such need for system level customization is fundamental to the multi-core architectures. The philosophy is as following: The future processors are not going to be faster, but much wider. The performance gains will be solely from architecture improvement. Thus the application software will have to capture most of the architecture features from the underlying hardware to justy the cost.

The decoupled layer design from general purpose system design method can hardly fit the upcoming large scale multi-core/many-core processors. Even the supercomputer designs are going towards hardware software co-design approaches.

Because of its vast design space, however, the specific co-design techniques of such customization approach need to be explored on specific system architectures and applications. This is where this work would make contribution to.

1.6 Dissertation Outline

This dissertation covers several embedded multi-core platform specific issues, including cache coherence protocols, inter-core communication, synchronization mechanism, and off-chip bandwidth. Most of these techniques also apply for general purpose multi-core platforms, at small to midium scale. In all these topics, benchmark studies on embedded application kernels are performed to reveal the inefficiencies with conventional design methodologies. New approaches are proposed under the cross-layer customization framework to address such inefficiencies, which often bring modifications to multiple system aspects. Extensive experiments are shown that such customization often improve system performance and power efficiency significantly as compared to baseline.

Chapter 2

Background and Related Work

2.1 Embedded Systems

Embedded processors have been reported to occupy more than 90% of the processor market while great amount money and man power have been invested into the research and development work[102]. The major concerns of many embedded systems are power consumption and real-time performance constraints. While performance specification requires worst case real-time guarantee, power consumption has a greater impact on battery life, microprocessor thermal effect, cooling cost, etc. This is especially true for most battery based products like smart phones and portable media players. This research will try to address these problems, from different perspectives of embedded multiprocessor systems, while causing little or no performance and cost overhead.

There have been a large number of research work on power-aware design techniques for computer systems. Many of them apply to both general-purpose and embedded computing systems. Moreover, embedded system design has the unique advantage of known target application and thus can exploit application specific information and achieve very efficient customization and optimization. As for the high-end embedded multiprocessor systems, because of their special architectural properties and requirements, more power saving opportunities are being exposed to

researchers to achieve even greater power reductions.

2.2 Power-saving Techniques for Embedded Systems

Embedded system design has traditionally adopted the layered system structure, including the hardware layer, the operating system layer, and user application layer. Various techniques have been developped to reduce system power consumption at different layers. All power saving techniques are based on the following equation:

$$\mathbf{P} = \frac{1}{2} \cdot C \cdot V_{DD}^2 \cdot f \cdot N + Q_{SC} \cdot V_{DD} \cdot f \cdot N + I_{leak} \cdot V_{DD} \tag{2.1}$$

Where P denotes the total power, V_{DD} is the supply voltage, f is the clock frequency.

The first term represents the dynamic power, where C is the load capacitance and N is the switching activity, i.e., the number of gate output transitions per clock cycle. The second term represents short-circuit power, where Q_{SC} is the quantity of charges carried by the short-circuit current per transition. The last term is static power dissipation due to leakage current I_{leak}.

There are many research work around trying to minimize every variable in this equation so as to reduce system power consumption.

2.2.1 DVFS and Clock-Gating

The DVFS(Dynamic Voltage and Frequency Scaling) method has been a very effective technique in tuning system power consumption at the minimum level. The

key idea is to lower clock frequency and scale down supply voltage when the system is running light weight tasks. It is clear from the above equation that, by lowering clock frequency and supply voltage, one can reduce dynamic switching power significantly. And for embedded systems that go into idle status once in a while, such as cell phone standby mode, this method proves to be very effective. Various DVFS techniques have been both proposed in academia[87, 103] and developed in industry [46, 113].

Although DVFS works well for dynamic power, it has little impact for the static power. As transistor density continues increasing and transistor feature size continues shrinking, static power is gradually becoming the dominant part in total power consumption. This is especially true for high-end multiprocessor systems, which have more transistors on a chip and have larger cache area.

Another technique, called clock-gating, also works on dynamic power. The idea is to temporarily cut off clock signals to certain structures that are inactive for some period of time and resume clock signals later. For example, a processor pipeline may decide to clock-gate part of its function units on seeing certain instructions coming through. This technique usually requires additional gates to control the clock signal of certain units and dynamically chooses to clock-gate on predefined conditions. As clock signal contributes to a very large part of the microprocessor total power consumption, this technique can effectively reduce energy to send the clock signals over the clock tree and reduce the load capacitance on the clock signal drivers. This should have greater effect on high-end embedded multiprocessors. However, clock-gating at coarser grain, such as a cache subsystem as a whole, and bigger structures may suffer from delay in resuming the clock signal, which may potentially harm

real-time performance. Such techniques have been discussed in various papers[23].

Both DVFS and clock-gating techniques are becoming available in newer verisons of microprocessors. Programmers are given the flexibility to adjust higher level policies and protocols to form various power-saving modes. Designers can also customize processors to apply these techniques to more micro-architecture components, depending on application specific requirements.

Such circuit level power saving techniques could also have significant application into future many-core processors. Due to the complexity of such systems, however, DVFS and clock-gating are often applied at coarse granularities, such as per individual cores or groups of cores.

Finding appropriate transistor size [106] or redesigning complex gate[90] will change the load capacity or reduce the total number of transistor count, thus reduce the power consumption. These techniques apply to both embedded systems and general-purpose computer systems.

2.2.2 Architecture Level Power Reduction

A large number of work has been done about processor power consumption at architecture level, which breaks down to individual components, such as pipelines, cache subsystems, register files, TLBs, etc., and analyzed according to run time data. Customizations at this level often give much greater flexity and finer granularity, with often more complicated algorithms.

Various power analysis and estimation techniques have been discussed [60, 89].

More power efficient designs are proposed.

Some techniques maintain transparency to the programmers. In [80], the authors introduced the RegionScout technique to dynamically monitor cache data sharing status at a coarse grain, in shared memory multiprocessor systems. Shared data activities are recorded in special hardware structures. The RegionScout controller thus can filter out unnecessary cache snooping from remote processors and save power.

However, for embedded systems, similar optimization would work much more effective when put in a hardware-software co-design framework. Application specific information is used to significantly cut unnecessary modules and reduce unnecessary complexity. Optimization techniques at this level sometimes require programmer assistance, which is common practice in the embedded system domain. For example, in Application Specific Instruction Processors (ASIPs), a designer can customize certain application function to be implemented in hardware and insert new instructions to the ISA. The programmer and the compiler needs to work together to utilize that facility for the best of system efficiency.

2.2.3 System Software Techniques for Power Saving

Depending on the embedded system physical resources and requirement, system software in embedded system design range from very small size with just basic interrupt handling routines, to middle size with relative more OS functionality for embedded process, to full fledged large size OS that has all general-purpose com-

puter's functionalities. Some embedded OS are also required to support hard real-time guarantees. Small size OSs are lack of many OS functions, which force software developpers to write many low-level code on hardware layer. Medium size OSs, such as Vxwork[119], INTEGRETY[38] and Neotrino[101], have relative larger size but with more system functions. With embedded systems trending to more complex and intelligent, "bigger" operating systems are also growing at faster speed than the embedded system market in general.

The use of operating system itself may be a big overhead in embedded systmes. Much effort has been contributed to analyzing this problem[107]. Techniques used at operating system level usually include operating system software, user application software, and compiler support. Application specific information needs to be carried down to processor architecture level by the operating system, usually when the programs are loaded into the system.

Operating systems also needs to be modified to reflect the change in the underlying hardware platform. More often, power saving protocols and policies are implemented at operating system level so as to give the programmer flexibility in adapting to specific application requirements.

2.3 Cross-Layer Customization for Embedded Multi-Cores

2.3.1 Cross-Layer Customization Approach for Embedded Systems

Although the above techniques work well for both embedded systems and general-purpose computer systems, they can become much more effective for embed-

ded systems, when put in a system-level customization framework. This is because embedded systems are dedicated to certain applications. By analyzing the application environment, the designer can remove significant redundancy in standard hardware parts and software routines to make the entire design extremely efficient. This, however, requires that the designers work concurrently at all system layers so that the extracted application specific information can be passed across the layers.

For example, in ASIP systems, the register file size and the register allocation schemes are designed during system synthesis[7]. The power aware scheduling approach in [132] is combined with the DVS technique and OS scheduling algorithm to reduce the overall energy consumption. In [88], the authors apply the application memory access information to the compiler and the cache architecture to reduce the total cache access energy.

In [116], the authors introduce the Temporal Streaming technique to dynamically identify sequences of memory accesses which correspond to a data stream. By moving the data stream to the requesting processor in advance, the overall performance is improved.

With multi-core processors being used for embedded applications, the systems are becoming much more complex, which presents more need for customization to achieve satisfactory power and performance efficiency.

This study covers several specific topics of multi-core system customizations.

2.3.2 Improving Snoop Protocol Power Efficiency

The current small to middle scale CMP processors often adopt shared memory model, which provides a very intuitive programming model. Most of them have private caches to individual cores and a much larger last level shared cache pool. This creates the problem of coherency among all the private caches on the same chip, which is resolved by the installation of cache coherency protocols.

The general purpose cache coherenc protocls, normally the snoop protocols, are known to be general and consumer significant amount of power[33, 69, 85].

This not only puts a heavy burden for embedded system applications, but also significantly limits the system scaling into larger scales, making coherent cache unavailable in many large-scale CMPs for both embedded and general-purpose markets.

There have been a number of research projects on this topic trying to augment traditional snoop protocols and their implementations towards better power efficiency. The basic idea is to catagorize different cache area according to data usage and sharing patterns[81, 80, 117, 21, 33]. Being general purpose, however, many of these approaches could meet advasary cases that yield very little improvement on power efficiency. In [130], the authors propose to tag shared data regions in virtual memory, with information provided by the programmer. This information is used to filter out unnecessary snooping into the caches. Application specific information is used to preclude the speculation effort needed in general-purpose domain while the effects are much more pronound, at the same time incurring little hardware

overhead.

2.3.3 Inter-Core Communication based on Producer-Consumer Patterns

As multi-core systems scale to larger number of cores, the inter-core communication and cache subsystem are becoming much more important than the CPUs, which often hold the key to performance and power efficiency[33, 70, 71]. The general-purpose shared memory model gives good high level abstraction but doesn't guarantee the best performance. For explicit producer-consumer type of communications, the data sharing based on passive shared memory and cache coherence is not efficient in multi-core systems.

A number of approaches have been explored to improve such mechanisms for better efficiency[105, 19, 117, 21]. Some of these approaches try to exploit common producer-consumer communication patterns [8, 45, 26] and modify the system consistency model to allow more compact communication operations [27, 55], often by modifying the cache structure and coherence protocols, with compiler and OS support for such augmented hardware mechanisms[52, 66, 73].

2.3.4 Hardware Based Synchronization Mechanisms

Synchronization in embedded multi-core processors is also becoming an important issue, especially in mid-large scale systems. Synchronization is fundamental in parallel computing systems. The conventional atomic instruction based imple-

mentation bears too much overhead in modern system designs in terms of both performance and power[10, 12, 122, 115, 64, 35].

For embedded applications, dedicated hardware synchronization controllers have been proposed to address such problems [78, 9, 98, 82, 74, 135, 100, 114, 36, 77, 84, 28, 43, 51]. They often change the underlying synchronization mechanism significantly, with sizable performance improvement. Additionally, such dedicated controllers would enable the use of power saving modes in modern multi-core processors when individual cores are waiting for synchronization[64].

2.3.5 Cache Partitioning for Memory Bandwidth Minimization

As multi-core architectures scale to mid-large scale at rapid pace, various bottlenecks are emerging that prevent people from fully utilize such platforms. Among the different bottlenecks, the off-chip memory bandwidth limitation is becoming a very common problem in modern multi-core processor applications. While such problem is well recognized and is due to the speed gap between processors and memory systems[20, 95, 42, 68], there are ways to help alleviate it, such as the cache partitioning technique[67, 37, 91, 54, 92, 83].

The cache partitioning is well known and well studied in uni-core era. The effects on multi-core system memory bandwidth is inadequately studied, which makes it an interesting research topic in this study.

Chapter 3

Low-Power Snoop Architecture for Synchronized Producer-Consumer Communication in Embedded Multiprocessors

3.1 Overview

The snoop-based cache coherence protocols are the most widely deployed as they rely on the inherent broadcast nature of the common bus connecting the processor nodes to the memory. Each cache controller "snoops" the bus for memory transfers, for each of which a cache lookup is performed in order to determine whether a cache block state should be changed in the local cache. Easily extendable multiprocessor structures and software-transparent implementation have made snoop protocols easy to understand, deploy, and reuse, with minimal impact on the performance of memory subsystem [76]. However, these protocols tend to be overly conservative in many real world programs, especially embedded applications.

Quite often in embedded systems data are cached in just a few nodes. Snooping in the others leads to a waste of energy. Previous research [85] has shown that only around 10% of the application memory references actually require cache coherence tracking. And it has been reported that snoop-related cache activities can contribute up to 40% of the total cache power [33, 69].

In contrast to general-purpose computing platforms, in embedded system de-

signs, fine-tuning hardware, compiler, and system software has been a common practice to maximize performance and achieve energy efficiency [96, 72, 50]. In this work[130, 134, 125], we propose a methodology that aggressively eliminates the majority of unnecessary snoop-induced cache look-ups and thus, achieves significant power reduction. The proposed technique explicitly exploits application-specific information regarding the exact *producer-consumer relationships* between various tasks as well as information regarding *the precise timing of synchronized accesses* to shared memory buffers by their corresponding producers and/or consumers. This program knowledge is used to eliminate a large number of snoop-induced cache lookups *even for references to the shared memory buffers.*

The conventional snoop controllers are augmented with small additional hardware which is controlled by the system software. This hardware dynamically identifies accesses to relevant memory areas, detects the timings of temporal sharing between producers and consumers, and thus precludes snooping to those regions when it is safe to do so. The end result of the proposed methodology is significant reductions of power spent in unnecessary snoop-induced cache lookups.

3.2 Related Work

The emergence of multi-core processors in embedded applications has exacerbated the power concerns with these applications but also has exposed more opportunities to reduce power consumption. Many research projects have addressed power-aware coherence protocol. However, most of them are in the domain of general

purpose systems, such as desktop and server computers. Few of them have been in the embedded system domain, which often times exhibit specific hardware architectures and program behaviors and usually exposes more stringent power constraints. The contribution presented in this work enables the application of snoop-based cache coherence solutions in energy-efficient embedded applications.

Jetty [81] is the name of a family of snoop filters designed to reduce energy consumption in snoopy bus-based multiprocessor systems. Jetty observes cache activities in local caches and records them in special cache-like hardware-structures. By doing this, Jetty can dynamically feed the snoop controller information as "what is present in the local cache and what is not". The snoop controllers in local caches subsequently decide whether it is safe to filter out certain cache loop-ups, which otherwise would consume a lot of power. The authors report an average of 29% energy reduction for L2 caches.

RegionScout [80] is another technique that exploits coarse-grain data sharing information to reduce energy for caches and bus traffic in server applications. In RegionScout, memory space is divided into a number of chunks(regions). Additional hardware structure is employed to dynamically record which region is holding useful data and which region is blank. By identifying memory references to different regions, the snoop controller can filter out remote references that are not relevant to its local activities, thus saving power to the cache system. In [117] the authors introduce Temporal Streaming technique to dynamically identify sequences of memory accesses which correspond to a data stream. By moving the data stream to the requesting processor in advance, a large number of coherence misses are eliminated.

Additionally, the program performance is improved in directory-based snoop protocols. In [21] the authors propose Coarse-Grain Coherence Tracking to monitor coherence status of large memory areas so as to avoid unnecessary broadcasts, so as to enhance performance in commercial, scientific and multiprogrammed workloads. In [33], the authors target low-power chip-multiprocessors and try to reduce TLB energy consumption by using virtually addressed data caches, as well as to reduce snoop energy loss by keeping track of the sharing set of each memory page.

While techniques for general purpose applications report significant power benefits, their hardware and power overhead would be non-trivial in most portable embedded devices. Moreover, in a general purpose environment, system designers have limited software information and need to assume worst-case situations, which typically leads to hardware-only methods which dynamically try to uncover important program properties. For example, in Jetty and RegionScout, additional hardware structures observe the local caches and extract possible data sharing information to snoop controllers. This may be less effective against programs that have shared data scattered in the address space or may lead to large overhead implementing the monitor hardware as compared to the energy budget of an embedded application.

In embedded applications, however, programs naturally have specific and deterministic behaviors according to their application environment and system designers usually have much more detailed control over software and hardware integration. By exploiting this advantage, we can avoid the speculative mechanisms usually present in general-purpose approaches and minimize area and power overhead by precisely determining when and to which specific region to block snooping

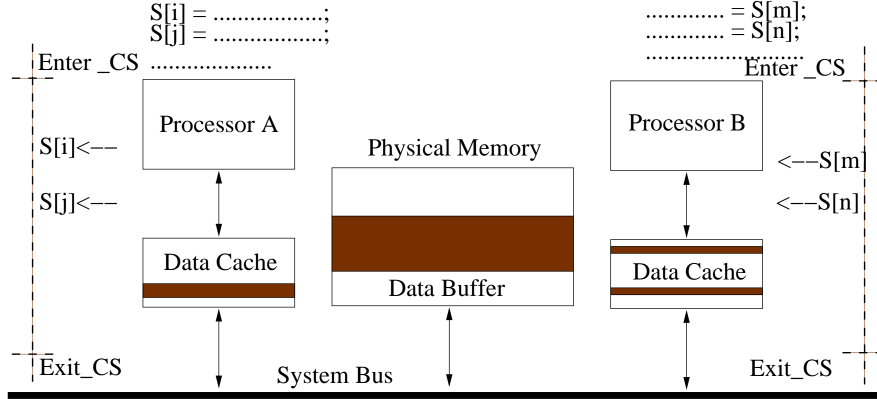

Figure 3.1: Synchronized producer-consumer communication with shared memory. At any time moment access to the shared buffer is exclusive required by the producer or the consumer only.

to local cache. Furthermore, since we are exploiting producer-consumer relationships in typical embedded applications, much stronger run-time sharing patterns can thus be exploited, yielding even greater energy reductions.

3.3 Functional Overview

The proposed technique benefits from the availability of precise application information regarding producer/consumer relationships in many embedded applications. The producer-consumer relationship between different processing nodes occurs naturally in the presence of data sharing. Quite often, sharing exists only within a small number of nodes rather than across the entire system. In this case, data would not be cached in other nodes' caches and probing those caches for shared data is unnecessary. Furthermore, shared data buffers are essentially *temporally "private"* when access right to them is acquired by a certain node. Consequently, for a certain

period of time even snooping for those shared data may be eliminated. Thus, by precisely differentiating different memory regions that store shared data and the exact relationships between them, we can enforce a more energy efficient coherence protocol implementation, which is active only during the ownership transition of shared data blocks.

There are three scenarios in which snooping can be safely eliminated. Clearly, if a data is private to a certain processing node, it is guaranteed not to be present in any remote caches. In this case, it is safe to eliminate snooping for this data in all remote processor nodes. Secondly, data sharing usually exists within only a subset of the nodes rather than across the entire MPSoC. In this case, the shared data is essentially private to the rest of the nodes and would not be cached there; consequently, it can be safely precluded from snooping. Furthermore, even between sharing processor nodes the shared data buffers can fundamentally be considered temporally "private" when access right to them is acquired by a certain node. This situation occurs when one of the participating nodes acquires exclusive access to the shared data by means of synchronization primitives. Consequently, for a certain period of time even snooping for these shared buffers may be eliminated at the shared nodes. Thus, by precisely differentiating shared memory buffers and by capturing the exact timing of the synchronized relationships between the sharing nodes, an aggressive snoop reduction can be achieved with significant reductions on the total power. The proposed application-aware low-power snoop coherence protocol is active only during *ownership transitions* of shared data blocks.

3.3.1 Synchronized Producer-Consumer Communication

In many multi-tasking embedded applications, especially stream applications, communication between different processing nodes constitutes Producer-Consumer (P/C) relationships, as illustrated in Figure 3.1. In this example, processor A performs a computation on batches of input data and for each batch it stores the output to the shared data buffer S. Data stored in this buffer are read by the task running on processor B performing subsequent computations or manipulations. With respect to data buffer S, processor A is producer, while processor B servers as consumer. Processor B, in turn can serve as producer to other shared memory buffers, which are "consumed" by other processors (possibly processor A). Since processors often read and write from different buffers during a program run, they may well be producers and consumers at the same time, with respect to different shared buffers. In order to prevent non-deterministic behavior and race conditions, accesses to the shared buffers must be fully synchronized. To obtain exclusive access to the shared buffer S, both producer and consumer use synchronization primitives such as locks, semaphores, or barriers. The portions of the code where the shared buffer is exclusively worked on are typically referred to as *Critical Sections (CS)* and are surrounded with synchronization operations in the forms of *ENTER_CS* and *LEAVE_CS*.

In the example in Figure 3.1, processor A is writing to buffer S, which injects write-misses to the system bus and results in invalidations in all others' caches, including the cache of processor B, and triggers snoop-induced cache lookups. Simi-

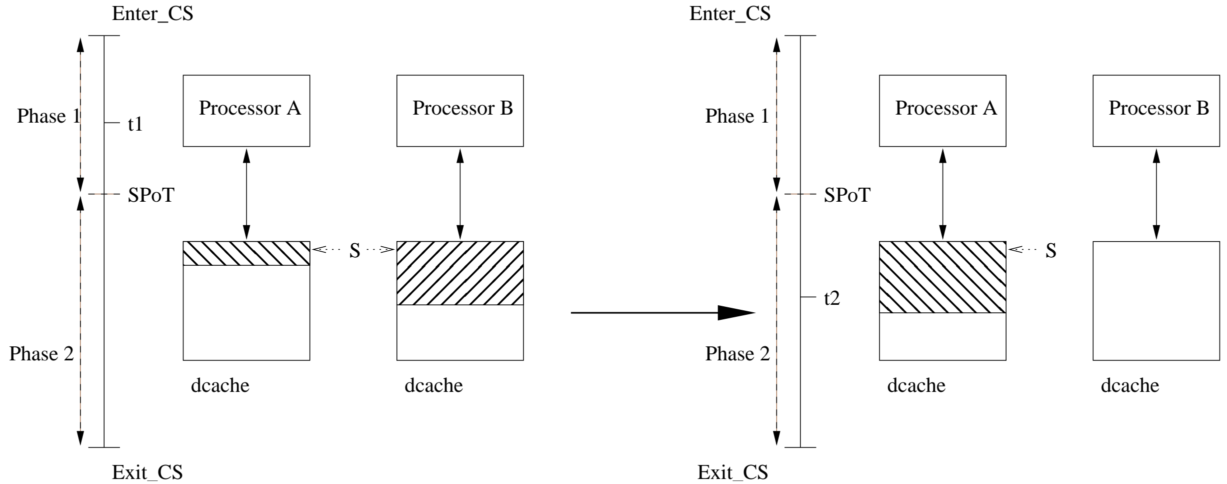

Figure 3.2: Producer-Consumer cache snooping activities.

larly, when processor B acquires access to buffer S and starts reading from buffer S, it generates a large number of read-misses, which in turn triggers snoop activities at all remote caches including the one of processor A. Clearly, all these snoop-induced cache lookups in nodes other than A and B are unnecessary, since buffer S is only shared between A and B and could not be cached anywhere else. Furthermore, even snooping in A and B's caches may be redundant when access right to S is exclusively acquired by either one of them while the other node does not hold any cache block of S.

3.3.2 Snoop-Phases in Producer-Consumer Communication

The snoop activities at both producer and consumer follow a certain well-defined pattern that constitutes of two phases.

Phase one occurs in the beginning of a critical section. As is shown, processor B's cache contains blocks belonging to the shared buffer S. These cache blocks have

been brought into B's cache while it was "consuming" S in the previous critical section. Since now processor A is writing to S, it is putting write-misses on the system bus and all these blocks will be invalidated by the end of the critical section. Time moment t_1, shown in Figure 3.2a, corresponds to such a state. Before these cache blocks are entirely invalidated, whether by snoop induced invalidation or by other address conflicts on processor B, snoop protocol should work to ensure data cache coherence between A and B.

As processor A's computation proceeds, the number of cache blocks containing S at B monotonically decreases, since B will not touch S until the next time it acquires access right to S. The rate of this decrease varies with applications behaviors and hardware configurations, as will be discussed later. At some point in time while A is its critical section, the number of cache lines at B holding data from S drops to zero. It can be seen that from this point on, no snoop activities related to accesses to S are needed at processor B, because no cache lines holding data from S exist to create incoherence with respect to S. In the proposed methodology, we refer to this moment in time as the *Snoop-Phase Transitioning (SPoT) point.*

The *second snoop phase* starts from the SPoT point and lasts until the end of the critical section; with respect to the snoop operations for the shared buffer, phase two effectively lasts until the beginning of the next data consumption iteration (critical section). The important characteristic of this phase is that no snooping at B is needed with respect to accesses to S generated by A. Time instance t_2, depicted in Figure 3.2b has occurred during the second phase as processor B's cache does not contain any data belonging to the data buffer S. It is evident that during the second

phase B does not have to snoop for remote references to S and is guaranteed to be safe to do so, since it is holding no valid lines belonging to that region and no new lines from S will be brought in until the next consumption iteration.

As can be seen from these observations, the potential saving from disabling snooping at B for references to S depends on how soon the SPoT point occurs in the production cycle of A. If SPoT comes near the end of the critical section, the benefits would be limited. However, our experiments on a set of benchmarks show that SPoT occurs quite early in actual applications. This can be easily explained by several observations. First, while A is producing to S, B is most likely doing other useful computations other than idling and thus will touch other shared data buffers as well as private data. These activities would thrash B's cache and evict S quickly. Second, A could be touching data from buffer S in some irregular fashion that would expedite invalidation in B's cache. For example, matrix multiplication usually touches different cache lines much faster than striding accesses and would cause much faster invalidations, which brings SPoT much earlier. Moreover, if buffer S is relatively big compared to the cache size, many blocks that contain S would have already been evicted at the very beginning of the critical section, due to its own evictions in the previous critical section. This in reality should leave only a few blocks to be of concern for the snoop controller.

The situation when B is consuming from S is similar. This time, B would start putting read misses on the bus which cause snooping on A's side and change blocks from S in A's cache from modified or "dirty" to shared state. The number of dirty cache blocks from S at A would determine how soon SPoT occurs. Following the

same argument as above, one can notice that the number of "dirty" cache blocks from S at A monotonically decreases. In our experiment, SPoT in this case also happens very soon in most critical sections, for similar reasons as discussed above. Consequently, snoop-induced cache lookups at A for references to S (generated at B) would be redundant after SPoT, and can also be safely eliminated.

3.3.3 Snoop-Phase Detection

By exploiting producer/consumer relationship and identifying SPoT for each shared memory buffer, the snoop controllers can aggressively filter subsequent snooping to these regions in their respective caches, and thus, achieve significant energy reduction. Snoop-induced cache lookups at each processor will be allowed only for small subsets of memory references to shared buffers of that processor, which occur only before SPoT points for associated memory regions. For all the remaining memory references that have occurred after SPoT or are not related to the shared buffers for the particular processor, no snoop activities shall be carried out.

In order to exploit the two snoop-phase pattern and, thus, filter snooping for accesses to particular shared buffers, a mechanism is needed to distinguish between references to various shared buffers in multiprocessor systems. The approach we propose here relies on the help of the operating system memory manager to assign such unique identifiers. The workings of this identification scheme are outlined in the next subsection.

We propose two methodologies for snoop elimination based on the snoop-

phases for P/C communication:

Passive SPoT Detection. This approach is based on a direct observation/detection of SPoT during run-time. The objective here is to simply detect the occurrence of SPoT and, subsequently, to disable all snooping activities for references to the particular shared buffer. The implementation involves several hardware counters which monitor the data sharing status with respect to buffer IDs associated to cache blocks. These counters keep track of the number of valid cache blocks (for consumer tasks) or dirty cache blocks (for producer tasks) belonging to different shared buffers in local caches. It is noteworthy that the counters used to detect the SPoT occurrence are updated only on a cache line replacement and as such are very efficiently in terms of cost, performance, and power overheads. In the next sections of this work we present a detailed description of the hardware architecture for SPoT detection.

Active SPoT Migration. In the second approach, which is an optimization over the passive SPoT detection, a special action is undertaken at each processor node when exiting a critical section to ensure that the SPoT is actively moved and will occur as early as possible. Instead of waiting for and detecting the SPoT, a simple hardware structure is employed to ensure that the cache blocks belonging to the specific shared buffer are either invalidated on the consumer node side, or changed to shared state with their content written back to memory on the producer node side. The hardware mechanism is activated when the task exits its critical section associated with the shared buffer. In effect, this ensures that the SPoT points occurs much earlier as compared to their natural timing. The aforementioned snoop

phase-1 coincides with the latency of this mechanism. The active SPoT migration approach also guarantees the effectiveness of the snoop filtering methodology to a broader range of applications where the snoop phase transition may naturally occur late in the critical section otherwise.

3.3.4 Shared Buffer Identification

The proposed approach distinguishes the shared buffers by letting each task inform the system software (through a simple API) as of which shared buffers are used in that task, where is the critical section for each of the buffers, and whether the buffer is being accessed in a producer or a consumer matter. The OS memory manager subsequently assigns an unique identifier for each such shared buffer and tags all the memory pages belonging to that buffer with this identifier. The buffer identifier associated to each page is captured in the Memory Management Unit (MMU) and the page table within the OS. For each memory reference that is placed on the bus, the buffer identifier is obtained from the MMU (together with the physical address of the location) and placed on the bus as part of the memory transaction.

When the program is loaded, the operating system obtains this information and assigns unique buffer identifiers to shared buffers used in the program at the granularity of pages. The buffer identifiers associated to each page are captured in the MMU and are managed within the page table entries by the operating system. The operating system also distinguishes between producers and consumers for

```
                        int S[100];
                        ..
                        ..
Thread 1                             Thread 2
..                                   ..
RegisterRegion(S, 100, producer);    RegisterRegion(S, 100, consumer);
..                                   ..
..                                   ..
ENTER_CS(S);                         ENTER_CS(S);
...                                  ...
...                                  ...
LEAVE_CS(S);                         LEAVE_CS(S);
```

Figure 3.3: Transferring to OS shared buffers information

shared buffers with respect to the processor nodes. The hardware counters that keep track of the number of valid or dirty cache blocks act differently for producer and consumer nodes. Since shared buffers are always written to by producers and read from by consumers they assume different roles on different processors, which is also maintained by the system software.

Figure 3.3 illustrates how the information regarding shared memory buffers and the relationship of task to buffers (producer or consumer) is transferred to the OS. Often times multitasked applications are developed by using multithreading libraries. Threads are created, terminated, and synchronized at the application level without intervention from the kernel, thus achieving high efficiency. Because the multiple threads comprising the application execute within the same address space (they share a single OS-level process), it is impossible for the OS to determine which memory buffers from that shared address space are actually shared between the threads. This information, however, can be easily provided by the embedded software developer by using a special function, which interfaces with the OS. Following this approach during the thread initialization phase, this function will be called to register all the shared buffers associated with each thread as shown in Figure 3.3.

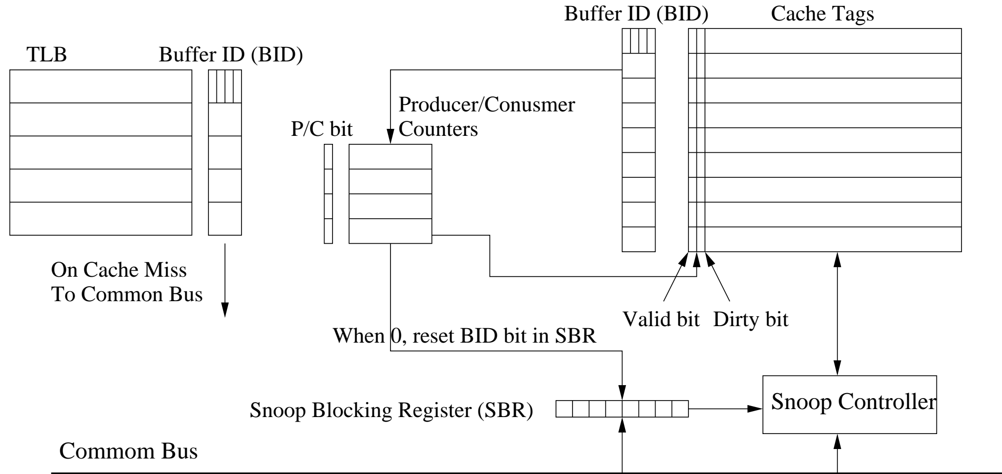

Figure 3.4: Hardware architecture for SPoT detection.

The OS subsequently assigns a unique buffer identifier for each shared buffer. As the identification occurs at page granularity level, shared buffers need to be aligned at page boundaries, which is easily done by the compiler. Our experiments show that supporting up to 16 different shared buffers in the system would be enough for many multitasked applications. Thus, a 4-bit *Buffer Identifiers (BID)* would be assigned to each memory page and captured in the TLB. In the subsequent section we outline how BIDs are used and explain (and later demonstrate by experimental results) that their small overhead is worth due to the large power savings in eliminating a large number of cache lookups.

3.4 Passive SPoT Detection

The passive SPoT detection approach relies on hardware to observe cache activity, detect SPoT occurrences and block snoop induced cache lookups for references to shared buffers. The hardware architecture is shown in Figure3.4.

As can be seen, the TLB entries are extended to contain a *Buffer ID (BID)*. These shared buffer IDs are associated with the page table entries and maintained by the operating system. This implementation requires that the shared buffers be aligned at page level, a requirement which is not restrictive as such alignment is required for other purposes as well such as better physical memory allocation and elimination of false sharing at cache block and page levels. When page table entries are loaded into the TLB, their associated BIDs are also obtained. The small table of BIDs associated to the TLB is implemented as a separate SRAM array and is not read and compared as a part of the normal TLB lookup which is needed for cache access. Whenever a cache miss occurs however, the associated BID is accessed and utilized by the hardware mechanism for SPoT detection; it is also placed on the bus together with the memory request (read-miss or a write-miss) transaction.

A similar small SRAM array is required in order to associate BIDs with each cache block. When a new cache block is brought into the cache, its BID is stored in that array. These buffer IDs are never compared like cache tags or even read when the cache block is accessed. They are only used as indices to a small set of up-down counters. In our experiments we have modeled 4-bit BIDs which index 14 counters. This is enough to handle a total of 14 shared buffers in the system - the remaining two values are reserved for two special purposes. One is for private data pages. Snooping to those pages can always be safely blocked, without any cache coherence concerns. The other is the opposite extreme, in which the programmer cannot determine the sharing status of the data, such as operating system data, etc. A conventional snooping for these memory regions is required in order to ensure cache

coherence. Because of the very small bit-width of the BIDs, the area overhead of these small BID SRAM arrays is minimal compared to other components in the cache system.

The BIDs associated with cache blocks are used in the process of SPoT detection. The hardware counters that BIDs index to are used to keep track of the number of cache blocks that hold data belonging to corresponding shared buffers. An additional Producer-Consumer (P/C) bit indicates what type of cache blocks must the counters track according to the role of the tasks as a producer or a consumer. This bit is also assigned by the operating system at task initialization, with information fed by the programmer.

The counters act differently according to the P/C bit. For producer tasks they keep track of the number of dirty cache blocks of the buffers. The producer (or "dirty") counter associated to a buffer increments on a write miss and a new block is brought into the cache, or when there is a write hit on a "read-only" block that would turn to "dirty" or "modified" as both actions introduce a new dirty cache block into that buffer. Note that in some snoop protocol implementations, a write hit also sends "write-miss" messages to the system, which can simplify our implementation. The counters decrement when a dirty block is evicted or when there is a read miss from other processor and a dirty block is changed from "modified" to "read-only/shared" state. Similarly, the consumer task counters keep track of the number of valid cache blocks that belong to the particular shared buffers. They increment when a new cache block is brought in on a read miss and decrement whenever a valid block is evicted by local activities or invalidated for remote write

misses. All the counters are set to 0 at the beginning of program execution, by the operating system.

When the value of a counter is zero, there are no cache blocks corresponding to the associated shared buffers in the local cache and snooping can be safely blocked for them. Such blocking is achieved through a *Snoop Blocking Register (SBR)*. This is a simple bit-mask register with a bit per shared buffer, where the bits are directly indexed by the buffer ID; a zero at bit position indicates that snoop-induced cache lookups for this buffer are blocked. When the counters reach zero, the corresponding bit in the SBR is set to 0.

When a memory request in the form of a read-miss or a write-miss is place on the bus, the BID of the address is obtained from the BID table and is placed on the bus as a part of the transaction. Note that no additional bus lines are needed as the data lines for such transactions are used to carry BIDs. The cache controllers snoop on the bus as usual, only that the SBR register is checked prior to performing a cache-lookup. References to private data are always blocked, while references to "unknown" data are always snoop-enabled. This also gives programmers great flexibility in focusing on most significant data buffers and let conventional snooping take care of the rest, which adds flexibility in adopting the proposed technique.

Overhead Analysis. The area overhead is dominated by the two 4-bit SRAM arrays that associate BIDs to TLB entries and individual cache blocks. Compared to the cache tag and TLB sizes, this area overhead is typically below 5%. The number producer-consumer counters is determined by the BID bitwidth. With 4-bit BIDs, 16 producer-consumer counters are used. The power overhead is comprised of reading

the BIDs and incrementing/ decrementing the buffer counters. Additionally, on a cache miss the BID is read and placed on the bus. All these events *occur only on cache misses* or certain cache block state changes and as such are not significant as compared to the savings achieved. In our experiment results we have accounted for all these overhead, including the power needed to transfer on the bus the 4-bit BIDs. It is also noteworthy that placing the BID on the common bus does not necessitate additional bus lines. The existing data bus lines are used instead since the memory transactions of interest comprise of read-misses and write-misses, which only carry an address of the memory location that is requested by the processor node.

3.5 Active SPoT Migration

Clearly, the earlier in the critical section the SPoT occurs, the better the savings achieved by the proposed methodology. In the passive SPoT detection scheme, the effectiveness of the filtering approach depends on how soon the transition between snoop phase-1 to phase-2 occurs. If the SPoT occurs very early in the critical section, then a larger number of snoop operation will be eliminated. If this is not the case for some applications, the overhead may exceed the achieved benefits of snoop elimination. Even though our experimental results show that the passive SPoT detection approach achieves extremely good reductions, such an assumption for early SPoTs can be eliminated by *actively ensuring* that SPoT occurs very early in the synchronized region.

The SPoT can be actively moved earlier in time by making sure that the local

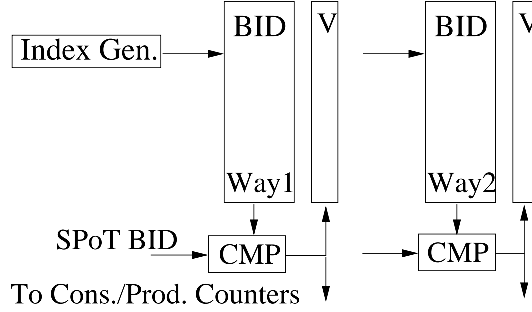

Figure 3.5: Hardware architecture for Active SPoT migration.

cache is released from cache blocks that hold content of shared buffers, instead of waiting for the natural occurrences of SPoTs. A small additional hardware can be employed for this purpose to forcefully expedite the occurrence of SPoT by either writing back dirty blocks (and thus changing them to “shared/read-only” state) for producer tasks, or by invalidating valid blocks for consumers. For this purpose, we have experimented with a simple hardware mechanism which simply traverses the BID array associated with the cache blocks and either write-backs (for a producer buffer) or invalidates (for a consumer) the cache blocks if the BIDs match; BIDs can be checked in parallel as their width is rather small. This procedure need to be initiated at the exit of the critical section.

The hardware organization required for the Active SPoT migration technique is shown in Figure 3.5. On the producer side, each buffer ID is checked if it matches the BID of the current critical section and, if positive, the state of the cache block is changed and the data is written to memory. This necessitates checking the Valid and the Dirty bits of the cache tag and clearing the Dirty bit if the BIDs match, thus minimizing the overhead. The procedure continues until the associated counter

reaches zero, which indicates the SPoT. At that moment snooping for that buffer can be safely blocked until the next cycle of production. Since BIDs are very short, this action can be performed in parallel for multiple associativity sets of the cache. The only limiting factor is the speed of write-backs supported by the memory subsystem on the producer caches. It is noteworthy that *no extra energy is consumed in this process* since those cache blocks would have to be written back to memory anyways, though maybe later in the execution cycle when the consumer requests them.

Similar steps are needed at the consumer's side - in this case, however, blocks from the shared buffer need only be invalidated if present, which only means clearing the valid bit of the matching cache blocks. Consequently, on the consumer side, since there is no concern of write-back speed, this action of active SPoT migration can be easily performed in parallel for multiple associativity sets. In a sense, the proposed hardware performs a cache invalidation for a particular memory regions - that corresponding to the shared buffer under consideration.

In our experiments, we have noticed that often times only a few dirty/valid blocks are left in the cache after exiting the critical section due to other memory activities in the previous production/consumption. Consequently, the expected number of write-backs or invalidations should be relatively small. We have seen that with our active approach, the SPoTs are almost pushed to the very beginning of their critical sections and snooping is almost completely eliminated as a result of that.

Overhead Analysis. The hardware/power overhead consists of the logic required to iterate through the cache associativity sets and compare the BIDs. This

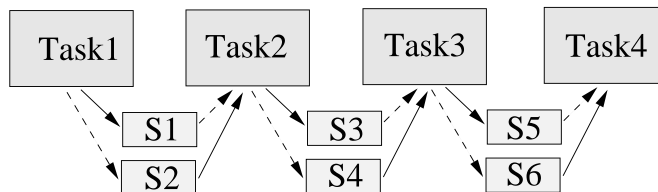

Figure 3.6: Application benchmarks organization.

includes a counter to generate the indices, and a comparator per associativity way. On the consumer side, in case of a BID match, the line valid bit (V) is reset, while on the producer side, a write-back is required. As mentioned above, no extra power is consumed when writing back such a dirty cache line as it eventually needs to be written to memory and our approach simply moves that event earlier. When a cache line is invalidate or written back to memory, the corresponding producer/consumer counters are updated accordingly. When the counter for the SPoT under consideration reaches zero, the cache traversal hardware is stopped. As our experiments suggest, it is very often the case that this hardware is only active several tens of cycles only, as only a few cache lines belonging to the particular shared buffer have remained in the cache.

3.6 Experimental results

To evaluate the proposed technique, we have conducted detailed experiments on a set of embedded multitasking applications. The simulated systems exhibit four processor nodes, each performing one computational task on a number of shared data buffers. The tasks mapped on the different processors work in a pipelined fashion; each task uses the output from another task in the system as its input and

itself produces the input for another task. Consequently, the four parallel tasks act as both producers and consumers with respect to other tasks in the system. This environment is typical for streaming data applications and systems [53], where the input data is streamed through several computational kernels for various manipulations until the final output is produced. Except for the very first task in the pipeline, all are both producers and consumers and work on more than eight different data buffers in different synchronization sections. The production are consumption cycles are interleaved so that at each moment of time the processor is executing in either a producer or in a consumer critical section. The benchmarks organization is depicted in Figure 5.7.

The individual tasks constitute of: *LU*, *FFT*, *ADPCM*, *matrix multiplication (MMUL)*, *data encryption (AES, SHA, blowfish)*, *LZO-compression (LZO)*, *g721 speech compression (G721)*, image processing - the *blur* and the *edge-detection*, and frequently used kernels in *multimedia processing*, such as the *Fast Fourier Transform (FFT)*, the Inverse Fast Fourier Transform (IFFT), and the *Fast Discrete Cosine Transformation (FDCT).* The tasks cover benchmarks from the Mediabench [61] and MiBench [39] suits, as well as from other open-source image and video processing tools [44]. The multitasking applications are combinations of individual tasks. The ones we have used are: *A1*={*LU, MMUL, AES, LZO*}; *A2*={*FFT, G721, blowfish, SHA*}; *A3*={*blur, edge-detection, AES, LZO*}; *A4*={*FFT, FDCT, IFFT, AES*}, which represent multi-tasked embedded applications in digital filtering, audio, image, video processing, and security arenas.

We have used the M5 [18] simulator to perform our experiments. The simula-

tor is used in system-call emulator mode and is extended with a collection of thread synchronization primitives, such as barriers, mutex locks, etc. Since M5 is based on Alpha ISA, we have used the *gcc* cross-compiler to compile all benchmarks. The simulated hardware platform configuration is of four processors connected to a shared memory through a common bus. Each processor has local instruction caches and L1 data caches. We have configured the system with write-back caches, since write-through caches generally consume more power and require more memory bandwidth and as such are less frequently deployed in embedded applications. We have experimented with four data cache organizations: caches of size 16KB and 32KB, either direct-mapped or 4-way set-associative. The data buffers used for communication between the processor nodes are of sizes 16KB and 64KB; experiments for both data sizes have been conducted.

The cache power expenditure of the four cache configurations have been obtained through the Cacti v4.2 tool [109] for $0.18\mu m$ technology. We have also accounted for the power overheads incurred by the introduced hardware structures. The BIDs are modeled as 4-bit SRAM arrays, whose energy consumption is similarly obtained by Cacti. The SBR is accessed on every bus transaction and is implemented as a 16-bit register. The hardware counters that track the number of dirty or valid cache lines are modeled as 12-bit up-down counters. The counters' actions are accounted for on a cache block replacement, in which a new block is brought in or an old block is evicted out, or state change in which a dirty block is changed to shared state. We have also taken into account the overhead of placing the 4-bit BIDs along with addresses on the bus. The energy data reported in [16]

	16K-4SA	16K-DM	32K-4SA	32K-DM
A1	16,202/ 1,241,013	67,146/ 2,351,109	29,043/ 1,000,890	42,210/ 1,173,708
A2	5,229/ 425,580	5,462/ 2,593,260	11,001/ 180,360	10,349/ 2,347,815
A3	20,493/ 1,058,058	34,750/ 1,125,651	20,863/ 846,108	31,061/ 973,050
A4	10,466/ 598,185	10,322/ 607,671	11,274/ 247,002	16,636/ 425,196

Table 3.1: Snoop-induced cache lookups for 16K shared data buffers; Passive SPoT v.s. baseline snoop protocol.

has been scaled to 4 bus lines with 50% bit-transition activities.

We have chosen the most widely used snoop-invalidation protocol with four symmetrical processors connected to a common bus as a baseline. The snooping activities for the baseline and the proposed *Passive SPoT* detection approach are shown in Table 3.1 and Table 3.2. The numbers in the tables come in pairs, the first showing the total number of snoop-induced cache lookups after applying the Passive-SPoT technique, while the second number reports the number of snoop-induced lookups for the baseline general-purpose coherence protocol.

It is clear from the results that the Passive-SPoT-Detection methodology significantly reduces the amount of snoop-induced cache lookups. The achieved reduc-

	16K-4SA	16K-DM	32K-4SA	32K-DM
A1	8,775/ 18,571,698	40,027/ 22,576,899	10,982/ 8,903,400	61,535/ 8,523,852
A2	20,589/ 3,952,833	32,336/ 23,960,067	32,798/ 3,818,868	33,128/ 23,303,937
A3	59,858/ 2,434,974	74,612/ 2,643,678	85,660/ 2,231,496	101,722/ 2,398,860
A4	40,147/ 7,124,247	89,426/ 30,993,432	41,834/ 6,864,186	89,566/ 30,685,677

Table 3.2: Snoop-induced cache lookups for 64K shared data buffers; Passive SPoT v.s. baseline snoop protocol.

tions vary with different shared buffer sizes and cache sizes and organizations and the nature of different kernels. In general, the larger the cache size is, the smaller the snoop reductions of the Passive-SPoT-Detection. This is because caches with larger capacity can hold more cache lines and hold them longer and, thus defer the SPoT occurrence. However, larger caches exhibit higher power consumption, which increases the contribution of each saved cache lookup. Increased associativity has a small impact on the snoop reductions. However, since more cache tags are to be checked in parallel on every access to the cache, the energy savings of the proposed methodology are higher. It is evident also that the result differ across the application benchmarks. The difference comes from the fact that the applications exhibit

	16K-4SA	16K-DM	32K-4SA	32K-DM
A1	6.7 / 68.1	12.8/ 61.9	6.5/ 62.6	7.0/ 42.2
A2	2.2/ 23.3	12.3/ 68.3	1.5/ 11.2	11.4/ 84.4
A3	6.1/ 58.1	6.2/ 29.6	5.2/ 52.9	5.6/ 35.0
A4	3.3/ 32.8	3.1/ 16.0	1.8/ 15.4	2.6/ 15.2

Table 3.3: Energy consumption (μJ) for 16K shared data buffers; Passive SPoT v.s. Baseline

different access patterns to the shared data buffers and also have different amount of private data accessed along with the shared. These properties will result in the SPoT occurring early or late during the critical section and since the Passive SPoT detection technique simply registers that event, the achieved snoop reductions will differ accordingly. Additionally, the size of the shared buffers has an impact on the reductions. Larger shared data buffers create significantly more misses in the baseline, which results in increased relative reductions.

The energy estimations for the baseline and the proposed Passive-SPoT detection approach are shown in Table 3.3 and Table 3.4. The numbers in the tables also come in pairs, the first showing the total energy consumption (μJ) by the Passive-

	16K-4SA	16K-DM	32K-4SA	32K-DM
A1	87.7/ 1,019.3	107.1/ 594.9	42.5/ 556.9	42.2/ 306.6
A2	19.7/ 216.9	113.4/ 631.3	20.0/ 238.89375	110.7/ 838.2
A3	14.7/ 133.6	14.3/ 69.6	15.8/ 139.59384	14.9/ 86.2
A4	35.6/ 391.0	148.0/ 816.7	34.8/ 429.39718	147.4/ 1,103.8

Table 3.4: Energy consumption (μJ) for 64K shared data buffers; Passive SPoT v.s. Baseline

SPoT technique, while the second number being the baseline energy consumption.

It can be seen that the percentage reduction of energy is smaller than the percentage reduction of the number of snoop activities. This can be explained by the introduced overhead of the proposed methodology. The reported energy numbers, as described earlier, take into account all the extra activities, such as BID access on cache-miss, producer-consumer counter increments and decrements, and transmitting the BID on the common bus to memory.

In general, since larger caches consumes more power per access they benefit more from snoop-induced cache loopup reductions. Caches with higher associativity need to check multiple cache tags simultaneously on every cache access, and thus

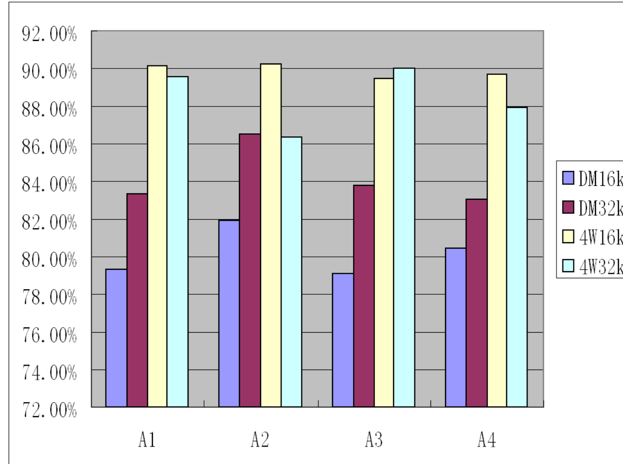

Figure 3.7: Energy reduction for 16K shared data buffers

should also gain more advantage from the proposed methodology. Consequently, the actual energy reductions for larger size and higher associativity caches would be more significant than for those with smaller sizes and associativities. Lower associativity caches, on the other side, cause significantly more misses and thus result in more snoop-induced cache lookups - the majority of which are eliminated by the proposed technique. For this types of caches, the absolute number of eliminated snoop-induced lookups is higher. Figures 3.7 and 3.8 show the energy reductions in percentage achieved by the Passive SPoT detection. The former represents all the configurations with 16K of shared data buffers, and latter shows the results for 64K shared data buffers.

As the data in the figures confirm, in general higher associativity caches feature higher reductions in snoop-related energy. One counter example for this general rule is application A2 as shown in Figure 3.7. For it greater reduction is achieved for direct-mapped 32KB cache as compared to 4-way associative 32KB cache. This

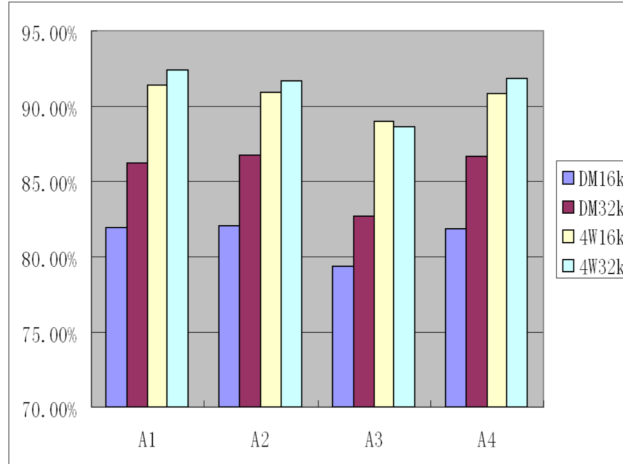

Figure 3.8: Energy reduction for 64K shared data buffers

situation can be explained by the fact that A2 exhibits many cache misses when using direct-mapped cache, which in the baseline translate to many snoop-induced lookups; these lookups are eliminated by the proposed technique. Even though, the energy contribution of each such lookup to the direct-mapped cache is smaller than the corresponding energy fir 4-was set associative cache, the significantly large number of such events in direct-mapped caches compensates and better energy reduction is achieved. For similar reasons applications A1, A2, and A4 achieve bigger reductions for 4-way 16KB caches than 4-way 32KB caches. Figure 3.8 confirms the aforementioned trends for the same cache configurations but for shared buffers of size 64K. One notable difference is that because of the larger size of shared buffers, all of the baseline configurations exhibit significantly higher number of misses and, thus higher power die to snooping. Consequently, eventhou the percentage reduction is similar, the absolute reductions in energy are significantly higher as compared to the case of 16K shared buffers.

The snooping activities for the baseline and the proposed Active-SPoT migration approach are shown in Tables 3.5 and 3.6. The numbers in the tables come in pairs, the first showing the total number of snooping achieved by the Active-SPoT technique, while the second number being the baseline snoop activities. It is clear from the results that the Active-SPoT-Migration methodology decreases snoop activities almost to zero. Although the remaining snoop activities still vary with cache organizations, cache sizes, different applications and different shared data buffer sizes, for all practical purposes they are essentially reduced to zero as compared to the baseline cases. It is evident that the active approach is very effective in pushing the SPoT very early in every critical section of the communicating tasks.

Nonetheless, the Active-SPoT migration methodology comes with an overhead, which needs to be taken into account when applied in real systems.

The energy estimations for the baseline and the Active-SPoT migration approach are shown in Table 3.7 and Table 3.8. The numbers in the tables also come in pairs, the first showing the total energy consumption (μJ) by the Active-SPoT technique, while the second number being the baseline energy consumption. We have assumed a hardware scheme, which checks one associativity set per cycle and invalidates cache lines from the shared data on the consumer side, while writing them back to memory on the producer side. The energy overhead of accessing the BID arrays each cycle until the SPoT is enforced is taken into account.

It is interesting to find that the active approach energy reductions for some applications benchmarks are very close to the reductions achieved by the passive SPoT detection approach. A reason for this is that the active approach was intro-

	16K-4SA	16K-DM	32K-4SA	32K-DM
A1	14/ 1,241,013	24/ 2,351,109	79/ 1,000,890	52/ 1,173,708
A2	14/ 425,580	8/ 2,593,260	14/ 180,360	11/ 2,347,815
A3	251/ 1,058,058	280/ 1,125,651	288/ 846,108	325/ 973,050
A4	4/ 598,185	12/ 607,671	18/ 247,002	12/ 425,196

Table 3.5: Snoop-induced cache lookups for 16K shared data buffers; Active SPoT v.s. baseline snoop protocol.

duced as an optimization over the passive SPoT detection approach, for the cases where the SPoT natural occurrence is rather late. However, our experiments have shown that for some application SPoT occurs very early, which does not leave much room for the active SPoT migration. Another factor is, although we eliminate almost all snoop activities by using the active approach, its higher overhead offsets the small benefits achieved by it. As a result, the absolute percentage reductions for the passive and the active techniques for such applications will be very close.

As the active approach pushes the SPoT almost to the very beginning of the critical section, regardless of the application benchmark or underlying cache configurations, the end results is that snoop-related energy is drastically reduced.

	16K-4SA	16K-DM	32K-4SA	32K-DM
A1	12/ 18,571,698	12/ 22,576,899	42/ 8,903,400	60/ 8,523,852
A2	14/ 3,952,833	10/ 23,960,067	26/ 3,818,868	28/ 23,303,937
A3	406/ 2,434,974	451/ 2,643,678	939/ 2,231,496	953/ 2,398,860
A4	4/ 7,124,247	12/ 30,993,432	30/ 6,864,186	54/ 30,685,677

Table 3.6: Snoop-induced cache lookups for 64K shared data buffers; Active SPoT v.s. baseline snoop protocol.

Figures 3.9 and 3.10 show the percentage of energy reductions of active approach compared to the passive approach. These two figures actually reveal the potential of the active approach with regard to different applications. It can be seen that for certain application/hardware configurations, such as application A2 with 32KB 4-way associative cache, and application A3 with 32KB 4-way associative cache, the active approach can further reduce energy by about 35%. Such cases are indicative that the active SPoT migration approach could further benefit applications for which the snoop phase transition point can be actively enforced to occur very early, and as such result in significant power reductions.

	16K-4SA	16K-DM	32K-4SA	32K-DM
A1	5.8 / 68.1	11.0/ 61.9	4.7/ 62.6	5.5/ 42.2
A2	2.0/ 23.3	12.1/ 68.3	0.8/ 11.2	11.0/ 84.4
A3	4.9/ 58.1	5.2/ 29.6	3.9/ 52.9	4.5/ 35.0
A4	2.8/ 32.8	2.8/ 16.0	1.1/ 15.4	1.9/ 15.2

Table 3.7: Energy consumption (μJ) for 16K shared data buffers; Active SPoT v.s. Baseline

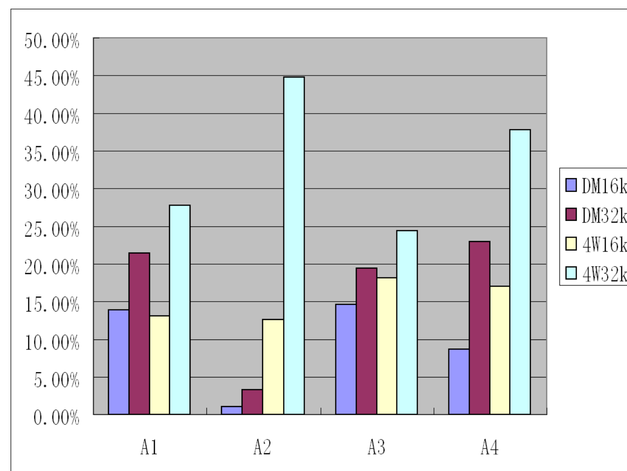

Figure 3.9: Active vs. Passive: Energy reductions for 16K shared data buffers

	16K-4SA	16K-DM	32K-4SA	32K-DM
A1	87.2/ 1,019.3	106.1/ 594.9	41.8/ 556.9	40.1/ 306.6
A2	18.5/ 216.9	112.6/ 631.3	17.9/ 238.89375	109.5/ 838.2
A3	11.4/ 133.6	12.4/ 69.6	10.5/ 139.59384	11.3/ 86.2
A4	33.4/ 391.0	145.6/ 816.7	32.2/ 429.39718	144.2/ 1,103.8

Table 3.8: Energy consumption (μJ) for 64K shared data buffers; Active SPoT v.s. Baseline

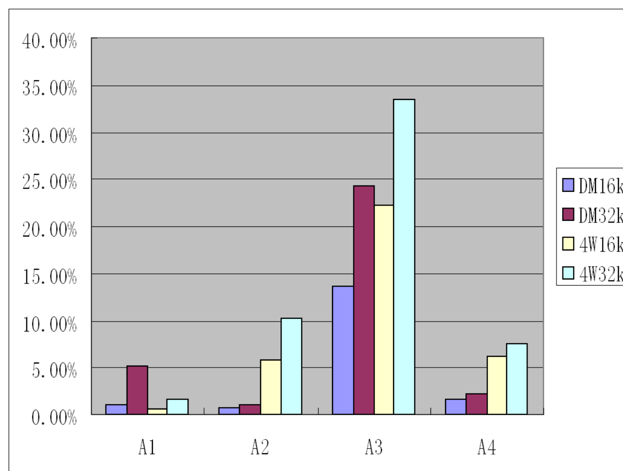

Figure 3.10: Active vs Passive: Energy reduction for 64K shared data buffers

Chapter 4

Energy and Performance Efficient Communication Framework for Embedded MPSoCs through Application-Driven Release Consistency

4.1 Overview

The increased integration densities coupled with the abundance of wireless connectivity have resulted in many modern applications implemented as complex computing systems. Such applications usually feature a large number of capabilities, such as aggregated multimedia data processing (speech, audio, video), communication protocols, security functions, user interfaces, and many others. Integrating multiple functionalities for applications with high demand for performance and data throughput is naturally achieved by a *Multi-Processor Systems-On-a-Chip (MPSoC)* implementation platform.

Each application task or a group of tasks is allocated to a processor core in the multiprocessor platform. Typically these platforms feature several processor cores, possibly of heterogeneous natures, with an access to a shared memory. The memory can be placed independently from the processors and accessed by all the cores in an identical manner, resulting in a Symmetric Multiprocessor (SMP). Alternatively, the memory can be distributed with each core having a local access to a portion of it resulting in a Non Uniform Memory Access (NUMA) platform. In both

cases, the memory address space is shared across the cores in a way transparent to the application software and the inter-core communication is implemented directly through the shared memory. The memory can be physically accessed through various interconnection networks. In an SMP platform, the most common solution is a shared high-speed bus or a hierarchy of buses, while in NUMA platforms more sophisticated point-to-point interconnects may be used. Each processor core typically employs a local cache to alleviate the interconnect bandwidth demand. Both private and shared data are cached locally, which typically results in sizable speedups and reductions in memory traffic. Locally caching shared data, however, introduces the cache coherence problem. When a processor updates a data after that same data is stored in a remote cache, it is evident that the remote copy becomes stale.

For bus-based platforms, snoopy cache coherence protocols [13] are the most widely used as they rely on the inherent broadcast nature of the shared bus. Each cache controller "snoops" the bus for memory operations, for each of which a cache lookup is performed based on the reference address in order to determine whether a cache line state change is required. Many variations of snoopy protocols have been explored. The NUMA platforms typically employ directory-based coherence protocols [63] where a directory controller maintains caches coherent through point-to-point control and data messages.

Since the available interconnect bandwidth is one of the limiting factors determining how many cores can be used, write-back local caches are typical. Write-through caches could quickly overwhelm the available communication bandwidth to main memory. In terms of data sharing coherence policies, two fundamental choices

exist: *remote-invalidate* and *remote-update*. The remote-update policy is usually combined with write-through caches as it propagates each write to all the remote caches. Consequently, it exhibits better inter-core communication latency as perceived by the consumer tasks since all the writes are immediately pushed to all the remote caches. However, due to the significant bandwidth and energy demand of the remote update schemes, such protocols are rarely used in practice, especially in embedded MPSoCs. The coherence policy of choice for practically all MPSoCs has been the remote-invalidate scheme. In such protocols the shared data is invalidated in all remote consumer caches when the producer writes into it. In the case of bus-based system, the write to shared data triggers an invalidation, e.g. a write miss causing all remote cores to invalidate this data in their local caches. In directory-based protocols, the directory sends the invalidation messages to remote cores sharing the data, after receiving a write message from the producer.

The cache coherence protocols, however, exhibit major shortcomings for embedded multiprocessors. First, they have been recognized as an *extremely energy inefficient*. For the snoop-based protocols, this is due to the need for all the processor nodes to "snoop" the bus for memory accesses and subsequently lookup the local cache and act accordingly. It has been reported that snoop-related cache activities can contribute for up to 40% of the total cache power [33, 70, 71]. Directory-based protocols are similarly power-consuming due to the excessive amount of traffic that occurs for tightly coupled producers and a set of consumers. Even though the remote-invalidate policy requires fewer transactions as compared to the other alternatives, the interconnect bandwidth is typically always utilized to its maximum by

increasing the number of active cores. Second, the *communication latency* between two processor cores is increased as each time a node writes a data, this data is invalidated in the other caches. A subsequent read to that data in a remote processor results in a miss (known as a *coherence miss*), which needs to be serviced by at least two subsequent transactions - a read-miss and a write-back from the producer.

In this article we introduce a framework for an efficient inter-core MPSoC data sharing. The proposed framework achieves *performance-*, *bandwidth-* and *energy-efficient* inter-core communication for groups of synchronized producer and (possibly multiple) consumers. The low-latency communication advantages of remote-update coherence policies is achieved and surpassed, yet the associated excessive traffic is avoided as only the last write to a cache line is allowed to generate a remote cache update transaction. The shared data is "*streamed*" from the producer's cache to the consumers' caches "*just-in-time*" so that it is present in the consumer's cache when the task enters its synchronization region where the data is read. The cache way partitioning methodology, which is a component in the proposed framework, ensures that shared data is not evicted from the consumers' caches. Moreover, since writes from the producer are guaranteed to be efficiently propagated to all the consumer caches as well as to the main memory, no snooping or additional directory-based transactions are needed for references to the shared data (and to private data as well), thus saving significant amount of energy.

For SMP platforms with snoop-based coherence protocols, the proposed framework results in a shared data communicated from a producer to a set of consumers with a single bus transaction per cache line in advance of the moment when it

will be needed at the consumer cores. For NUMA platforms, a single transaction per consumer core is executed prior to the consumer task requesting a read. The shared data is automatically updated on the consumer cores, and in the subsequent consumption phase no coherence misses ensue.

Furthermore, the proposed framework leverages limited cache configurability in terms of mapping the shared buffers to specific associativity ways. In this way the frequently accessed shared data is prevented from leaving the cache due to conflicts with private or other shared data. Supporting such cache mapping ensures that from the consumer process point of view, when entering the corresponding synchronized program region the updated shared data is guaranteed to be present in the local cache. Isolating shared data in a subset of the cache ways results in ***significant energy reductions*** as only a part of the cache ways needs to be looked up for each cache access.

The proposed methodology integrates supports from the programmer, compiler, OS, and hardware. Simple loop transformations based on loop unrolling and utilizing a "remote-update" version of the store instruction, ensure that when the producer writes into a cache line of a shared data for the *last time prior to exiting the synchronization region*, that write is propagated to the interconnect and to all the remote consumer nodes where the data is updated in their local caches.

The introduced methodology and system organization will be extremely beneficial and relevant for embedded systems since the application, or set of applications, to be deployed on the MPSoC will be known in advance and often times even developed in-house (or purchased as software IPs) by the MPSoC manufacturer. This

would enable the software tuning required for the precise identification of the shared buffers and producer/consumer relationships.

At its core, the introduced inter-core communication framework implements a form of *release consistency* [34]. Release consistency is one of the most popular relaxed consistency memory models. It requires that programs use explicit synchronization primitives as provided by the system and that updates to shared data are made visible to remote processors only after the producer exits its synchronized critical section, e.g. executes a lock release. The definition of release consistency allows for various implementation solutions. Most of the hardware-based approaches follow an *eager* release consistency policy where writes to shared data immediately trigger coherence transactions [63]. *Lazier* policies are implemented mostly in software-based protocols - the coherence actions are postponed in order to reduce the total number of transactions [22, 48].

The proposed framework implements a "just-in-time" lazy release consistency that combines the low synchronization latency benefits of eager release consistency with the low traffic and elimination of unneeded invalidations and updates of lazy release models. The proposed cross-layer approach, based on a cooperation between compiler, OS, and a hardware architecture, enables a low-cost and energy-efficient data sharing and communication in *resource and power constrained embedded MP-SoCs*.

The proposed methodology is applicable to both bus-based SMPs and distributed memory organizations with more generic interconnect architectures. The framework does not use specific properties of the underlying interconnect and can

be easily applied in any multiprocessor platform with local caches. The application-driven, "just in-time" remote update is either placed on the common bus and acted upon by the remote cache controllers or handled by the directory, which sends the update messages to the remote consumer nodes. The communication latency between producer and consumer cores, as perceived by the consumer tasks for their read operations to shared data, is mostly reduced to a local cache hit for either platform, while the impact on power is expected to be more significant in bus-based systems as no coherence-triggered cache lookups are required on misses placed on the bus. In this work[128, 131], we focus on the design requirements and evaluation of the proposed methodology in multi-core platforms with a snoop-based cache coherence architecture.

4.2 Related Work

Multiprocessor systems, in general, have been the focus of many research projects. The importance of these platforms for the future of general-purpose computing and high-end embedded systems has become clear in recent years [29, 120]. Power and thermal limitations coupled with the effect of diminishing returns of aggressive uni-processor optimizations have made multiprocessor systems an attractive implementation choice in many application domains. However, MPSoCs have introduced several important research challenges by their own [75], such as the need for new programming models and abstractions, efficient compiler and operating system support, and the ever growing need for power-aware architectures.

Inter-core Communication in MPSoCs. The inter-core communication latency problem has been recognized and addressed in the general-purpose architecture community [105, 19]. Temporal streaming organization was introduced in [117] to dynamically identify sequences of memory accesses which correspond to a data stream. By moving the data stream to the requesting processor in advance, the overall performance is improved. A coarse-grain coherence tracking technique was proposed in [21]. The coherence status of large memory areas is monitored so as to avoid unnecessary broadcasts and enhance performance in commercial, scientific and multiprogrammed workloads. An evaluation study of fine-grained producer-consumer communication for cache coherent multiprocessor has been presented in [8]. An ISA-support, similar to the one used in our framework, for remote write instructions has been studied, where the programmer manually inserts these instructions as well as prefetch instructions at the consumer tasks.

Speculative techniques have been presented in [45], which decouple the cache coherence in two phases and allow the processor to continue execution when the block coherence status is not guaranteed. An interconnect-aware coherence protocol has been introduced in [27]. Coherence techniques capable of exploiting a heterogeneous interconnect architecture has been described. A hardware mechanism for detecting producer-consumer sharing has been proposed in [26]. The mechanism enables the producer node to function as a home directory node in order to reduce the number of transactions needed to communicate the shared data block as well as to eliminate the need for remote memory access on the producer node. A hardware implementation of a form of lazy release consistency for multiprocessors has been

presented in [55]. Writes are buffered into the write buffer and serviced by the directory without stalling the producer. Only when releasing the lock, the producer may be stalled until all outstanding writes have been confirmed by the memory.

Low-Power MPSoCs Architectures. Energy-efficient cache coherence methodologies have been recently proposed for general-purpose computing systems. Jetty [81] is a family of snoop filters designed to reduce power in snoopy bus-based multiprocessors. Local data cache activities are monitored and recorded accordingly in a special cache-like hardware buffer, in order to infer the sharing and cached status of particular cache blocks. Jetty targets large server machines and its application in embedded systems, where significantly smaller on-chip memories are used may result in significant overhead in terms of hardware area and power. RegionScout [80] is another technique that exploits coarse-grain data sharing information to reduce energy for caches and bus traffic. By identifying memory references to different regions, the snoop controller can filter out remote references that are not relevant to its local activities, thus saving power to the cache system. A snoop filtering technique is proposed in [124]. Hardware and OS support is introduced to keep track of the cache content with respect to shared data and disable snooping when determined that the local cache no longer contains valid blocks of shared data. Region-based snoop filtering mechanism has been proposed in [86]. The technique eliminates a large number of redundant snoop operations. Program access semantics are exploited in [15] to reduce snoop power. Access patterns to shared variables are used to eliminate unneeded snoop cache probings.

Compiler Support for MPSoCs. Developing efficient parallel programs or

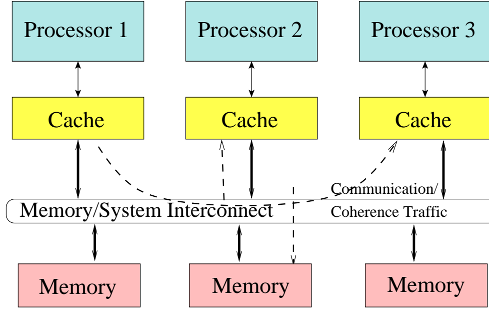

Figure 4.1: Shared memory multiprocessor organization

porting existing sequential programs to efficiently use multiple processors has been a major challenge. Automatic extraction of parallelism has been developed and well-understood for scientific programs [52, 66, 73]. However, efficiently uncovering and modeling parallelism in less regular programs has proved to be a significant research problem. An automatic MPSoC design space exploration approach is proposed in [111]. The system starts from a sequential application specification and guides the designers through a fast and early in the design exploration of parallel implementation platforms. Compiler-based loop transformation techniques for improved inter-core data reuse have been proposed in [25]. Loop iterations are reorganized so that the shared data is accessed within short periods of time. In [32], an infrastructure is proposed where the programmer can explicitly specify the intended parallelism in the program.

4.3 Motivation and Overview

The shared memory multiprocessor organization, as depicted in Figure 4.1 has gained popularity in the embedded domain due to its simplicity of implementation

and well-understood and efficient programming and communication models. The use of local caching is required in order to control the heavy bandwidth demand on the shared interconnect, which is typically implemented as a high-speed system bus. Caches, however, introduce the problem of coherency, which is resolved through coherence protocols.

4.3.1 Inter-Core Data Sharing: To Invalidate or to Update?

The remote-invalidate coherence policy requires snoop-induced cache lookups for both read-miss and write-miss transactions. Every cache write-miss or write into a clean cache line results in a write-miss transaction interpreted as an *invalidate* by all other processor nodes "snooping" the bus or by the directory controller. On the other hand, a read miss for a data that is modified in a remote cache, similarly results in a transaction snooped by the remote owner of the data (and every other processor node in the system as well), which in turn writes back the modified cache line to memory and to the requesting processor. Such lookups are needed since no information exists whether a particular memory reference is to a private or a shared data, and also whether at the time of the memory reference that data is present in the local cache. Write-misses are essentially treated as invalidate requests, while read-misses need to be acted upon since the requested data may be present and modified in the local cache. This protocol has the distinct advantage of requiring much less interconnect bandwidth thus enabling more processor cores in the system.

The remote-invalidate protocol, however, exhibits two major shortcomings.

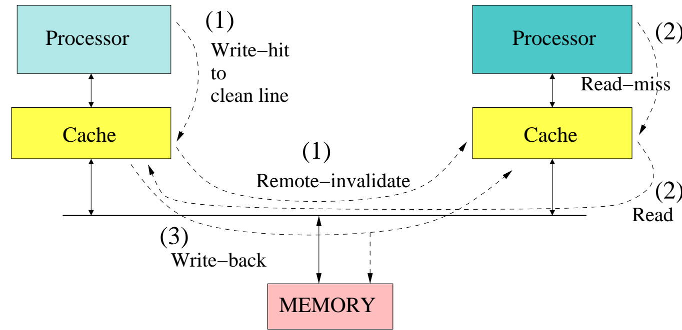

Figure 4.2: Bus transactions involved in communicating data

First, since each transaction results in cache lookups at all other processor cores, *significant amount of power* is consumed - up to 40% of the total cache power [33, 69]. Second, communicating data between a producer and a consumer processor is achieved by three transactions per cache line, resulting in both *performance impact* on the consumer tasks in terms of *coherence misses for all the shared data cache lines* and an *increased number of transactions*. Figure 4.2 illustrates the steps involved in communicating a data item placed in one cache line. The first step involves the producer processor modifying the data in its local cache. The first write results in the invalidation of any outstanding copies of that data in all remote caches. Subsequently, when a consumer processor tries to read that data, *a read miss always occurs*, which is then transmitted and observed by the producing processor. In result of this, the producer writes-back the requested data as a third transaction involved; the data would then be written to both memory and the consumer's cache. As this process repeats for each production-consumption cycle and involves three transactions per cache line, the performance impact on the consumer is significant.

On the other hand, remote-update coherence protocols require no snoop-

induced cache lookups for write- or read-misses by the consumer cores. They, however, require that all the writes are snooped (or handled by the directory) and propagated to the local cache if the block written by the reference is present in the cache. Remote-update protocols also require write-through caches so that all writes are immediately observed by all the processors in the system or by the directory. The advantage of these protocols is that no coherence misses exist since the cached data is updated in all the consumer processors for each write. Conflict misses for shared data are still possible, however, if no special actions are taken to lock or preserve the shared cache lines in the cache. Nonetheless, the requirement that each write is immediately propagated through the system to all the remote consumer caches and the memory creates an overwhelming demand for interconnect bandwidth. This effect has made the remote-update coherence protocols impractical for many system.

4.3.2 Cross-Layer Integration for Data Communication

The proposed framework achieves the advantages of both protocol classes through a customization process that spans the layers of the compiler, operating system, and hardware architecture. No snoop-induced cache lookups are required on read- and write-misses or directory actions. The final state of the shared-data caches lines are propagated and updated in the remote consumer caches when the producer task has completed its update on each line. Moreover, since there may be multiple writes to the same cache line before the producer task releases the shared

data, a simple compiler support is introduced, which ensures that only the last write to a shared cache line, prior to exiting the producer's synchronization region, is propagated to the common bus and triggers an update at the consumers' caches. This is achieved with the help of a *store-with-remote-update* instruction, which behaves as a normal store with the addition of enforcing a write-through and a remote-update coherence transaction. Such an instruction support can be trivially added to any modern microprocessor core. From now on we refer to such an instruction as a *st.update* instruction. The system software is provided with information regarding the shared buffers for each task, and it allocates them in a pre-set memory region as described below, or assigns a unique identifier for each such shared buffer.

Communication between tasks is always implemented through the careful use of synchronization mechanisms, such as locks and barriers. Synchronization allows the producing and the consuming tasks to acquire exclusive access to the shared data - it is possible for a set of consumers to acquire a simultaneous read-only access to the shared data. In most of the cases, the code which accesses the shared data is surrounded by synchronization primitives for acquiring and releasing a lock. The code between these two points is usually referred to as a *critical section* and the synchronization primitives enforce mutual exclusion of the critical sections on the producer and the consumer tasks. Figure 4.3 depicts the code structure of a synchronized communication between the two tasks. When the producing task is within its critical section, it is guaranteed that the consumer is outside his critical region and vice versa. The proposed technique requires that producer and consumer tasks are synchronized in this way. This requirement is fundamental for the release

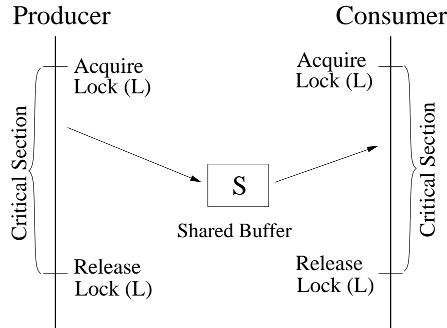

Figure 4.3: Synchronized inter-task communication

consistency memory model, a form of which the proposed methodology implements. Cache coherence for the shared data will not be guaranteed to the consumer core, until it has successfully entered its critical section.

With such synchronized producers and consumers, coherence actions are needed since it is possible that the producer cache is left with modified cache line after the producer task has left the critical section; similarly on the consumer cache side there may be valid cache lines left containing parts of the shared buffer after the consumer task has left its critical region. Such situations are handled by the write-invalidate coherence protocol, which writes back cache lines from the producer cache and invalidates cache lines from the consumer cache. As explained above, this behavior leads to excessive power consumption, longer communication latency (requiring three bus transactions), and increased bus traffic.

Yet, if it can be ensured that upon exiting its critical region the producer core has pushed the latest write for each cache line of the shared buffer to the consumer caches and to the memory, no coherence actions would be required and the most recent copy of the data will be present either in the consumer core caches (if it

is not evicted by local cache traffic) or in the memory. Figure 4.4 illustrates the main aspects of the idea. The fundamental components of the proposed cross-layer specialization methodology can be outlined as follows:

1. Invalidate elimination. A write hit into a clean cache line does not generate a write-miss on the bus as in the invalidate coherence protocol. Such a miss, which triggers remote invalidates, is not needed since the content of the cache line is guaranteed to be propagated to both remote caches and memory efficiently and in time.

2. Application-triggered remote update. Multiple writes, if present, to a cache line are performed with normal store instructions. The write-back cache will not have them propagate to the bus (unless the line is evicted) until the last write for the critical section to that cache line is executed for which the compiler has used the *st.update* store instruction. Instead, if the cache line data is present in the consumer's cache, the last write to it by the producer is propagated and the consumer cache has the most up-to-date copy of the data. If, on the other hand, that cache line has been evicted on the consumer side, the memory has the most recent copy. Note that the consumer processor can only try to read the shared data after the producer task has exited its critical region. In order to perform the remote update, the consumer cache controller must look for bus transactions triggered by a *st.update* instruction; this is the only bus transaction for which a consumer cache needs to look for on the common bus. Furthermore, a local cache lookup for an update is performed only if the write transaction refers to a shared buffer associated with the consumer task.

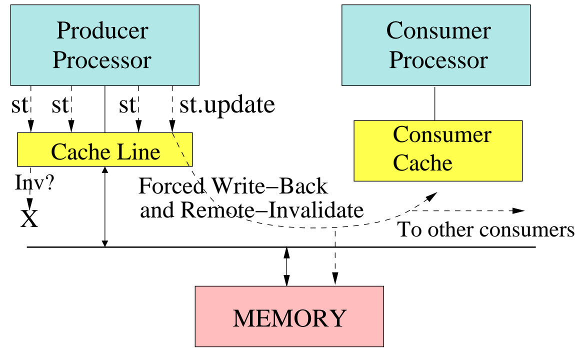

Figure 4.4: Propagating updates with explicit store.update

3. Eliminating coherence transactions. Since all the cache lines modified by the producer task are guaranteed to leave the local cache before the producer critical section is exited and the consumer section entered, there is no need for the producer node to snoop for read-misses on the bus. Similarly, there is no need for a consumer node to snoop for write-misses since it is guaranteed that the shared data present in the local cache will be eventually updated and this update will always occur before the consumer task enters its critical section. For the directory-based protocols, there will be no coherence messages from the producer, while in its critical section, other than the single final update per shared cache line, and no coherence transactions regarding this same shared data from the consumers.

Through the mechanisms outlined above, performance is improved, and more specifically the latency in reading shared data, due to the significantly reduced transactions needed in producer-consumer form of communications, namely from three down to one performed in advance of reading. Moreover, performance is further improved by the reduction of interconnect traffic, which results in fewer

communication scheduling conflicts.

To implement the aforementioned methodology, a number of issues need to be resolved. First, a compiler support (or software designer support) is required so that the *st.update* instruction is used efficiently and correctly. As described above, its purpose is to propagate the last write to a shared cache line before the producer critical section is left. This is achieved through a loop transformation which unrolls the appropriate loop dimension in order to have the last access to the cache line isolated and translated to a *store.update* instruction. Additionally, the OS needs to be informed about the shared buffers involved in the communication, so that each such buffer is specially allocated, or assigned a unique identifier used at the hardware level to recognize memory references to each specific shared array. At hardware layer, the cache controllers need to be adjusted accordingly to recognize the references to the different shared buffers and perform the remote-update operations triggered by *store.update*.

4.3.3 Cache Way Partitioning for Low-Power Data Sharing

Even though the framework outlined about achieves improved latency and reduced interconnect bandwidth demand, one can also expect a positive impact on energy as a result of the reduced communication transactions. Since the mechanisms for distinguishing shared from private data are required by the proposed inter-core communication technique, it is natural to extend the proposed methodology with a support for cache way-allocations, which would map and confine private and shared

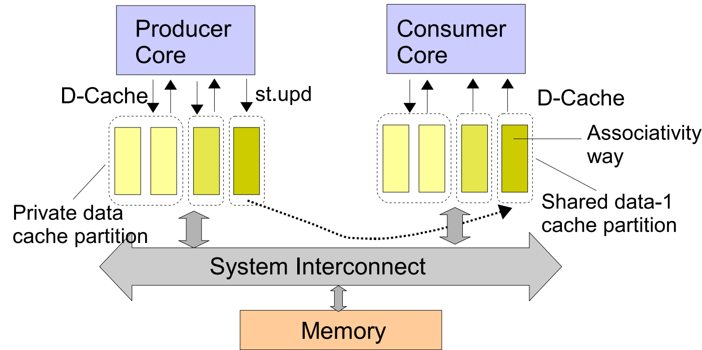

Figure 4.5: MPSoC cache partitioning

data in different cache ways, thereby saving significant amount of power on cache accesses and eliminating interference between private and shared data.

Reconfigurable cache architectures which allow configuring the associativity ways and way size of a cache have been proposed so as to customize the cache to the requirements of the application [11, 37]. If a task requires fewer associativity ways, only the required ways are kept active, reducing the energy consumption of the cache memory. Previous work on reconfiguring caches have mostly focused on a single task and processor core [133].

To this end, the proposed framework leverages recent configurable cache architectures, more specifically selective cache way allocation, to isolate shared data buffers and private data into separate cache partitions and thus achieve sizable power reductions. The full advantages of remote-update policies are achieved through shared data isolation in the consumers' caches. Figure 4.5 illustrates the fundamental operating principles of the proposed methodology. The coupling of cache partitioning and the utilization of store-update instruction that propagates a write

in the producer core implements a low-power remote-update cache coherence policy that not only ensures that the updated shared data at the consumers does not interfere with other data but also performs this update only when it is necessary for the correct execution of the consumer thread. Such partitioning will result in the following benefits: First, data cache accesses triggered either by the processor or the coherence mechanism will need to access only a cache partition instead of the entire cache structure, resulting in ***significant power reductions***; And second, interference (caused by both processor and coherence activities) across private data and the several shared data buffers is eliminated - this in turn enables the efficient implementation of the proposed inter-core communication framework.

4.4 Compiler Support

Knowledge about the shared data is readily available to the embedded system designer and to a certain degree is also available to the system compiler. Typically, the embedded application is designed as a set of interacting parallel threads. Each thread works on different pieces of the data set and communicates with the other threads. The multiple producer/consumer threads run within the same address space, and as such, access the same data memory. The actual communication between the threads, however, is achieved through the common access to a small set of data items, typically a few data buffers.

4.4.1 Shared Memory Identification

Clearly, in order for the compiler to perform the required transformations, it needs to know which array is the shared array that the producer task writes to. It is possible that there are multiple arrays, some used for intermediate results, but only one is used to communicate shared data with remote tasks. This information can be easily provided to the compiler by the software developer who has complete knowledge regarding the shared data organization - an interface identical to the C-language *pragma* directive can be used for this purpose. The manual annotation of the shared buffers can be done very efficiently for a large class of software kernels with regular (affine) access patterns, such as signal, image, voice, and numerical computations.

On the other hand, automatically identifying the set of shared data objects between parallel threads of computation has been one of the prominent problems in the compiler research community for the last decade, known as the *pointer analysis problem* [41]. The objective of pointer analysis is to determine all possible run-time values of a pointer in order to obtain information regarding the memory objects which can be potentially accessed by that pointer. Pointer analysis together with *escape analysis*, i.e. determining when a pointer leaves the scope of a procedure and therefore can be used to access the data object within the procedure assigned to it, are used to determine the shared data items across threads [97]. Even though it has been shown that the static pointer analysis is an undecidable problem [59, 93], various approximation algorithms have been offered which produce solutions with

different precision. An imprecise solution may be quite conservative and include data objects, which are not accessed by the pointer. Recent solutions have achieved efficient run-time complexity with high accuracy of the analysis results [30, 99, 17].

Since the proposed methodology requires information regarding the shared data buffers between producer/consumer threads any of the aforementioned alias analysis techniques can be applied to extract the required knowledge. In many cases, nonetheless, the information regarding the shared memory objects between the software threads can be easily provided by the software developer to the compiler/linker infrastructure. We have followed such an approach in our experimental study.

4.4.2 Loop Transformations for Software-triggered Remote Updates

The proposed methodology relies on the producer task to utilize the *st.update* instruction efficiently so that only the last write to a cache line from a shared data array is propagated to memory and remote consumer caches. It is noteworthy that this needs to be performed only at the producer processor side and on the shared array that is written by that producer. Such transformations on the producer's code can be very efficiently performed for the cases where the shared array is traversed (for writing) by using linear or *affine* indices, which are functions of the loop dimensions.

Figure 4.6 demonstrates one such case where the shared array is traversed linearly and written to an element at a time. Even though this example is very simple, many numerical and digital signal processing kernels access their output buffers in

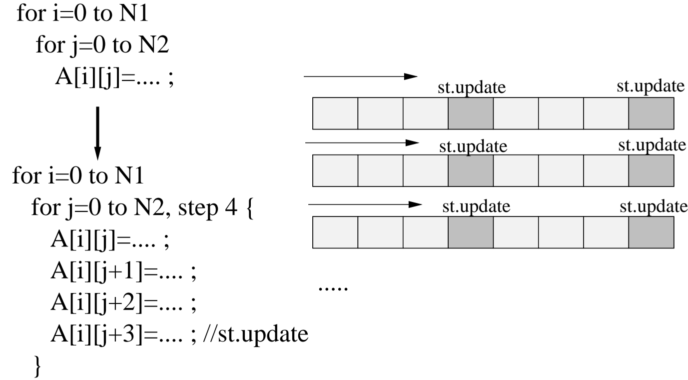

Figure 4.6: Transformations for row-wise array traversal with *st.update* support

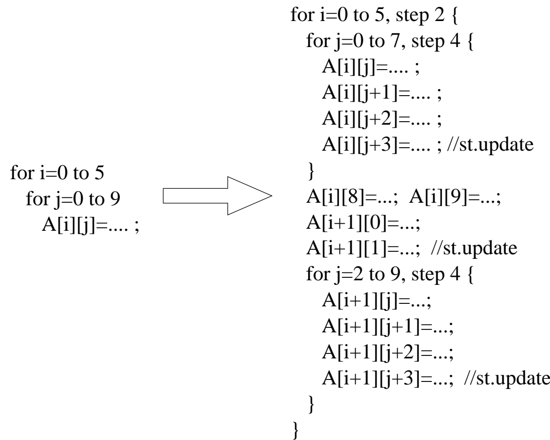

Figure 4.7: Transformations for row-wise traversal with "irregular" row sizes

this way. As Figure 4.6 demonstrates, the loop can be simply unrolled (if it is not already unrolled for scheduling purposes) which exposes the last write to the cache line. Of course, it has been assumed here that the array has been allocated on cache line boundaries in memory following a row-major order and also that the cache line contains four words. The alignment property can be easily established by the compiler, while the cache line size must be known by the compiler or software developer performing the transformation. For the cases where the row size is not multiple of

the cache line size, the outer loop dimension needs to be unrolled accordingly in order to have several instances of the innermost loop, each accessing the array row starting from an index, which is aligned at cache line boundaries. Store instructions, one for each word within the unaligned cache line, are subsequently used in between the instances of the innermost iterations. Figure 4.7 illustrates this principle.

A more general example of a loop nest accessing an array for writing is shown in Figure 4.8. Here the leading array index is not from the innermost loop dimension, which implies that the array is not accessed linearly through the cache lines - in this case it is accessed column-wise. For this situation, in order to expose the last access to a cache line, the leading loop dimension, the one that forms the rightmost and least significant array index, needs to be unrolled accordingly. After unrolling the outer loop, the traversal of four columns is exposed with one of them marking the last access to a cache line. It can be seen that if the array is indexed using a linear function of a loop index for each array dimension, then the loop dimension corresponding to the leading (i.e. rightmost) array index needs to be unrolled appropriately to expose the last write to a cache line.

The scenario described above implies that all loop dimensions are used in the formation of the array index, i.e. the array is traversed only once within that loop. However, if some subset of the outermost loop indices are not used in the array index formation, as shown in Figure 4.9, then within that loop nest the array is traversed multiple times. Since our methodology requires that the last write to a cache line is propagated to memory and remote caches, it needs to identify the last traversal of the array and perform the transformation described above. Note that executing the

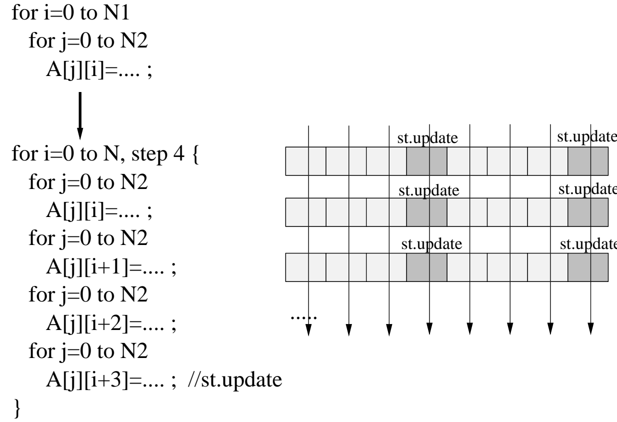

Figure 4.8: Transformations for column-wise traversal with *st.update* support

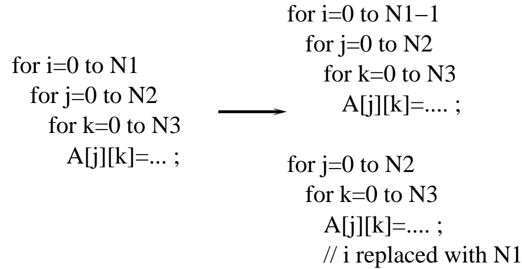

Figure 4.9: Loop peeling for *st.update* support

st.update instruction for each array traversal would not result in incorrect results in terms of coherence - it will only result in redundant bus transactions and remote updates. Eliminating these redundant write updates can be achieved by separating the loop dimensions responsible for the last traversal of the array in a transformation that we refer to as *loop peeling*. Figure 4.9 shows the initial loop with the last array traversal "peeled out" from the main loop. The peeled-out loop nest, which follows the main loop is the one on which the unroll-based transformations have to be applied. In the new peeled-out loop nest each of the new loop iterations participate in the array index formation as a separate dimension. The outer loop dimensions are

```
for i=0 to N1                    for i=0 to N1-1
  for j=0 to N2                    for j=0 to N2
    for k=0 to N3        ---->       for k=0 to N3
      A[j][k]=... ;                    A[j][k]=.... ;

                                 for j=0 to N2
                                   for k=0 to N3
                                     A[j][k]=.... ;
                                     // i replaced with N1
```

Figure 4.10: Transformation for *while* loops with unknown at compile-time upper bounds

no longer present in the peeled-out loop and in the body of that loop they need to be replaced with their respective upper bounds. In the example shown in the figure, the loop is peeled from the outer loop dimension i. The variable i is then replaced with $N1$ within the body of the peeled loop. The unrolling transformation can now be applied only on that loop, which performs that last traversal of the shared array before exiting the critical section.

Finally, *while* loops with unknown at compile-time upper iteration bounds can be handled with a simple transformation, resembling in spirit the loop peeling transformation. Figure 4.10 demonstrates the transformation. First, a new while loop is extracted such that its upper bound is reduced so as to ensure that it is a multiple of four (the assumed number of words within the cache line; this constant, however, can be trivially parameterized). Without a loss of generality we have also assumed that the starting index of the while loop is either 0, or a multiple of four. If this is not the case, then it can be easily aligned through a simple "prologue" code that executes only the first few iterations until the index becomes of the required

form. The resulting while loop is then trivially unrolled in order to expose the last write to a cache line, which will utilize the *st.update* instruction. Afterwards, a simple "epilogue" code is inserted that executes the last few iterations of the original loop that were separated in order to align the upper bound of the unrolled while loop to a multiple of four. The last write in that code is ensured to use the *st.update* instruction.

The majority of signal processing and numerical kernels access their output buffers in the manner shown in the examples above, i.e. using linear (affine) indices to access the shared buffer. If, however, some of the stores to the shared buffer are inside conditional blocks, e.g. *if-then-else*, it is possible that the *st.update* instruction may not execute for some cache lines. Such situations can be treated in the following way. If the execution of the last write to a shared cache line is predicated on a condition then *st.update* instructions will be used for both this write and for the last preceding store instruction to that cache line that is not predicated on a condition. In this way, it will be ensured that regardless of the condition value, the updated shared cache line will always be propagated to the remote consumers. It can be noted that this approach is conservative and may result in multiple *st.update* instructions being executed per cache line. Since the majority of modern high-end processors feature a write buffer in order to avoid blocking the processor when a cache line needs to be written-back, if several *st.update* instruction to a cache line are executed close to each other, they will be naturally combined by the write buffer to a single write/remote-update operation, which will absorb most of the overhead of multiple remote-updates.

4.5 System Software Support

It is essential for the proposed framework that each processor core and its cache controller are aware of the shared buffers involved in any communication for that core. It is also important that the cache controllers are able to identify whether a memory reference placed on the bus belongs to a shared buffer operated by the core. Such an identification is needed in order to perform the remote-update triggered by a remote producer's *st.update* instruction, as well as to filter out all the unnecessary snoop operations. For the case of directory-based protocols, such an identification is only needed at the directory controller. This information is maintained and provided to the hardware support by the system software.

4.5.1 Memory Reference Identification

In the very beginning of its execution, each task is required to notify and register with the system software all of its shared buffers. This can be easily accomplished with a system call provided for that purpose. The software developer would use that function to register each of the shared buffers by specifying their start address and size to the OS. At this step, a mechanism is needed that will enable the hardware cache controller, or the directory, to uniquely identify references to each such shared array. Two alternative approaches are possible in terms of operating system support for this functionality.

1. Explicit Identification. The OS assigns an unique identifier for each of the shared buffers - the *Shared Data Identifier (SDI)*. The SDI is assigned to the page

in which the shared data is allocated and maintained as part of the memory mapping for that page in the page table. Since the SDI is used as an identifier, its size will be determined by the total number of shared arrays in the system to be supported by our approach. For instance, a 4-bit SDI covers a total of 14 shared buffers with one value (0000) reserved for private memory references, and one value (1111) for references outside the targeted application tasks, such as kernel data, global scalar data outside the tasks, etc. For these type of references (with SID=1111), a default coherence protocol mechanism is triggered.

The SDI is also provided to the hardware Memory Management Unit (MMU) when the mapping for that page is stored. From the MMU the SDI is used by the hardware support for two important purposes. First, it is used by the *st.update* instruction when placing the special remote-update transaction. Second, on a cache miss when a memory request transaction is initiated, the SDI for that address is used to inform the other nodes, or the directory, of the nature of the referred region. As explained above, an SDI of 1111 refers to regions containing, for instance, kernel data structure, for which the default coherence mechanism is to be enabled. For all references belonging to the shared buffers targeted by our methodology and the task's private data, the memory requests do not require any coherence related activities at remote processors. At steady execution state of the system, the interconnect traffic will consist almost entirely of coherence-unrelated memory references.

2. Implicit Identification through Allocation. The alternative strategy for identifying the shared arrays without the need for explicitly assigning an SDI is to allocate these arrays at dedicated segments in the address space of the task.

For instance, a large segment from this space can be allotted for shared arrays only. This segment can be identified by a small number of most significant address bits. For instance, an extreme case would be to use the most significant bit for this, in which case the entire upper half of the address space is dedicated for shared arrays. Clearly, in practice the size of this segment can be much smaller and thus defined through a group (fixed) of most significant bits. Within this segment, each shared array will be assigned to an unique sub-segment, which is further identified by another small group of address bits. In a sense, these small group of bits will correspond to and represent the SDI for that array as described above. With this approach, the hardware required at each cache controller or the directory for identifying references to shared arrays is minimal and consists of comparator for the group of most significant address bits defining the shared memory segment. The particular SDIs will be subsequently used by the cache controller or the directory to execute the required cache update.

4.5.2 Multi-Tasking Support and False Sharing Avoidance

In modern MPSoCs, each core typically executes multiple tasks by using the widely available support for preemptive multitasking in modern embedded operating systems. Allowing multiple tasks to share a core is clearly a required feature for modern applications, which typically consist of multiple tasks with varying complexity that are usually significantly more in number than the available processor cores.

The proposed inter-core communication technique does not require any special support from the system software and will operate correctly when multiple producers share a processor core. It is possible that the multiple producers interfere in the cache for their producer buffers. This, however, will not introduce inconsistencies since a partially filled cache line by a producer that was preempted by another producer, will be brought back to the cache when the original producer is scheduled for execution afterwards. Furtheremore, it is possible that the producers share the same buffer. In this case, however, the producers will be synchronized so that they do not overwrite each others data, thus effectively rendering this situation similar to the case where each producer operates in its own subset of the shared buffer, which as described above is handled efficiently by the proposed technique. Since write-allocate policy is often used in local caches of multi-core systems, the partially filled cache line will be brought back into the cache (because it was marked as dirty when it was evicted and therefore written back to memory or to the next level cache), the original producer will continue its execution from where it left off and when the cache line is complete, the *st.update* instruction will propagate that cache line to all relevant consumer cores. Meanwhile, the consumer threads expecting to read that data will, of course, either be executing outside the critical section for that data or be blocked. The different producers will operate on distinct memory buffers, identified as such by using any of the identification mechanisms described in the previous sub-section.

Certain situations of false sharing in cache lines, however, could result in data inconsistency if not handled properly. This could happen if a producer task's shared

data is mapped to the same cache line with a private, or unrelated shared data, of a remote consumer or a producer task. The cache line could be brought in to the remote cache and subsequently updated through *st.update* with a stale value of the unrelated (to the first producer) data. This situation can be easily avoided if shared (and private) data are allocated at cache line boundaries. Such a restriction will not impose any practical constraints as shared data blocks are typically much larger in size than a single cache line and can be easily aligned by the compiler at cache line boundaries; standard compiler "pragma" directives exist for requesting such alignment in all modern compilers.

4.6 Cache Partitioning for Low-Power Data Sharing

The proposed framework aims at low-latency communication for the shared data by elimination of the coherence misses. Clearly, the improvements on performance would strongly depend on whether the consumer cache can capture the shared data in its cache and prevent it from being evicted by other unrelated local accesses. This would depend on the shared data size and the cache organization - size and associativity. Isolating the shared data in a separate subsection of the cache would not only prevent this interference, but it will also have the tremendous benefit of significantly reducing dynamic power as each access to the cache would lookup only a subset of the cache resources.

4.6.1 Functional Overview

We include in the proposed framework such a support for isolating shared and private data by leveraging simple cache configurability in the form of way selection. The shared arrays (or groups of them) are allocated into fixed associativity ways of the cache and thus preventing evictions caused by references to other shared buffers or private data. Depending on the cache organization and way size, this approach allows multiple shared buffers to be mapped into the same associativity way. However, these arrays must be properly aligned and allocated in memory next to each other so that they would not evict each other in their allotted way, which essentially behaves as a separate and smaller direct-mapped cache. In this way, not only are interferences with private data eliminated but also significant energy reductions are obtained by eliminating coherence misses and distinguishing private and shared data accesses.

To implement such a cache partitioning functionality, the proposed technique can leverage already existing and evaluated configurable cache architectures. Cache organizations that allow configuring and partitioning the associativity ways and way size have been proposed with the goal of customizing the cache to the requirements of the application [11, 37, 133]. If a task requires fewer associativity ways, only the required ways are kept active, reducing the energy consumption of the cache memory.

Since shared and private data are mapped to different cache ways, one can expect minimum interference between them and thus much fewer cache misses due

to interferences between local and shared data. In order for this approach to provide positive results, a careful trade-off analysis is needed to ensure that for the given cache size the majority of shared and private data can fit in their allotted ways. Otherwise, either shared data or private data may start evicting itself in its cache ways and thus increase number of conflict misses. Energy is saved through both the elimination of coherence misses and the separation of private data and shared data arrays across the cache associativity ways. By distinguishing whether an access is for private data or for shared data, the cache controller does not have to probe unrelated cache ways for a possible hit, thereby saving power. Since this applies for every memory access, the power savings would be significant.

4.6.2 Cache Way Partitioning: Advantages and Pitfalls

The cache way partitioning allocates cache ways across shared and private data, in order to prevent eviction of shared data and reduce power. Several trade-offs, however, need to be carefully examined. By confining the shared and private data to fixed associativity ways, cache misses due to interference between them will be eliminated. However, if the cache size is not large enough, fixed cache partitioning may introduce interference within shared or private data, which can in turn result in extra misses. For example, if a large part of the cache is allocated to shared data, what is lost on the private data ways due to increased missed may outweigh what is gained on the improved shared data handling, and vice versa. Since we are targeting embedded systems, cache size should also be one of the parameters

to such a trade-off. Thus, the flexibility to satisfy both private and shared data is limited. One needs to achieve a balance between possible gains and losses on both sides, which are largely dependent on the number and patterns of accesses to each type of data. In general, way partitioning is beneficial if the shared buffers fit within their allotted ways, while the remaining local data suffers negligible or no miss-rate impact when isolated in the remaining cache ways.

Furthermore, although performance benefits may be limited due to relatively smaller cache sizes in embedded systems, the power reductions are expected to be significant. The cache subsystem constitutes a significant fraction of the total number of transistors in a modern microprocessor and consumes a large part of the total power. For example, it has been shown that in StrongArm-110 [79] the cache contributes to as much as 43% of the total power. A significant portion of the cache power is typically contributed by lookup of multiple cache ways on every access. By having shared and private data residing in different cache ways, the energy cost per access is significantly reduced. For example, if two associative ways are allocated to shared data and two for private data then the energy cost of each cache access is nearly reduced by half. Such energy reductions with no practical impact on performance (and in many cases actually reducing the total number of misses thus improving performance) would be of significant advantage to many power-constrained high-end embedded applications.

The ultimate decision of determining how many cache ways to be allotted to each type of data is delegated to the application programmers who have full understanding and knowledge of the application's data volumes and access frequencies.

Our experience indicates that for most of the application kernels that we have used in our experimental study, accesses to private and shared data are fairly balanced (i.e. with largely equal frequencies of access) and for such situations, the most efficient solution is to assign to each type of data half of the associativity ways. In the cases of unbalanced accesses, a cache profile information can be easily generated and subsequently the application developer can select the point in the fairly small design space that best matches the application requirements in terms of power v.s. performance. Furthermore, shared data arrays must be properly aligned in memory in order to completely eliminate evictions due to self-interference. The information regarding the partitioning of cache ways is passed to the OS and hardware support in a simple bit-mask control register used by the cache controller at run-time to decide which ways need to be looked up or the new data allocated to.

4.7 Hardware Support

A hardware support is need to capture the information regarding the shared buffers and to allow for this information to be used efficiently and in a way specified by the system software. We first describe the hardware support required for the explicit SDI identification and how is this identifier propagated to the remote caches. The implicit identification alternative, which relies on special allocation does not require such propagation of the SDI since each controller can infer the SDI from the reference address.

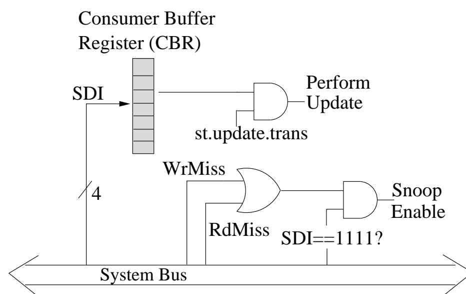

Figure 4.11: Cache controller support for bus-based systems

4.7.1 Shared Data Communication Support

As explained earlier, the system software is provided with and captures the information for all the shared buffers involved in communications. For each shared buffer a unique identifier (SDI) is maintained. The SDI is used by the hardware to distinguish and identify the memory references to specific shared arrays mapped to the particular processor. Any read-miss and write-miss bus transaction would carry on the bus the SDI associated with the missed address. This would enable the processor nodes to disable snooping for all but the references with SDI=1111. As explained above, we support this special SDI for the cases of kernel data or rare cases where the SDI space is exhausted - references to such data (and only that data) would require a fall back to the default coherence mechanism supported by the system. Assigning the SDI to read-miss and write-miss transactions requires no extra bus lines since data lines are not used in these requests and a subset of them can be used to carry the SDI.

One distinct feature that our methodology requires is the support for the *st.update* instruction. It requires no extra hardware beyond the extra encoding

point and the trivial decoder adjustment. In terms of basic cache support it does not require any extra functions as well - whenever decoded it simply forces a write-back/remote-update transaction to the interconnect. In order to eliminate any redundant cache lookups and updates for the case of bus-based snoopy protocols, it is important that a write-back transaction caused by *st.update* is distinguished from ordinary write-backs caused by cache line evictions - this can be achieved by using a special bus transaction control value. Furthermore, in order to avoid redundant update attempts for shared data that belongs to other processors, the SDI needs to be carried as well with the remote-update transaction. Since the write-backs caused by caches are typically distinguished through the bus control signals from special I/O byte writes generated from peripherals to the memory, such byte writes will not interfere with the cache update policy for bus-based systems. It is noteworthy that in the case of the *implicit identification scheme* being used, the aforementioned mechanisms of carrying the SDI within the transaction is not required, since the shared data is allocated in a segment in the address space that can be easily checked by the cache controller or the directory.

Figure 4.11 depicts the required hardware support at the cache controllers for bus-based systems. Similar hardware is needed at the directory controller for non-bus systems. This is the only hardware required by the implicit identification scheme. Since this hardware requires the SDI of the shared buffer referred by the bus transaction, it is extracted from the memory address as explained in the previous section. Clearly, for the explicit identification scheme, the SDI is part of the transaction itself.

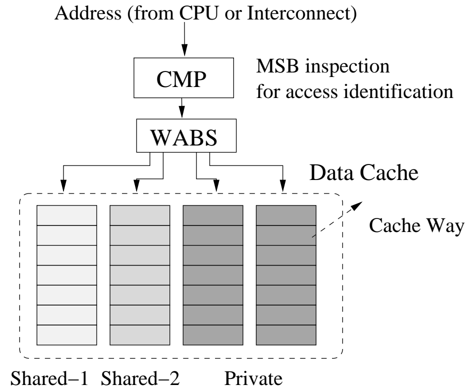

Figure 4.12: Shared/Private-data cache way allocation architecture

The *Consumer Buffer Register (CBR)* captures which shared buffers are consumed by the local processor. The CBR is a bitmask register indexed with the SDI attached to the *st.update* on the producer processor or by the group of address bits determining this identifier in the case of implicit identification through allocation. The corresponding bit in the CBR is set if the shared array assigned to that SDI is consumed in the local processor and, consequently, has to be updated when such write-back occurs on the common bus. The CBR register is written by the system software when it acquires from the application task the set of shared buffers and after assigning them the unique identifiers (SDI). For directory-based systems, one CBR per processor needs to be maintained within the directory controller. If a consumer task is migrated to another core, the CBR bits corresponding to its shared buffers (consumed data) need to be migrated as well to the new core.

The logic that monitors for write-miss and read-miss transaction identifies the cases with SDI=1111 that need to be handled by the default coherence mechanism. These correspond to kernel data structures that can be cached in various caches

and at the steady state of application execution will not occur. Note that the logic formed by these two gates does not constitute a conventional snooping as it does not trigger undiscriminated cache lookups. Similarly the logic that checks for whether a valid remote-update is present on the interconnect allows updates only for write-back transactions and only for shared buffers consumed at that processor.

4.7.2 Cache Way Partitioning Hardware

Figure 4.12 illustrates the general architecture of the proposed selective cache way allocation methodology. The *Way Allocation Bitmap (WAB)* register captures the information regarding which ways are allotted to which type of data. This is a simple bitmap register consisting of a bit per associativity way and written to by the OS. Additional hardware support is required to distinguish at run-time whether a memory reference is an access to shared or private data. For this, the *implicit identification scheme* as described in Section 4.5 could be used in a straightforward manner. A comparator logic would check the few bits from the effective address to determine whether the reference is to a shared or to a private data. Another approach would be to use a special bit within the load/store instruction encoding to specify whether it refers to a private or shared data. Clearly, that approach would have the disadvantage of requiring a static mapping between a load/store instruction and a type of memory reference, which can (albeit in very rare cases) restrict the usage of pointers to access both private and shared data. In our experimental setup we have followed the implicit identification scheme, where the shared arrays are

allocated in a dedicated segment from the address space.

4.8 Experimental Results

To evaluate the proposed technique, we have conducted detailed experiments on a set of embedded multitasking benchmarks representing six different applications. The constituent kernels of these applications are mapped for execution as separate threads on different cores and the communication between them is carried out in a producer/consumer manner. The simulated platform consists of four processor cores and a bus-based interconnect. Each processor executes its thread, which operates on a number of shared data buffers; each thread clearly operates as a both producer and consumer, with the first one reading its data from the system input, and the last one producing its data into the system output (we assume a memory mapped I/O for these). The threads mapped on the different processors execute in a pipelined fashion; each thread uses the output from another thread in the system as its input and it produces the input for another thread. Clearly, the four parallel threads act as both producers and consumers with respect to other threads in the system. This environment is typical for streaming data applications and systems [53], where the input data is streamed through several computational kernels for

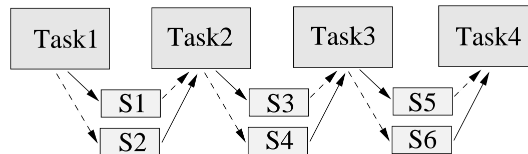

Figure 4.13: Application benchmarks organization.

	Cache Config.	C0 Misses/ Reduct.	C1 Misses/ Reduct.	C2 Misses/ Reduct.	C3 Misses/ Reduct	Total Misses	Reduct.
JPEG-E	2K-2SA	5,048 74.23%	9,100 89.18%	9,301 89.83%	4,566 93.25%	28,015	87.36%
	8K-2SA	18,573 77.95%	39,893 73.09%	37,329 85.49%	18,006 92.56%	113,801	81.03%
	2K-4SA	4,967 75.44%	9,084 89.16%	9,301 89.83%	4,566 93.25%	27,918	87.61%
	8K-4SA	18,028 69.53%	38,816 68.23%	38,220 80.70%	18,006 94.02%	113,070	76.76%
JPEG-D	2K-2SA	1,124 10.14%	9,375 24.54%	13,752 45.84%	5,043 84.39%	29,294	44.29%
	8K-2SA	3,380 3.37%	37,172 22.69%	54,956 45.36%	20,894 80.54%	116,402	43.22%
	2K-4SA	1,069 10.66%	9,375 24.54%	13,752 45.84%	5,043 84.39%	29,239	44.38%
	8K-4SA	1,890 6.03%	38,030 22.00%	59,453 34.98%	21,092 77.75%	120,465	37.92%

Table 4.1: Cache Misses: Baseline vs. Achieved Reductions (%); 32K D-Caches

	Cache Config.	C0 Misses/ Reduct.	C1 Misses/ Reduct.	C2 Misses/ Reduct.	C3 Misses/ Reduct	Total Misses	Reduct.
M-INV	2K-2SA	4,871 76.21%	8,725 92.42%	9,235 90.09%	4,757 88.80%	27,588	88.15%
	8K-2SA	18,062 79.96%	34,882 90.95%	37,329 87.60%	19,081 86.13%	109,354	87.15%
	2K-4SA	4,871 76.21%	8,725 92.42%	9,235 90.09%	4,757 88.80%	27,588	88.15%
	8K-4SA	17,927 80.07%	34,888 89.29%	37,296 86.27%	19,543 83.58%	109,654	85.74%
SPEECH	2K-2SA	5,217 70.29%	8,761 92.04%	9,062 91.81%	4,806 87.89%	27,846	87.18%
	8K-2SA	18,627 65.86%	34,960 85.59%	37,731 78.56%	19,625 77.85%	110,943	78.52%
	2K-4SA	4,707 78.86%	8,761 92.04%	9,062 91.81%	4,806 87.89%	27,336	88.97%
	8K-4SA	17,103 77.83%	34,996 76.50%	36,065 79.23%	21,737 71.50%	109,901	76.61%

Table 4.2: Cache Misses: Baseline vs. Achieved Reductions (%); 32K D-Caches (continued)

	Cache Config.	C0 Misses/ Reduct.	C1 Misses/ Reduct.	C2 Misses/ Reduct.	C3 Misses/ Reduct	Total Misses	Reduct.
SEC-IMG	2K-2SA	4,949 74.80%	1,069 72.10%	55,315 6.87%	3,442 56.91%	72,739	21.96%
	8K-2SA	18,519 77.86%	36,349 45.39%	246,147 0.52%	36,775 9.34%	337,790	10.55%
	2K-4SA	4,949 74.80%	9,017 69.15%	54,712 6.35%	3,442 56.91%	72,120	21.31%
	8K-4SA	18,005 80.08%	36,474 45.64%	242,556 0.78%	38,301 7.27%	335,336	10.66%
SEC-SPCH	2K-2SA	4,929 76.02%	6,659 90.85%	6,903 60.77%	16,638 13.29%	35,129	46.12%
	8K-2SA	18,406 75.98%	26,533 87.78%	27,497 34.80%	46,376 2.64%	118,812	40.46%
	2K-4SA	4,925 76.10%	6,659 90.85%	6,903 88.12%	10,656 38.42%	29,143	68.54%
	8K-4SA	17,981 82.61%	33,075 92.24%	27,893 30.06%	58,521 0.00%	137,470	39.10%

Table 4.3: Cache Misses: Baseline vs. Achieved Reductions (%); 32K D-Caches (continued)

	Cache Config.	C0 Misses/ Reduct.	C1 Misses/ Reduct.	C2 Misses/ Reduct.	C3 Misses/ Reduct.	Total Misses	Reduct.
JPEG-E	2K-2SA	4,970 75.39%	9,084 89.16%	9,301 89.83%	4,566 93.25%	27,921	87.60%
	8K-2SA	18,027 82.56%	35,964 89.79%	37,312 88.00%	18,006 92.56%	109,309	88.44%
	2K-4SA	4,967 75.44%	9,084 89.16%	9,301 89.83%	4,566 93.25%	27,918	87.61%
	8K-4SA	18,026 82.56%	35,964 89.79%	36,949 90.16%	18,006 94.02%	108,945	89.42%
JPEG-D	2K-2SA	1,069 10.66%	9,375 24.54%	13,752 45.84%	5,043 84.39%	29,239	44.38%
	8K-2SA	2,987 3.82%	37,040 23.37%	54,956 45.16%	20,217 83.24%	115,200	43.77%
	2K-4SA	1,069 10.66%	9,375 24.54%	13,752 45.84%	5,043 84.39%	29,239	44.38%
	8K-4SA	1,857 6.14%	37,023 23.85%	54,840 45.81%	20,019 84.56%	113,739	44.83%

Table 4.4: Cache Misses: Baseline vs. Achieved Reductions (%); 64K D-Caches

M-INV	2K-2SA	4,871 76.21%	8,725 92.42%	9,235 90.09%	4,757 88.80%	27,588	88.15%
	8K-2SA	17,928 81.04%	34,876 91.51%	36,982 89.73%	19,064 88.11%	108,850	88.59%
	2K-4SA	4,871 76.21%	8,725 92.42%	9,235 90.09%	4,757 88.80%	27,588	88.15%
	8K-4SA	17,927 82.82%	34,837 92.59%	36,883 90.23%	18,965 89.09%	108,612	89.57%
SPEECH	2K-2SA	4,707 78.86%	8,761 92.04%	9,062 91.81%	4,806 87.89%	27,336	88.97%
	8K-2SA	18,485 71.23%	34,883 91.38%	36,507 86.25%	19,625 77.85%	109,500	83.84%
	2K-4SA	4,707 78.86%	8,761 92.04%	9,062 91.81%	4,806 87.89%	27,336	88.97%
	8K-4SA	17,103 86.82%	34,873 92.50%	35,946 92.58%	19,014 88.86%	106,936	90.97%

Table 4.5: Cache Misses: Baseline vs. Achieved Reductions (%); 64K D-Caches (continued)

	Cache Config.	C0 Misses/ Reduct.	C1 Misses/ Reduct.	C2 Misses/ Reduct.	C3 Misses/ Reduct.	Total Misses	Reduct.
SEC-IMG	2K-2SA	4,949 74.80%	9,017 83.66%	42,755 13.27%	3,442 56.91%	60,163	31.38%
	8K-2SA	18,005 82.41%	35,897 48.91%	181,016 1.54%	36,410 9.76%	271,328	14.28%
	2K-4SA	4,949 74.80%	9,017 88.23%	42,697 16.17%	3,442 56.91%	60,105	34.14%
	8K-4SA	18,005 82.41%	35,897 53.87%	181,705 2.85%	11,762 67.24%	247,369	19.11%
SEC-SPCH	2K-2SA	4,924 76.10%	6,659 90.85%	6,903 90.44%	9,869 43.16%	28,355	71.59%
	8K-2SA	18,133 81.44%	26,435 91.04%	27,464 34.96%	38,535 3.17%	110,567	44.91%
	2K-4SA	4,924 76.10%	6,659 90.85%	6,903 90.44%	6,143 69.33%	24,629	82.42%
	8K-4SA	17,980 82.78%	26,435 91.16%	27,447 89.55%	21,119 78.76%	92,981	86.25%

Table 4.6: Cache Misses: Baseline vs. Achieved Reductions (%); 64K D-Caches (continued)

	Cache Config.	C0 Trans./ Reduct.	C1 Trans./ Reduct.	C2 Trans./ Reduct.	C3 Trans./ Reduct.	Total Trans.	Reduct.
JPEG-E	2K-2SA	9,213	13,452	13,653	4,602	40,920	91.17%
		85.11%	92.68%	93.07%	93.29%		
	8K-2SA	35,090	57,831	54,755	18,042	165,718	84.82%
		86.61%	76.75%	89.65%	92.57%		
	2K-4SA	9,096	13,436	13,653	4,602	40,787	91.44%
		86.22%	92.67%	93.07%	93.29%		
	8K-4SA	34,482	56,242	55,646	18,044	164,414	81.92%
		83.86%	74.29%	84.50%	94.03%		
JPEG-D	2K-2SA	1,294	13,727	22,456	5,044	42,521	51.28%
		18.47%	32.61%	57.14%	84.40%		
	8K-2SA	3,661	54,599	89,791	21,354	169,405	50.33%
		6.56%	30.71%	57.28%	78.81%		
	2K-4SA	1,201	13,727	22,456	5,044	42,428	51.40%
		19.98%	32.61%	57.14%	84.40%		
	8K-4SA	2,041	55,457	94,288	21,655	173,441	44.30%
		11.76%	30.27%	46.04%	75.73%		

Table 4.7: Bus Transactions: Baseline vs. Achieved Reductions (%); 32K D-Caches

	Cache Config.	C0 Trans./ Reduct.	C1 Trans./ Reduct.	C2 Trans./ Reduct.	C3 Trans./ Reduct.	Total Trans.	Reduct.
M-INV	2K-2SA	8,968 87.07%	13,077 94.95%	13,587 93.27%	4,759 88.80%	40,391	91.91%
	8K-2SA	34,510 88.99%	52,309 93.74%	54,755 91.45%	19,202 85.58%	160,776	90.97%
	2K-4SA	8,968 87.07%	13,077 94.95%	13,587 93.27%	4,759 88.80%	40,391	91.91%
	8K-4SA	34,314 89.58%	52,315 92.70%	54,722 90.55%	20,123 81.18%	161,474	89.87%
SPEECH	2K-2SA	9,479 81.41%	13,115 94.69%	13,414 94.47%	4,807 87.87%	40,815	90.73%
	8K-2SA	35,332 76.10%	52,387 89.18%	55,667 79.97%	19,951 76.58%	163,337	81.67%
	2K-4SA	8,804 88.69%	13,115 94.69%	13,414 94.47%	4,807 87.87%	40,140	92.48%
	8K-4SA	33,521 88.59%	52,423 83.00%	53,559 85.48%	23,114 67.24%	162,617	82.73%

Table 4.8: Bus Transactions: Baseline vs. Achieved Reductions (%); 32K D-Caches (continued)

	Cache Config.	C0 Trans./ Reduct.	C1 Trans./ Reduct.	C2 Trans./ Reduct.	C3 Trans./ Reduct.	Total Trans.	Reduct.
SEC-IMG	2K-2SA	9,048 85.62%	13,385 81.17%	79,471 6.18%	3,528 55.61%	105,432	24.17%
	8K-2SA	34,974 86.86%	53,775 62.77%	339,166 0.67%	47,371 7.25%	475,286	14.69%
	2K-4SA	9,047 85.66%	13,369 79.19%	78,703 5.97%	3,528 55.61%	104,647	23.88%
	8K-4SA	34,406 89.53%	53,900 63.13%	335,545 0.88%	49,235 5.65%	473,086	14.92%
SEC-SPCH	2K-2SA	9,038 86.74%	8,870 93.13%	11,255 75.94%	19,622 11.27%	48,785	55.06%
	8K-2SA	34,864 85.94%	35,291 90.76%	44,924 59.10%	50,228 2.43%	165,307	54.30%
	2K-4SA	9,024 86.93%	8,870 93.13%	11,255 92.71%	12,480 32.80%	41,629	73.59%
	8K-4SA	34,394 90.83%	41,816 93.78%	45,320 56.91%	76,141 0.00%	197,671	48.69%

Table 4.9: Bus Transactions: Baseline vs. Achieved Reductions (%); 32K D-Caches (continued)

	Cache Config.	C0 Trans./ Reduct.	C1 Trans./ Reduct.	C2 Trans./ Reduct.	C3 Trans./ Reduct.	Total Trans.	Reduct.
JPEG-E	2K-2SA	9,102 86.17%	13,436 92.67%	13,653 93.07%	4,602 93.29%	40,793	91.42%
	8K-2SA	34,466 90.72%	53,372 93.12%	54,720 91.39%	18,042 92.57%	160,600	91.95%
	2K-4SA	9,096 86.22%	13,436 92.67%	13,653 93.07%	4,602 93.29%	40,787	91.44%
	8K-4SA	34,446 90.77%	53,372 93.12%	54,358 93.31%	18,042 94.03%	160,218	92.78%
JPEG-D	2K-2SA	1,199 20.02%	13,727 32.61%	22,456 57.14%	5,044 84.40%	42,426	51.40%
	8K-2SA	3,207 7.48%	54,467 31.60%	89,791 56.91%	20,319 82.82%	167,784	50.89%
	2K-4SA	1,196 20.07%	13,727 32.61%	22,456 57.14%	5,044 84.40%	42,423	51.40%
	8K-4SA	1,990 12.06%	54,433 32.21%	89,658 57.14%	20,019 84.56%	166,100	51.74%

Table 4.10: Bus Transactions: Baseline vs. Achieved Reductions (%); 64K D-Caches

	Cache Config.	C0 Trans./ Reduct.	C1 Trans./ Reduct.	C2 Trans./ Reduct.	C3 Trans./ Reduct.	Total Trans.	Reduct.
M-INV	2K-2SA	8,968 87.07%	13,077 94.95%	13,587 93.27%	4,759 88.80%	40,391	91.91%
	8K-2SA	34,332 90.03%	52,303 94.30%	54,390 93.02%	19,166 87.64%	160,191	92.15%
	2K-4SA	8,968 87.07%	13,077 94.95%	13,587 93.27%	4,759 88.80%	40,391	91.91%
	8K-4SA	34,312 91.02%	52,247 95.06%	54,291 93.36%	18,967 89.09%	159,817	92.91%
SPEECH	2K-2SA	8,804 88.69%	13,113 94.68%	13,414 94.47%	4,807 87.87%	40,138	92.48%
	8K-2SA	35,182 79.41%	52,291 93.86%	54,443 87.34%	19,951 76.58%	161,867	86.40%
	2K-4SA	8,804 88.69%	13,115 94.69%	13,414 94.47%	4,807 87.87%	40,140	92.48%
	8K-4SA	33,489 93.26%	52,283 94.99%	53,354 95.00%	19,017 88.86%	158,143	93.89%

Table 4.11: Bus Transactions: Baseline vs. Achieved Reductions (%); 64K D-Caches (continued)

	Cache Config.	C0 Trans./ Reduct.	C1 Trans./ Reduct.	C2 Trans./ Reduct.	C3 Trans./ Reduct.	Total Trans.	Reduct.
SEC-IMG	2K-2SA	9,047 85.66%	13,369 88.98%	58,995 12.86%	3,528 55.61%	84,939	34.37%
	8K-2SA	34,410 90.60%	53,305 65.59%	249,751 2.39%	46,853 7.59%	384,319	19.69%
	2K-4SA	9,047 85.66%	13,369 92.06%	59,168 16.03%	3,528 55.61%	85,112	37.02%
	8K-4SA	34,391 90.65%	53,305 68.94%	251,336 2.67%	11,915 66.69%	350,947	23.53%
SEC-SPCH	2K-2SA	9,023 86.92%	8,870 93.13%	11,255 94.14%	11,220 37.96%	40,368	76.69%
	8K-2SA	34,582 89.62%	35,174 93.26%	44,891 59.22%	39,202 3.12%	153,849	59.54%
	2K-4SA	9,022 86.93%	8,870 93.13%	11,255 94.14%	6,143 69.33%	35,290	87.72%
	8K-4SA	34,366 90.98%	35,176 93.36%	44,855 93.61%	21,120 78.76%	135,517	90.56%

Table 4.12: Bus Transactions: Baseline vs. Achieved Reductions (%); 64K D-Caches (continued)

various manipulations until the final output is produced. This is also referred to as pipeline parallelism, the efficient exploitation of which has been the focus of recent research projects in the general-purpose architectural community as well [110]. Except for the very first task in the pipeline, all are both producers and consumers and work on more than eight different data buffers in different synchronization sections. The production and consumption cycles are interleaved so that at each instance the processor is executing in either a producer or in a consumer critical section. The benchmarks organization is depicted in Figure 5.7. This organization is followed in all of the benchmarks that we have used.

The first two benchmarks, *JPEG-E* and *JPEG-D*, are JPEG encoder and decoder. We have constructed by functionally partitioning the standard applications and mapping the corresponding kernels to different cores. For *JPEG-E*, the constituent kernels (in their functional order) are: *left-shift*, *fdct*, *zig-zag-quantization*, and *huffman*. *JPEG-D* has been functionally partitioned into: *dehuffman*, *inverse-zig-zag-inverse-quantization*, *idct*, and *inverse-levelshift*. Our third benchmark, *M-INV*, computes the inverse of the product of two input matrices by using the LU decomposition approach. Matrix inversion is a fundamental operation in many signal processing algorithms. The first computational kernel is *mmul*, which computes the product of two input matrices: *mmul(A,B)*$\rightarrow$*C*. The second kernel, *lu* computes the LU factorization of C: *LU(C)*$\rightarrow$ L, U. Subsequently, the third kernel, *inv*, computes the inverse of the triangular matrices L and U produced by the previous kernel. The fourth kernel, *mmul*, computes the product of *inv(U)* and *inv(U)*, produced by the third kernel. The fourth benchmark, *SPEECH*, represents a speech coder, which

applies the *g721* after applying low-pass filter on the input signal. The kernels of the *SPEECH* benchmark are: *fft*, *low-pass filter*, *ifft*, *g721*. The fifth benchmark, *SEC-IMG*, processes an image by first detecting the edges in it (and extracting them), *edge-detect*, followed by smoothing, *smooth*. It finishes up by compressing the resultant image, *lzo*, and encrypting it through a symmetric cypher, *rijndael*. The sixth benchmark, *SEC-SPCH*, first processes a speech through a low-pass filter, and *adpcm* speech encoder. Afterwards, it digitally signs the data with *sha*, and finishes up by encrypting it with *blowfish*.

We have used the M5 [18] simulator to perform our experiments. Since the proposed methodology is largely independent from the underlying type and class of processors, we have focused our study on the memory and cache subsystem. The simulator is used in system-call emulator mode and is extended with a collection of multi-threading libraries. The simulated hardware configuration is of four processors connected to a shared memory through a common bus. We have experimented with four cache organizations: caches of size 32K and 64K which are representative of modern and future high-end embedded processors, with 2-way set-associative and 4-way set-associative organizations, such as the $IntelXScale$ [47] and $ARM11$ [14]. We have considered two cases for the upper-limit of the amount of data communicated between the tasks; the shared data is below 2K and below 8K. These limits are achieved by providing an input data with the appropriate dimensions/resolution. A processor may use 2 to 4 shared buffers to communicate with its neighbors, depending on specific applications and its role in the data pipeline. The baseline is also configured with four processors connected to a system bus, with conventional

invalidate-based snoop coherence protocol. In the proposed technique all the snoop-induced cache lookups are completely eliminated since the communication in the applications is clearly synchronized and no operating system code is simulated.

To model and evaluate the proposed technique we have modeled the OS and the hardware support by augmenting the M5 simulation infrastructure with the required features. For instance, the functionality of cache partitioning, the *st.update* instruction and its identification by the consumer core's snoopy cache controllers is faithfully modeled inside the simulator. Both the required OS and hardware support, outlined in Sections 4.5, 4.6, and 4.7, are modeled in details and their impact evaluated in a cycle o bus-transaction accurate manner. The required simple loop transformations that expose the last write to a cache line and use the *st.update* instruction to propagate just-in-time the updated data to the consumer cores, have been manually applied on the benchmark kernel codes. Our application benchmarks, as described above, consist of well-known and frequently used signal-processing, numerical, and image processing kernels. All these kernels feature regular and affine loop iteration spaces and data access patterns. As such, it was straightforward to manually apply and model the required simple loop transformations.

In our experimental study we have first focused on evaluating the basics of the proposed framework, that is the inter-core communication ***without applying cache way partitioning***. We have evaluated the impact of the proposed data sharing scheme on the cache misses and the number of bus transactions, which consist of read/write misses and write-back/remote-update events. These two components are a direct consequence of the proposed methodology for inter-core data

communication. The proposed technique eliminates coherence misses, which in turn eliminates a large number of bus transactions, which would have been caused by these coherence misses. Furthermore, the proposed approach also eliminates write-miss (invalidate) transaction, which are needed in the baseline for writes into clean cache lines. As described in the previous sections, the proposed technique does not introduce new transactions since the "forced" write-back by the *st.update* instruction is simply executed earlier in time as compared to the baseline.

Tables 4.1 and 4.4 report the total number of misses for the baseline and the corresponding reductions (in percentage) achieved by the proposed inter-core communication technique with traditional caching organization, i.e. without applying the proposed cache way partitioning. Table 4.1 reports the data for 32K caches, while Table 4.4 focuses on 64K caches. The results for each benchmark are broken down to its kernels components, i.e. for each application benchmark we show the baseline misses and the reductions (in percentage) for each of its tasks mapped to one of the processor cores, e.g. C0-Misses represents the misses incurred by the first core in the platform executing the first task from the corresponding benchmark. The last two columns for each benchmark report the total number of baseline misses and the percentage reductions achieved by the proposed data communication technique. For each benchmark we have also evaluated different cache configurations and shared data sizes. The first row for each benchmark (2K-2SA) reports the achieved miss reductions for a 2-way set-associative cache and a shared data buffers of size up to 2K used for communication between the constituent tasks. Similarly, the other three rows for each benchmark report the experimental data for 4-way

set-associative caches and shared buffers of size up to 8K.

It is evident from the results that the proposed technique significantly reduces the number of cache misses. On average, the miss-rates are reduced about 60%. The achieved reductions vary with different shared buffer sizes and cache sizes and organizations and the nature of different kernels. In general, the larger the cache size is with respect to the shared buffer size, the more significant the achieved reductions are. This is because large caches are more likely to capture the shared data and not evict it due to local cache traffic. Such shared data would be then updated by the producer's *st.update* write-backs and thus provide for repeated cache hits on the consumer side. Higher associativity similarly achieves larger miss reductions as they also tend to preserve shared data.

The miss reductions greatly vary with the different applications due to various factors. The applications have different cache requirements, especially for the private data. With shared buffers and number of accesses to them being of the same magnitude, the coherence misses the technique eliminates are comparable in numbers. Consequently, the remaining private data accesses would determine the overall miss rate. Additionally, the different tasks exhibit fairly different access patterns. Some of them are more prone to cause address conflicts, especially for lower associativity. The relatively small miss rate reductions at the third processor for *SEC-IMG* are due to the fact that *lzo*, which is the third kernel in these benchmarks, exhibits relatively high volume of private data traffic that evicts the shared data thus precluding the benefits of remote updates.

In the baseline cache coherent organization (invalidation-based), the read to a

shared data by a consumer always results in a miss (a coherence miss) that is subsequently handled by the memory system by either fetching the data from memory where it has been written back by the producer or directly from the producer cache if it is still available there. Thus the latency of a load instruction to a shared data by a consumer is the latency of two bus (or any other interconnect) transactions. On the other hand, our technique "preemptively" updates the shared data at the consumer caches in a way driven the producers. Consequently, when the consumer reads a shared data, the large majority of the load instructions hit in the cache, thus the latency of the read operation as perceived by the consumer thread has decreased from two interconnect transactions to a single-cycle hit in the L1 data cache.

Tables 4.7 and 4.10 report the total number of bus transactions for the baseline architectures featuring a write-invalidate snoop cache coherence protocol. The tables also report the reductions (in percentage) of the bus transactions achieved by the proposed framework. The reduced number of transactions accounts for the introduced application-triggered remote-updates, which as described earlier can be thought of as moving the inevitable write-back request from the consumer caches earlier in time. The structure of these tables is identical to the previous tables.

It is evident that the reductions of transactions are proportional and strongly correlate to the reductions in cache misses. On one hand, bus transactions are caused by the read/write misses and write-back/write-update memory requests. Cache misses account for a large part of the overall transaction numbers, especially for write-back caches. On the other hand, since most write-backs/write-updates occur for shared data buffers, they generally follow the number of writes to shared buffers,

which are proportional to the coherence misses which the proposed approach eliminates. For the same reasons of preserving from evictions more of the shared data, the proposed framework achieves better transactions reductions for caches with large sizes and higher associativity. The reported interconnect transaction reductions will be identical to the bandwidth reductions. This follows from the fact that for the majority of signal processing and numerical applications, the sustained bandwidth utilization by the kernels is mostly fixed in time. e.g. the bus transactions generated by the kernels are uniformly distributed in time. Since bandwidth is a measurement of the number of bus transactions for a unit of time, the percentage reduction of bus transactions in this case is identical to the percentage reduction of bus bandwidth.

We now proceed with evaluating the proposed framework in its entirety, i.e. including the mechanism for application-driven cache way partitioning. The next two tables report the achieved energy reductions and impact on miss-rate of the selective cache way allocation methodology. Way allocation is applied *in addition to* the inter-core data communication, which was so far evaluated in isolation. As expected, way allocation can significantly reduce cache energy, while not impacting the already reduced miss-rate or actually in many cases reducing it even further.

Tables 4.13 and 4.15 report the achieved energy reductions and the impact on cache misses of the cache way partitioning mechanism. This data reflects the application of the cache way partitioning scheme compared to the corresponding baseline cache organization. The hardware configurations are largely identical to the ones we used to evaluate the communication mechanism without cache way partitioning. The structure of this and the next tables is identical to the previous

tables. They report the per-core data for each benchmark, while in different rows presenting the data for different cache configurations and shared data buffer sizes.

Clearly, to use in practice and to evaluate the cache way partitioning technique, the underlying cache organization needs to implement at least several cache associativity ways. For this reason, we have assumed a baseline cache organization consisting of 4-way set-associative cache. As the frequency of access and volume of the shared and private data are largely identical for all the application benchmarks we have used, the four ways of the underlying cache are evenly divided into two groups, one for shared data and the other for private data. Each group has two cache ways. Within each groups, the LRU replacement policy is assumed. For similar reasons of feasibility, we do not report results for shared data arrays of 8K in 32K cache configuration, because it is impossible to isolate the shared and the private data within the cache ways and the way-allocation technique simply does not apply in such situations. For the rest of the configurations (shared buffer sizes and cache volumes), we ensure that the shared data fits within the two cache ways. Private data, however, may evict itself at run-time, because of reduced available cache size. This effect can be seen in the results for the third phase (lzo) of *SEC-IMG*, where for some of the configurations the cache misses are slightly increased even though the misses to shared data are entirely eliminated (only cold misses to shared data are suffered in the beginning of execution) and in overall the total number of misses are reduced. However, for most of the cache configurations, the applications can place both the shared data and the private data within the allotted cache ways. For such cases, not only is energy significantly reduced but also are the total number

of misses. The cache misses for these cases are mostly due to cold misses when shared data and private data are first brought in to the cache. Table 4.15 reports the number of misses achieved after applying the cache way partitioning technique and the impact on miss rate. On average, the miss-rates are reduced about 67%.

Table 4.13 reports the achieved energy reductions. These results are based on the energy estimates per access as provided by Cacti-5 [112] for a process technology of 70nm. The format is identical to the previous table, however, showing in pairs the total cache energy (in mJ) for a baseline cache organization with invalidate-based snoop coherence protocol, and the energy reductions (in percentage) achieved by the proposed cache way allocation methodology. It can be seen that the energy reductions for all the applications and configurations are fairly stable and close to 50%, which is due to the fact the cache power per access is almost reduced by half. Both the reduced number of cache ways in which the lookup is performed as well as the reduction of total number of misses are factors in the achieved energy improvements. The dynamic energy for the cache misses is modeled as a sum of the cache access and an access to an SRAM memory of size 256K.

Tables 4.17 and 4.19 report the improvement on system performance. Such impact on performance is measured by comparing average memory read access latency from CPU side against the baseline. The paired numbers in each cell represent percentage improvements of the proposed scheme versus the baseline, with L1 cache miss penalty of 10 and 20 CPU cycles, respectively. The hit latency is assumed at 1 CPU cycle. In general, we observe improvements across the benchmark kernels on memory access latency, up to 29%. This is due to the reduced cache misses

that make the average latency longer. Some of the kernels do not exhibit large latency reductions due to the fact that they spend the large majority of their execution time operating on private data, and as such do not benefit as much from the proposed technique as other kernels which extensively operate on the shared data. The reported latency reductions are quite conservative, as they do not take into account the significantly reduced traffic on the on-chip interconnect as a result of the proposed technique. In our simulations, we have assumed constant L1 miss penalty - 10 and 20 cycles. Also, longer cache miss penalties make such reduction more significant in terms of memory performance, which can be clearly seen by the reported reductions.

Table 4.21 reports the performance numbers after applying the introduced cache partitioning methodology. The table format is identical to the previous tables reporting average memory access latency reductions. The achieved results are within similar ranges. We observe mostly improvement on memory access performance. However, there is a single case where the proposed scheme under-performs the baseline. This is due to slightly increased number of cache misses on the private data part, as private data is restricted to a subset of the cache space now. Such issue is inherited from the cache partitioning technique itself. Nevertheless, such overhead is negligible even in these rare cases, and can be fixed by applying more sophisticated cache partitioning techniques that allow partial sharing between the partitions.

4.9 Conclusion

We have presented a framework for performance, bandwidth, and energy efficient inter-core communication in embedded multiprocessors. The framework achieves a cost-efficient and in-time remote cache update of shared cache blocks through the integrated efforts of compiler, system software, and hardware. Cache way partitioning policy isolates shared and private data in separate associativity ways thus preventing evictions of shared data and resulting in significant energy reductions. The methodology seamlessly integrates the system layers as the application information is captured and utilized through well-defined interfaces and software controlled hardware structures, enabling its application to a broad range of multiprocessor embedded applications.

	Cache Config.	C0 Enrg./ Reduct.	C1 Enrg./ Reduct.	C2 Enrg./ Reduct.	C3 Enrg./ Reduct.	Total Enrg.	Reduct.
JPEG-E	32K, 2K-4SA	10,559 52.14%	205,807 50.73%	30,841 52.27%	21,004 52.03%	268,211	51.07%
	64K, 2K-4SA	13,245 51.51%	261,323 50.35%	38,874 51.62%	26,538 51.45%	339,980	50.62%
	64K, 8K-4SA	50,519 51.62%	1,045,064 50.35%	155,271 51.61%	105,092 51.45%	1,355,946	50.63%
JPEG-D	32K, 2K-4SA	19,914 50.06%	17,285 42.33%	50,300 48.22%	15,448 52.50%	102,947	48.23%
	64K, 2K-4SA	25,279 49.84%	21,637 42.24%	63,451 47.90%	19,458 51.88%	129,825	47.93%
	64K, 8K-4SA	94,831 49.84%	86,282 42.18%	253,605 47.89%	77,614 51.87%	512,332	47.89%
M-INV	32K, 2K-4SA	26,773 52.18%	20,948 53.27%	29,531 52.34%	20,225 52.04%	97,478	52.43%
	64K, 2K-4SA	33,861 50.67%	26,318 52.48%	37,211 51.67%	25,542 51.47%	122,932	51.53%
	64K, 8K-4SA	246,436 50.44%	202,367 51.36%	272,351 50.98%	199,243 50.82%	920,396	50.88%

Table 4.13: Cache way partitioning: Cache energy (mJ) and reductions

	Cache Config.	C0 Enrg./ Reduct.	C1 Enrg./ Reduct.	C2 Enrg./ Reduct.	C3 Enrg./ Reduct.	Total Enrg.	Reduct.
SPEECH	32K, 2K-4SA	87,527 50.73%	11,266 55.66%	100,066 51.08%	1,975,848 50.49%	2,174,707	50.55%
	64K, 2K-4SA	111,106 50.35%	14,007 54.54%	126,891 50.85%	2,511,804 50.15%	2,763,808	50.21%
	64K, 8K-4SA	530,684 50.32%	52,796 54.81%	603,958 50.73%	10,046,272 50.15%	11,233,711	50.21%
SEC-IMG	32K, 2K-4SA	416,263 50.51%	65,394 51.29%	59,209 47.97%	124,630 50.54%	665,496	50.37%
	64K, 2K-4SA	529,033 50.17%	82,813 50.80%	70,930 48.36%	158,324 50.20%	841,099	50.08%
	64K, 8K-4SA	2,273,309 50.17%	348,331 50.77%	336,202 48.73%	620,128 50.21%	3,577,970	50.10%
SEC-SPCH	32K, 2K-4SA	10,166 52.22%	99,027 50.89%	8,095 55.66%	343,371 50.49%	460,659	50.71%
	64K, 2K-4SA	12,746 51.57%	125,657 50.48%	10,043 54.50%	435,267 50.21%	583,713	50.37%
	64K, 8K-4SA	46,077 51.77%	502,173 50.48%	39,971 54.50%	1,142,539 50.25%	1,730,760	50.46%

Table 4.14: Cache way partitioning: Cache energy (mJ) and reductions (continued)

	Cache Config.	C0 Misses/ Impact	C1 Misses/ Impact	C2 Misses/ Impact	C3 Misses/ Impact	Total Misses	Impact
JPEG-E	32K, 2K-4SA	4,967 75.38%	9,084 89.16%	9,301 89.83%	4,566 93.25%	27,918	87.60%
	64K, 2K-4SA	4,967 75.44%	9,084 89.16%	9,301 89.83%	4,566 93.25%	27,918	87.61%
	64K, 8K-4SA	18,026 82.58%	35,964 89.79%	36,949 90.16%	18,006 94.02%	108,945	89.42%
JPEG-D	32K, 2K-4SA	1,069 2.62%	9,375 24.54%	13,752 45.84%	5,043 84.39%	29,239	44.08%
	64K, 2K-4SA	1,069 10.66%	9,375 24.54%	13,752 45.84%	5,043 84.39%	29,239	44.38%
	64K, 8K-4SA	1,857 7.22%	37,023 23.85%	54,840 45.81%	20,019 84.56%	113,739	44.85%
M-INV	32K, 2K-4SA	4,871 76.21%	8,725 92.42%	9,235 90.09%	4,757 88.80%	27,588	88.15%
	64K, 2K-4SA	4,871 76.21%	8,725 92.42%	9,235 90.09%	4,757 88.80%	27,588	88.15%
	64K, 8K-4SA	17,927 82.82%	34,837 92.59%	36,883 90.23%	18,965 89.09%	108,612	89.57%

Table 4.15: Cache way partitioning: Cache misses and impact on miss-rate

	Cache Config.	C0 Misses/ Impact	C1 Misses/ Impact	C2 Misses/ Impact	C3 Misses/ Impact	Total Misses	Impact
SPEECH	32K, 2K-4SA	4,707 78.84%	8,761 92.04%	9,062 91.81%	4,806 87.89%	27,336	88.96%
	64K, 2K-4SA	4,707 78.86%	8,761 92.04%	9,062 91.81%	4,806 87.89%	27,336	88.97%
	64K, 8K-4SA	17,103 86.82%	34,873 92.50%	35,946 92.58%	19,014 88.86%	106,936	90.97%
SEC-IMG	32K, 2K-4SA	4,949 74.80%	9,017 89.85%	54,712 3.64%	3,442 56.91%	72,120	21.84%
	64K, 2K-4SA	4,949 74.80%	9,017 89.85%	42,697 -6.74%	3,442 56.91%	60,105	18.11%
	64K, 8K-4SA	18,005 82.41%	35,897 89.96%	181,705 -3.84%	11,762 67.80%	247,369	19.45%
SEC-SPCH	32K, 2K-4SA	4,925 76.10%	6,659 90.60%	6,903 90.44%	10,656 32.84%	29,143	66.99%
	64K, 2K-4SA	4,924 76.10%	6,659 90.85%	6,903 90.44%	6,143 68.26%	24,629	82.15%
	64K, 8K-4SA	17,980 82.78%	26,435 91.16%	27,447 90.60%	21,119 79.86%	92,981	86.81%

Table 4.16: Cache way partitioning: Cache misses and impact on miss-rate (continued)

	Config.	C0 impact	C1 impact	C2 impact	C3 impact
JPEG-E	1K-2SA	7.89% / 14.89%	0.89% / 1.87%	6.07% / 11.93%	4.56% / 9.13%
	4K-2SA	7.99% / 15.14%	0.80% / 1.67%	5.80% / 11.39%	4.51% / 9.03%
	1K-4SA	7.90% / 14.94%	0.89% / 1.86%	6.07% / 11.93%	4.56% / 9.13%
	4K-4SA	6.94% / 13.19%	0.73% / 1.52%	5.60% / 10.98%	4.58% / 9.17%
JPEG-D	1K-2SA	0.13% / 0.27%	2.95% / 5.50%	2.81% / 5.56%	6.17% / 12.05%
	4K-2SA	0.03% / 0.07%	2.72% / 5.06%	2.78% / 5.50%	6.10% / 11.87%
	1K-4SA	0.13% / 0.27%	2.95% / 5.50%	2.81% / 5.56%	6.17% / 12.05%
	4K-4SA	0.03% / 0.07%	2.69% / 4.99%	2.31% / 4.54%	5.94% / 11.56%
M-INV	1K-2SA	3.12% / 6.31%	8.59% / 16.43%	6.31% / 12.37%	4.70% / 9.36%
	4K-2SA	1.68% / 3.47%	4.46% / 8.92%	3.42% / 6.93%	2.36% / 4.84%
	1K-4SA	3.12% / 6.31%	8.59% / 16.43%	6.31% / 12.37%	4.70% / 9.36%
	4K-4SA	1.67% / 3.45%	4.38% / 8.76%	3.37% / 6.82%	2.35% / 4.80%

Table 4.17: Average memory access latency reduction (32K D-Cache)

	Config.	C0 impact	C1 impact	C2 impact	C3 impact
SPEECH	1K-2SA	0.95% / 1.97%	15.73% / 27.90%	1.88% / 3.88%	0.05% / 0.10%
	4K-2SA	0.67% / 1.39%	15.43% / 27.14%	1.41% / 2.92%	0.04% / 0.09%
	1K-4SA	0.96% / 2.00%	15.73% / 27.90%	1.88% / 3.88%	0.05% / 0.10%
	4K-4SA	0.72% / 1.51%	13.80% / 24.27%	1.36% / 2.81%	0.04% / 0.09%
SEC-IMG	1K-2SA	0.20% / 0.42%	2.25% / 4.59%	1.40% / 2.41%	0.36% / 0.75%
	4K-2SA	0.18% / 0.39%	1.35% / 2.77%	0.10% / 0.17%	0.16% / 0.33%
	1K-4SA	0.20% / 0.42%	2.15% / 4.39%	1.28% / 2.21%	0.36% / 0.75%
	4K-4SA	0.18% / 0.39%	1.37% / 2.79%	0.15% / 0.26%	0.13% / 0.26%
SEC-SPCH	1K-2SA	8.20% / 15.46%	1.38% / 2.87%	11.35% / 19.84%	0.15% / 0.30%
	4K-2SA	8.44% / 15.86%	1.33% / 2.77%	6.50% / 11.37%	0.03% / 0.06%
	1K-4SA	8.20% / 15.46%	1.38% / 2.87%	16.46% / 28.77%	0.27% / 0.57%
	4K-4SA	8.99% / 16.93%	1.74% / 3.60%	5.68% / 9.91%	0.00% / 0.00%

Table 4.18: Average memory access latency reduction (32K D-Cache) (continued)

	Config.	C0 impact	C1 impact	C2 impact	C3 impact
JPEG-E	1K-2SA	7.90% / 14.94%	0.89% / 1.86%	6.07% / 11.93%	4.56% / 9.13%
	4K-2SA	8.24% / 15.66%	0.89% / 1.86%	6.06% / 11.91%	4.58% / 9.17%
	1K-4SA	7.90% / 14.94%	0.89% / 1.86%	6.07% / 11.93%	4.56% / 9.13%
	4K-4SA	8.24% / 15.66%	0.89% / 1.86%	5.97% / 11.73%	4.51% / 9.03%
JPEG-D	1K-2SA	0.13% / 0.27%	2.95% / 5.50%	2.81% / 5.56%	6.17% / 12.05%
	4K-2SA	0.03% / 0.07%	2.84% / 5.30%	2.81% / 5.54%	6.15% / 12.02%
	1K-4SA	0.13% / 0.27%	2.95% / 5.50%	2.81% / 5.56%	6.17% / 12.05%
	4K-4SA	0.03% / 0.07%	2.79% / 5.20%	2.77% / 5.48%	6.11% / 11.93%
M-INV	1K-2SA	3.12% / 6.31%	8.59% / 16.43%	6.31% / 12.37%	4.70% / 9.36%
	4K-2SA	1.73% / 3.57%	4.53% / 9.07%	3.49% / 7.06%	2.43% / 4.97%
	1K-4SA	3.12% / 6.31%	8.59% / 16.43%	6.31% / 12.37%	4.70% / 9.36%
	4K-4SA	1.69% / 3.49%	4.48% / 8.98%	3.48% / 7.03%	2.41% / 4.94%

Table 4.19: Average memory access latency reduction (64K D-Cache)

	Config.	C0 impact	C1 impact	C2 impact	C3 impact
SPEECH	1K-2SA	0.96% / 2.00%	15.73% / 27.90%	1.88% / 3.88%	0.05% / 0.10%
	4K-2SA	0.81% / 1.68%	16.64% / 29.27%	1.58% / 3.28%	0.05% / 0.10%
	1K-4SA	0.96% / 2.00%	15.73% / 27.90%	1.88% / 3.88%	0.05% / 0.10%
	4K-4SA	0.71% / 1.49%	16.44% / 28.93%	1.50% / 3.10%	0.04% / 0.09%
SEC-IMG	1K-2SA	0.20% / 0.42%	2.75% / 5.61%	2.66% / 4.75%	0.36% / 0.75%
	4K-2SA	0.19% / 0.40%	1.59% / 3.25%	0.42% / 0.77%	0.37% / 0.77%
	1K-4SA	0.20% / 0.42%	2.60% / 5.32%	2.19% / 3.91%	0.36% / 0.75%
	4K-4SA	0.19% / 0.40%	1.44% / 2.95%	0.23% / 0.41%	0.16% / 0.34%
SEC-SPCH	1K-2SA	8.20% / 15.46%	1.38% / 2.87%	16.89% / 29.53%	0.28% / 0.59%
	4K-2SA	9.01% / 16.96%	1.38% / 2.86%	16.71% / 29.22%	0.42% / 0.88%
	1K-4SA	8.20% / 15.46%	1.38% / 2.87%	16.89% / 29.53%	0.28% / 0.59%
	4K-4SA	8.93% / 16.80%	1.38% / 2.86%	6.53% / 11.41%	0.03% / 0.06%

Table 4.20: Average memory access latency reduction (64K D-Cache) (continued)

	Config.	C0 impact	C1 impact	C2 impact	C3 impact
JPEG-E	16K, 1K-4SA	7.89% / 14.93%	0.89% / 1.86%	6.07% / 11.93%	4.56% / 9.13%
	32K, 1K-4SA	7.90% / 14.94%	0.89% / 1.86%	6.07% / 11.93%	4.56% / 9.13%
	32K, 4K-4SA	8.24% / 15.66%	0.89% / 1.86%	6.06% / 11.91%	4.58% / 9.17%
JPEG-D	16K, 1K-4SA	0.03% / 0.07%	2.95% / 5.50%	2.81% / 5.56%	6.17% / 12.05%
	32K, 1K-4SA	0.13% / 0.27%	2.95% / 5.50%	2.81% / 5.56%	6.17% / 12.05%
	32K, 4K-4SA	0.04% / 0.09%	2.84% / 5.30%	2.81% / 5.54%	6.15% / 12.02%
M-INV	16K, 1K-4SA	3.12% / 6.31%	8.59% / 16.43%	6.31% / 12.37%	4.70% / 9.36%
	32K, 1K-4SA	3.12% / 6.31%	8.59% / 16.43%	6.31% / 12.37%	4.70% / 9.36%
	32K, 4K-4SA	1.73% / 3.57%	4.53% / 9.07%	3.49% / 7.06%	2.43% / 4.97%

Table 4.21: Average memory access latency reduction with cache way allocation

	Config.	C0 impact	C1 impact	C2 impact	C3 impact
SPEECH	16K, 1K-4SA	0.96% / 2.00%	15.73% / 27.90%	1.88% / 3.88%	0.05% / 0.10%
	32K, 1K-4SA	0.96% / 2.00%	15.73% / 27.90%	1.88% / 3.88%	0.05% / 0.10%
	32K, 4K-4SA	0.81% / 1.68%	16.64% / 29.27%	1.58% / 3.28%	0.05% / 0.10%
SEC-IMG	16K, 1K-4SA	0.20% / 0.42%	2.80% / 5.71%	0.73% / 1.26%	0.36% / 0.75%
	32K, 1K-4SA	0.20% / 0.42%	2.80% / 5.71%	-1.11%/ -1.98%	0.36% / 0.75%
	32K, 4K-4SA	0.19% / 0.40%	2.65% / 5.42%	-0.57%/ -1.04%	0.37% / 0.78%
SEC-SPCH	16K, 1K-4SA	8.20% / 15.46%	1.38% / 2.86%	16.89%/ 29.53%	0.23% / 0.48%
	32K, 1K-4SA	8.20% / 15.46%	1.38% / 2.87%	16.89%/ 29.53%	0.28% / 0.58%
	32K, 4K-4SA	9.01% / 16.96%	1.38% / 2.86%	16.90%/ 29.56%	0.43% / 0.89%

Table 4.22: Average memory access latency reduction with cache way allocation (continued)

Chapter 5

Low-Cost and Energy-Efficient Distributed Synchronization for Embedded Multiprocessors

5.1 Overview

The ever increasing demands of many modern applications for consolidated functionality, including multimedia, data, communication, security and many other capabilities coupled with increased integration densities have resulted in the adoption and utilization of embedded multiprocessor implementation platforms. Such application domains include smart phones, portable media players, navigation devices, and many others. While trying to meet the performance requirements of such applications, embedded multiprocessor systems have encountered challenges that are specific to these architectures and application domains, such as *energy efficiency* concerns in battery-powered devices and *real-time performance* requirements for many time-critical tasks. These domain specific requirements have resulted in new lines of research efforts aiming at adopting and optimizing general-purpose hardware and software organizations to the low-power and real-time requirements of the modern embedded applications.

Multiprocessor architectures for the embedded domain have given rise to some unique problems not present in uni-processor embedded systems, such as inter-core

communication, synchronization, data/code sharing, etc. These issues present challenges for embedded system designers and open new frontiers for developing novel embedded system architectures. The typical availability of application-specific information present at design time has enabled a new set of optimization strategies that aim at capturing and exploiting this information at run-time, in order to achieve energy-efficiency and time-deterministic performance for the particular application program or set of tasks to be executed. One such problem that arises in embedded multiprocessors is the typical need for synchronization among the threads executing on the processor cores. Such functionality is needed in almost all instances where execution progress, data sharing, and communication between the parallel threads need to be carefully orchestrated. It is usually the responsibility of the programmer (or in recent developments of parallel compilation environments, the compiler) to properly use the set of available synchronization operations in order to ensure deterministic event ordering and proper communication between the threads.

Several well known synchronization primitives are usually made available to the software developers/compilers by system libraries or directly by the operating system. Frequently used synchronization primitives include *locks*, *barriers*, *semaphores*, and *monitors*. At hardware level, however, various implementation approaches are being used based on the underlying hardware architecture. Their implementation is often based on certain atomic operations provided by the hardware. Conventional examples of such atomic operation implementations include the paired instructions of *load-linked* and *store-conditional*, or an atomic *test-and-set* instruction. Such atomic primitives ensure that a software implementation of a

synchronization primitive would access a certain *synchronization variable* and subsequently modify it, without this variable being modified by another core in the process.

While such implementation provides a general-purpose support for a comprehensive set of synchronization primitives, its generality comes with the price of significant power and inter-core communication overheads. It has been known that such synchronization can result in severe bus traffic contention when multiple processors compete for the same synchronization variable [10, 12]. In the case when no local caching is available (or when no cache coherence mechanism exists) the processors need to poll the synchronization variable, thus polluting the interconnect to memory with a large amount of traffic and also expending a significant amount of power. When coherent caches are present, the polling is performed at the local cache by the *spin-lock* synchronization primitive, which, however, does not resolve the power problem; significant bus contention ensues too when a processor releases the synchronization variable, which leads to invalidations and subsequent misses in all remote caches. For all these cases the latency of acquiring a synchronization variable is significant. Even requesting an available synchronization variable may take several bus transactions before the processor acquires it. Moreover, even in the presense of coherent caches, the need to read and modify the synchronization variable normally results in two bus transactions.

All these problems stem from the need that all the processors compete to read and modify a shared variable (the synchronization variable) without any imposed ordering. Recently several research projects have proposed hardware based

solutions [78], where a special and centralized hardware controller is introduced that keeps track of the participating tasks and communicates with them to manage synchronization. Such a solution normally achieves much better synchronization performance compared to the general-purpose implementations as it imposes an order in acquiring the synchronization variable by effectively maintaining a queue of the requesting processors for each synchronization variable. In such solutions, however, all the communication in acquiring, releasing, and granting the synchronization variable is routed through the centralized controller. Consequently, to acquire an available lock two communication transactions are still required: one from the requesting processor and one from the controller to grant the access. The controller can also result in silicon area and chip routing overheads, especially in smaller scale multiprocessors if it is to be connected with each core through dedicated communication links.

In this work[127, 126], we propose a completely distributed and decentralized synchronization architecture to address the aforementioned problems. The proposed organization is specifically applicable to shared memory multiprocessors with the cores accessing the memory through a shared system bus. Local to each processor, a very light-weight hardware controller is introduced, which captures the synchronization variables of interest to the local processor. In this way, each processor participates in a completely decentralized and distributed protocol of acquiring and releasing a synchronization variable. Each such local controller monitors the bus for "acquires" and "releases" of synchronization variables of local interest and maintains a precise state of the global status for each variable. The proposed orga-

nization requires no atomic operations for accessing and modifying main memory as it relies on the dedicated hardware controllers to manage synchronization and on the inherent serialization that the system bus imposes on the one-way transactions from processors to memory and vice versa. Best case lock acquisition latency (when the synchronization variable is available) of zero bus transactions is achieved, with the only delay of the core acquiring the bus for communication. Significant performance and power improvements are achieved through this organization. The end results is that the semantic of *queued locks* is implemented in a completely distributed manner with a **near-zero latency lock acquisition and release**. The approach also eliminates bus contention due to synchronization competition, while providing opportunities for precise and **fine-grained power management** to significantly reduce the energy expended while the processors wait on synchronization.

5.2 Related work

A large body of research works exist related to the synchronization problems in multiprocessor systems. The performance impact of synchronization due to bus contention and global communication has been recognized from various perspectives [9, 10, 74, 122, 115]. As observed in these studies, synchronization operations result in significant communication overhead, when multiple processors compete for a synchronization variable, thus causing performance degradation. Furthermore, synchronization operations usually result in elevated power consumption [64, 35]. When a local processor is waiting for its turn to acquire certain synchronization

variable, it needs to repeatedly access the cache or memory in order to check for the latest status, while the rest of the processor has no useful work to do but idling. This may lead to significant waste of energy [64].

In [122], the authors propose a light-weight distributed synchronization method in point-to-point communication applications. The approach encodes the global data dependencies between two processors directly in their memory accesses. In this way, PEs dynamically check load/store instructions and infer dependencies set at compile time. Control and communication with global storage architectures are improved, enabling finer grained parallelism and synchronization for *applications with strong dependencies*. In [9] and [98], the authors propose the Lock Cache organization. The lock cache mechanism implements synchronization in a dedicated and centralized hardware controller. Synchronization sections are distinguished into long Critical Sections (CS) and Short CSs, which are handled by the Lock Cache Controller. With task preemption support from the RTOS, the Lock Cache achieves good performance for *database like applications*. This approach, however, implements the Lock Cache Controller in a centralized fashion, which may lead to potential scalability concerns with increasing number of processors in the system. In [82], a compiler-only solution for multitasking synchronization has been presented. The efficiency of a serializing compiler is analyzed in terms of memory usage and performance.

A light-weight barrier-based parallelization support for non-cache-coherent MPSoC platforms has been proposed in [74]. A cost-efficient barrier implementation for the specific targeted architecture is outlined. In [64], the authors propose the *thrifty* barrier mechanisms addressing the power problem in general-purpose

multiprocessor systems. By carefully predicting and monitoring barrier stall times, processors are placed in low-power modes and speculatively resumed when the barrier release is predicted. The authors lay out solid analysis of the various possibilities with existing power-saving mode in existing CPUs and the trade-offs between cost of entering/existing a power-saving mode and the potential benefit from that particular mode. The results are promising in terms of reduced energy consumption. However, that approach only deals with power-saving with barriers. We think other synchronization primitives such as spin-locks are also commonly used in parallel programs and should be given equal importance in addressing power concerns.

In [135], the Synchronization State Buffer (SSB) is proposed for fine-grain synchronization on many-core architectures. The hardware buffers are introduced to the memory controller of each memory bank. Based on the observation that only a small number of data units participate in synchronization activities, the SSB mechanism achieves efficient fine-grain synchronization on many-core platforms through a combination of hardware and software mechanisms. A fast barrier synchronization implementation for exploiting fine-grained data parallelism with chip multiprocessor platforms is proposed In [100]. A barrier filter mechanism is introduced, which includes hardware tables residing at cache banks as well as operating system support. It observes cache invalidation requests from the bus and performs barriers by filling specific barrier cache lines. In [114], techniques are proposed that aim at improving performance of multiprocessor's synchronization mechanisms, especially for simultaneous multithreaded (SMT) machines. The lock box mechanism is introduced which optimizes synchronization between threads on the same pro-

cessor. In [36], the authors propose a set of architectural primitives for process synchronization in large scale multiprocessors. Queuing method is used with these primitives to reduce number of operations on the interconnect. A software solution for the contention and scalability problem of multiprocessor system synchronization is proposed in [77]. Synchronization algorithms are presented that keep processors spinning on locally accessed flag variables while minimizing remote communications. Generic techniques to accelerate software synchronization primitives are proposed in [84]. The approach uses atomic memory Read-Modify-Write (RMW) instructions. Hybrid primitives are proposed which exploit the uncached RMW instructions to reduce the latency of the arbitration phase. In [28], the authors identify and quantify the performance deficiencies of conventional barrier implementations on large shared memory machines, based on which they propose a queue based barrier implementation which aims at reducing the round trip network latencies. In [43], the hot-spot accesses to the memory modules caused by synchronization are targetted so as to reduce their latencies. A single-stage shuffle-exchange combining network is proposed as a compromise between multistage combining networks and dedicated synchronization buses, so as to trade between performance and cost. In [51], the communication and synchronization challenges with the MIT Multi-ALU Processor (MAP) chip are described. The architecture aims at achieving good fine-grain thread-level parallelism. Thread synchronization is implemented by blocking on specific registers or by executing a special fast barrier instruction.

All these projects address various aspects of synchronization in multiprocessor systems. Nonetheless, a few of them focus on embedded multiprocessors. Such em-

bedded platforms are increasingly becoming the implementation of choice for many modern embedded applications and devices with strict power, data throughput, and real-time performance requirements.

The synchronization architecture proposed in this work is completely distributed and achieves very fast lock acquisition, eliminates bus contention traffic, and enables very fine-grained and flexible power management. It is suitable for multithreaded applications, as well as programs exploiting fine-grained parallelism, which normally incur a high performance and bus bandwidth overhead due to synchronization.

The remaining of the paper is as follows. Section 5.3 analyzes the advantages and disadvantages of conventional synchronization implementations and introduces the fundamental idea for a distributed synchronization implementation. In Section 5.4 we detail our technique, which spans across hardware, compiler and operating system layers. We outline the implementations for locks and barriers using the proposed approach. In Section 5.6 we present our experimental study.

5.3 Functional Overview

Synchronization is one of the major challenges in parallel systems. The implementation of synchronization primitives has a direct and significant impact on the system performance and power, and can thus influence the way many applications are actually parallelized. In this work we introduce a novel distributed synchronization organization for fast and power-efficient synchronization primitives, especially

suited for shared-memory, symmetric multiprocessor architectures.

Conventional general-purpose synchronization implementations rely on atomic operations to access and modify certain memory locations. Such a support enables the processors to exclusively access a synchronization variable and setting it up as "acquired". In many implementations, such as for the most common *lock*, the processors compete for such an access and whoever succeeds is "granted" the synchronization variable, while the other processors continue their attempts. Many other higher level primitives, such as barriers, semaphores, and monitors are built in a similar way or by using locks. While such synchronization implementations are general-purpose and impose small hardware and ISA requirements (some assume coherent caches), they can be extremely inefficient in terms of both performance and power consumption.

Since the processors have no knowledge as of the global status of the synchronization variables, they all compete for the access to the shared synchronization variable by overwhelming the system interconnect with transactions. Three major problems and overheads ensue when the traditional synchronization primitives, such as locks or barriers, are employed in shared-memory, symmetric multiprocessor organizations. *First*, even in the presence of coherent caches, at the moment a synchronization variable is released, all the processors waiting for it enter another competing cycle, which results in a burst of bus traffic. Such bursts of bus bandwidth utilization are due to the cache coherence traffic. *Second*, when a processor attempts to acquire a synchronization variable when it is not available, it needs to continuously poll for it and thus generate significant bus traffic, or in the case

of coherent caches, to continuously read it from the local cache until its remote invalidation. This can be extremely energy inefficient, due to the large number of bus transactions and the polling to the cache and memory structures. *Third*, the latency of acquiring an available synchronization variable is quite high, since it usually entails at least two bus transactions. Since this is the most common situation when performing synchronization operations, it can contribute to a significant performance overhead especially for applications that exploit fine-grain parallelism with frequent synchronization points.

A centralized solution [78] can alleviate the polling energy and the bus traffic caused by competing accesses, especially for distributed memory organizations and network-on-chip designs. However, as mentioned in the previous section, it does not solve the best-case (and also common case) acquisition latency problem since lock acquisitions and releases will always have to be controlled by the remote centralized controller. For the most common case of obtaining an available synchronization variable, the processor must send a request transaction and subsequently receive a grant from the remote synchronization controller. Furthermore, such a controller will have to compete for the memory bus or use dedicated communication lines to the processors, which could result in a significant hardware area cost and chip routing overhead.

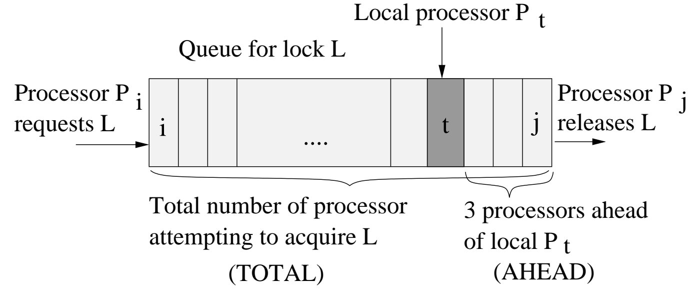

Figure 5.1: Distributed lock queue information

5.3.1 Distributed Queue Abstraction Model

The proposed distributed synchronization architecture addresses the performance, latency, and power problems of the traditional synchronization implementations. Each processor is assigned a local light-weight controller that observes the sequences of remote acquisition attempts and synchronization releases and participates in a very low-cost and efficient distributed protocol for lock acquisition and release. By monitoring the system bus, each local controller is able to construct a state (per synchronization variable) representing how many remote processors are waiting for the synchronization variable and have requested it *before the local processor*, as well as the *total number of processors* currently waiting for the variable. A remote release observed on the bus results in decrementing the number of processors waiting for the variable before the local processor. When this number reaches zero, the local processor ***immediately*** acquires the lock. After the local processor exits its synchronization section, it informs the local controller to release the lock, which results in a bus transaction informing the remote processors that the lock has been released. In this way, the next processor in the global queue with a local state

indicating that there is no processors in-front of it can, thus, immediately acquire the lock.

Fundamentally, a distributed and completely decentralized implementation of a *queuing* mechanism is implemented as each local controller maintains the *minimal amount of information needed* to represent the relevant queue of remote processors waiting to acquire the synchronization variable and the position of the local processor within it. Figure 5.1 illustrates the data structure that is captured by each local controller, and the minimal information needed to capture the structure with respect to the local processor. Each processor needs to maintain the information regarding its position within the queue; it also needs to be able to update this information as remote processors are requesting and releasing the lock. The relevant information about this can be captured by two variables represented through up-down counters, shown in Figure 5.1. The TOTAL register captures the total number of processors in the queue i.e that have requested the lock, but are still waiting for their turn. The AHEAD register captures the number of processors that are ahead in the lock queue with regards to the local processor. Clearly, when AHEAD becomes zero (down from a non-zero) it is the local processor's turn to acquire the lock. The same mechanism is implemented at all the processors in the system. In this way, no single processor captures the entire queue of waiting processors. However, from the pieces of information captured at each processor the entire queue can be easily constructed if needed. This distributed queue abstraction enables the utilization of distributed hardware controllers at very low cost with highly efficient synchronization performance.

TOTAL register:

```
(1)     if (RemoteAcquire)
(2)        TOTAL=TOTAL + 1;
(3)     if (RemoteRelease)
(4)        TOTAL = TOTAL –1;
```

AHEAD register:

```
(1)     if (LocalAcquire) {
(2)        send Acquire(Lock) on the bus;
(3)        if(TOTAL == 0)
(4)           Grant local access;
(5)        else
(6)           AHEAD = TOTAL; }
(7)     if (RemoteRelease) {
(8)        if(AHEAD != 0)
(9)           AHEAD = AHEAD – 1;
(10)           if (AHEAD == 0) Grant local access; }
```

Figure 5.2: Local lock queue management

Maintaining the TOTAL and AHEAD registers can be achieved by only monitoring the bus for remote acquire attempts and releases to that particular lock. Clearly, acquires and releases generated by the local processor would also have to be taken into account in this process; they also need to be placed on the bus so that the remote processors can accordingly adjust their local state regarding this lock. Figure 5.2 illustares the functionality required to maintain the TOTAL and the AHEAD registers for each lock, as well as the detection mechanisms for when it is the local processor's turn in acquiring the lock. It is evident that this functionality can be achieved through a rather simple finite-state machine controller and the pair of registers per synchronization variable.

5.3.2 Synchronization Efficiency with Distributed Queues

The proposed protocol has the distinct advantage of *near zero-latency lock acquisition*. When the lock is globally available, the processor does not have to

wait for a synchronization variable to be atomically brought back from memory or to be acquired from some remote centralized controller; the only latency incurred would be the latency for the local controller to acquire the system bus in order to send a lock acquire announcement - this operation is represented by Step 2 in the functional description for the AHEAD register. In the case of a lock just being released by the last processor that has requested it before the local processor, the synchronization variable is acquired at the moment the local controller observes the release operation on the common bus. Such lock acquisition is in effect instantaneous as it can be triggered in the same clock cycle during which the remote release was observed. Furthermore, since there is no contention through atomic operations for the synchronization variable, it takes *only two bus transactions* per task/processor in its operation to acquire and release the synchronization variable *regardless of the timings of parallel requests*. This is the direct benefit from the queuing mechanism. Acquire requests may occur concurrently with remote processors. However, the system bus will serialize the requests and all the local controllers will update their TOTAL and AHEAD counters accordingly.

Since the local synchronization controllers have the necessary information regarding the global status of the synchronization variable, they can provide for *fine-grained power management policies* on the local processor. This is also due to the very low cost implementation of each hardware synchronization controller which handle synchronization operations while the rest of the functional units are idle. If, for example, the local task needs to wait for the lock to be acquired (as in the case of spin-locks), the controller can gate either the entire pipeline or the most power

consuming components, such as the access to caches. In this way, the processor can be switched into a low-power mode very efficiently, and resumed at the exact moment when the synchronization variable is to be acquired. In the cases of non-trivial wake-up logic, the procedure can be initiated in advance as the local controller has a complete information as of the number of tasks/processors, which are in front of the local task in the lock queue.

The local synchronization controllers can also work cooperatively with the OS. The OS allocates the synchronization variable information in the controller's internal structures. Furthermore, the OS can utilize different power-saving policies and resume policies based on (and controlled by) the particular application requirements in order to make the best trade-off between performance and power. On the other hand, due to the fact that many power saving techniques, such as clock-gating and power-gating, are often strongly dependant on specific hardware implementations, the actual parameters may differ greatly between different implementations. There are often considerable penalties associated with entering and exiting the different levels of power saving modes, which necessitate flexibility and tuning to the actual implementations. In view of these, it will be beneficial to provide interface to the operating system and to the system designers who can then implement a custom power-saving policy on the particular multiprocessor platform.

5.4 System Architecture

The proposed distributed synchronization architecture requires the hardware support in the form of the local synchronization controllers. We refer to this hardware block as a *Distributed Synchronization Controller (DSC)*; an identical instance of it (of course, with different run-time state) is assigned locally to each processor node in the system and operates independently from the other controllers by reacting to the synchronization requests/releases placed on the bus by the remote processors.

There exist several synchronization primitives that have been used in the area of parallel programing and systems. In this work, we present how our distributed organization implements ***locks*** and ***barriers***. We demonstrate the DSC implementation similarities of these two primitives which reduces the DSC complexity. Most of the other synchronization primitives can be derived from locks and barriers; thus, they can be either implemented in the DSC in a similar manner, or emulated by software in the synchronization library.

5.4.1 Synchronization Variable Identification

Since each lock/barrier is assigned an entry in the DSC, a mechanism is needed to identify the locks/barriers and their DSC entries. Conventional synchronization implementations utilize atomic operations to access certain memory locations and set them to special values to denote ownships of associated synchronization variables. In the proposed approach, a synchronization variable is still assigned a memory

location within a known page/segment of the memory address space. Since our approach does not require that any particular value is written to or read from that memory location, its address is used for the sole purpose of broadcasting the lock acquire and releases on the bus by means of normal read and write memory transactions to that address. A group of the least significant bits of this memory address is used to uniquely identify the corresponding synchronization variable. It is also used to identify the specific DSC entry. We refer to this value as a *LockID* or a *BarrierID*. The number of bits that is actually used to represent a *LockID* is determined by the maximum number of synchronization variables that will be used in the system. In our experimental benchmarks, including Splash-2, Mediabench, and a number of signal processing applications, this number never exceeds 10, and thus we have adopted 4-bit IDs.

It is noteworthy that the DSCs need not simultaneously capture all the locks and barriers used by the program. Often times, the set of locks/barriers used by the set of parallel threads comprising the program are actively used only in small sets as the program executes different phases. At any moment, the DSCs only need capture the set of locks/barriers that are currently used by some threads. That is, if at least one thread attempts to acquire a lock, an entry for this lock must be allocated and activated at *all* the DSCs associated to processors that execute a thread using this lock. This will ensure that the queue state of that lock is properly maintained and when the local threads attempt to use that lock, the proper queue state will be captured by the DSC and the synchronization operation carried out successfully. If for some application the DSCs entries are exhausted then traditional

lock implementation can be used instead for the remaining locks/barriers that cannot fit in the DSC.

A lock acquire operation can be modeled as a normal read from the address of that lock, while a lock release is modeled as a write operation to that location. The particular values written or read are of no importance. The synchronization variable memory segment or page number, which corresponds to a group of the most significant address bits of the variables, is used by the DSCs to determine whether a read/write request to this location is an acquire/release operation rather than actual memory accesses. A write or a read command to that memory location, coupled with the known memory page or segment where locks are allocated is sufficient to represent an "acquire/release lock" command or "enter/exit barrier" command, which will be identified and reacted to by the DSCs. In a subsequent subsection of this work we discuss an alternative approach for lock acquisition/release operations that uses instead two dedicated lock acquire/release operations. DSCs are activated only on those commands, which eliminates the need to snoop on the bus for multiple address bits. Clearly, this approach to lock/barrier identification does not impose any extra requirements on the bus organization, as only traditionally supported read/write operations are used. If the system bus supports additional control operations, special Acquire and Release transactions can be used, with a parameter corresponding to the lock-ID. These transactions can be triggered by special *lock_acquire* and *lock_release* instructions. In this way, the local DSCs would monitor the bus for such transactions only.

5.4.2 Distributed Synchronization Controller

The Distributed Synchronization Controllers (DSC) are small controllers associated to each processor core that manage the synchronization variables used by the tasks on that processor. They receive synchronization requests from the local processors such as "acquire a lock", "release a lock", "enter a barrier" and also monitor the system bus for relevant synchronization activities from remote processors, so as to gather enough information for local synchronization operations.

The major part of every DSC is a table of entries for synchronization variables which are mapped to memory locations. Apart from the synchronization variable IDs, each entry contains the two registers, AHEAD and TOTAL, which capture the queue status with respect to the local processor. The number of bits required for these registers is determined by the total number of processor cores in the system. A 4 processor system, for instance, requires only 2 bits for each register, which sums to 4 bits for each entry. Likewise, a 16 processor system requires only 4 bits per register. These arrangements have made the additional hardware support trivial in terms of both area and power as well as scalability.

The *lock* or *barrier ID*, corresponding to a group of least significant bits from the address is used to lookup the DSC for that entry. One approach would be to use a CAM-based parallel lookup. Since the DSC size is very small (16 entries in our study) and the *LockID* used as a key is relatively short (4-6 bits wide - corresponding to the total number of lock/barrier that need to be supported through the DSC), such a parallel lookup will be very fast and consume a trivial amount of power. An

alternative organization will be to use the LockID/BarrierID directly as an index into the DSC entry. This approach will be very efficient if all the processors in the system work with the same set of synchronization variables. This situation occurs when the application expoits loop parallelism with worker threads mapped to all the processors. However, if the processors use different synchronization variables, this may result in underutilization of the DSC entries. This can occur since the locks/barriers have unique identifiers mapping them to DSC entries. If a processor does not use a particular lock, its DSC entry will remain unused. In this case, the synchronization variable IDs are not physically stored in the hardware. Yet, another implementation approach that still uses the variable IDs as an index into the DSC while avoiding the DSC underutilization problem is to employ an additional mapping register/table, which will be indexed through the *LockID* and will provide the actual DSC index, if that lock/barrier is relevant to the local processor. However, since the mapping can be managed by the operating system, DSCs on different processors could have different mappings, which gives the system designer more flexibility to make more efficient use of hardware DSC entries, when subsets of processors synchronize on different synchronization variables.

Figure 5.3 illustrates the DSC internal organization. It consists of a number of entries that correspond to synchronization variables, such as locks and barriers. For each entry, the controller registers the synchronization variable's ID. This ID can be set when the thread/task is loaded onto this processor by the operating system or the program loader. Alternatively, the DSC can be setup just prior to entering a program phase that uses a particular set of synchronization variables. Here we have

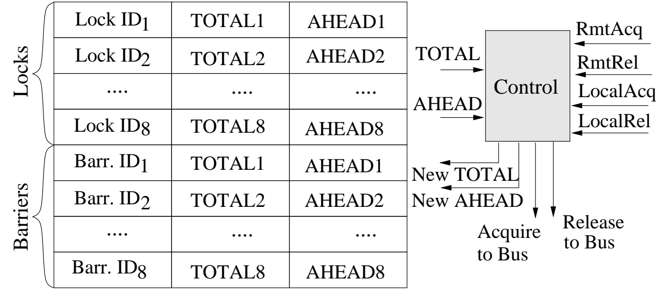

Figure 5.3: Distributed Synchronization Controller (DSC) organization

assumed that the Lock/Barrier ID is captured in the DSC and a parallel lookup is performed. As discussed above, an alternative implementation is also possible where the ID is mapped to a DSC index through a small set of registers. In this case, even the ID field is eliminated from the table.

Two counters/registers are associated with each synchronization variable. The TOTAL and AHEAD counters for each lock/barrier used by the local task are allocated an entry in the DSC. They together denote the position of local processor in a lock queue. These two registers record synchronization activities both from remote processors and from the local processor that access the corresponding synchronization variable. The algorithm of how they function is described in the following sub-sections.

On a bus transaction representing a remote lock acquire or release, or on a local acquire or release, a simple control logic is used to update the registers. The DSC controller implements the functionality described in Figure 5.2 for lock implementation. This controller contains two comparators and an increment/decrement module for the TOTAL register, and a decrement unit for the AHEAD register.

The DSC controllers are managed by the operating system. When a program is loaded onto a processor, the OS sets the ID field of each synchronization variable that is used in the specific thread/task. Initial value of TOTAL and AHEAD registers are set to zero, which means no other processor is waiting for the lock and no one is waiting ahead in the queue. As for barriers, the OS sets the TOTAL counter value according to program information and sets AHEAD equal to TOTAL. Although the DSC controller can implement and control simple power saving techniques such as blocking accesses to certain structures like caches, or power-gating certain functional units, the actual implementation can delegate some of these capabilities to the operating system. In this case the system developers are given more flexibility to fine-tune the implementation to the underlying system platform. Also, since many power saving techniques, such as clock-gating and power-gating, exhibit considerable penalties associated with their applications, which are dependant on several physical parameters specific to the underlying hardware, it may be more prudent to let the developers to tune the DSCs and the corresponding power saving policies to these parameters so as to maximize the power savings.

5.4.3 Lock implementation

An algorithmic description for lock implementation is shown in Figure 5.2. For lock implementation, the TOTAL register captures the total number of remote processors that are waiting to acquire the particular lock. TOTAL is incremented when a remote processor sends an “acquire(lock)” command on the bus - for a

typical bus this could be a read command to the location for that lock. Similarly, it decrements when a remote processor broadcasts a release for the lock in the form of a write to the lock address. In this way, when the local processor issues "acquire(lock)" command, it knows immediately how many processors are waiting in the queue, without the need to request information from other processors or from the shared memory. The TOTAL register is initialized to zero when the lock is created by the local thread and allocated into the DSC. When the local processor issues an acquire to that lock, TOTAL is copied into the AHEAD register, since at that moment all that have requested are in front of the local processor in that logic queue. The AHEAD register indicates how many other processors are still waiting before the local processor can acquire a lock. It decreases monotonically on observing remote processors releasing the lock. When the AHEAD counter reaches zero, the DSC determines that all remote processors that were before the local one in queue have released it and it is safe for the local processor to immediately be granted the lock, without any bus transactions. As is evident from this description, the TOTAL and AHEAD registers for a particular lock throughout the system represent the queue for that lock in a consistent way. Each processor keeps track of its place in the queue locally without global communication with the other processors. This leaves out the necessity for a centralized controller that manages the entire system, thus making the design completely distributed and easy to implement in terms of global chip routing and performance/power overheads. Additionally, atomic memory operations and synchronization competition, which constitute the fundamental reason for the bus traffic contention problem, are no longer needed to construct locks and other

TOTAL register:

```
(1)    TOTAL = constant_num_threads_hit_bar;
(2)    (Set by OS during program load)
```

AHEAD register:

```
(1)    if (LocalHitBar) {
(2)        send HitBar(Bar) on the bus;
(3)        AHEAD = AHEAD – 1;
(4)        if(AHEAD == 0) {
(5)            Grant local access;
(6)            AHEAD = TOTAL; }
(7)    if (RemoteHitBar(Bar)) {
(8)            AHEAD = AHEAD – 1;
(9)            if (AHEAD == 0) {
(10)               Grant local access;
(11)               AHEAD = TOTAL; }
```

Figure 5.4: Local barrier queue management

primitives that are based on locks, as the bus serialization property is sufficient for the DSCs operation.

5.4.4 Barrier implementation

While there exist barrier implementations that are based on locks, which in turn are based on atomic operations of the underlying platform, they are often very inefficient in terms of performance and power. At the same time, barriers are frequently used to orchestrate parallel threads during different stages of program execution and are also frequently used in data parallel loops. Consequently, efficient barrier implementation will enable the fine-grain parallelization of many applications. Here we describe a barrier implementation using the same DSC entries, the cost of which is as low as the locks described in the previous section. The algorithm description for barrier implementation is shown in Figure 5.4. The distributed implementation scheme for barriers follows the same concept as locks. This time the

TOTAL register captures the constant that corresponds to the number of threads that must reach the barrier before it is released, which is specified in the program code. In this scheme, the value in TOTAL does not change during program execution.

When the barrier entry is loaded into the DSC, the AHEAD register is initialized with the constant held in TOTAL representing the number of threads required to reach the barrier - this constant is typically provided when the barrier variable is instantiated and initialized. Subsequently, when the DSC observes on the bus that another processor has reached the barrier, it decrements the AHEAD register. Similarly, when the local processor reaches the barrier, the AHEAD counter is decremented and a bus transaction is initiated by the DSC to notify the remote processors that the barrier has been reached locally. This bus transaction can be modelled, for instance, as a read transaction from the memory location of the barrier (of course, no value is expected from that memory location). Similar to locks, if the bus can support extra commands, a dedicated synchronization transaction type can be defined for reaching a barrier, which will carry the Barrier-ID needed for the remote DSCs to update their states. When the AHEAD register reaches zero, it means that all the threads have reached the barrier (and for all of them the AHEAD register will decrement to zero) and all the DSCs signal their local processor to proceed execution and leave the barrier. At that moment the AHEAD register is loaded back the constant from the TOTAL and in this way the distributed procedure is re-initialized for that barrier.

It is noteworthy that the bus transaction overhead for implementing barriers

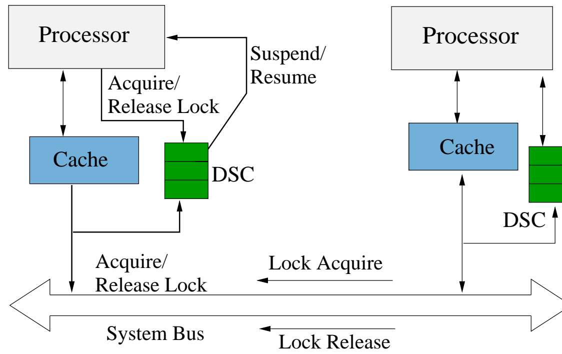

Figure 5.5: Overall system organization

through the DSC is drastically reduced *to a single bus transaction per thread*. Each thread/processor notifies the rest of processors that is has reached the barrier by placing the "reach-barrier" bus transaction. When the last thread reaches the barrier the AHEAD counter is decremented to zero, the "reach-barrier" transcation is placed, and the local processor is signaled to continue execution, i.e. the barrier instruction returns control immediately. When that bus transaction is observed by the remote processors, their AHEAD counters similarly are decremented to zero and the execution control is released at these processors as well.

It is evident from the above description that the implementation of barriers is very similar to that of locks, based on the same DSC structures. This helps greatly reduce the complexity of the DSC controllers and lower the cost of DSCs that support multiple types of synchronization primitives. Most other synchronization primitives can be constructed in a similar way.

5.4.5 Power Management

By having the DSCs handle most synchronization operations, it becomes possible to place the local processor in various power down modes, including a complete shut-down, while the local thread is blocked in a synchronization operation. The DSC continues its operation while the processor is suspended and resumes the processor execution when the synchronization conditions are met. As the local DSC controllers are extremely small and simple, the achieved power savings would be significant. We quantify this in our experiments.

ISA-transparent approach. Various power-saving schemes can be explored. One approach would be to disable the accesses to the data cache for a spin-lock software implementation, where the proposed distributed synchronization architecture is implemented in an ISA-transparent way. In this case, the software implementation of the lock consists of a small loop, which iteratively attempts to read and set the lock in an atomic way. The DSC will intercept the execution of the lock read operation by matching the address with the known segment/page for synchronization variables and will then proceed with its function. Note that such an identification can be achieved very efficiently, for instance by using a status bit in the TLB or in the cache line (if virtual memory is not supported) to indicate that the memory reference is to a synchronization variable. In this way, load instructions will not trigger address comparison each time they are executed. Subsequently, the lock availability and its status are determined by the DSC based on the lock entry in the DSC lock table. When the lock is not yet available the DSC will enforce that

the lock load operation returns a result indicating that the atomicity of the operation has failed. The software implementation will proceed with the next attempt of acquiring the lock. From this moment on, the DSC with the cooperation of the cache controller will simply return a lock unavailable status to that lock instruction. For quick identification of this load instruction, the DSC at this moment can place its program counter into a latch and until the lock is made available by the remote processor, return status unavailable to this lock load instruction. The spin-access to the local data cache and/or remote memory through the bus will be blocked and thus no power will be spent in such energy inefficient operations. While the task executes this cache-oblivious spin-lock, the thread library or the operating system may decide to preempt that task and schedule another task for execution. The DSC controller can thus handle multiple tasks all waiting for a synchronization variable. When the DSC detects that a lock has become available due to a remote release, it will inform the system software (if needed to enable the resumption of the task) and on the next execution the spin-load will return a lock available status to the application.

Explicit ISA support. Alternatively, the processor ISA can be augmented to support dedicated *instructions for lock acquisition and release.* The two new instructions are very simple and they would have two register operands. The first register will contain the *LockID* of the lock being manipulated while the second register will indicate the status of the lock acquisition operation (the lock release instruction will not require such a result register). The lock acquisition instruction can be implemented with a blocking and non-blocking versions. The blocking version

would simply stall the pipeline until it is signalled by the DSC to continue execution when the lock has become available to the local processor. Clearly, while the pipeline is stalled the processor will be in a low-power mode as no execution activities are present. The blocking version will be usefull for systems with no system software support for power management or no operating system at all. The non-blocking version will enable the system developer with the help of OS or the thread library to develop user-controlled power management policies. It will also enable the system software to perform context switch when a lock is not available and thus schedule for execution another task while the preempted task waits on the synchronization. Similar instructions can be introduced for handling barriers. Note that with such an explicit ISA support, no memory locations need be allocated for actual lock/barrier variables. A trivial compiler (or OS) support may be needed to allocate unique identifiers to all the locks and barriers in the system. These special instructions will also result in special bus transactions to announce to all the processors in the system whether a lock is acquired or released. This does not require any special hardware support on the bus, but only a few new control values to be used as indicators for these types of bus transactions. The system-level power management policies are described in Section5.5.

5.5 Compiler and OS Support

The role of the compiler/software developer is limited to the instantiation of the locks/barriers and the allocation of unique addresses for each such synchroniza-

```
Parallel Program:

MainTask( ) {
    .....
    init_barrier(NumThreads, BarrID1)
    ....
    init_lock(LockID1)
    init_lock(LockID2)
    start(WorkerThread1)
    start(WorkerThread2)
    ....
}

WorkerThread1( ) {
    .....
    barrier(BarrID1);
}

WorkerThread2( ) {
    acquire_lock(LockID1);
    .....
    release_lock(LockID1);
    .....
    barrier(barrID1);
    .....
}
```

Figure 5.6: Example parallel application

tion variable. As explained in the previous section, this address will be used to form the unique *LockID* or *BarrierID*. In the case of dedicated instructions for the synchronization operation and system bus support for acquire/release commands, the role of the compiler is to generate the globally unique identifier for each lock and barrier. This can be easily achieved with an operating system support. As this is performed during program initialization, no performance overhead will be incurred in practice.

Subsequently, the operating system (or thread library) ensures that the state for each lock/barrier is stored in the local DSC. This is performed when the threads are loaded on the processor nodes. The example code structure presented in Figure 5.6 illustrates this. The main program initializes the lock and barriers and, subsequently, spawns the worker threads that use them to synchronize across the processors. The DSC controllers on different processors can have different sets of synchronization variables allocated to them depending on what threads are operating on and the way synchronization is performed amongst them.

5.5.1 OS Power Management Role

The proposed methodology does not inherently require an operating system and can be readily applied to systems with no OS support. In this case, as explained in Section 5.4.5, the DSCs are used to generate the power management signals. For instance, the pipeline execution may be suspended as in the case of special lock acquisition instruction. Alternatively, the access to the power hungry data cache can be disabled as was described in the previous section, while the processor iteratively spins in its attempt to acquire the lock. While the lock is not available, the result of the spin-load instruction will be defined by the DSC (and the cache controller) to indicate the unavailability of the lock. However, in the presence of OS, more sophisticated power management techniques can be employed as then the OS can make informed decisions regarding when and how/whether to suspend the current task or processor, in order to arrive at the best solution for particular applications.

An efficient OS support can be implemented when non-blocking versions of the lock acquire and release instructions are supported, as the OS can have a full control in implementing the power saving policies. When the lock or barriers are released, the DSC will simply generate an interrupt and inform the local OS. There are various trade-offs between power and performance that need to be considered. Many power saving techniques bring along side effects that may harm real-time performance considerably. For example, the techniques of dynamic voltage and frequency scaling, clock-gating and power-gating can significantly reduce dynamic and leakage power. However, when a processor, or some portion of it, is clock-

gated or power-gated completely, it takes significant amount of time to power down the clock trees and other structures and even more time to resume their normal operation. Some power-gating applications may lead to loss of circuit states. All these requires sophisticated parameter tuning by system developers, according to specific physical processor implementations.

Application-specific properties must also be taken into account when deciding which power management policy to follow. For example, some locks are used in very short but frequent critical sections. In this case, the application must inform the system software that the processor must not be placed in deep power saving mode as the latency of resuming execution will negate the power saving benefits. In this case, the threads would exhibit very short stall times and thus it will be more beneficial to just stall the processor pipeline or disable the accesses to the cache structures for the spin-load instructions. This information transfer can be performed at the time when the locks/barriers are initialized (during the program/phase initialization) by the means of control values passed to the OS/thread library. Some locks and barriers, however, may be used for longer critical sections, such as in many parallel programs where load-balancing is difficult to achieve. In this case, certain processors in the system may consistently spend significant amount of time waiting for synchronization at every iteration. Consequently, such situations create good opportunities that large processor structures (or even the entire processor) be turned off during that time to save power. In such cases a thread may stall for a significant amount of time waiting for its turn; consequently, more aggressive power down techniques must be considered. In such power-down policies where the processor is brought down to

a very low-power mode through voltage/frequency scaling, the latency of resuming it may be non-trivial. Since the penalty parameters are specific to the processor microarchitecture and the underlying process technology, it may be best for the programmer and the OS to decide what power-saving policies are to be employed to achieve maximum benefits.

For such cases of long synchronization latencies, the OS could program the DSC to generate a pre-resume signal earlier in time and just-in time to allow the processor to restore its voltage/frequency level. Based on an application-specific information regarding how long it takes for a thread to execute its critical section, a number of different policies can be adopted. Using profile information as of how long a thread blocks on a lock or barrier can be one option. This information is provided to the operating system, which subsequently chooses which power-saving technique to apply on specific synchronization variables. Furthermore, the operating system can also initiate the waking up procedure earlier so as to hide its latency from the actual execution time. In this case the DSC can be programmed to generate a pre-resume signal. When profiling information is not readily available or the latency is dynamic, the operating system can still collect run time statistics and make earlier wake-ups based on that, as proposed in [64]. Such an approach would require additional hardware support for timing measurements. For short critical section applications, the DSC can also be programed to do early wake-ups based on AHEAD register values. For instance, the pre-resume signal can be generated when there is only one processor left in front of the local processor in the lock queue, which is reflected by a value of 1 in the AHEAD register. In our experimental study,

we have assumed the most conservative power-saving approach.

5.5.2 Multi-tasking support per core

The proposed approach can also be supported in more complex multitasking environments where multiple tasks can share each processor. If all the locks and barriers in the system can be allocated within the DSCs then no special attention is needed in this case. If a resume signal is received at a processor for a task that is currently preempted, the OS will register this and later when the task is resumed it will succeed in acquiring the lock. In the case when not all system-wide synchronization variables can be allocated in the DSCs, then the OS when performing a task preemption on a processor, will also have to preempt that task in the other processors (its worker threads) as well. This represents a global task switch, where no parts of the task (as it can run multiple worker threads by itself) are left executing. In this case, multiple independent tasks, each executing multiple threads within it, can be easily supported. During such global task switch, the OS will have to preserve the relevant DSC state for all the processors where threads from that task are allocated.

5.6 Experimental results

We have conducted detailed simulations on a set of multitasked parallel applications. We have chosen the kernel programs from the SPLASH-2 [104], and the parallelized MPEG encoding/decoding applications from ALPBench [65]. The

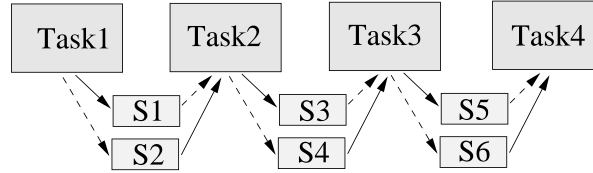

Figure 5.7: Data-streaming benchmarks organization.

parallelization method of these programs mostly follows Single Program Multiple Data (SPMD) fashion, where identical threads are spanwed to operate on a subset of the input data. We have also constructed a set of benchmarks where each benchmark is configured to consist of four parallel threads, each performing one stage of computation in a stream processing pipeline. The threads communicate through butterfly buffers and synchronize using standard locks and barriers. The individual tasks constitute of: *FFT*, *ADPCM*, *matrix multiplication*, *data encryption tasks*, *lzo-compression*, *g721*, image processing - the *blur* and the *edge-detection*, and *video processing*. The tasks cover benchmarks from the MediaBench [62] and MiBench [40] suits, as well as from other open-source image and video processing tools. This set of programs represents the software pipeline parallelization method, which is another major type of parallelization for embedded systems. The production and consumption cycles are interleaved so that at each moment of time the processor is executing in either a producer or in a consumer critical section. The benchmarks organization is depicted in Figure 5.7. The multitasking applications are combinations of individual tasks. The ones we have used are: *A1*={*LU, MMUL, AES, LZO*}; *A2*={*FFT, G721, blowfish, SHA*}; *A3*={*blur, edge-detection, AES, LZO*}; *A4*={*FFT, FDCT, IFFT, AES*}, which represent multi-tasked embedded applications in digital filtering, audio, image, video processing, and security arenas.

	P1-l			P2-l		
	Baseline	DSC	Reduct.	Baseline	DSC	Reduct.
4p	529,168	401,172	24.19%	589,168	461,172	21.72%
5p	669,166	525,166	21.52%	773,166	629,166	18.62%
6p	793,168	601,172	24.21%	953,168	761,172	20.14%
7p	933,166	725,166	22.29%	1,161,166	953,166	17.91%
8p	1,057,168	801,172	24.22%	1,365,168	1,109,172	18.75%

Table 5.1: Performance characteristics (in number of cycles) and DSC reductions - Increasing data set

	P3-b			P4-b		
	Baseline	DSC	Reduct.	Baseline	DSC	Reduct.
4p	525,109	113,128	78.46%	525,111	145,127	72.36%
5p	625,113	113,137	81.90%	625,120	157,136	74.86%
6p	725,106	113,146	84.40%	725,119	169,145	76.67%
7p	825,106	113,155	86.29%	793,121	181,154	77.16%
8p	925,099	113,164	87.77%	893,126	193,163	78.37%

Table 5.2: Performance characteristics (in number of cycles) and DSC reductions - Increasing data set (continued)

	P1-l			P2-l		
	Baseline	DSC	Reduct.	Baseline	DSC	Reduct.
4p	64,526	48,505	24.83%	64,543	48,505	24.85%
5p	64,543	52,527	18.62%	64,563	52,518	18.66%
6p	96,554	72,525	24.89%	96,575	72,543	24.88%
7p	96,571	76,540	20.74%	96,569	75,768	21.54%
8p	128,582	96,545	24.92%	128,580	96,560	24.90%

Table 5.3: Bus bandwidth characteristics (in number of bus transactions) and DSC reductions - Increasing data set

	P3-b			P4-b		
	Baseline	DSC	Reduct.	Baseline	DSC	Reduct.
4p	128,529	64,516	49.80%	132,526	64,522	51.31%
5p	156,541	80,518	48.56%	164,545	80,523	51.06%
6p	184,561	96,529	47.70%	196,556	96,530	50.89%
7p	212,575	112,535	47.06%	228,563	112,541	50.76%
8p	240,587	128,541	46.57%	260,583	128,550	50.67%

Table 5.4: Bus bandwidth characteristics (in number of bus transactions) and DSC reductions - Increasing data set (continued)

As explained before, the proposed approach reduces greatly synchronization latency and has the potential to significantly reduce performance overheads associated with synchronization. However, most of the benchmarks outlined above are parallelized at task level and communication/synchronization operations are not as frequent so as to observe significant performance improvements for the entire program. With the emergence and prevalence of chip multiprocessors, applications exploiting fine-grain parallelisation have become feasible. In view of this, we have created four such kernel programs, P1-l, P2-l, P3-b, P4-b, that perform frequent data accesses to shared data. These programs utilize frequent synchronization operations to access shared data and are able to more adequately evaluate the performance advantage of the proposed synchronization schemes. In P1-l and P2-l, a lock is used for mutual exclusion by a set of tasks which iteratively access and modify a shared state. The benchmarks consist of identical threads, each mapped to a different processor. In P1-l, the short critical section is accessed iteratively by all tasks with identical delay (just the loop overhead), which results in both high lock utilization frequency and high amount of contention for the lock. On the other hand, P2-l introduces a randon short delay (up to 10 cycles) before the next attempt to enter the critical section. In this case, the lock is still utilized with great frequency, while the amount of contention between the threads will be reduced. Benchmarks P3-b and P4-b have similar structure but are using barriers instead at the end of each loop iteration to synchronize their execution progress. Both P3-b and P4-b consist of identical threads which iteratively perform a load, an increment, and a store instructions. All the loop iterations execute in parallel by performing this update to

a memory location and then synchronize on a barrier before executing the next iteration. In this way, all the loop iterations across the tasks are executed in parallel, while for P1 and P2 the critical sections across the threads are naturally serialized as only one task is allowed to enter the critical section. Similarly to P2-l, P4-b introduces a slight variation (within several cycles) in the execution cycles for the iteration body. In this way, the threads do not arrive at the barrier at exactly the same time and much less bus contention is introduced as compared to P3-b. These synchronization-heavy benchmarks are used to evaluate the performance and bus bandwidth impact of the proposed technique. By measuring the total bus transactions and execution cycles, we demonstrate the performance improvements and bus bandwidth reductions of the proposed distributed synchronization architecture.

We have used the M5 [18] simulator to perform our experiments, extended with a collection of thread synchronization primitives. The simulated hardware configuration is of 4 to 8 processors connected to a shared memory through a common system bus. Our simulated architecture includes 16K, 4-way set-associative data caches. The baseline architecture assumes traditional lock and barrier implementations based on load-linked/store-conditional paired instructions with coherent caches support.

Tables 5.1 and 5.3 report the performance characteristics and the bus bandwith utilization (as a total number of bus transactions) achieved by the proposed distributed synchronization organization (DSC) when compared to the baseline for the four fine-grain parallel benchmarks. In this first study, all the threads in the four benchmarks execute 4000 iterations regardless of the number of threads instantiated

	P1-l			P2-l		
	Baseline	DSC	Reduct.	Baseline	DSC	Reduct.
4p	529,168	401,172	24.19%	589,168	461,172	21.72%
5p	535,566	420,366	21.51%	618,766	503,566	18.62%
6p	529,036	401,072	24.19%	635,676	507,712	20.13%
7p	533,571	414,751	22.27%	663,816	544,996	17.90%
8p	529,168	401,172	24.19%	683,168	555,172	18.74%

Table 5.5: Performance characteristics (in number of cycles) and DSC reductions - Fixed computational workload

for the particular run. Our study comprises configurations from 4 processors up to 8 processors. This setup corresponds to a situation where the size of the input data set increases linearly with the number of processors available in the system in order to achieve higher troughput. From the way our benchmarks are constructed, it is to be expected that for P1-l and P2-l, the total time needed to perform the computation must increase linearly with the number of processors since each thread performs a computation within a critical section accessed through a single lock. Consequently, the critical sections executed by all the threads are serialized and increasing the number of threads (each executing a fixed number of iterations) will increase the total number of critical sections to be executed. On the other hand, P3-b and P4-b are expected to maintain roughly the same execution time regardless of the num-

	P3-b			P4-b		
	Baseline	DSC	Reduct.	Baseline	DSC	Reduct.
4p	525,109	113,128	78.46%	525,111	145,127	72.36%
5p	500,313	90,737	81.86%	500,320	125,936	74.83%
6p	483,652	75,794	84.33%	483,665	113,117	76.61%
7p	471,816	65,135	86.19%	453,551	103,979	77.07%
8p	463,099	57,164	87.66%	447,126	97,163	78.27%

Table 5.6: Performance characteristics (in number of cycles) and DSC reductions - Fixed computational workload (continued)

ber of processors, since all the loop iterations across the threads execute in parallel and only synchronize with barriers at the end of each iteration. The increase in performance will be only due to the synchronization overhead.

In Table 5.1 the performance is measured as a total number of cycles to execute each benchmark. When comparing the performance achieved by the proposed approach to the baseline for each configuration, the DSC achieves around 24% performance improvement for P1-l due to the improved lock acquisition latency and the reduced bus contention overhead achieved by the DSC approach. For P2-l the reductions in performance are slightly smaller since this benchmark exhibits much less synchronization-based bus contention. The performance reductions for the barrier benchmarks P3-b and P4-b are greater and in the range of 78% - 87%. This

	P1-l			P2-l		
	Baseline	DSC	Reduct.	Baseline	DSC	Reduct.
4p	64,526	48,505	24.83%	64,543	48,505	24.85%
5p	51,743	42,124	18.59%	51,756	42,118	18.62%
6p	64,538	48,521	24.82%	64,559	48,536	24.82%
7p	55,411	43,958	20.67%	55,409	43,976	20.63%
8p	64,582	48,546	24.83%	64,580	48,567	24.80%

Table 5.7: Bus bandwidth characteristics (in number of bus transactions) and DSC reductions - Fixed computational workload

higher reductions are due the higher barrier overhead as compared to single locks. Since the actual iteration body is rather short (a memory update operation executed through a load, increment, and a store) and that the iterations are executed in parallel across the threads, the barrier overhead greatly dominates the execution time on the baseline implementation. The DSC approach almost completely eliminates this overhead by reducing it to a single bus transaction per thread/processor. Similarly to the lock benchmarks, the reductions for P4-b are smaller than for P3-b due to the smaller amount of bus contention produced by P4-b.

Since in P3-b the loop iterations across the threads are executed in parallel, increasing the number of processors/threads that execute the same number of iterations must result in identical total run-time for an ideal synchronization imple-

	P3-b			P4-b		
	Baseline	DSC	Reduct.	Baseline	DSC	Reduct.
4p	128,529	64,516	49.80%	132,526	64,522	51.31%
5p	125,341	64,518	48.53%	131,745	64,523	51.02%
6p	123,193	64,517	47.63%	131,190	64,521	50.82%
7p	121,680	64,520	46.98%	128,523	64,521	49.80%
8p	116,589	64,541	44.64%	130,583	64,546	50.57%

Table 5.8: Bus bandwidth characteristics (in number of bus transactions) and DSC reductions - Fixed computational workload (continued)

mentation with no overhead. It can be observed, however, that due to the very high barrier overhead in the baseline, the total number of cycles is significantly increased when more processors are introduced. The more processors result in more cache coherence traffic to handle the contention for the lock used to implement the barriers. On the other hand, since the DSC approach reduces this overhead drastically, it is seen that the total number of cycles for P3-b is only minimally increased when more processors are included.

Bus bandwidth results are reported in Table 5.3. As can be seen from the results, the DSC architecture significantly reduces the total number of bus transactions. For the lock-based benchmarks, the reductions are in the range of 20%-24%, while for the barrier-based benchmarks, the reductions increase to 55%-60%. This

can be explained by the fact that the baseline (and traditional) barrier implementation requires a significant amount of bus transactions, while the proposed DSC approach requires only a single bus transaction per thread for the barrier implementation and two bus transactions per thread for locks.

Tables 5.5 and 5.7 report the performance and bus characteristics for the same set of benchmarks but with a fixed input data set, i.e. the total number of iterations executed by all the threads in the system is constant and does not depend on the number of processors. This scenario corresponds to a situation where more processors (computing power) is used to speed up the computation time for a fixed amount of input data per benchmark. For this experiment we have assumed a total of 16000 iterations to be executed by the benchmarks. That is for the case of 4 processors, each thread executed 4000 iterations. The other border case is the 8 processor configuration for which each processor/thread executes 2000 iterations. This setup demonstrates the scalability of the proposed synchronization architecture, i.e. its increased utility when dealing with larger number of processors as compared to the baseline lock/barrier implementation with coherent caches. These two tables have an identical organization to the first two tables and report performance (in terms of cycles) and bus transactions, respectively.

The results show consistent reductions in terms of performance and bus transactions. This is a very clear and strong indication of the efficiency and the good scalability of the proposed DSC approach. This time, however, due to the constant amount of computation performed by the benchmarks, the performance for lock-based benchmarks is independent from the number of processors/threads, while

the barrier-based benchmarks achieve better performance as the number of processors grow. P3-b for example emphasizes the case where the performance must improve linearly with number of processors assuming no overhead synchronization and inter-core communication. The performance is significantly improved by the proposed DSC approach; in the range of 20%-25% for the lock-base benchmarks, and 72%-87% for the barrier-based. Similarly to the case of increasing data set, the improvements for P2-l and P4-b are slightly smaller due to the less bus contention these two benchmarks exhibit.

For this case of fixed computational workload, it is to be expected that the performance for the barrier-based benchmarks will be improved when more processors are added. For a perfect, no-overhead synchronization and inter-core communication, the total number of cycles must linearly decrease with the number of processors added. However, it can be observed that for the baseline implementation this reduction is far worse than linear. On the other hand, the DSC approach eliminating a large fraction of the barrier overhead, achieves performance improvements much closer to the linear rate.

Table 5.7 reports the total number of bus transactions for the four benchmarks and the reductions achieved by the DSC approach. Significant reductions are achieved across the benchmarks and multi-processor platforms, ranging from 18% to 60%. Much greater reductions are observed for the barrier-based benchmarks as the overhead of barriers is significant for the baseline architectures.

It is evident from Tables 5.1, 5.3, 5.5, and 5.7 that the proposed approach not only significantly reduces bus bandwidth and improves performance but it also

provides for better scalability when more processors are to be used within the system.

	Idle Cycles (I)	Total Cycles (T)	I/T
FFT	78,045	228,722,336	0.03%
LU_con	393,799,327	1,117,336,393	35.24%
LU_noncon	151,248,064	788,640,101	19.18%
RADIX	222,772	23,251,486	0.96%
CHOLESKY	78,175	924,734,914	0.01%
Mpegenc	62,020,947	192,784,501	32.17%
Mpegdec	1,312,658	15,388,463	8.53%
APP1	10,594,745	18,069,904	58.63%
APP2	72,411,385	112,554,584	64.33%
APP3	23,963,057	36,836,670	65.05%
APP4	5,899,673	12,160,014	48.52%

Table 5.9: Thread load imbalance: 4-processor system

In the subsequent part of this section we report on the power savings that can be achieved by the proposed distributed synchronization architecture. The cache power expenditure have been obtained through Cacti v4.2 tool [109] for $0.18\mu m$ technology. The energy associated with the additional hardware structures for the proposed approach is also accounted for. The DSCs are modeled as small SRAM

Energy (mJ)	DSC	Baseline	Reduction
FFT	28,043	28,057	0.05%
LU_con	139,786	214,661	34.88%
LU_noncon	132,593	161,350	17.82%
RADIX	4,411	4,454	0.95%
CHOLESKY	100,796	100,811	0.01%
Mpegenc	29,531	41,324	28.54%
Mpegdec	3,056	3,305	7.55%
APP1	1,406	3,420	58.89%
APP2	7,625	21,394	64.36%
APP3	2,458	7,014	64.96%
APP4	1,174	2,296	48.86%

Table 5.10: Energy characteristics: 4-processor system

tables, which energy is similarly obtained by Cacti. The counters are modeled as 4-bit up/down counters, in order to support a maximum of 16 processors.

We have evaluated the most conservative power management technique as described in Section 5.4.5, i.e. disabling the accesses to the data cache while the application performs a spin-lock implementation. This power management technique is relatively easy to implement as the DSC can directly control the cache access for the lock/barrier variables while the lock/barrier is not yet available for the local processor. As described in Section 5.4.5 this can be easily implemented in an ISA-transparent or, in an even more straightforward way, with a special ISA support for lock acquire and release instructions. For this study we have used the Splash benchmarks, as well as the parallel Alp-Mpeg and the four data streaming parallel applications as described in the beginning of this section. We have experimented with system configurations consisting of 4 and 8 processors.

Table 5.9 reports the number of cycles that each benchmark spends while waiting to acquire a lock or be released from a barrier. The reported numbers are for a 4-processor system. The total number of execution cycles are also reported. As can be seen, the results vary significantly across the benchmarks. Some kernels, such as FFT, RADIX, and CHOLESKY from Splash2, are well balanced and execute a relatively small number of synchronization operations. Consequently, the idle cycles spent on synchronization are minimal. In other benchmarks, however, a significant thread disbalance is observed that can be exploited for sizable energy savings.

Table 5.10 reports the potential energy reductions for a 4-processor system. The DSCs gate the cache accesses to the synchronization variables; consequently,

	Idle Cycles (I)	Total Cycles (T)	I/T
FFT	182,147	393,749,964	0.05%
LU_con	897,590,421	1,626,303,087	55.19%
LU_noncon	334,876,812	976,744,941	34.28%
RADIX	698,940	24,808,530	2.82%
Mpegenc	37,055,282	209,898,754	17.65%
Mpegdec	3,073,405	20,074,903	15.31%

Table 5.11: Thread load imbalance: 8-processor systems

during the spin loop for acquiring a lock, no data cache energy is expended. Clearly, the energy reductions are strongly related to the thread load disbalance. The more balanced the threads, the less amount of time is spend idling on a lock or within a barrier. The average energy reduction across these benchmarks is 29%. Tables 5.11 and 5.12 report the load-imbalance and energy results for a 8-processor system.

It must be noted that as the number of processors increases, the DSC improvements increase as well. This is because as the system scales up, the synchronization contention as well as the associated overheads increase faster, which makes the benefits of the DSC controllers even more prominent. This indicates the good scalability and efficiency achieved by the proposed distributed synchronization architecture.

Energy (mJ)	DSC	Baseline	Reduction
FFT	13,236	13,270	0.26%
LU_con	36,178	206,843	82.51%
LU_noncon	45,835	93,379	50.92%
RADIX	1,562	1,695	7.84%
Mpegenc	15,603	22,649	31.11%
Mpegdec	879	1,464	39.92%

Table 5.12: Energy characteristics: 8-processor system

5.7 Conclusions

In this work we have presented a novel synchronization implementation approach that significantly improves performance, bus bandwidth, and power efficiency of the fundamental synchronization primitives. The proposed methodology features light-weight distributed controllers across the system that cooperatively implement a distributed synchronization protocol. Operating system and compiler support are integrated to provide flexible management of the synchronization controllers. Experimental results demonstrate significant performance, bus bandwidth, and energy improvements for a broad range of benchmarks.

Chapter 6

Off-Chip Memory Bandwidth Minimization through Cache Partitioning for Multi-Core Processors

6.1 Overview

Uniprocessor systems have encountered enormous design difficulties as processors reached GHz frequencies. Design complexity, circuit synchronization, power consumption, and thermal issues have hindered the rate of further advancements. To continue the growth of computer system performance, the industry has turned to multi-core platforms [94]. Multi-core processors, which often consist of multiple but simpler cores running at lower frequencies, naturally address many of the above problems with promising and steady increases of theoretical peak performance [57]. However, multi-core architectures impose significant challenges of their own [94] that have been the focus of the state-of-the-art research in industry and academia.

One of the most challenging problems facing multi-core systems is the widening gap between the ever increasing memory bandwidth demand due to the increasing number of processor cores and the limited speed of accessing off-chip memory structures [20, 95, 42, 68]. Shrinking transistor dimensions have enabled unprecedented integration density, however, the off-chip memory access time has remained relatively unchanged. To keep up with the scaling trend, large on-chip caches are implemented

accompanied with many other hardware and software optimization techniques. In the absence of sufficient off-chip memory bandwidth, the number of cores that can be integrated into a single chip will be severely restricted. This is especially true for most memory demanding applications, which achieved parallelism often falls far short from the expected teoretical levels, due to serialization and wasted cycles on the memory traffic. Nonetheless, even applications with moderate memory access demand would soon face the very same problem as multi-core architectures scale up quickly and the gap with respect to available off-chip bandwidth and ability to further increase cache sizes keeps growing.

On-chip caching has been employed as a very effective approach to reduce memory bandwidth requirement. While caching helps reduce off-chip memory bandwidth significantly, the effective use of caches is not always guaranteed. Useful content may be evicted due to address conflicts, which adds to the off-chip memory pressure. The problem is more significant in multi-core systems, in which multiple tasks are executing simultaneously. The contention of cache resources would potentially lead to significant inter-task cache interferences and thus much higher memory bandwidth requirements.

Cache partitioning techniques have been a well studied area in recent years. These cache organizations aim at improving cache utilization to achieve better performance and power efficiency [67, 37, 91, 54, 92, 83]. Most existing cache partitioning techniques focus on reducing cache miss rates in order to increase system throughput or overall performance. However, the majority of these approaches do not consider off-chip memory bandwidth as a primary optimization goal. As we

show in the later sections, memory bandwidth based cache partitioning is superior in terms of performance and power since off-chip memory bandiwdth is a bottlenck that is reached easily by modern data-intensive applications and is the most significant contributor to system perfromance, throughput, and power.

In this work[129], we propose a framework for an L2 cache partitioning in multi-core platforms. The novelty of our approach is that it considers the off-chip memory bandwidth demand of the system as a primary optimization goal; the proposed partitioning algorithm is tailored for off-chip bandwidth demand functions exhibited by the different tasks in the system as a function of the cache resources assigned to the task. A miss-rate metric may result in suboptimal solutions as it does not precisely capture the density (in time) of the off-chip memory accesses, which is by definition what the memory bandwidth requirement captures. It may happen that a task with higher miss-rate may exhibit lower bandwidth demand than another task that has a lower miss-rate. The proposed technique judiciously partitions and allocates last level cache resources to concurrently running tasks according to their specific memory bandwidth requirement characteristics which are functions of the cache resources used by the task. By isolating each task in its own cache partition, inter-task interference is eliminated. Overall system memory bandwidth requirement is minimized, which helps the designers to keep the workloads within the bandwidth budget as much as possible. Our experimental results demonstrate the significant reductions of memory bandwidth demand of a set of multiprogrammed and/or multithreaded benchmarks.

6.2 Related work

The problem of limited off-chip memory bandwidth in multi-core systems has attracted a lot of attention. With transistor density continuing to shrink, a large number of processor cores can be integrated into a single chip, which leads to increasingly powerful processors. On the other hand, the bandwidth to the off-chip main memory hasn't improved much, compared to the processor core scaling. While caching has long been employed as the most effective approach to reduce bandwidth pressure, its effectiveness in the mid/large scale multiprocessor systems could be highly suboptimal due to significant contention across the parallel tasks.

In [20] the authors discuss extensively the problem of limited pin bandwidth to multiprocessor systems. They have focused on program performance in multiprocessor systems and make detailed decomposition of program execution times. They conclude that even more complex on-chip cache structures would prove to be cost-effective, with the limited pin bandwidth severely restricting performance increases. In [95] the authors have carefully studied the requirements for on-chip cache structures along with all sorts of optimization techniques that come along with scaling of processor core numbers. Their study shows that cache size need to grow much faster than processor core numbers to compensate for the limited off-chip bandwidth. Because of that, the near future processors need to allocate a huge percentage of chip area for caches, which means much less core counts than expected. The study also shows that effective bandwidth optimization techniques can help reduce cache size requirement and thus help scaling processor cores. [68] acknowledges the critical

role of last level cache and studies the application of different organizations of L2, including processor based split-L2, private L2 and address-interleaved shared L2 designs. [42] also does extensive research into cache/memory structures, and provide insights into the metrics for performance/bandwidth for multi-core processors.

The topic of cache partitioning has been well studied, in the context of both uni-processor and multi-processor systems. While most of the previous projects consider cache miss rates and per-task performance as an optimization goal, a large number of them also report the reduced bandwidth pressure that comes as a result of proper cache partitioning schemes. In [67], the authors take an efficient software approach and use OS based memory address mapping to thoroughly experiment with multiple cache partitioning schemes that target performance, fairness and quality of service (QoS). This work confirms the effectiveness of cache partitioning schemes that are previously obtained through simulation based studies. The study also provides new insights into cache partitioning technique, one of which indicates that bandwidth requirement can be reduced by using the proper partitioning schemes such that the overall memory bandwidth contention is reduced. This could be more important to an application's performance than sheer cache space. A cooperative cache partitioning for chip multiprocessors is proposed in [24]. The time-sharing between cache partitions for concurrently running threads allows multiprocessor cache partitioning to unfairly supress some threads for greater performance improvement for the entire system, while at the same time maintain fairness among the threads. This method can actually be extended to help the other cache partitioning schemes that focus on overall throughput to achieve better fairness and QoS. [31] also tries

to address the last level shared cache contention through cache partitioning method in a software method.

Very few cache partitioning techniques target off-chip memory bandwidth of multi-core system designs as a primary optimization goal. Considering only cache miss-rates can result in suboptimal solutions with respect to bandwidth since tasks with higher miss rate may exhibit lower bandwidth requirements than tasks with lower miss rates. This paper contributes to a better understanding of the impact on cache partitioning on the system off-chip memory demand as well as a specific algorithm that judiciously identifies a cache partitioning based on the tasks bandwidth demands as a function of the cache resources assigned to them.

6.3 Memory Bandwidth and Last Level Caches in Multi-Core Systems

In this work, we address multi-core systems with a shared last level (L2) cache. Our primary optimization goal is the last level cache utilization with respect to off-chip memory bandwidth demand. This optimization goal, rathen than miss-rate reduction, is selected because of the significant gap between on-chip network and off-chip memory. This is true for most modern multi-core processor systems. Regardless of the type of memory system used, the off-chip memory bandwidth is very likely to be the single largest bottleneck impacting the system performance than any other resource, especially as the system scales to more processor cores [20, 95]. Since the proposed partitioning technique is for last level caches, the type of on-chip network

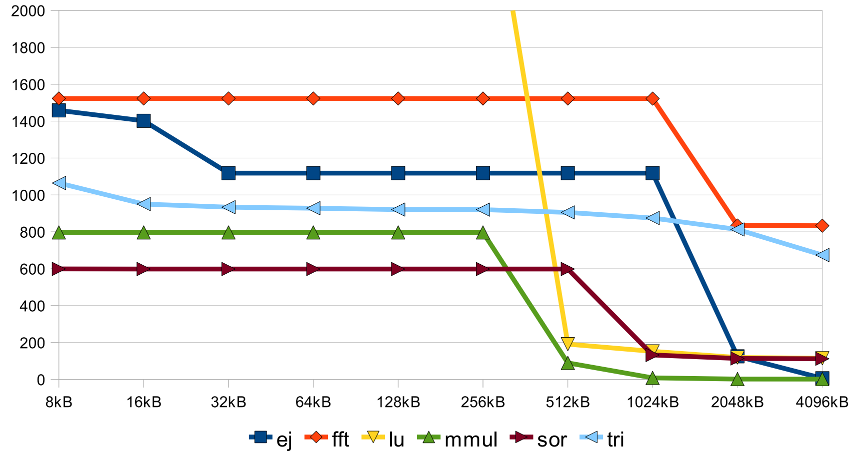

Figure 6.1: Memory Bandwidth Requirement Curves

and its bandwidth is of no special importance.

6.3.1 Bandwidth Demand and Cache Resources

Although many studies exist of the relation between cache organization and cache misses, the direct relation of cache organization and the memory bandwidth requirement in the context of multi-core systems, is a new problem that requires a special attention. As one can naturally expect, the memory bandwidth demand, i.e. the total number of bytes transferred within a fixed time period, correlates well with the total number of cache misses. In our study, we have evaluated the bandwidth demand, cache misses, and miss-rates as a function of the cache resources for a set of applications. This data is shown in Figures 6.1, 6.2, and 6.3

Due to the time factor involved in the off-chip bandwidth requirement, the applications that exhibit larger number of cache miss rates (or misses) do not necessarily consume more memory bandwidth. This can be easily explained with more

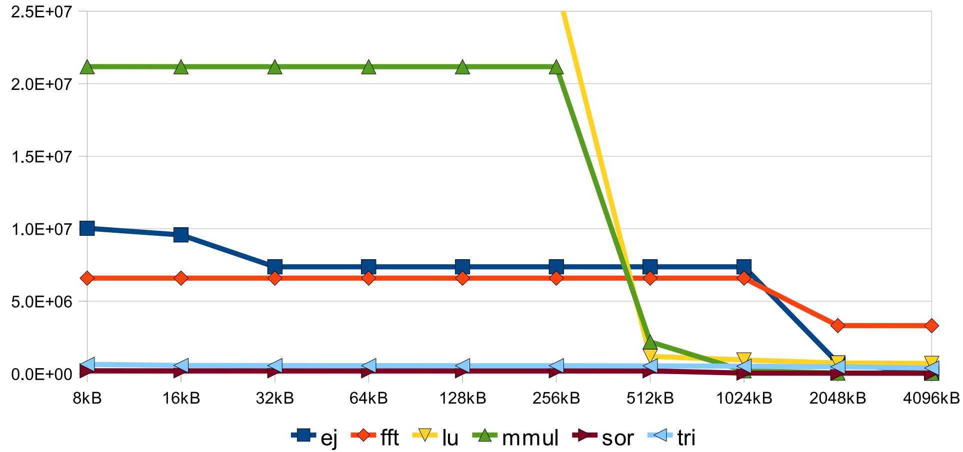

Figure 6.2: Cache Misses Curves

computational instructions that stretch the overall execution and/or good L1 utilization. Meanwhile, applications that exhibit less significant cache miss rates (or misses) could exhibit much higher bandwidth requirements, due to denser memory accesses. Depending on the memory access patterns, some applications, like matrix multiplications (MMUL), incur earlier declines in the bandwidth/misses curve than others, which lend themselves better targets for bandwidth optimizations as they can more efficiently use smaller cache resources without polluting the remaining cache.

Figure 6.1 shows the profiling information of memory bandwidth requirement of several applications as a function of the L2 cache size. The applications work on data arrays with sizes in the range from 0.5MB to 1.5MB. The horizontal axis is the varying L2 cache size that changes from 8kB to 4MB. The vertical axis is the recorded off-chip memory bandwidth, in Mbit/s. For these data we have used the M5 [18] system simulator configured as a single-core processor. A simple CPU model

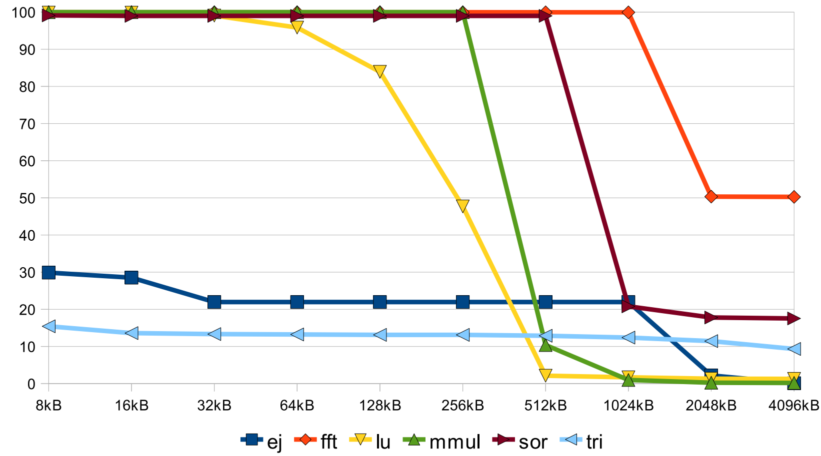

Figure 6.3: Cache Miss-rate Curves

was used to reduce simulation times, which can result in slightly smaller bandwidth demands due to lower IPCs as compared to more sophisticated out-of-order processors that may attempt to speculatively load data. The system bandwidth is set to a very large number so that the required bandwidth from different applications doesn't saturate bus capacity; in effect this setup measures the bandwidth demand of each application.

Figure 6.2 shows the data regarding L2 cache misses for the same applications with the same cache configurations. The vertical axis in this figure depicts the total L2 cache misses recorded during the same execution pass. Figure 6.3 reports the cache miss rate as a function of L2 cache size for the same applications.

As can be seen from these charts, the individual bandwidth curves are correlated (with respect to their shapes) with the cache miss. Comparing across the applications, however, one can find that bandwidth requirement does not correlate linearly with cache misses or miss-rates. For example, LU decomposition consumes

much more bandwidth than MMUL for the same amount of cache misses. Likewise, TRI is more bandwidth demanding than EJ. MMUL, although working across 1.5MB data regions, exhibits the earliest decline in memory bandwidth/cache misses than LU decomposition (LU), which covers 1MB of data. These observations clearly indicate that bandwidth demand need to be considered as a primary minimization goal in partitioning the L2 cache resources between the tasks in the system.

6.3.2 Cache Sharing in Multi-Core Processor Systems

For multi-programmed/multi-threaded workloads, however, there is one added layer of complexity regarding cache sharing when the individual tasks are scheduled to run simultaneously, as is suggested in [67]. With the last level cache being shared among the processor cores, the applications are expected to compete for cache space and could evict each other's cache blocks, therefore generating a sizable number of cache misses in addition to normal self-evictions. This often leads to much more severe memory access contention and thus more severe off-chip bus congestion. An off-chip bus congestion, in turn, will result in severely deteriorated system performance, as can be seen from the above figures, when more cores are integrated into the system.

Whenever such demand exceeds the bandwidth that can be supported by the underlying memory sub-system, the memory bus becomes saturated, and the off-chip memory access becomes the bottleneck of the entire system, i.e. the workload becomes memory bound. At this point, the memory service speed would become

the sole resource that dictates the system performance. Such contention-induced memory accesses directly translate to performance slow-downs, until the bandwidth requirement drops and can thus be sustained by the off-chip memory interface. With ever increasing number of cores to be integrated into a single multi-core platform, it becomes very easy to saturate the off-chip bandwidth with multiple copies of even moderate (with respect to their working set) applications running simultaneously. This makes it even more important to keep control of memory bandwidth demand and reduce unnecessary cache misses. We discuss the cache partitioning method for such purpose in the next section.

6.4 Partitioning Algorithm

6.4.1 Cache Partitioning Basics

Several configurable cache architectures have been proposed in the research literature recently [11, 123, 133]. In what follows we assume a configurable cache that supports a "set"-based partitioning. The algoirthm can be easily extended to support the other forms of cache configurability such as the "way"-based or the combination of the two. The proposed technique identifies a fixed cache partitioning that is used throughout the execution of the entire workload, i.e. the cache is configured in the very beginning when the set of tasks is loaded for execution. Other partitioning mechanisms, such as dynamic partitioning, multiple sharing partitioning (MTP), can also be implemented to give better dynamic bandwidth reductions and more fair benefits to the individual components of a workload.

The cache partitioning implementation used in this work relyes on "set" based partitioning, i.e. each partition is allocated a certain number of sets, each set keeps a fixed number of associativity ways. Due to cache implementation constaints, the number of sets in each partition are required to be a power of two. A particular configurable cache woul support a certain minimal partition size, which our algorithm takes into account. We have set this minimal partition size to 8kB (or 128 cache blocks) in our experiment setup. The power of two size requirement for each partition and the minimum partition size set the granularity of the proposed partitioning algorithm. These parameters, however, even partitioning implementations, can change, to satisfy different partitioning granularity needs and can be handled by (a modified version of) our algorithm.

To find an optimal partitioning scheme, one can certainly perform an exhaustive search to test all possible solutions. However, such a method could become very expensive at finer partitioning granularities and large number of tasks. In view of this, we have devised a heuristic algorithm, which exploits common characteristics of the bandwidth-cache curves in 6.1. The cost of this heuristic algorithm is fixed at O(logS), where S is the size of total L2 cache. We describe this algorithm in the following subsection.

6.4.2 Algorithm Overview

Our partitioning algorithm optimizes towards minimum overall bandwidth and takes as input a) the total available cache size S, and b) the bandwidth-cache charac-

```
PARTITIONING[W, BWdrop_tbl, S]:

 To find a partition scheme with  workload W of K tasks,

bandwidth drop table BWdrop_tbl  with N partition sizes

and total L2 size S

1. FOR i = 1 to K

2.  P[i] = p_min;

//init to min partitions

3. SUM = K*p_min; res = S - SUM;

4.

5. WHILE(res > 0) do

6.  MAX = 0;

7.    FOR k = 1 to K, j = 1 to N

    // find max bandwidth drop and record position

8.     IF SIZE[k][j] < res

and BWdrop_tbl[k][j] > MAX

9.     THEN MAX = BWdrop_tbl[k][j];

10.  p = k; q = j;

11. P[p] = p_min * (2**q);

12 BWdrop_tbl[p][q] = 0;

13. SUM = SUM + P[k]-p_min; //calculate remaining resources

14. res = S - SUM;
```

Figure 6.4: Partitioning Heuristic Pseudocode

teristics of each task, which can be obtained through offline profiling or simulations.

The nature of such an algorithm is NP-hard. This can be proved by reducing the "knapsack" problem to an instance of our problem, as following. The knapsack problem consists of P objects $\{p_i\}$, each associated with a weight $\{w_i\}$, and value $\{v_i\}$. One wants to maximize the total value V under a certain capacity W. If we let the weight be the partition size of a particular task among all possible configurations and value be the bandwidth reduction achieved with that partition size, we get an instance of our partitioning problem, with the target of maximum bandwidth reduction under the constraint of limited total cache space and a total of P tasks. Because of the huge search space of such a problem, we have developed a heuristic algorithm.

The algorithm iteratively attempts a distribution of cache resources to different tasks in order to identify a mapping that minimizes the total bandwidth demand. Due to the power of two contraint of the hardware based partitioning technique, the different partitions can only grow to twice as large between consecutive configurations. Thus for a total cache size of S, there will only be $N = logS$ possible partition sizes. The bandwidth-cache functional relation is used to generate a PxN look-up table that records the bandwidth changes (drop) between consecutive steps for each constituent application in the workload.

The algorithm works in the following iterative steps: It begins by assigning each task the minimum partition size, p_min as the initial partitioning scheme. For each iteration, it calculates the difference of total cache size, S, and the sum of partition sizes of the current partitioning scheme $SUM(P_k)$. This difference is the

amount of available cache resources the algorithm can explore. This number is also used to calculate the maximum number of partition sizes, n, an individual partition can grow from the minimum size.

Subsequently, the algorithm searches for the maximum bandwidth drop that can be reached under current cache resource constraint for all K tasks. Suppose the maximum bandwidth drop belongs to the curve of task p at q-th partition size. In this case, task p is assigned the partition size of $P_min * 2^q$. Note that in each step, the algorithm finds and selects the task that exhibits the *steepest drop in bandwidth demand* when moving from one partition size to the next larger one. For that task, the algorithm selects the cache size for which this maximal bandwidth drop is achieved. The algorithm continues in the same way to identify the partition sizes for the rest of the tasks within the remaining cache resources. Because the partition sizes must be a power of two, this algorithm is guaranteed to complete within $O(logS)$ iterations.

6.4.3 Intuitive and Formal Description

Although this heuristic algorithm is based on a greedy search, it has proven to be very effective for the benchmark workloads in our experiments. And there is a good intuitive reason for this. This heuristic exploits a characteristic that is typical for the majority of tasks' bandwidth-cache functions. As can be seen from the Figure 6.1 in the previous section, most functions start from (the smallest partition size) very high bandwidth requirement/misses/miss-rates and stay on the plateau

until they reach a threashold size, then dive quickly to another much lower plateau.

Furthermore, most functions have only one such dramatically diving phase which represents the biggest bandwidth/misses drop near a favorable partition size. In the heuristic algorithm, this property would show up (in the bandwidth-drop table) as the few largest bandwidth-drop numbers for each task with all possible partition sizes. Selecting the smallest cache size that causes the largest bandwidth drop is clearly very benefitial because the smaller partition sizes before this sharp drop are still too small and the bandwidth demand is close to the worst case of severe cache contention. The larger partition sizes beyond the sharp decline, however, will result in a waste of cache reources since the gains from increased partition size would remain moderate. In this case, it may be better to use the available resource wisely on the other tasks if they can achieve a larger drop in bandwidth.

When the total cache size is relatively small, the algorithm will run out of cache resources shortly after finding a good partition size for very few tasks. The rest of the tasks will keep their initially assigned minimum partition size. Although seemingly unfair to satisfy the needs of the few, these schemes often turn out to achive much lower overal system bandwidth. Judging from the bandwidth-cache functions, most applications stay very close to the high plateau before they reach certain threashold. Taking or giving any resource before that does not result in sizable difference to them. When the total cache size is very large, the algorithm could end up with extra cache space not assigned to any task. Again referring to the bandwidth-cache curves, keeping it or evenly distributing this space will not make much difference either, since most tasks would be sitting on their lower plateau. The

input: 3 tasks, 64kB L2 cache

BWRD	8kB	16kB	32kB	64kB
APP1	20	100	100	100
APP2	0	20	200	200
APP3	0	10	50	500

bandwidth reduction table

BWRD	8kB	16kB	32kB	64kB
APP1	20	80	0	0
APP2	0	0	***180***	0
APP3	0	10	40	450

bandwidth drop table & step1

BWRD	8kB	16kB	32kB	64kB
APP1	***20***	80	0	0
APP2	0	0	***180***	0
APP3	0	10	40	450

bandwidth drop table & step2

BWRD	8kB	16kB	32kB	64kB
APP1	***20***	80	0	0
APP2	0	0	***180***	0
APP3	0	10	40	450

final partitioning

Figure 6.5: Algorithm Walkthrough on Example

real interesting part in between of these two scenarios, however, is very short. Most applications exhibit in the bandwidth drop table only one or two very significant numbers, which is very well suited and targetted by this greedy based algorithm. Figure 6.4 gives a detailed description of this algorithm.

Figure 6.5 provides an example of an input bandwidth table with different partitioning sizes. As can be seen, the algorithm finds the best partitioning scheme in two steps.

Because the nature of the problem is NP-hard, the greedy heuristic algorithm does have adversary cases. This often happens to particular combination of tasks, which is very dependent on the specific workloads. This can be improved significantly, however, by further optimizing the heuristic program with more hints from the programmers. In the experiment section below, a comparison is made between the simpliest form of heuristic algorithm and exhaustive search method to show such differences.

6.5 Experimental Results and Discussion

APP1:	MMUL	MMUL	MMUL	MMUL	TRI	TRI	TRI	TRI
APP2:	MMUL	MMUL	MMUL	MMUL	SOR	SOR	SOR	SOR
APP3:	EJ	EJ	EJ	EJ	FFT	FFT	FFT	FFT
APP4:	EJ	EJ	FFT	FFT	SOR	SOR	TRI	TRI
APP5:	EJ	EJ	LU	LU	SOR	SOR	TRI	TRI
APP6:	MMUL	MMUL	FFT	FFT	TRI	TRI	SOR	SOR

Table 6.1: Benchmark Workloads

6.5.1 Experimental Setup

We have conducted detailed experiments to evaluate the effectiveness of the proposed cache partitioning methodology. We have used the M5 [18] simulator platform to conduct our experiments. M5 is a cycle-accurate full-system simulator and has been extensively enhanced for our study. The simulated machines are configurated as 8-processor systems running at 1GHz, with 16kB split L1 caches and a shared L2 cache. We have evaluated the proposed technique for L2 cache sizes from 256kB to 16MB with a power of two increment. At each configuration, the proposed memory bandwidth-based cache partitioning is applied to compare it with a baseline architecture in which the cores share the entire L2 cache.

6.5.2 Benchmark Applications

Each benchmark workload constitutes a set of parallel applications. Each of these application in turn can be data parallel application and naively parallelized into several identical threads each running on its own part of the input data set. The input data size and the memory access patterns are the biggest factors determining their execution. The individual tasks execute one of the following computational kernels that are broadly used in many numerical and signal processing applications: EJ, FFT, FDCT, LU, MMUL, SOR, TRI. The combinations that we have used as multi-tasked benchmarks are listed in Table 6.1. Among the individual kernels in the workloads, matrix multiplication (MMUL) executed the multiplication of two square matrices; successive overrelaxation method (SOR) is a program for solving a linear system of equations; fast Fourier transform (FFT) computes discrete Fourier transform of input signals; LU decomposition (LU) is a matrix decomposition algorithm used in many communications applications; EJ is the extapolated jacoby method, and TRI is a transformation that converts a matrix into an upper triangular form.

Each of the computational kernels operates on an input data buffer with a size from 0.5MB to 1.5MB. The off-chip bandwidth requiremetns as well as the cache miss characteristics for each task have been reported in a previous section of this work. The entire applications cover data arrays ranging from 4MB to 12MB. All caches are warmed up before the main execution in order to exclude cold cache misses.

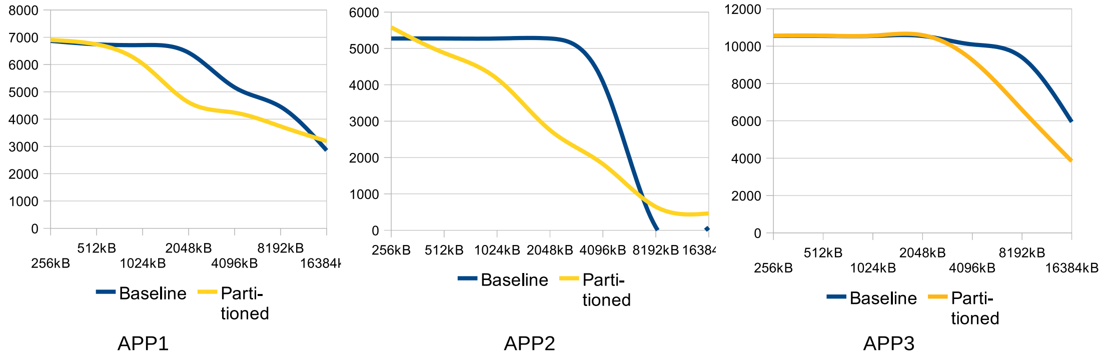

Figure 6.6: Achieved bandwidth v.s. baseline: APP1, APP2 and APP3

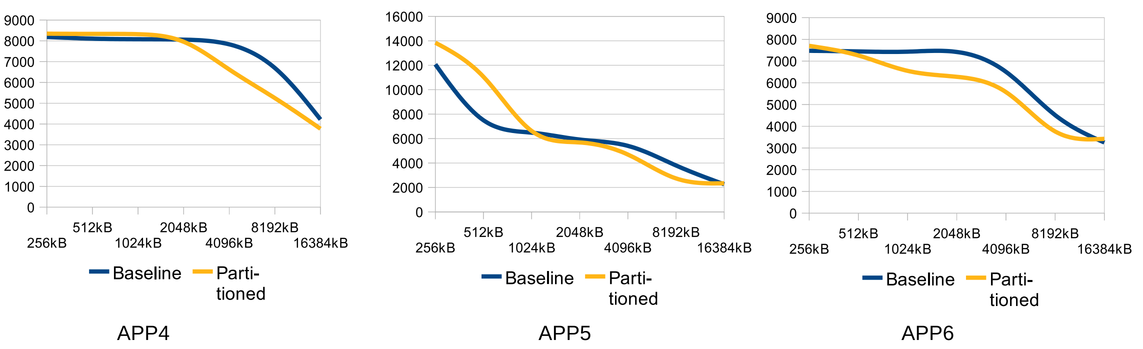

Figure 6.7: Achieved bandwidth v.s. baseline: APP4, APP5 and APP6

6.5.3 Results and Analysis

Figure 6.6 and Figure 6.7 report the results of the six benchmarks compared to the baseline architecture in terms of bandwidth requirements. The yellow lines are the bandwidth requirements from baseline executions. The blue lines represent the overall bandwidth demand when the proposed cache partitioning is applied.

The bandwidth reductions achieved by the proposed technique are clearly demonstarted by the experimental data. Bandwidth requirements reductions of up to 50% are achieved by the proposed cache partitioning.

The experiment results clearly demonstrate three distinct situations with respect to the total L2 cache size. The first situation corresponds to the initial points in the L2 cache size space, i.e. when the entire L2 cache is significantly smaller than 1) the total workload data size and 2) to any individual kernel's earliest bandwidth drop point of its specific bandwidth-cache function. In this situation, cache partitioning cannot help with such limited resources and the algorithm ends up with a configuration with bandwidth requirements close to the baseline - sometimes worse due to the enforced cache partition size limitations.

The proposed cache partitioning exhibits its benefits in the second situation, in which it is possible to allocate sufficiently large fractions of the cache resources to most of the kernels while the rest may remain in their worst scenarios. In this stage, the cache partitioning algorithm is able to exploit the opportunities to move individual kernels off the high plateaus in the bandwidth demand functions. It should be noted that this algorithm minimizes the overall systen bandwidth requirement but it does not do so uniformaly accross all the tasks. In this way, a particular task that can provide the most bandwidth benefit to the system will be selected for most cache resources, especially when such resource are somewhat limited. Replicas of the same tasks are treated as different tasks with the same bandwidth-cache curves. For L2 caches with relatively large sizes, i.e. between 1M and 8M, the algorithm can afford to make more flexible choices. The largest reductions can be seen for such L2 cahes, as the partitioning algorithm is able to move towards the direction of and achieve the fastest bandwidth drops for all the tasks in the workload.

The third distinct situation is when the L2 cache size approaches 16MB. In

this case the bandwidth reductions compared to the baseline are minimal. This can be easily explained by the fact that such large L2 caches can be easily shared by all the tasks as then can fit their working sets with very few or no inter-task conflicts. The difference between these three different situation is often determined by the earliest significant bandwidth-cache reduction point of a task with sizable bandwidth requireemnts. This is also the earliest point when the algorithm can identify an effective partitioning scheme.

Of cousre, it is also possible to have workloads for which the baseline may slightly outperform any cache partitioning. This is largly due to increased contention when seperate cache partitions are enforced. In these cases, multiple kernels do not interfere significantly and actually benefit from larger shared L2 resources. However, even in these cases, the actual bandwidth requirements from the baseline and the one with cache partitioning tend to be very close. This case is illustrated by benchmark APP5. Overall, the experimental results clearly demonstrate that our partitioning technique is effective in achieving significant off-chip bandwidth reductions.

6.5.4 Comparison Between Heuristic Algorithm and Exhaustive Search

As mentioned in the previous section, the heuristic algorithm is greedy based and thus could meet adversary cases for a real NP-hard problem. The study that compares heuristic algorithm with a exhaustive search algorithm is conducted in 6.8.

Shown in 6.8 is the additional amount of total bandwidth achieved through

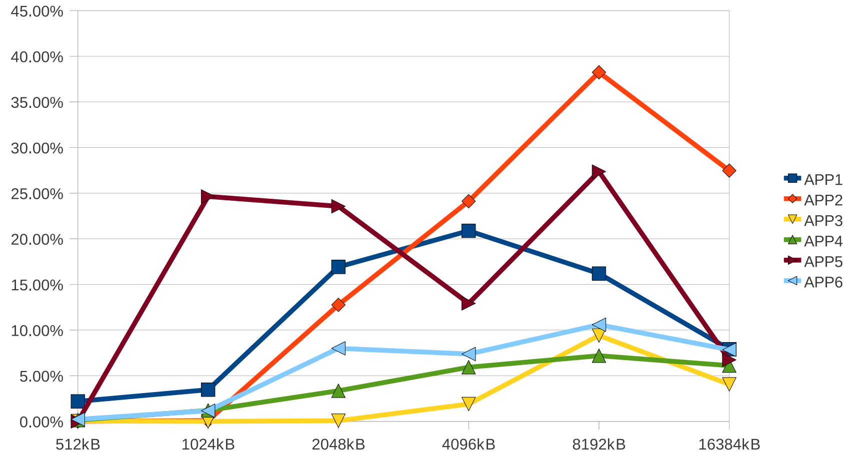

Figure 6.8: Heuristic Algorithm Compared to Exhaustive Search

heuristic algorithm as compared to the exhaustive search.

From this chart, one can see that the heuristic algorithm achieves much of the benefit as an exhaustive one. It starts to behave less effective as the total cache space grows, which corresponds to significantly larger total search space of all possible partitioning schemes . It is also very clear that different workloads lead to different heuristic effectiveness.

Overall, the heuristic algorithm does often come up with reasonably good partitioning scheme with constant time, while the exhaustive search, even with simple tuning, runs considerably longer as the search space grows.

6.6 Conclusions

In this work we have studied the relation between off-chip memory bandwidth requirement and L2 cache partitioning for multi-core processor systems. Off-chip memory bandwidth limitation is becoming a pressing problem for multi-core archi-

tectures, which significantly undermines the scaling trend for future platforms.

We have shown that L2 cache partitioning can be an effective technique in reducing system bandwidth pressure. The proposed partitioning algorithm utilizes the bandwidth-cache functions of individual programs in a workload and identifies a partitioning scheme that significantly reduces overall bandwidth requirement. We have shown convincing experimental data that our bandwidth based cache partitioning approach is effective in alleviating overall memory bandwidth requirements.

Chapter 7

Conclusions

7.1 Embedded Multi-Core Architecture Challenges

In this thesis study, a number of issues about embedded multi-core architectures and their applications are covered, including the cache coherence protocols, the shared memory based inter-core communication, the synchronization mechanism, and last level cache space partitioning for bandwidth reduction. For all these different topics, benchmark studies show the inefficiencies of conventional implementations, in terms of power and performance. Much of such inefficiencies come from the general-purpose architectures that such mechanisms are first implemented for.

There is strong motivation to improve on these inefficiencies. On the one hand, chip multiprocessors are developing into unprecedented and unanticipated level of integration. The quick pace of system scaling into much larger core counts is exposing the conventional implementations to very serious challenges. Such challenges must be addressed before further scaling can be meaningful to system designers.

On the other hand, many of the multi-core mechanisms have been developed towards general-purpose platforms. The trade-offs in embedded system domain make it necessary to tailor such mechanisms for specific applications, thereby reducing overhead and improving power and performance efficiency.

This study shows how much can be achieved by such hardware-software cus-

tomization method.

There is more to certainly more areas to explore and more different techniques to integrate into this framework.

7.2 Cross-Layer Customization for Embedded Multi-Cores

This work also shows the effectiveness of cross-layer customization or hardware-software co-design method for embedded multi-core systems. Such heavy system level tuning often exists in embedded system designs, to make the most potential out of the existing platform.

Embedded systems traditionally adopt a lot from the general-purpose architectures. However, their dedicated nature makes it possible for highly specialized optimizations. The efficiency improvement over the general-purpose counter parts is often very significant.

While general-purpose system designs don't modify hardware layer, they can still tune the software layer to the specifications of hardware platforms. Such practice is more common with large scale CMPs even in the general-purpose systems and the design of future supercomputer systems.

In that regard, this work could also provide some insights into the proper co-design method for such systems.

Bibliography

[1] Cavium octeon multi-core processor family.

[2] *Grand Central Dispatch (GCD) Reference*.

[3] http://www.picochip.com/products/pc102.

[4] http://www.tilera.com.

[5] http://www.vdc-corp.com/ehw/default.asp.

[6] *Intel Threading Building Blocks*.

[7] Xtensa architecture and performance, http://www.tensilica.com.

[8] H. Abdel-Shafi, J. Hall, S.V. Adve, and V.S. Adve. An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors. In *High-Performance Computer Architecture, 1997., Third International Symposium on*, pages 204–215, Feb 1997.

[9] B. Akgul, J. Lee, and V. Mooney. A system-on-a-chip lock cache with task preemption support. In *Conference on Compilers, Architecture, and Aynthesis for Embedded Systems (CASES)*, pages 149–157, 2001.

[10] B. Akgul and V. Mooney. Parlak: Parameterized lock cache generator. In *Design Automation and Test in Europe (DATE)*, pages 1138 – 1139, 2003.

[11] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *International Symposium on Microarchitecture (MICRO)*, pages 248–259, November 1999.

[12] T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[13] James Archibald and Jean-Loup Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems (TOCS)*, 4(4):273–298, 1986.

[14] ARM Ltd. *ARM11 Family*.

[15] C. Ballapuram, A. Sharif, and H-H. Lee. Exploiting access semantics and program behavior to reduce snoop power in chip multiprocessors. In *ASPLOS*, pages 60–69, 2008.

[16] Rizwan Bashirullah, Wentai Liu, and Ralph K. Cavin. Low-power design methodology for an on-chip bus with adaptive bandwidth capability. In *Design Automation Conference (DAC)*, pages 628–633, 2003.

[17] M. Berndl, O. Lhotak, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using bdds. In *Conference on Programming language design and implementation (PLDI)*, pages 103–114, 2003.

[18] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[19] J. Brown, R. Kumar, and D. Tullsen. Proximity-aware directory-based coherence for multi-core processor architectures. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 126–134, 2007.

[20] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors. In *International Symposium on Computer Architecture (ISCA)*, pages 78–89, 1996.

[21] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. Improving multiprocessor performance with coarse-grain coherence tracking. *SIGARCH Computuer Architecture News*, 33(2):246–257, 2005.

[22] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of munin. In *Symposium on Operating Systems Principles (SOSP)*, pages 152–164, 1991.

[23] Anantha P. Chandrakasan. *Low-Power Digital CMOS Design*. PhD thesis, University of California at Berkeley, 1994.

[24] Jichuan Chang and Gurindar S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *International Conference on Supercomputing (ICS)*, pages 242–252, 2007.

[25] G. Chen and M. Kandemir. An approach for enhancing inter-processor data locality on chip multiprocessors. In *Springer-Verlag Lecture Notes in Computer Science*, pages 214–233, 2007.

[26] L. Cheng, J. Carter, and D. Dai. An adaptive cache coherence protocol optimized for producer-consumer sharing. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 328–339, 2007.

[27] L. Cheng, N. Muralimanohar, K. Ramani, R. Balasubramonian, and J. Carter. Interconnect-aware coherence protocols for chip multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, pages 339–351, 2006.

[28] Liqun Cheng and John B. Carter. Fast barriers for scalable ccnuma systems. In *ICPP*, pages 241–250, 2005.

[29] P. Cumming. The ti omap platform approach to soc. In *Winning the SOC Revolution*. Kluwer Academic Publishers, 2003.

[30] M. Das. Unification-based pointer analysis with directional assignments. In *Conference on Programming language design and implementation (PLDI)*, pages 35–46, 2000.

[31] Livio Soares David Tam, Reza Azimi and Michael Stumm. Managing shared l2 caches on multicore systems in software. In *WIOSCA '07*, 2007.

[32] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 223–234, 2007.

[33] M. Ekman, F. Dahlgren, and P. Stenstrom. Tlb and snoop energy-reduction using virtual caches in low-power chip-microprocessors. In *ISLPED*, pages 243–246, August 2002.

[34] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *International Symposium on Computer Architecture (ISCA)*, pages 15–26, May 1990.

[35] O. Golubeva, M. Loghi, and M. Poncino. On the energy efficiency of synchronization primitives for shared-memory single-chip multiprocessors. In *Great Lakes Symposium on VLSI (GLSVLSI)*, pages 489–492, 2007.

[36] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. *SIGARCH Comput. Archit. News*, 17(2):64–75, 1989.

[37] A. Gordon-Ross and F. Vahid. A self-tuning configurable cache. In *Design Automation Conference (DAC)*, pages 234–237, 2007.

[38] Green Hill Software. *INTEGRITY: The most advanced RTOS technology.*

[39] M.R Guthaus, J. S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC-4: Workshop on Workload Characterization*, pages 3–14, December 2001.

[40] M.R Guthaus, J. S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC*, pages 3–14, Dec 2001.

[41] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2001.

[42] Lisa R. Hsu, Steven K. Reinhardt, Ravishankar Iyer, and Srihari Makineni. Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 13–22, 2006.

[43] William Tsun-Yuk Hsu and Pen-Chung Yew. An effective synchronization network for hot-spot accesses. *ACM Trans. Comput. Syst.*, 10(3):167–189, 1992.

[44] http://www.gimp.org. *The GNU Image Manipulation Program.*

[45] J. Huh, J. Chang, D. Burger, and G. Sohi. Coherence decoupling: making use of incoherence. *ACM SIGARCH Computur Architecture News*, 32(5):97–106, 2004.

[46] Intel. *Intel XeonTM Processor at 1.40 GHz, 1.50 GHz, 1.70 GHz and 2 GHz.*

[47] Intel Corporation. *Intel XScale Microarchitecture.*

[48] Vadim Iosevich and Assaf Schuster. A comparison of sequential consistency with home-based lazy release consistency for software distributed shared memory. In *International Conference on Supercomputing (ICS)*, pages 306–315, New York, NY, USA, 2004. ACM.

[49] Ahmed A. Jerraya. Long term trends for embedded system design. In *DSD '04: Proceedings of the Digital System Design, EUROMICRO Systems*, pages 20–26, Washington, DC, USA, 2004. IEEE Computer Society.

[50] V. Kathail, S. Aditya, R. Schreiber, B. R. Rau, D. C. Cronquist, and M. Sivaraman. Pico: Automatically designing custom computers. *IEEE Computer*, 35(9):39–47, 2002.

[51] S. Keckler, W. Dally, D. Maskit, N. Carter, A. Chang, and W. Lee. Exploiting fine-grain thread level parallelism on the mit multi-alu processor. In *International Symposium on Computer Architecture (ISCA)*, pages 306–317, 1998.

[52] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[53] B. Khailany, W. Dally, U. Kapasi, P. Mattson, J. Namkoong, J. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, 2001.

[54] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 111–122, 2004.

[55] L. Kontothanassis, M. Scott, and R. Bianchini. Lazy release consistency for hardware-coherent multiprocessors. *Supercomputing Conference (SC)*, pages 61–61, 1995.

[56] Claus Kuhnel. *Avr Risc Microcontroller Handbook*. Elsevier, 1998.

[57] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *MICRO*, pages 81–92, 2003.

[58] Edward L. Lamie. *Real-Time Embedded Multithreading : Using ThreadX and ARM*. CMP Books, 2004.

[59] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[60] P. Landman and J. Rabaey. Black-box capacitance models for architectural power analysis. In *International Workshop on Low Power Design*, page 65170, April 1994.

[61] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *International Symposium on Microarchitecture (MICRO)*, pages 330–335, December 1997.

[62] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *MICRO*, pages 330–335, Dec 1997.

[63] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *International Symposium on Computer Architecture (ISCA)*, pages 148–159, 1990.

[64] J. Li, J. Martinez, and M. Huang. The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2004.

[65] M-L. Li, R. Sasanka, S. Adve, Y-K. Chen, and E. Debes. The alpbench benchmark suite for complex multimedia applications. In *International Symposium on Workload Characterization*, pages 34–45, October 2005.

[66] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Symposium on Principles of Programming Languages*, pages 201–214, 1997.

[67] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *International Symposium On High Performance Computer Architecture (HPCA)*, pages 367–378, 2008.

[68] Chun Liu, Anand Sivasubramaniam, and Mahmut Kandemir. Organizing the last line of defense before hitting the memory wall for cmps. In *International Symposium on High Performance Computer Architecture (HPCA)*, page 176, 2004.

[69] M. Loghi, M. Letis, L. Benini, and M. Poncino. Exploring the energy efficiency of cache coherence protocols in single-chip multi-processors. In *GLSVLSI*, pages 276–281, 2005.

[70] M. Loghi and M. Poncino. Exploring energy/performance tradeoffs in shared memory mpsocs: Snoop-based cache coherence vs. software solutions. In *Conference on Design, Automation and Test in Europe (DATE)*, pages 508–513, 2005.

[71] M. Loghi, M. Poncino, and L. Benini. Cache coherence tradeoffs in shared-memory mpsocs. *ACM Transactions on Embedded Computing Systems*, 5(2):383–407, 2006.

[72] D. Lyonnard, S. Yoo, A. Baghdadi, and A. Jerraya. Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip. In *Design Automation Conference (DAC)*, pages 518–523, 2001.

[73] S. A. Mahlke, W. Y. Chen, J. C. Gyllenhaal, and W.-M. W. Hwu. Compiler code transformations for superscalar-based high performance systems. In *Conference on Supercomputing*, pages 808–817, 1992.

[74] A. Marongiu, L. Benini, and M. Kandemir. Lightweight barrier-based parallelization support for non-cache-coherent mpsoc platforms. In *Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 145–149, 2007.

[75] Grant Martin. Overview of the mpsoc design challenge. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 274–279, 2006.

[76] M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: decoupling performance and correctness. In *International Symposium on Computer Architecture (ISCA)*, pages 182–193, 2003.

[77] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[78] M. Monchiero, G. Palermo, C. Silvano, and O. Villa. Efficient synchronization for embedded on-chip multiprocessors. *IEEE Transactions on Very Large Scale Integration Systems*, 14(10):1049–1062, October 2006.

[79] J. Montanaro et al. A 160mhz, 32b 0.5w cmos risc microprocessor. In *IEEE ISCC*, pages 214–229, February 1996.

[80] A. Moshovos. Regionscout: Exploiting coarse grain sharing in snoop-based coherence. In *International Symposium on Computer Architecture (ISCA)*, pages 234–245, 2005.

[81] A. Moshovos, G. Memik, A. Choudhary, and B. Falsafi. Jetty: Filtering snoops for reduced energy consumption in smp servers. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 85–96, 2001.

[82] A. Nacul and T. Givargis. Lightweight multitasking support for embedded systems using the phantom serializing compiler. In *Conference on Design, Automation and Test in Europe (DATE)*, pages 742–747, 2005.

[83] Kyle J. Nesbit, James Laudon, and James E. Smith. Virtual private caches. *SIGARCH Computuer Architecture News*, 35(2):57–68, 2007.

[84] Dimitrios S. Nikolopoulos and Theodore S. Papatheodorou. Fast synchronization on scalable cache-coherent multiprocessors using hybrid primitives. In *In Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, pages 711–719, 2000.

[85] Jim Nilsson, Anders Landin, and Per Stenstrom. The coherence predictor cache: A resource-efficient and accurate coherence prediction infrastructure. In *International Symposium on Parallel and Distributed Processing*, pages 10–17, 2003.

[86] A. Patel and K. Ghose. Energy-efficient mesi cache coherence with pro-active snoop filtering for multicore microprocessors. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 247–252, 2008.

[87] T. Pering, T. Burd, and R. Brodersen. Dynamic voltage scaling and the design of a low-power microprocessor system, 1998.

[88] P. Petrov, D. Tracy, and A. Orailoglu. Energy-efficient physically tagged caches for embedded processors with virtual memory. In *Design Automation Conference*, pages 17–22, June 2005.

[89] S. Powell et al. Estimating power dissipation of vlsi signal processing chips: The pfa technique. In *VLSI Signal Processing*, page 250259, 1990.

[90] S. C. Prasad and K. Roy. Circuit optimization for minimization of power consumption under delay constraint. In *Intl Workshop on Low Power Design*, page 1520, April 1994.

[91] M. Qureshi and Y. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *International Symposium on Microarchitecture (MICRO)*, pages 423–432, 2006.

[92] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Architectural support for operating system-driven cmp cache management. In *International Cconference on Parallel architectures and Compilation Techniques (PACT)*, pages 2–12, 2006.

[93] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994.

[94] R. M. Ramanathan. Intel multi-core processors: Making the move to quad-core and beyond. *Technology@Intel Magazine*, pages (1):2–4, 2006.

[95] Brian M. Rogers, Anil Krishna, Gordon B. Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. Scaling the bandwidth wall: challenges in and avenues for cmp scaling. *SIGARCH Computer Architecture News*, 37(3):371–382, 2009.

[96] C. Rowen. *Engineering the Complex SOC. Fast, Flexible Design with Configurable Processors*. Prentice Hall, New Jersey, 2004.

[97] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 34(5):77–90, 1999.

[98] B. Saglam and V. Mooney. System-on-a-chip processor synchronization support in hardware. In *Design, Automation and Test in Europe (DATE)*, pages 633–641, 2001.

[99] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *Symposium on Principles and practices of parallel programming (PPoPP)*, pages 12–23, 2001.

[100] Jack Sampson, Ruben Gonzalez, Jean-Francois Collard, Norman P. Jouppi, Mike Schlansker, and Brad Calder. Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers. In *MICRO*, pages 235–246, 2006.

[101] D. C. Sastry and M. Demirci. The qnx operating system. *Computer*, 28(11):75–77, 1995.

[102] M. Schlett. Trends in embedded microprocessor design. *Computer Magazine*, 31(8):44–49, August 1998.

[103] T. Simunic, L. Benini, A. Acquavia, P. Glynn, and G. De Micheli. Dynamic voltage scaling and power management for portable systems. In *38th DAC*, pages 524–529, June 2001.

[104] J. Singh, W-D. Weber, and A. Gupta. Splash: Stanford parallel applications for shared-memory. *SIGARCH Computer Architectures News*, 20(1):5–44, 1992.

[105] K. Strauss, X. Shen, and J. Torrellas. Flexible snooping: Adaptive forwarding and filtering of snoops in embedded-ring multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, pages 327–338, 2006.

[106] C. H. Tan and J. Allen. Minimization of power in vlsi circuits using transistor sizing, input ordering, and statistical power estimation. In *Intl Workshop on Low Power Design*, pages 75–80, April 1994.

[107] T. Tan, A. Raghunathan, and N. Jha. Embedded operating system energy analysis and macro-modeling, 2002.

[108] Andrew Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2001.

[109] D. Tarjan, S. Thoziyoor, and N. Jouppi. Cacti 4.0: An integrated cache timing, power and area model. Technical report, HP Laboratories Palo Alto, June 2006.

[110] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *International Symposium on Microarchitecture (MICRO)*, pages 356–369, 2007.

[111] M. Thompson, H. Nikolov, T. Stefanov, A. Pimentel, C. Erbas, S. Polstra, and E. Deprettere. A framework for rapid system-level exploration, synthesis, and programming of multimedia mp-socs. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 9–14, 2007.

[112] S. Thoziyoor, N.Muralimanohar, J. Ahn, and N. Jouppi. Cacti 5.3. Technical report, HP Laboratories Palo Alto, April 2008.

[113] Transmeta. *Crusoe LongRun Power Management White Paper.*

[114] Dean M. Tullsen, Jack L. Lo, Susan J. Eggers, and Henry M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, page 54, Washington, DC, USA, 1999. IEEE Computer Society.

[115] O. Villa, G. Palermo, and C. Silvano. Efficiency and scalability of barrier synchronization on noc based many-core architectures. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 81–90, 2008.

[116] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. In *ISCA*, 2005.

[117] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. In *International Symposium on Computer Architecture (ISCA)*, pages 222–233, 2005.

[118] James Y. Wilson and Aspi Havewala. *Building Powerful Platforms with Windows CE*. Addison-Wesley Professional, 2001.

[119] WindRiver. *VxWorks, http://www.windriver.com*. Wind River.

[120] W. Wolf. The future of multiprocessor systems-on-chips. In *DAC*, pages 681–685, June 2004.

[121] Karim Yaghmour. *Building Embedded Linux Systems*. O'Reilly Media, Inc., 2003.

[122] C. Yang and A. Orailoglu. Light-weight synchronization for inter-processor communication acceleration on embedded mpsocs. In *Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, pages 150–154, 2007.

[123] S-H. Yang, B. Falsafi, M. D. Powell, and T. N. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. *Symposium on High-Performance Computer Architecture (HPCA)*, 00:0151, 2002.

[124] C. Yu and P. Petrov. Aggressive snoop reduction for synchronized producer-consumer communication in energy-efficient embedded multi-processors. In *CODES+ISSS*, pages 245–250, 2007.

[125] C. Yu and P. Petrov. Low-power snoop architecture for synchronized producer-consumer embedded multiprocessing. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17, Issue:9:1362 – 1366, September 2009.

[126] C. Yu and P. Petrov. Low-cost and energy-efficient distributed synchronization for embedded multiprocessors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 18 Issue:8:1257 – 1261, August 2010.

[127] Chenjie Yu and Peter Petrov. Distributed and low-power synchronization architecture for embedded multiprocessors. In *CODES+ISSS '08: Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 73–78, New York, NY, USA, 2008. ACM.

[128] Chenjie Yu and Peter Petrov. Latency and bandwidth efficient communication through system customization for embedded multiprocessors. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 766–771, New York, NY, USA, 2008. ACM.

[129] Chenjie Yu and Peter Petrov. Off-chip memory bandwidth minimization through cache partitioning for multi-core platforms. In *DAC '10: Proceedings of the 47th Design Automation Conference*, pages 132–137, New York, NY, USA, 2010. ACM.

[130] Chenjie Yu and Peter Peter Petrov. Aggressive snoop reduction for synchronized producer-consumer communication in energy-efficient embedded multi-processors. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 245–250, New York, NY, USA, 2007. ACM.

[131] Chenjie Yu, Xiangrong Zhou, and Peter Petrov. Low-power inter-core communication through cache partitioning in embedded multiprocessors. In *SBCCI*

'09: Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design, pages 1–6, New York, NY, USA, 2009. ACM.

[132] W. Yuan and K. Nahrstedt. Integration of dynamic voltage scaling and soft real-time scheduling for open mobile systems, 2002.

[133] C. Zhang, F. Vahid, and W. Najjar. A highly configurable cache architecture for embedded systems. In *International Symposium on Computer Architecture (ISCA)*, pages 136–146, 2003.

[134] X. Zhou, C. Yu, A. Dash, and P. Petrov. Application-aware snoop filtering for low-power cache coherence in embedded multiprocessors. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 13(1), January 2008.

[135] Weirong Zhu, Vugranam C Sreedhar, Ziang Hu, and Guang R. Gao. Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures. In *ISCA*, pages 35–45, 2007.