INTEGRATION AND EVALUATION OF CACHE COHERENCE PROTOCOLS FOR MULTIPROCESSOR SOCS

A Thesis
Presented to
The Academic Faculty

by

Taeweon Suh

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
December 2006

INTEGRATION AND EVALUATION OF CACHE COHERENCE PROTOCOLS FOR MULTIPROCESSOR SOCS

Approved by:

Professor Hsien-Hsin S. Lee,
Committee Chair
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor David E. Schimmel
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Douglas M. Blough
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Gabriel H. Loh
College of Computing
Georgia Institute of Technology

Professor Linda M. Wills
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Shih-Lien L. Lu
Microprocessor Research Lab
Intel Corporation

Date Approved: 6 November 2006

To my loving wife,

Soojung Lee

for her support, encouragement, and sacrifice

to get through the agonizing times

and accomplish Ph.D.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude and appreciation to everyone who helped me get through all the difficulties and accomplish this Ph.D.

First and foremost, my adviser, Dr. Hsien-Hsin Sean Lee, helped and guided me professionally and personally during my Ph.D. journey. He taught me everything I needed for Ph.D. study: how to do research, how to write papers, even how to deal with people and how to live as a Ph.D. His clear teachings, reasonings, and diligence inspired me a lot. It was a privilege to finish my Ph.D. under his guidance.

I also want to thank Dr. Shih-Lien Lu. He supervised me at Intel for more than a year. I was amazed by his warm and generous understanding. He changed an unfamiliar environment to a fun place to work. Anytime I had problems, I was more than welcomed to discuss with him. His fun and positive stance, his knowledge, his experience always led me to the right direction.

Dr. Douglas Blough supervised me for the first 2 years in Georgia Tech. I still can not forget his thoughtful consideration and supervision when I set foot in the US. His kindness and guidance gave me the strength to get through one of the most difficult times in the study.

I thank Dr. Linda Wills, Dr. David Schimmel, and Dr. Gabriel Loh for serving as the thesis committee members. They taught me and made me rethink what a Ph.D. should be like with sharp, tough, and inspiring questions. Those experiences will be my precious treasure and guide in my life forward.

Our MARS lab members always gave me their friendship, support, and feedback. I specially thank Mrinmoy Ghosh and Chinnakrishnan Ballapuram for their friendship and help academically and personally. I also thank Dong Hyuk Woo and Richard

Yoo for their help in processing paperwork.

My Korean friends who joined Georgia Tech in 2001 and Korea University Alumni in Georgia Tech were all my supporting grounds. When exhausted with classes, work and research, the chats and gatherings with them gave me the times to relax and refresh. I specially thank HongKyu Kim, DongHoon Han, and Chang Ho Lee for their friendship.

I can not forget our small group members in Korean First Presbyterian Church. I specially thank Hyungie Lee and Yungwon Suh for their help and prayers. In times of troubles, their prayers were a great strength to me.

My family gave me incredible support. I could not have accomplished this Ph.D. without their love and support. I first thank my father, Il-Hwan Suh, my mother, Soon-Ja Oum for their encouragement, unconditional love, support and prayers. My grandmother, Myeong-Jeon Choi, had always prayed for me until she passed away. I wish I could see here again and thank for what she did for me. I thank my brother, Gi-Won Suh, my sister, Jeong-Hee Suh for their love and encouragement and serving our parents when I was gone for the study. My mother-in-law, Eun-Sook Lee, my father-in-law, Je-Soo Lee were always great supporters. I thank them for their prayers, support, and encouragement.

I do not know how to express thanks to my wife, Soojung Lee. She has been always with me during the entire Ph.D. years. She waited for me at night, prepared what I needed, helped and encouraged me all the time. She knows and understands me than anyone in the world. She was a good counselor in agonizing times and she doubled the joy in delightful times. I can not image this accomplishment without her.

Finally, I thank my Lord, Jesus Christ, who saved me and promised Heaven by sacrificing Himself, and gave me the meaning and purpose to live in this world.

TABLE OF CONTENTS

DEDICATION iii

ACKNOWLEDGEMENTS iv

LIST OF TABLES x

LIST OF FIGURES xi

SUMMARY xiii

I INTRODUCTION 1

1.1 Problem Statement 3

1.1.1 Integration of Cache Coherence Protocols in MPSoCs 3

1.1.2 Evaluation of Coherence Traffic Efficiency 5

1.2 Thesis Contributions 6

1.3 Thesis Organization 7

II CACHE COHERENCE PROTOCOLS 9

2.1 Snoop-based Cache Coherence 9

2.1.1 Invalidation-based Protocols 11

2.1.2 Update-based Protocols 14

2.2 Directory-based Cache Coherence 16

2.2.1 Memory-based Schemes 17

2.2.2 Cache-based Schemes 19

2.3 Coherence Protocol Evaluation 22

2.3.1 Snoop-based Cache Coherence 22

2.3.2 Directory-based Cache Coherence 23

2.4 Emulation Initiatives for Evaluation 24

2.4.1 RPM 24

2.4.2 MemorIES 25

2.4.3 Other Cache Emulators 25

III SOC INTEGRATION AND COMMUNICATION ARCHITECTURES . 27
3.1 SoC Communication Architecture 28
3.1.1 AMBA 28
3.1.2 CoreConnect 30
3.1.3 SiliconBackplane μNetwork 31
3.1.4 WISHBONE 32
3.1.5 CoreFrame 33
3.1.6 LOTTERYBUS 34
3.1.7 Discussion 34
3.2 SoC Interface Protocols 37
3.2.1 OCP 37
3.2.2 VCI 38
3.2.3 Discussion 39
3.3 MPSoC Architecture & Methodology 40
IV CACHE COHERENCE PROTOCOL INTEGRATION ON SHARED-BUS-BASED MPSOCS 45
4.1 Motivational Examples 46
4.2 Protocol Integration Techniques 47
4.2.1 Read-to-write Conversion 48
4.2.2 Shared Signal Assertion and De-assertion 49
4.2.3 Detailed Descriptions According to Protocol Combinations . 51
4.3 Architectural Features for Performance Enhancement 54
4.3.1 Snoop-hit Buffer 54
4.3.2 Region-based Cache Coherence 57
4.4 Additional Issues in Heterogeneous MPSoCs 59
4.4.1 Synchronization Mechanism for Heterogeneous Processors . 59
4.4.2 Real-time Operating Systems 61
4.4.3 DMA 61
4.5 Case Studies 62

4.5.1 PowerPC755 and Intel486 Integration 62
4.5.2 PowerPC755 and ARM920T Integration 63
4.6 Experimental Setup 66
4.7 Simulation Results 69
4.7.1 Performance of Integration Techniques with Snoop-hit Buffer 69
4.7.2 Performance of RBCC 72
4.8 Conclusion and Discussion 76
V CACHE COHERENCE SUPPORT ON NON-SHARED-BUS-BASED MP-SOCS 80
5.1 Introduction 80
5.2 Coherence Support 82
5.2.1 Bypass Approach 83
5.2.2 Bookkeeping Approach 87
5.3 Hardware Cost Evaluation 91
5.4 Experimental Setup 92
5.5 Simulation Results 93
5.5.1 Performance of the Bypass Approach 93
5.5.2 Performance of the Bookkeeping Approach 96
5.6 Conclusion and Discussion 98
VI EVALUATION OF COHERENCE TRAFFIC EFFICIENCY 100
6.1 Coherence Traffic 102
6.1.1 Coherence Traffic in Generic Snoop-based Coherence Protocols 102
6.1.2 Coherence Traffic on Pentium-III 103
6.2 Evaluation Methodology 105
6.2.1 Shortcomings in Multiprocessor Environment 105
6.2.2 Methodology for Intrinsic Delay Measurement 108
6.3 Experiment Infrastructure 110
6.4 Hardware Design 111
6.4.1 State Machine 112

6.4.2 Cache . . . 113
6.4.3 Statistics Registers . . . 113
6.4.4 FSB Interface . . . 114
6.5 Experiment Procedure . . . 114
6.6 Experiment Results . . . 115
6.6.1 Cache-to-cache transfer . . . 115
6.6.2 Invalidation Traffic . . . 117
6.6.3 Execution Time . . . 118
6.6.4 Intrinsic Delay Estimation . . . 120
6.6.5 Opportunities for Performance Enhancement . . . 121
6.7 Conclusion and Discussion . . . 122
VII CONCLUSION . . . 124
7.1 Cache Coherence Protocol Integration on Shared-bus-based MPSoCs 124
7.2 Cache Coherence Support on Non-shared-bus-based MPSoCs . . . 125
7.3 Evaluation of Coherence Traffic Efficiency . . . 126
REFERENCES . . . 129
VITA . . . 139

LIST OF TABLES

1 Definitions of the notations in the state transition diagrams of Figure 2. 13

2 Definitions of additional notations in the state transition diagram of Figure 3. 15

3 Characteristics of SoC communication architectures. 36

4 Classification of MPSoC platforms according to coherence support. . 46

5 Incompatibility problems and solutions. 48

6 State transition percentages (SPLASH2 was executed on a 16 processor configuration, and Multiprog was executed on a 8 processor configuration). 57

7 Simulation environment. 68

8 Problem of the E state and solution in the bookkeeping approach. . . 89

9 Hardware cost of the bookkeeping approach (cache line=32 bytes). . . 92

10 Processor combinations in Figure 35 and integrated protocols. 93

11 State transition in the cache line and bus signal behavior in the snoop phase on the P-III FSB. 104

12 BRAM usage in the Virtex-II FPGA (XC2V6000) according to the cache sizes. 113

13 Evaluation metrics for coherence traffic evaluation. 115

14 Run-time estimation for each coherence traffic according to latencies (256KB cache in the FPGA). 121

LIST OF FIGURES

1 A symmetric multiprocessor system based on a shared bus. 10

2 State diagrams of invalidation-based protocols. 12

3 State diagram of the Dragon protocol. 15

4 Structure of DSM machine and implementation of the directory structures. 21

5 AMBA communication architecture. 29

6 CoreConnect-based SoC architecture. 30

7 SiliconBackplane μNetwork SoC architecture. 31

8 WISHBONE interconnection. 32

9 CoreFrame-based SoC architecture. 33

10 A system with a wrapped bus and OCP instances. 37

11 Integration of IPs using the VCI protocol. 39

12 PROPHID heterogeneous high-performance multiprocessor architecture. 41

13 Generic MPSoC architecture template. 43

14 Shared-bus-based MPSoC 46

15 Read-to-write conversion to remove the S state. 50

16 Shared signal assertion and de-assertion 51

17 Snoop-hit buffer. 56

18 Region-based cache coherence. 58

19 Case study platforms for shared-bus-based MPSoCs. 64

20 Hardware deadlock problem in PF1 and PF2. 65

21 Worst case results on a two-processor platform. 69

22 Worst case results on a four-processor platform. 70

23 Best case results on a two-processor platform. 71

24 Best case results on a four-processor platform. 72

25 Typical case results on a two-processor platform. 73

26 Typical case results on a four-processor platform. 74

27 Worst case results of RBCC. 75
28 Best case results of RBCC. 76
29 Typical case results of RBCC. 77
30 RTOS kernel simulation results of RBCC. 78
31 Multiprocessor and multiple-bus SoC architecture. 81
32 The bypass approach. 83
33 The bypass approach with the integration techniques. 85
34 The bookkeeping approach. 88
35 Simulation platform for the evaluation of the bypass and bookkeeping approaches. 93
36 RTOS kernel simulation results of the bypass approach (2 tasks on each CPU). 95
37 RTOS kernel simulation results of the bypass approach (32 tasks on each CPU). 96
38 Simulation results of the bookkeeping approach. 98
39 Timing diagram of cache-to-cache transfer on the P-III FSB. 104
40 Bus arbitration delay in SMP systems. 106
41 Delay caused by stalls in the pipelined bus. 107
42 Evaluation methodology of coherence traffic. 109
43 Experiment equipment for coherence traffic efficiency measurement. . 111
44 Hardware design schematic in the Virtex-II FPGA for efficiency evaluation. 112
45 Coherence traffic measurement results. 116
46 Execution time increase compared to the baseline (5 runs, baseline = 5635 seconds). 119

SUMMARY

The objective of this thesis is twofold. The first objective is to provide generic methodologies for enabling efficient communication among heterogeneous processors in multiprocessor system-on-a-chips (MPSoCs). The second objective is to evaluate the coherence traffic efficiency based on a novel emulation platform using FPGA.

Embedded systems have several properties for system-on-a-chip (SoC) designers to abide by: low-cost, low-power, soft or hard real-time constraint, and short time-to-market requirement. These properties coerce industries into embracing the IP-based design concept. Exhibiting this design trend, international consortia such as OCP-IP and VSIA devised standard SoC interface protocols for the seamless integration of heterogeneous IP blocks. To meet their performance and cost constraint, SoC designers integrate multiple, sometimes, heterogeneous processor IPs to perform particular functions. Nevertheless, the integration of heterogeneous processors arouses complications because of different interface protocols and incompatible communication mechanisms, in particular, cache coherence protocols. Whereas the interface problems are being well studied in academia and industry, the communication problems among heterogeneous processors have not been addressed.

The first two contributions of the thesis provide efficient communication mechanisms among heterogeneous processors via the integration and support of cache coherence protocols in MPSoCs. Heterogeneous processors have incompatible coherence protocols. Hence, special care should be taken to efficiently make use of existing hardware in processors' IPs. Our contributions addressed coherence problems for two main MPSoC architectures: *Shared-bus-based MPSoCs* and *Non-shared-bus-based MPSoCs*. In shared-bus-based MPSoCs, the integration techniques guarantee data consistency

among incompatible coherence protocols. An integrated protocol will contain common states from distinct coherence protocols. A snoop-hit buffer and region-based cache coherence were also proposed to further enhance the coherence performance. For non-shared-bus-based MPSoCs, the bypass and bookkeeping approaches were proposed to support cache coherence in a new cache coherence-enforced memory controller. The simulations based on micro-benchmark and RTOS kernel showed the benefits of the methodologies over a generic software solution.

The third contribution of the thesis evaluated the coherence traffic efficiency. As the memory wall becomes higher, it is imperative to understand the impact of communication among processors and enhance future communication architectures based on observations. Traditionally, the evaluations of the snoopy protocols focused on reducing bus traffic using trace-based or execution-driven simulations, and the impact of coherence traffic on system performance has not been explicitly investigated.

Using an Intel server system and an FPGA, our novel method measured and quantified the intrinsic delay of coherence traffic and evaluated its efficiency. The intrinsic delay was measured by completely isolating the impact of coherence traffic on system performance. The technique eliminated non-deterministic factors in measurements such as bus arbitration delay and stall in the pipelined bus. The experimental results showed that the cache-to-cache transfer in the Intel server system is less efficient than the main memory access.

CHAPTER I

INTRODUCTION

With the advances in the process and design integration technologies, an entire system is now possible to fit onto a single chip called system-on-a-chip or SoC. Even though this ever-increasing chip capacity offers embedded system designers much more flexibility, the typical requirements of an embedded system such as high performance, low power, low cost, meeting real-time constraint, and fast time-to-market, etc., can still restrain SoC designs. To facilitate these often contradictory requirements, embedded processors continue to evolve in both the general-purpose processor's side (e.g., ARM [3] and MIPS [9]) and the configurable processor's side (e.g., Tensilica [16], ARC [2], and Improv [5]). Regardless of the evolution path of the embedded processors, the design of SoC systems are driven by the needs of applications. In other words, processors are chosen and integrated based on the nature of the required tasks running on the systems [71].

Typically, general purpose processors are used to perform control-centric and some signal processing tasks with low-to-medium performance requirement. On the other hand, SoCs also integrate application specific processors such as digital signal processors to achieve high performance. Time-critical tasks with extremely high performance demands such as inverse discrete cosine transform and fast Fourier transform, typically require dedicated hardware, often provided in the form of intellectual properties (IPs) to reduce the time-to-market design cycle.

The latest trend of SoC designs involves the integration of heterogeneous processors. Applications in the domains of multimedia, wireless, network, and gaming,

etc., demand such an approach to attain maximal performance by exploiting the distinctive computing strength offered by different processors. Several commercial SoCs are available using this approach, for example, Texas Instruments' OMAP 2 [11], LSI's DiMeNsion 8650 [4], Analog Devices' GSM baseband processor AD6525 [1], Philips' Nexperia pnx8500 [10], to name a few. Therefore, it is imperative to provide efficient and effective design methodologies for heterogeneous processors and IPs to reduce time-to-market design cycle. Reflecting this trend, international consortia formed by industry partners such as OCP-IP and VSIA has devised standard SoC interface protocols to enable seamless integration of heterogeneous IP blocks. In academia, various multiprocessor research thrusts on embedded systems are being actively conducted [33, 95]. Nevertheless, the integration of heterogeneous processors in multiprocessor SoCs (MPSoCs) can be rather complex due to their distinct interface protocols and incompatible communication mechanisms, in particular, in the cache coherence protocols. Although the interface problems were well studied in academia and industry, the communication problems among heterogeneous processors have not been properly addressed. In this thesis, we addressed the communication problems and provided generic solutions to enable efficient communication via cache coherence protocols.

The performance of future computing systems is becoming less scalable as the disparity between processor and DRAM memory continues to grow. It is referred to as the *memory wall* problem [112]. To bridge the memory discrepancy, a large number of on-die transistors are dedicated to increasingly larger caches and other memory-related architectural features [67, 64, 28, 53, 63, 93, 37, 82, 94]. Furthermore, many bus protocols are pipelined to improve the overall throughput. For example, the design of the front-side bus (FSB) [6] in the P6 processor family consists of 7 pipeline stages. To further expedite data processing, applications are parallelized or multithreaded on multiprocessor systems. Server-class multiprocessor systems are often

based on shared-bus architecture. Recently, even desktop computers are sold with shared-bus multiprocessor configuration, often referred to as symmetric multiprocessor (SMP). To maintain data consistency in SMP systems, multithreaded applications communicate among processors via cache coherence protocols. Due to the sharing of the memory bus, the communication becomes a limiting factor in performance as the number of processors increases. Despite the importance, the efficiency of coherence traffic has not been explicitly evaluated with real systems.

More recently, the capacity, speed, and complexity of field programmable gate array (FPGA) have been substantially improved. It is not uncommon these days to have FPGAs with logic gate capacity in the order of millions. A recent study at Intel showed that the original Pentium processor (P5) fits into a single FPGA. It occupies only about 40% of the Virtex-4 LX200 [114] device. The FPGA vendors also provide built-in memory and a variety of IP blocks so that the design cycle can be greatly shortened. Furthermore, the operating frequency of today's FPGA approaches roughly one-tenth of a high-end processor's speed, making real-time pre-silicon verification of the SoC/ASIC development possible. By leveraging the advantages of the emulation technique using FPGA and combining with an Intel server system, we proposed a novel emulation method for the evaluation of coherence traffic efficiency.

1.1 Problem Statement

1.1.1 Integration of Cache Coherence Protocols in MPSoCs

The use of IP components in an SoC design provides two major advantages. First, it accelerates the processing speeds by exploiting the distinctive property of each IP component. Secondly, it creates an opportunity for achieving higher power efficiency while meeting the specific computing needs. Both are essential goals for delivering an embedded system. For efficient communication among IP cores, several companies such as ARM, IBM, Sonics, and PALMCHIP proposed using their proprietary or

open protocols such as AMBA, CoreConnect, SiliconBackplane μNetwork, and CoreFrame as standard communication architectures. These communication architectures mainly based on bus topologies are typically designed for interfacing their own processors such as ARM and PowerPC. Such lack of interoperability circumscribes the integration of heterogeneous IPs in SoCs. Consequently, IP vendors have to provide versatile IPs tuned into variegated communication architectures. To bridge the gap, international consortia such as OCP-IP and VSIA proposed using "socket," so designers can exercise "plug-and-play" integration practices. The basic concept behind socket is to define a point-to-point interface between two communicating entities such as IP cores and bus interface modules. This approach provides a seamless integration of IPs that conform the interface standards, regardless of the on-chip communication standards.

Nevertheless, the design complexity of integrating heterogeneous processors in MPSoCs is not trivial since it introduces several challenges in both design and validation as a result of distinct interface protocols and incompatible communication mechanisms, in particular, cache coherence protocols. First, to make the integration seamless, the interface protocols should provide the superset functionality of all different processor interfaces. In other words, it should not only support standard operations such as memory read and write operations, but also provides processor specific operations such as machine-dependent exceptions. Second, even with the socket approach, incompatible coherence protocols cannot be integrated and used together because they result in data inconsistency.

To address the communication challenges among heterogeneous processors, the first two contributions of this thesis provide generic methodologies to enable efficient communication. The methodologies integrate incompatible coherence protocols and support data coherence for both shared and non-shared bus architectures with minimal hardware addition.

1.1.2 Evaluation of Coherence Traffic Efficiency

The performance of the cache coherence protocols were evaluated in several literatures [57, 66, 88, 107, 39, 40, 26, 49, 51, 50, 48, 21, 85, 58, 44]. Traditionally, the evaluations of coherence protocols focused on protocols themselves, and the system-wide performance impact of coherence traffic has not been explicitly investigated using off-the-shelf machines. When workloads are parallelized and run natively on SMP systems, the speedup is dependent on three factors: (1) how efficiently workloads are parallelized; (2) how much communication is involved among processors; (3) how efficiently the communication mechanism manages communication traffic (for example, cache-to-cache transfer between processors). While programmers make every effort to efficiently parallelize workloads, the underlying communication mechanism of the architectural implementation remains unmanageable in the software layer and it becomes the limiting factor of the speedup as the number of processors increases.

Despite of the importance of the communication, it has not been feasible to separate its contribution from the speedups measured on SMP machines. Oftentimes, because of the difficulty of the direct evaluation on real machines, software simulators [36, 76, 86, 92, 84] were developed to characterize the multiprocessor system performance. However, the software-based simulation is sometimes difficult to reach an unbiased conclusion since the exact real-world modeling such as I/Os is difficult. In addition, it hinders the broad range measurement of the system behavior due to the intolerable simulation time.

Toward these issues, we proposed a novel emulation technique using an off-the-shelf system and an FPGA. Our study evaluated the exclusive impact of coherence traffic on the system performance. Notably, our methodology made it possible to measure the intrinsic delay of coherence traffic by completely eliminating the non-deterministic factors such as arbitration delay and stall in the pipelined bus in a multiprocessor system.

1.2 Thesis Contributions

The contributions of this thesis are threefold.

- We studied generic solutions for enabling efficient communication among heterogeneous processors by integrating incompatible cache coherence protocols in a *shared-bus-based MPSoC*. The integration techniques include *read-to-write conversion* and *shared-signal assertion/de-assertion*. To further enhance the coherence performance, two low-cost architectural techniques were proposed: *snoop-hit buffer* and *region-based cache coherence*. Using Verilog-HDL, the proposals were applied to simulation platforms with commercial embedded processors. Synthesis and simulation results showed the cost-effectiveness and performance improvements over a generic software solution. We also discussed the hardware deadlock problem present when integrating processors with no native coherence support.
- The previous study was extended to address the communication problem among heterogeneous processors in a *non-shared-bus-based MPSoC*. We studied two approaches that support coherence in a non-shared-bus MPSoC — *bypass* and *bookkeeping* approaches. Using Verilog-HDL, we implemented simulation platforms with the proposed techniques. We reported the implementation costs and quantified the performance benefit over a conventional software solution.
- To quantify the impact of coherence traffic, a novel approach for measuring the intrinsic delay of coherence traffic was proposed. For the experiment, cache, statistics modules and the front-side bus protocol in Pentium-III were implemented in the Virtex-II FPGA. Then, statistics information was gathered during the native execution of the benchmarks. After compiling collected data, we analyzed data to evaluate the coherence traffic efficiency on an Intel server system. We found that the cache-to-cache transfer in the Intel server system is less efficient than the main memory access.

1.3 Thesis Organization

This thesis is organized as follows.

- *Chapter II: Cache Coherence Protocols*

  This chapter introduces classic coherence protocols. Snoop-based coherence protocols and directory-based coherence protocols were surveyed and summarized. Emulation initiatives for the coherence protocol evaluation were also surveyed.

- *Chapter III: SoC Integration and Communication Architecture*

  This chapter reports the survey of SoC communication architectures proposed by several companies, and the SoC interface protocols from OCP-IP and VSIA. Additionally, it presents academic researches on design methodologies for MPSoC architectures.

- *Chapter IV: Cache Coherence Protocol Integration on Shared-bus-based MPSoCs*

  This chapter presents the first contribution of the thesis. We proposed a generic methodology of integrating incompatible cache coherence protocols in shared-bus-based MPSoCs. Then, the proposal is extended to cover two architectural features to enhance the coherence performance. This chapter also discusses the implication and limitation of integrating processors with no native coherence support.

- *Chapter V: Cache Coherence Support on Non-shared-bus-based MPSoCs*

  This chapter presents the second contribution. Here, we extended our scope to non-shared-bus-based MPSoCs and proposed ccMC. Then, two approaches to support the coherence were presented in details.

- *Chapter VI: Evaluation of Coherence Traffic Efficiency*

  This chapter presents the third contribution of this thesis. Using an Intel server system and an FPGA, we proposed a novel methodology to measure the intrinsic delay of coherence traffic and to evaluate its efficiency. With the coherence cache

implemented in FPGA, this methodology evaluated coherence traffic efficiency by natively running standard benchmarks. We also discussed the opportunities to enhance the coherence traffic efficiency.

- *Chapter VII: Conclusions*

  This chapter concludes this thesis.

CHAPTER II

CACHE COHERENCE PROTOCOLS

Cache coherence protocols are required for maintaining data consistency and guaranteeing data correctness in a multiprocessor system. A large number of cache coherence protocols [44, 88, 26, 106, 57, 78, 107, 65, 58, 52, 73, 68] were proposed and developed in the '80s and early '90s. More recently, research thrusts such as the token coherence protocol [77] have continued to improve the efficiency of coherence. Depending on the scale and requirements of a system design, different coherence protocols may be used. Large-scale multiprocessor systems based on distributed shared memory usually employ a directory-based coherence scheme for data consistency. On the other hand, small- or medium-scale multiprocessor systems commonly use a shared bus architecture, where a snoop-based coherence protocol is implemented to make caches coherent. In the subsequent sections, we detail these two major cache coherence protocols.

2.1 Snoop-based Cache Coherence

Multiprocessor systems based on a shared bus employ a snoop-based protocol for cache coherence. Figure 1 shows a basic structure of such multiprocessor systems. All bus transactions go through a common resource, e.g., the shared bus. Depending on the cache line status and bus transaction type, the processors snoop each bus transaction and respond with appropriate state changes for the corresponding cache lines. The snoop-based coherence protocols can be classified into two main policies: the *invalidation-based protocol* and the *update-based protocol*. In general, a write-back cache is used as the underlying scheme for both policies. The difference between these policies lies in whether to invalidate or update the shared cache lines when

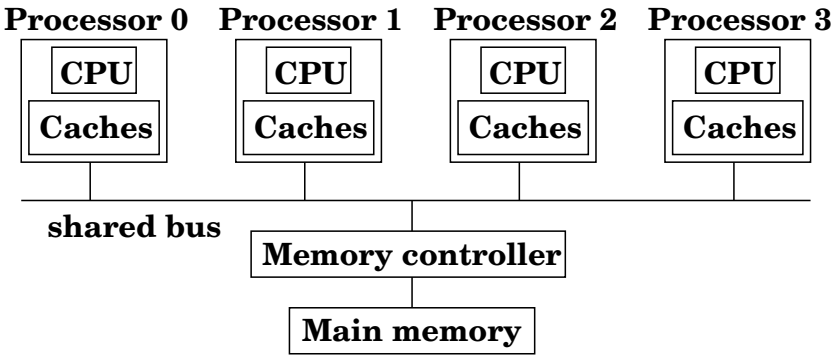


Figure 1: A symmetric multiprocessor system based on a shared bus.

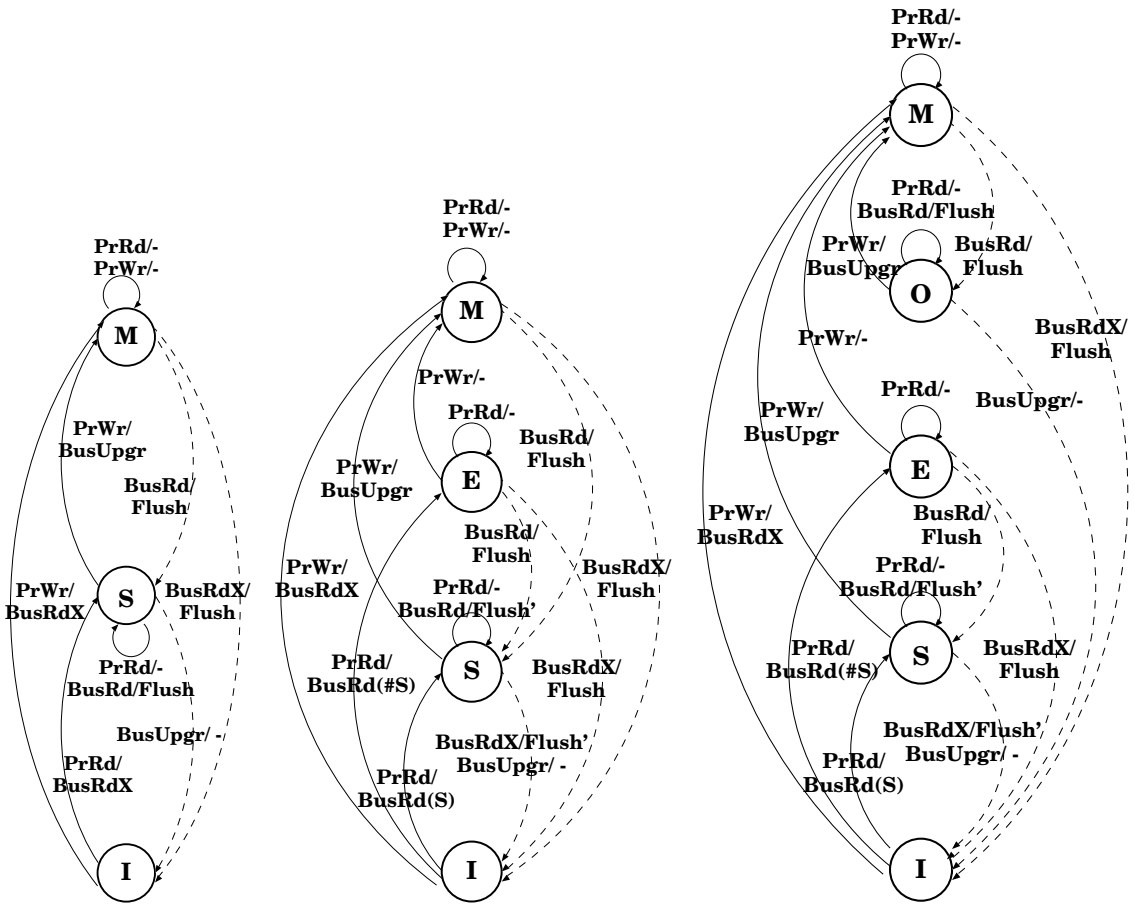
a processor writes to the same memory block. In the invalidation-based protocol, the shared cache lines held by other processors are all invalidated, whereas in the update-based protocol, data is updated in all caches sharing the same memory block. It is controversial to argue with respect to which policy delivers better performance since the performance of each policy is highly correlated to the data-sharing patterns exhibited by the application's workload. In [65, 58], a hardware scheme was proposed to support both policies in one single platform. Given the behavior of an application's workload, it changes policies on a per-cache-line granularity or a per-page granularity in a dynamic manner. Such a hybrid approach, however, is rather expensive in terms of extra hardware needed. In general, the invalidation-based strategies are more robust, and thus most vendors use them as the default protocol [44].

In the 1980s, several invalidation-based protocols (e.g., the write-once [57], the Synapse [52], the Berkeley [66] and the Illinois [26]) and update-based protocols (e.g., the Firefly [107] and the Dragon [78]) were proposed and evaluated [78]. As the multiprocessor system design and understanding become matured over time, the industry vendors adopt some variants of the most efficient protocols for their own systems. In the following sections, we focus on these protocols.

2.1.1 Invalidation-based Protocols

In general, an invalidation-based protocol consists of the following states: **M**odified, **E**xclusive, **S**hared, **I**nvalid, and **O**wned. The *I* state indicates that a data block in the cache is invalid. Multiple processors can share the same data block in their respective caches in the *S* state, but only one processor can have the block exclusively in its own cache in the *E* state. The *E* state also indicates that the cached block has not been modified since it was brought in from the main memory. A processor can have a modified data block in the *M* or the *O* state. The *M* state indicates that the processor owns the modified data block exclusively in its cache, while the *O* state specifies that the modified block may be shared with other processors. For different performance requirements and the affordable complexity, different states can be combined to establish a coherence protocol. Instead of listing all possible combinations, three common coherence protocols, as shown in Figure 2, will be discussed. An optimization called *BusUpgr* [44] for reducing data traffic on the bus is also illustrated in Figure 2. In the figure, a solid line in the state transition indicates that the transaction is initiated by its own processor. On the other hand, each dotted line indicates that the state transition is initiated by a remote processor, i.e., the transition is incurred by the snooping result. The notation *A/B* in Figure 2 means that transaction *B* takes place after the observation of transaction *A*. For example, *PrWr/BusRdX* indicates that *BusRdX* is generated by observing *PrWr*. The definition of each notation is summarized in Table 1.

MSI protocol: As depicted in Figure 2(a), the $I \rightarrow S$ transition occurs when a processor's read operation misses its local cache. When a processor's write operation misses its cache, the $I \rightarrow M$ transition is made with the *BusRdX* transaction. When a processor's write hits a clean line (the *S* state), *BusUpgr* is generated to invalidate the same block in other processors' caches, followed by the $S \rightarrow M$ transition in its own cache. The cache-to-cache transfer could occur during the $M \rightarrow S$ and $M \rightarrow I$ state



(a) MSI protocol (b) MESI protocol (c) MOESI protocol

Figure 2: State diagrams of invalidation-based protocols.

transactions. It also could occur when a processor observes *BusRd* on the *S* state line. Nevertheless, in this case, since several processors might have the same block in the *S* state, a selection algorithm is needed to determine which cache will provide the data [44], resulting in complex hardware and potential performance degradation because of the arbitration mechanism. Silicon Graphics 4D series multiprocessor machines [32] use a protocol similar to MSI.

MESI protocol: The MESI protocol was proposed by Papamarcos and Patel [88] and is often referred to as the Illinois protocol [26]. The state transition of the MESI protocol is similar to the one of the MSI protocol with an additional *E* state included. By introducing the *E* state, as shown in Figure 2(b), the MESI protocol has an

Table 1: Definitions of the notations in the state transition diagrams of Figure 2.

Notation	Definition
PrWr	Local processor's write operation
PrRd	Local processor's read operation
BusRd	Read transaction on the bus
BusRdX	Read exclusive transaction on the bus for the ownership of a memory block, i.e., a remote processor intends to modify a memory block after read
BusUpgr	Same as *BusRdX*, but no data involved since the purpose is to invalidate the same memory block in remote caches
BusRd(S)	Read transaction on the bus, and the shared signal is asserted by remote processor(s)
BusRd(#S)	Read transaction on the bus, and the shared signal is de-asserted by remote processor(s)
Flush	Cache line data supply to the bus for cache-to-cache transfer
Flush′	Same as *Flush*, but data is supplied by the cache responsible for supplying the data
–	No action taken

advantage over the MSI protocol in terms of bus bandwidth savings, but it requires a new bus signal generically named the *shared signal*. When a processor's read misses its own cache, the processor gets the data from the memory or remote caches. If a valid copy exists in other cache(s), which is indicated by the assertion of the shared signal, the $I \rightarrow S$ state transition is incurred for the new cache line. In contrast, if the data cannot be found in any other caches, then the $I \rightarrow E$ state transition occurs for the new line. When there is a subsequent write operation to the same block by the same processor, the $E \rightarrow M$ transition occurs without generating bus traffic because no other caches have the same block as indicated by the E state. Variants of the MESI protocol are implemented in many commercial microprocessors, including Intel's IA32 Pentium class processors [62], AMD K6 [24], and PowerPC 601 [34], to name a few. Depending on the implementation, the behavior of the state machine may vary slightly from the generic one shown in Figure 2(b). For example, the P6

family, including Pentium Pro, Pentium II, and Pentium III, supports the cache-to-cache transfer only for the $M \rightarrow S$ and $M \rightarrow I$ transitions. Note that in the Illinois MESI protocol, the cache-to-cache transfer could also occur during the $E \rightarrow I$, $E \rightarrow S$, and $S \rightarrow I$ transitions. In P6, the *shared signal* is implemented with an *#HIT* front side bus (FSB) signal, and *BusUpgr* is implemented with a special encoding called *0-byte memory read with invalidation* using multiplexed *#REQ[4:0]* FSB signals [6].

MOESI protocol: This protocol was proposed by Sweazey and Smith [106] to further enhance the coherence performance at the cost of additional hardware complexity. By introducing the O state, this protocol allows for some memory blocks in the main memory to not necessarily be the most up-to-date when some processors might have the same block in the S state. This occurs when a cache with the M state line observes *BusRd*, which initiates a cache-to-cache transfer without updating the memory. This transaction also incurs the $M \rightarrow O$ transition in the snooping processor's cache and the $I \rightarrow S$ transition in the requesting processor's cache. Since the SRAM-based cache is considerably faster than the DRAM main memory, such a new state inclusion enables faster transfer compared to a simultaneous update to the memory with the cache-to-cache transfer. Note that in the MESI protocol, the $M \rightarrow S$ transition updates the main memory simultaneously with the cache-to-cache transfer. Variants of the MOESI protocol are also used in modern microprocessors, e.g., AMD64 architecture [25] and SUN Microsystems' UltraSPARC [105].

2.1.2 Update-based Protocols

As mentioned in Section 2.1, the update-based protocol broadcasts data whenever a processor writes to a shared block of memory, so processors always contain the most up-to-date data in their caches. The Firefly [107] and the Dragon protocol [78] are representatives of the update-based protocol.

Figure 3 depicts the state transition diagram of the Dragon protocol, and Table 2

Table 2: Definitions of additional notations in the state transition diagram of Figure 3.

Notation	Definition
PrWrMiss	Local cache miss for processor's write operation
PrRdMiss	Local cache miss for processor's read operation
BusUpd	Data update request from a remote processor's write operation
BusUpd(S)	Same as *BusUpd*, and shared signal is asserted by remote processor(s)
BusUpd(#S)	Same as *BusUpd*, and shared signal is de-asserted by remote processor(s)
Update	Update the cached data according to *BusUpd* request

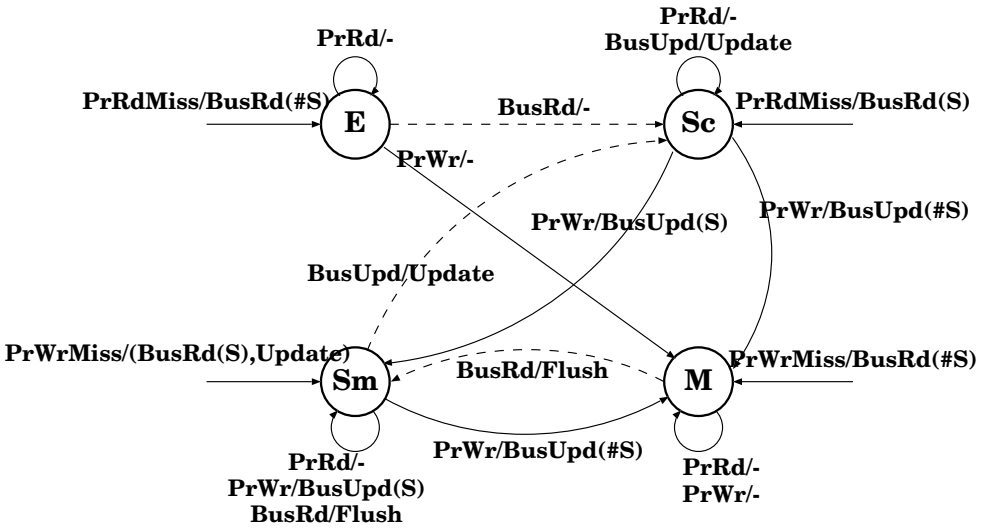


Figure 3: State diagram of the Dragon protocol.

details the additional notations in the state transition diagram. Compared to the states of the invalidation-based protocols, the *Sc (Shared Clean)* and *Sm (Shared Modified)* states are new to this protocol. The *Sc* state is semantically similar to the *S* state, while the *Sm* state is similar to the *O* state. The *Sm* state indicates that the main memory is not coherent, and the cache is responsible for updating the main memory upon eviction. The *Sc* state means that the main memory may or may not be up-to-date since the remote caches may have the same line in the *Sm* state.

Since every write to a potentially shared line (*Sc*, *Sm*) needs to be updated in the remote caches, a new bus transaction called *BusUpd* is also introduced, as shown

in Figure 3. For example, the $Sm \rightarrow M$ transition occurs when *PrWr/BusUpd(#S)* is observed, meaning that the bus update (*BusUpd*) is initiated by a processor's write operation (*PrWr*), but the remote caches inform that the line is not shared by them through (*#S*). When the data update occurs as a result of a write operation to a shared line, the main memory is not updated. Through this scheme, this protocol expects a faster data transfer as the cache is faster than the DRAM main memory. This is the same argument as the one made in the MOESI protocol. The Dragon protocol ensures that data is always valid if the tag matches. Hence, there is no explicit invalid state even though it reserves *a miss mode bit* for compulsory misses.

2.2 Directory-based Cache Coherence

The bus-based multiprocessor makes use of a snoop-based protocol and communicates through a bus, relying on a broadcast mechanism to invalidate or update the data in remote caches. As such, this basic mechanism is less scalable in terms of the number of processors allowed on a single bus because of limited bandwidth and electrical load (capacitance). This lack of scalability is a critical obstacle to the construction of a large-scale system with more than hundreds or even thousands of processors.

During the 1990s, the research on distributed shared memory (DSM) machines [22, 73, 72, 68, 31, 81, 70, 40, 55, 74, 21] concentrated on scaling beyond the number of processors that may be sustained by a single shared bus. The DSM uses a shared address space, and the main memory is distributed across the entire system. As such, unlike the bus-based machine, the data access latency can be different, depending on the location of the data. This type of architecture is often referred to as *Cache-Coherent, Non-Uniform Memory Access (ccNUMA)*. For maintaining cache coherence, the DSM typically employs a directory-based cache coherence protocol [39]. Instead of resorting to broadcasting, the DSM explicitly sends requests to appropriate processing nodes after looking up the directory through network transactions.

Even though the DSM machine was the dominant form of large-scale multiprocessor systems in the 1990s, its popularity diminished after the emergence of cluster computing. The availability of high-speed networks and increasingly powerful commodity microprocessors is making clusters of computers and networks an appealing solution for cost-effective parallel computing. Nevertheless, as the mainstream of future computer architecture research migrates to the many-core (currently implying more than eight processors on a chip) architecture, the directory-based cache coherence is being adopted as a coherence protocol among distributed caches (e.g., L2 or L3) on a chip.

Figure 4(a) shows the basic structure of a DSM machine. As shown, the main memory is distributed across several processing nodes, and each distributed memory is associated with a directory. A node assigned for a given data block is referred to as the *home node* of that block. Each node maintains a directory for its allocated main memory. The granularity of association is typically on a per-cache line basis. A cache miss in a local node could be serviced from either the local memory or a remote memory, depending on the location of the missed address. Like the snoop-based protocol, the directory-based protocol could be an invalidation-based, an update-based, or a hybrid configuration. In most of the proposed implementations for large-scale multiprocessors, the invalidation-based protocols are commonly used. Spurious updates in the update-based protocol incur a separate network transaction for each destination, and such protocols make it more difficult to preserve the desired memory consistency model in directory-based systems [44]. This section discusses common directory structures and operations adopted in the implementation of DSM machines.

2.2.1 Memory-based Schemes

The memory-based scheme maintains the directory information along with the main memory. Figure 4(b) illustrates a simple approach to implementing the directory, where one dirty bit and a presence bitvector are designated for each cache line. The

size of the presence bitvector corresponds to the number of nodes on the system. This directory structure is often referred to as the *full-bitvector directory* or *full-map directory*. The presence bitvector indicates which nodes are currently sharing the same memory block, and the dirty bit implies that only the node with its corresponding presence bit set contains the valid data. A cache read miss in a node i sends a message to the *home node*. If the line is in the clean state (i.e., the corresponding dirty bit is off), the *home node* supplies data along with a response message and turns on the i^{th} presence bit. If the line is in the modified state (i.e., the corresponding dirty bit is on), the *home node* sends the owner's node information to the requester. The requester then sends a request message to the owner node. The owner node supplies the data back to both the requesting node and the *home node*, changing the cache line to the shared state. In the *home node*, the dirty bit is turned off and the i^{th} presence bit is turned on. A cache write miss or a write hit in a local node i involves the invalidation of all sharing nodes indicated by the presence bits of the *home node*. After the invalidation, the corresponding dirty bit and the i^{th} presence bit are turned on in the *home node*. If a cache line in the shared state is replaced in a local node i, a message called the *replacement hint* may or may not send to the home node to turn off the i^{th} presence bit. This helps to reduce unnecessary invalidation messages the next time the block is modified. Note that whether the *replacement hint* is sent or not has no effect on the correctness.

The main disadvantage of the memory-based schemes is the storage overhead of the directory. As the number of nodes increases, the number of presence bits also increases. To overcome the storage problem, Agarwal [21] proposed keeping a fixed number of pointers for each line, which is referred to as a *limited pointer directory* or simply a *limited directory*. This comes from the observation that typically only a few caches share the same block of memory. LimitLESS [40] also has the same structure as a *limited directory* except that the directory overflow is handled by the

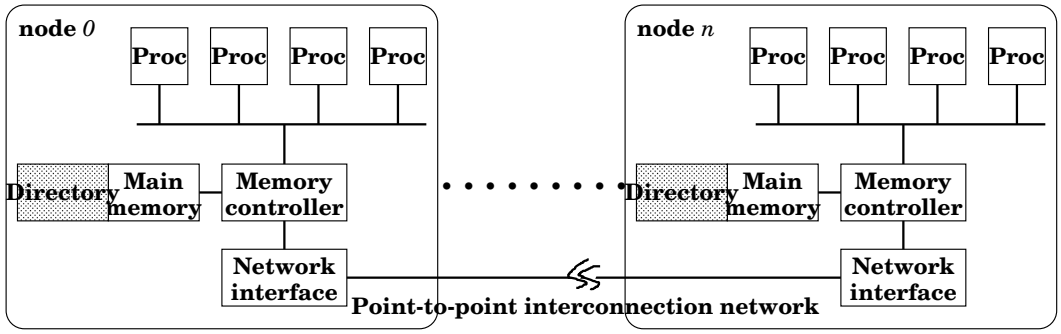
software. Gupta et al. [59] and O'Krafka et al. [85] proposed reducing the directory height, taking advantage of the fact that the size of the cache (i.e., total cached blocks) is much smaller than the one of the main memory. Several DSM machines, including Stanford DASH [73], FLASH [68], MIT Alewife [22], and SGI origin [70], were implemented based on this scheme.

2.2.2 Cache-based Schemes

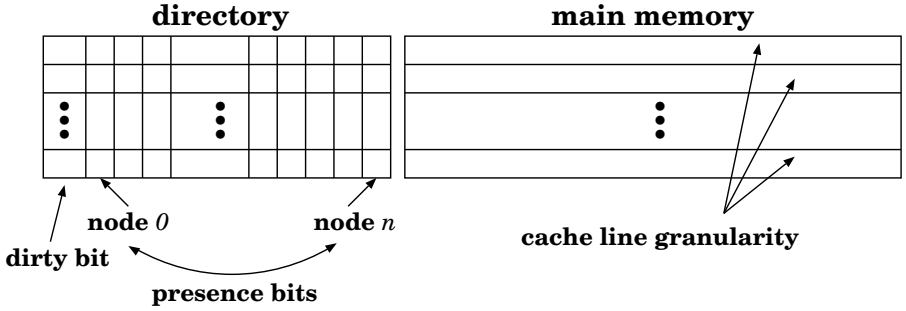
In contrast, the cache-based scheme only keeps the *head pointer* for each block in the main memory, as illustrated in Figure 4(c). The head pointer points to the first sharing node of the block. Then, the forward and backward pointers in the caches of each node direct the next sharer and the previous sharer, respectively, in a doubly linked list fashion. This scheme is called *chained directory*. A cache read miss in a local node sends a message to the *home node* to obtain the *head node* information of that block. After receiving a response from the home node, the requester sends a message to the head node, asking to insert itself to the head of the doubly linked list. The home node or old head node sends the requested data to the requesting node, and the requester becomes the new head node of that block. A cache write miss or a write hit causes a breakup of the doubly linked list since the requester becomes the only owner of that block (no other sharers) in the invalidation-based protocol. This breakup involves a series of invalidation messages from the head node to the end of the linked sharing nodes. Unlike the memory-based scheme, this scheme requires sending messages (*replacement hint* in the memory-based scheme) to neighbors even when a clean line is replaced since the new cache line will need the forward and backward pointers.

Even though the cache-based scheme exhibits some disadvantages such as the long latency resulting from the serialization of the invalidation messages, it has several advantages over the the memory-based scheme. First, the directory overhead is smaller

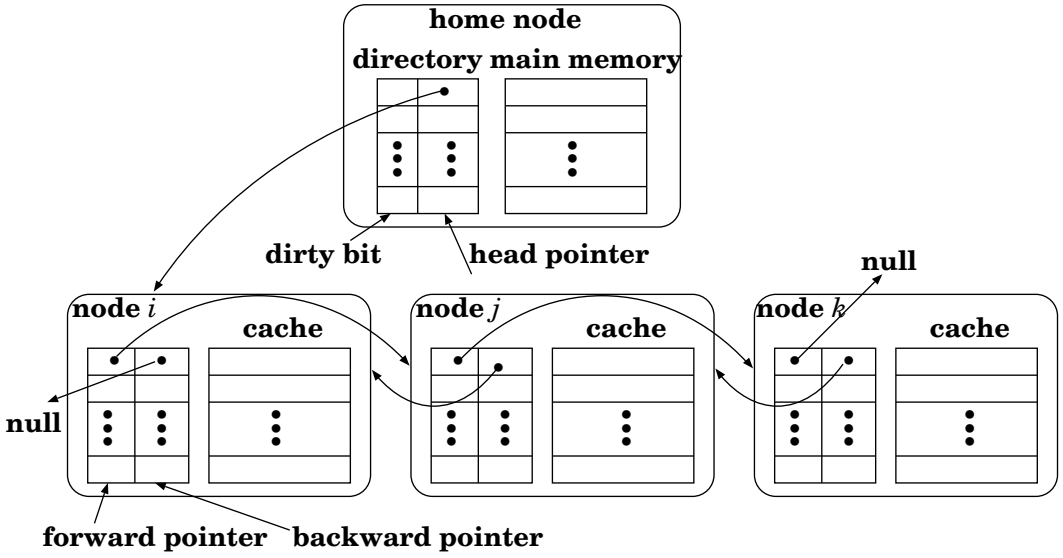
since it only keeps the *head pointers*. Second, it keeps track of the order of the block access through the doubly linked list, making it easier to provide fairness. Third, the invalidation messages are sent by distributed nodes, whereas it is centralized in the memory-based scheme. Several DSM machines, including Sequent NUMA-Q [74], Convex Exemplar [108], and Data General [42], implemented this scheme.



(a) Basic structure of DSM machine



(b) Memory-based schemes



(c) Cache-based schemes

Figure 4: Structure of DSM machine and implementation of the directory structures.

2.3 Coherence Protocol Evaluation

The cache coherence protocol is a crucial design choice for multiprocessor systems, directly affecting the overall system performance. Depending on the selection of coherence protocols and the target application workloads, several system factors can impact the overall performance to different certain levels such as the maximum achievable bus bandwidth (in snoop-based protocols) and the number of network transactions (in directory-based protocols). Since the introduction of coherence protocols, they have been thoroughly evaluated over the past two decades [57, 66, 88, 107, 39, 40, 26, 49, 51, 50, 48, 21, 85, 58, 44]. As multiprocessor research matured in the 1990s, parallel benchmarks such as SPLASH2 [111], NAS [30], ScaLAPACK [41], TPC [18], PARKBENCH [14], and Perfect Club [35] were developed to facilitate the study of centralized and distributed shared memory multiprocessor systems. Most protocol evaluations were performed via trace-driven or execution-driven software-based simulation. This section summarizes evaluation methodologies, metrics used for the protocol evaluations, and design options associated with such investigations.

2.3.1 Snoop-based Cache Coherence

Goodman [57] used six traces from PDP-11 and VAX machines under Unix to conduct experiments on the write-once protocol. This work reported the correlation of bus traffic and miss ratio, and discussed the cache block size effect. Katz [66] also utilized traces to evaluate the Berkeley protocol and reported the bus traffic reduction over the write-once protocol. Papamarcos and Patel [88], on the other hand, used an analytical model to demonstrate the effectiveness of the Illinois protocol. Firefly [107] implemented by Digital Equipment Corporation (DEC) was again evaluated using trace-driven simulation to assure its effectiveness.

In the late 1980s, several researchers [26, 50, 49, 51] evaluated coherence protocols under the same environment. Archibald [26] used the probability function to

generate memory references and compared the write-once, Synapse, Berkeley, and Illinois protocols with the write-through policy. This work reported the results using a metric called *system power*, which is the sum of the processor utilization in the system. Eggers and Katz used trace-driven simulations to characterize sharing patterns and their effect on cache coherence performance [49, 51]. They also evaluated the impact of block size in the Berkeley write-invalidation protocol and the impact of cache size in the Firefly write-update protocol [50]. Using a trace-driven simulation of SPLASH [97] benchmarks, Dubois et al. [48] coined the term *false sharing*, representing useless misses in a shared memory multiprocessor. Using execution-driven simulation with the SPLASH2 benchmarks, Culler et al. [44] extensively explored the design trade-offs in protocols by analyzing the bus bandwidth requirement according to the state transitions and the cache block size. They also classified cache misses in detail according to the benchmarks and block sizes.

2.3.2 Directory-based Cache Coherence

Chaiken et al. [39] evaluated memory-based schemes (full-map directory and limited directory) and cache-based schemes (chained directory) using trace-driven simulations with address traces from a variety of parallel applications. In this study, the processor utilization, in which computation is based on an analytical model [90], is used as an evaluation metric. LimitLESS [40] implemented in MIT Alewife [22] was evaluated using both execution-driven simulation and trace-driven simulation to report the execution cycle times of applications. Agarwal et al. [21] used the trace-based simulation to evaluate directory schemes, where the communication cost per memory reference is used as a metric. O'Krafka and Newton's evaluation in [85] is based on the execution-driven simulation using ssim, genie, and verf benchmarks, where the average communication requirements and the average memory access time were used

as evaluation metrics. Using the execution-driven simulation with SPLASH2 benchmarks, Culler et al. [44] examined the *full-bitvector directory* protocol, where the experiment showed the number of network traffic patterns according to the number of processors. They also reported the cache block size impact on network traffic with a 32-processor configuration. With a 64-processor configuration, their experiment also explored the data-sharing patterns by observing the invalidation size (the number of active sharers) upon each invalidating write.

2.4 Emulation Initiatives for Evaluation

To overcome the inherent limitation of the software-based simulations, several researches devoted their efforts to emulations using field-programmable gate array (FPGA). Compared to the software simulation, the emulation achieves several orders of magnitude speedup in evaluation. Hence, it is possible to perform the broad-range analysis of the target systems' behavior.

2.4.1 RPM

Rapid prototyping engine for multiprocessor (RPM) [47] is an emulator developed at University of Southern California in the mid-to-late '90s. The major objective of the RPM project is to develop a common, configurable hardware platform to emulate the different models of multiple instructions multiple data streams (MIMD) systems with up to eight execution processors. The RPM platform emulates *ccNUMA* architectures under strong ordering of shared-memory accesses. It is organized with nine identical boards connected through Futurebus+. Each board contains a CPU, L1 and L2 caches, main memory, and several FPGAs. The Sparc IU/FPU is used as CPU, and L1/L2 caches were implemented with SRAM and FPGAs. To emulate variable interconnection delays, the delay unit was implemented using an FPGA and sits between the Futurebus+ and the network interface. The RPM platform has inherent limitations due to its infrastructure. For example, it cannot prototype systems with

more than eight execution processors and/or with more than two levels of caches. The RPM cannot implement various interconnection topologies due to the connection of the nodes through Futurebus+ and the processor architecture is fixed. In spite of these limitations, RPM is much more flexible than traditional hardware prototypes and can help explore a large number of practical multiprocessor architectures.

2.4.2 MemorIES

Memory instrumentation and emulation system (MemorIES) [83] is an emulator developed by IBM T.J. Watson Research Center in 2000, for evaluating large caches and SMP cache coherence protocols for their future server systems. The emulation board can be directly plugged into the 6xx bus of an RS/6000 SMP server, in which each processor has L1 and L2 caches. L3 caches can be designed in the FPGAs on the MemorIES board with various configurations and protocols. Then, by passively monitoring 6xx bus transactions, MemorIES is able to perform on-line emulation of the cache behavior while the system is running commercial workloads without slowing down the execution speed of the applications. However, since MemorIES is a passive emulator, one cannot inject transactions on the bus. This nature limits the usage of MemorIES. For example, when emulating an L3 cache and/or an L3-level coherence protocol, MemorIES cannot perform invalidations of cache lines in L1 and L2 caches because it cannot inject invalidation traffic on the bus. In other words, MemorIES cannot emulate a fully-inclusive cache. In addition, since the latency information of the L3 hit and miss is not reflected in the emulation, the emulated cache's behavior could be slightly different from the one of a real L3 cache. Nevertheless, MemorIES complements simulation techniques by providing fast results for a wide design space.

2.4.3 Other Cache Emulators

Reconfigurable address collector and flying cache simulator (RACFCS) [120] is a cache emulator developed at Yonsei University in 1997. Using an Intel 486 system, the latch

board is directly connected to the pins of the microprocessor. At the same time when the board is collecting traces through the connected pins, the RACFCS performs an on-line cache emulation. Like MemorIES, it is a passive emulator and also has the same limitations. That is, RACFCS cannot emulate a fully-inclusive cache and cannot perform the latency studies.

Hardware accelerated cache simulator (HACS) [109] is an L3 cache emulator developed at Brigham Young University in 2002. In terms of functionality, HACS is exactly same as RACFCS. The difference is that HACS uses a more advanced system equipped with a Pentium®-Pro processor. By passively monitoring bus transactions, HACS performs an L3 cache emulation while running SPECint2000 natively.

Active cache emulator (ACE) [60] is also an L3 cache emulator developed by Intel in 2005. Based on a dual Pentium®-III processor server system, ACE replaces one of the two Pentium®-III processors with FPGA for cache emulation. The Pentium®-III processor has an L1 and an L2 inside, and the ACE emulates an L3 cache by monitoring bus transactions on front-side bus (FSB). The major difference from other cache emulators (MemorIES, HACS, and RACFCS) is that ACE is an active emulator whereas the rest is passive ones. The ACE is able to inject delays on the bus by the snoop stall protocols of FSB. Thus, it is able to perform a real L3 cache emulation while running benchmarks natively. Sitting on the FSB, ACE keeps track of the memory transactions and stores appropriate TAG information from memory transactions on the FSB for the given emulated cache size. If a memory transaction misses the L3 TAG stored in the FPGA, a default L3 miss latency is inserted onto the FSB based on the snoop stall protocol. If a hit is detected, zero or a default hit latency is inserted in the same way. The ACE enables the L3 cache behavior modeling and analysis for commercial workloads while natively running workloads. However, there is one limitation in the ACE. It cannot emulate a complete non-blocking L3 cache since the FSB does not allow for out-of-order completion of bus transactions.

CHAPTER III

SOC INTEGRATION AND COMMUNICATION ARCHITECTURES

According to the 2005 International Technology Roadmap for Semiconductors (ITRS) [8] report, a system-on-a-chip (SoC) closely resembles an application specific integrated circuit (ASIC), and is evolved most directly from an ASIC with the principal goals of design cost reduction and higher level system integration. The primary distinction between an ASIC and a SoC is that the SoC design advocates the concept of reusing existing blocks or cores to minimize the need and effort of creating new circuits blocks. The reusable intellectual property (IP) components in an SoC include high-volume custom cores or blocks, analog or digital, as well as software modules. Such a design methodology can improve design productivity, reduce verification and validation effort, and accelerate time-to-market. Nevertheless, it demands the integration of heterogeneous components designed potentially even by incompatible process technologies. Since heterogeneous components such as microprocessor cores, DSPs, memory, and various peripherals are integrated in one single chip, the major challenges in the SoC design are compatibility and communication issues among heterogeneous IPs. Reflecting its importance, several companies, including ARM, IBM, Sonics, and PALMCHIP, proposed proprietary or open-source communication architectures, and research institutes such as TIMA Lab [17] are actively investigating compatibility and efficient ease-of-integration issues. Furthermore, international consortia such as VSI Alliance [20] and OCP-IP [12] are developing common standards for IP core interfaces to facilitate "plug and play" SoC design practices. The purpose of these protocols aims at providing an efficient communication design methodology

for integrating heterogeneous IPs.

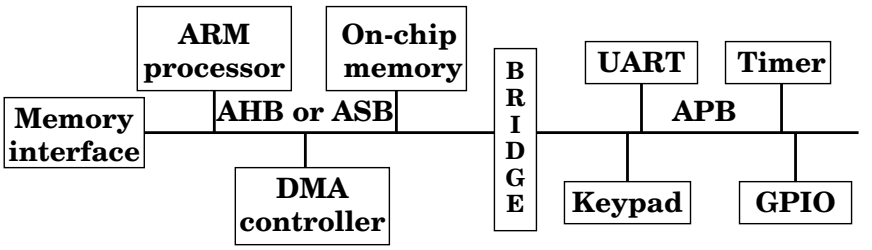
This chapter summarizes these communication architectures and interface protocols by major hardware vendors. It begins by introducing the SoC communication architectures in Section 3.1. Section 3.2 discusses the compatibility issues by presenting SoC interface standards, and Section 3.3 covers multiprocessor SoC architectures and design methodologies.

3.1 SoC Communication Architecture

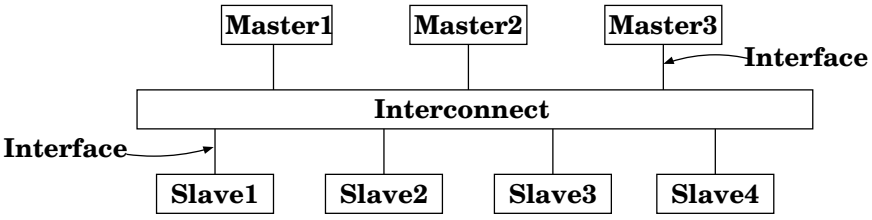
SoCs are mainly designed and deployed for embedded systems. Due to the nature of many embedded applications, these systems typically have real-time requirements. Thus, a small real-time kernel, often referred to as a real-time operation system (RTOS), is generally used to manage and schedule applications' tasks. While an RTOS is used to guarantee a real-time response on the software side, the underlying layer (i.e., the hardware) should also provide sufficient computing power and efficient communication mechanism. Especially, as heterogeneous components are integrated in SoCs, it is imperative to provide an efficient communication mechanism. This section presents the communication architecture proposed by major hardware vendors.

3.1.1 AMBA

Advanced Microcontroller Bus Architecture (AMBA) [27] from ARM is an open standard, on-chip communication specification, which has been evolved since the mid-90s. Figure 5 shows the current AMBA communication architecture, which includes AMBA 2.0 and AMBA 3 Advanced eXtensible Interface (AXI). AMBA 2.0 is based on the bus architecture as depicted in Figure 5(a). It consists of two levels of bus hierarchy: Advanced High-performance Bus (AHB) or Advanced System Bus (ASB) for high-performance communication, and Advanced Peripheral Bus (APB) for slow peripheral devices. The pipelined AHB connects embedded processors (e.g. an ARM core) to high-performance peripherals such as DMA controller, on-chip memory, and



(a) AMBA 2.0



(b) AMBA 3 AXI

Figure 5: AMBA communication architecture.

memory interface. The ASB is an old version of the pipelined system bus and is being replaced by AHB. APB is mainly used to connect slow peripheral devices such as UART, timer, keyboard/mouse controller, etc. APB is accessed by bus masters in AHB or ASB via a bus bridge depicted in Figure 5(a). Unlike ASB, AHB provides separate bus channels for read and write, eliminating the bus load (capacitance) problem incurred by the tri-state based shared bus. AHB also allows for the split and retry transactions, which can be used by slave devices to issue a *SPLIT* or *RETRY* response if a slave is unable to supply data immediately for a transfer. This mechanism improves the overall utilization of the bus. For arbitration, AHB makes use of the priority-based mechanism.

The AMBA 3 AXI protocol, of which the concept is based on *interconnect* and *interface*, provides more advanced features — including unaligned data transfer, multiple outstanding transactions, and out-of-order completion. A typical system consists of a number of masters and slave devices connected together through some form of interconnect as shown in Figure 5(b). The AMBA 3 AXI provides a single interface

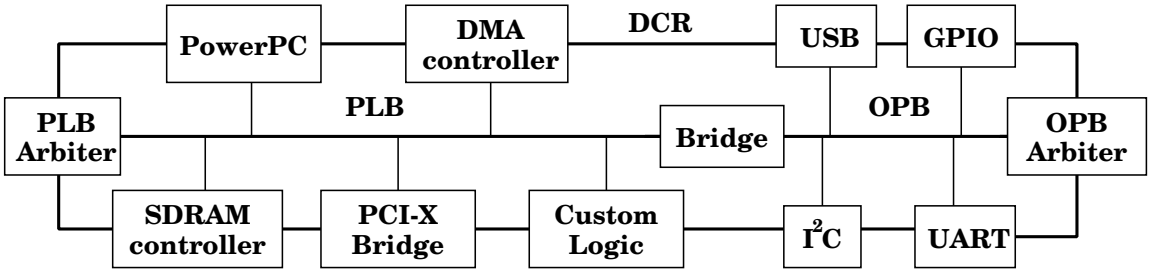


Figure 6: CoreConnect-based SoC architecture.

definition for describing interfaces: between a master and the interconnect, between a slave and the interconnect, and between a master and a slave. This interface definition enables a variety of different interconnect topologies. Nevertheless, for most of the systems, the channel bandwidth requirement of addresses is significantly less than that of data. Such systems can achieve a good balance between system performance and interconnect complexity by using a shared address bus with multiple data buses to enable parallel data transfers.

3.1.2 CoreConnect

CoreConnect [61] from IBM shares many similar characteristics with the AMBA architecture. It is an open standard and consists of three buses for interconnecting IPs: Processor Local Bus (PLB), On-chip Peripheral Bus (OPB), and Device Control Register bus (DCR), as shown in Figure 6. The pipelined PLB, which corresponds to the AHB in AMBA, interconnects high-bandwidth devices such as processor cores, external memory interfaces, and DMA controllers. It supports separate read and write buses and allows split transactions. The OPB, which corresponds to APB in AMBA, is a secondary bus designed to alleviate system performance bottlenecks by reducing capacitive loading on the PLB. Similar to the APB, low-bandwidth devices such as UART, Timer, and I^2C reside on OPB. The slight difference is that the OPB supports multiple masters whereas the APB supports only one master (the APB bridge). CoreConnect provides a separate bus (DCR) for configuring and checking

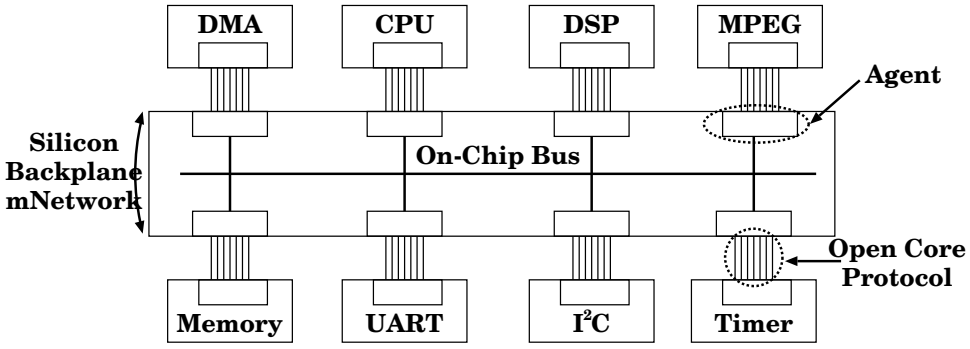


Figure 7: SiliconBackplane μNetwork SoC architecture.

registers of on-chip modules. The DCR lessens PLB and OPB traffic. For arbitration, PLB relies on the priority-based scheme.

3.1.3 SiliconBackplane μNetwork

Figure 7 shows SiliconBackplane μNetwork [98] from Sonics. Each IP in a system communicates with an attached agent via ports implementing OCP [12], and the agents communicate with each other using a network that implements the SiliconBackplane protocol. The SiliconBackplane μNetwork makes use of agents to decouple the performance of the communication network from the individual IP cores, enabling the cores to be designed independently. The key concept governing the Sonics architecture is the combination of a fully pipelined, fixed-latency bus and a time-division multiple access (TDMA)-based bandwidth allocation scheme into a single communication protocol. The pipeline is quite common in other bus architectures such as AMBA and CoreConnect. For the fixed latency, the architecture allows for the retry mechanism, which is also a feature in AMBA and CoreConnect. However, unlike AMBA and CoreConnect in which arbitration is based on assigned priorities to bus masters, SiliconBackplane relies on TDMA-based arbitration. It is also noticeable that only one common bus accommodates all necessary IPs in a system. Therefore, the usable bandwidth must satisfy the aggregated simultaneous bandwidth required

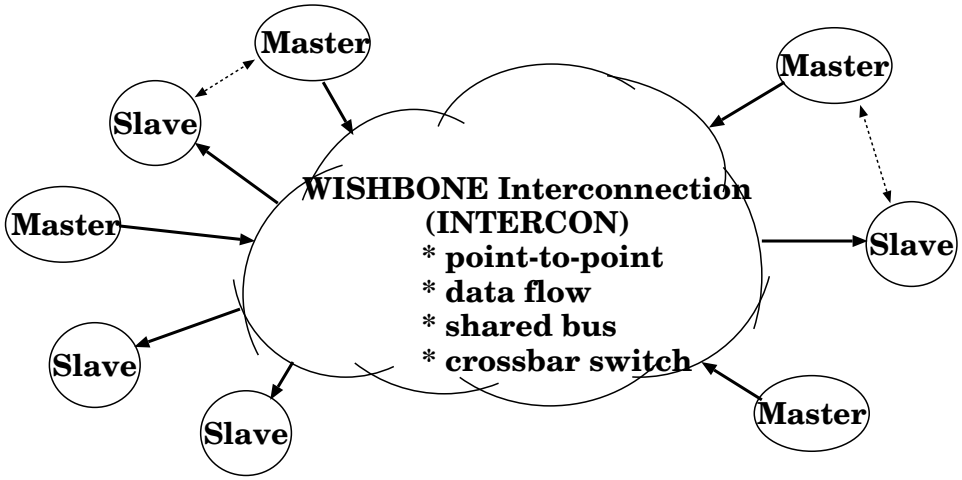


Figure 8: WISHBONE interconnection.

by all integrated IP cores. SiliconBackplane μNetwork also features the dynamic modification of many system parameters as the system requirements change. To avoid a tri-state bus, it uses a variant of a multiplexed bus topology where the signal to be driven on a bus is qualified (through an AND gate) by the enable signal, and the resulting qualified signals are OR-ed together.

3.1.4 WISHBONE

WISHBONE [15] is an open standard from OpenCores [13]. Figure 8 shows its interconnection model for connecting IPs. As shown, the masters and slaves communicate through an interconnection interface referred to as *INTERCON*. The WISHBONE interconnection allows the system integrator to change the way that IP cores connect to each other. The AMBA 3 AXI has the same feature. There are four types of interconnections in *INTERCON*: point-to-point, data flow, shared bus, and crossbar switch. The point-to-point interconnection is the simplest way to connect two IP cores together. The data flow interconnection is used when data is processed in a sequential and pipelined manner. The shared bus, like other bus architectures, provides a way to connect masters with slaves using a common medium. The crossbar switch interconnection is useful when multiple masters need to use the interconnection at the

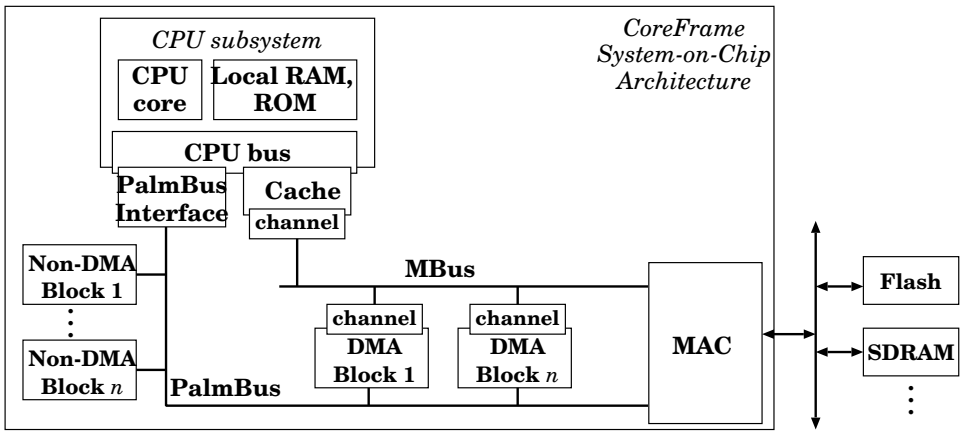


Figure 9: CoreFrame-based SoC architecture.

same time. The IP cores' interfaces conforming the WISHBONE protocol have "in" and "out" ports for address, data, and other control interface signals. This allows the interface to be adapted to the multiplexed or tri-stated interconnection. Even though multiplexed interconnection requires a large number of routing spaces and logic gates, it is a more portable design choice since using tri-state logic is inherently slower and is not friendly to EDA tools. Finally, the arbitration mechanism in WISHBONE is left to be defined by end users.

3.1.5 CoreFrame

CoreFrame [43, 87] from PALMCHIP is not significantly different from AMBA and CoreConnect. Its architecture is based on MBus and PalmBus, as shown in Figure 9. MBus is the interface between the memory access controller (MAC) and the memory channels (DMA channels). PalmBus is the interface for communications between the CPU and peripheral blocks and is not used to access memory. PalmBus is accessed via the PalmBus interface block, which is similar to the bridges in AMBA and CoreConnect. CoreFrame is optimized for devices requiring extensive DMA with high bandwidth data streams. It supports point-to-point signals instead of shared tri-state buses. CoreFrame defines a processor-independent architecture, whereas AMBA and CoreConnect are most likely to be used with their own processors such as ARM

and PowerPC unless a wrapper [118] is adopted. CoreFrame opens the arbitration mechanism to be application-specific.

3.1.6 LOTTERYBUS

LOTTERYBUS [69] is an arbitration mechanism for efficient and fair communication. The development of LOTTERYBUS is motivated by two observations: First, the static priority-based arbitration does not provide a means of controlling the fraction of communication bandwidth assigned to each component, possibly resulting in starvation for lower priority components in some situations. Second, TDMA-based arbitration could lead to significant latency increase resulting from variations in the time profiles of communication requests, sometimes high-priority communication incurs longer latency. To provide efficiency and fairness in communication, Lahiri proposed an arbitration scheme called LOTTERYBUS. Its mechanism heavily relies on a uniformly distributed random number generator, which can be generated using a linear feedback shift register. Using the notation in [69], let the bus masters in bus-based system be C_1, C_2, ... C_n, the number of tickets held by each master be t_1, t_2, ... t_n, and at any bus cycle let the set of pending requests be represented by a set of Boolean variables r_i *(i=1,2,...n)*, where r_i*=1* if the master C_i has a pending request, and r_i*=0* otherwise. Then, the master to be granted is chosen depending on the generated random number, by comparing it with the probability of granting master C_i given by Eq 1. The experimental results show that LOTTERYBUS offers low access latencies and effective bandwidth guarantees for each system component.

$$P(C_i) = \frac{r_i \cdot t_i}{\sum_{j=1}^{n} r_j \cdot t_j} \tag{1}$$

3.1.7 Discussion

Table 3 summarizes the SoC communication architectures we discussed. LOTTERYBUS is not included in the table since it is only used for the arbitration mechanism. In

general, the basic communication architecture is not significantly different from each other. It is reasonable to say that the current dominant SoC communication architectures are mostly based on shared bus. However, as embedded applications demand more computing power and their datapaths become diverse, communication architectures tend to leave more freedom to end-users for the choice of communication architecture. For example, AMBA 3 AXI and WISHBONE allow users to configure the interconnection network such as multi-level bus, point-to-point network, and crossbar switch-based architectures.

In SoC designs, a predictable data access latency is also important to meet the real-time constraint of embedded applications. To satisfy this demand at the hardware level, new standards employ more aggressive and advanced communication mechanisms. For example, the AMBA 3 AXI adopts the split transaction and out-of-order completion of the bus transactions. CoreConnect and SiliconBackplane μNetwork also provide the split transaction to prevent slow devices from holding the shared medium for a long time.

From an implementation standpoint, one noticeable change in design practices compared to those in the 90s is that the shared bus is implemented with a variant of multiplexors instead of tri-state buffers. In the past, when silicon real estate was precious, the bus was implemented using tri-state buffers since the same physical line could be shared among multiple agents. However, the disadvantages of using the tri-state buffer are that it is inherently slow because of turn-around time and it is not friendly to the EDA tools. Nowadays, as process technology advances, designers have more luxury to explore silicon real estate for complex designs. For example, the evolution from 0.5μm to 0.18μm technology frees 88% of the chip space [45]. Moreover, the system requires faster clock frequency and the time-to-market requirement becomes more stringent. A tri-state-based design could be problematic in meeting those conditions, so a multiplexor-based unidirectional bus becomes more prevalent.

Table 3: Characteristics of SoC communication architectures.

Company	Communication Protocol	Topology	Interconnection Type	Arbitration Mechanism	Latency-reducing Mechanism
ARM	**AMBA 2.0**	Hierarchical bus -AHB(ASB),APB Pipelined bus	Multiplexor-based bus (ASB: tri-state based)	Priority-based	Split transaction
	AMBA 3 AXI	Configurable - shared bus - multilayer bus	Multiplexor-based	N/A*	Split transaction, Out-of-order completion
IBM	**CoreConnect**	Hierarchical bus -PLB,OPB,DCR Pipelined bus	Multiplexor-based bus	Priority-based	Split transaction
Sonics	**SiliconBackplane μNetwork**	Single bus Pipelined bus	Multiplexor-based bus	TDMA-based	Retry
OpenCores	**WISHBONE**	Configurable - point-to-point - shared bus - crossbar	Configurable - tri-state based - multiplexor-based	Configurable - priority-based - round robin	None
Palmchip	**CoreFrame**	Hierarchical bus	Multiplexor-based bus	Application-specific	N/A*

** It is not described in documents.*

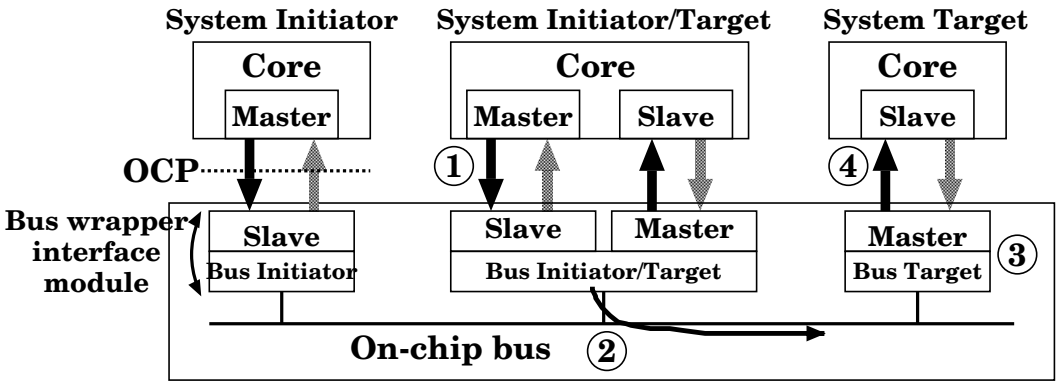


Figure 10: A system with a wrapped bus and OCP instances.

It also possibly shortens the time-to-market span because it conforms well to EDA tools. However, the advantages come at the expense of silicon area.

3.2 SoC Interface Protocols

As mentioned, an SoC system is characterized by the heavy reuse of pre-built IPs, and it requires the mix and match integration of heterogeneous IPs on a single chip. In achieving this goal, there are two conflicting camps [116]. Companies such as ARM, IBM, Sonics, and Palmchip advocate their own communication protocols as potential standards. On the other hand, international consortia such as OCP-IP and VSI Alliance argue for using standardized communication links. In general, companies that promote their own architectures claim that a standard protocol incurs performance and area overhead, whereas the advocates of the standard communication protocol believe that no single bus protocol can address the needs of all SoC integration. This section summarizes two standard interface protocols proposed by OCP-IP and VSI Alliance.

3.2.1 OCP

The open core protocol (OCP) [12] defines a high-performance, bus-independent interface for IPs. It defines a point-to-point interface between two communicating entities such as IP cores and bus interface modules (bus wrappers). Figure 10 shows

a simple system containing a bus wrapper and three IPs: a system target, a system initiator, and an entity that acts as both. The on-chip bus in Figure 10 could be any bus protocol such as AMBA, CoreConnect, SiliconBackplane μNetwork, WISHBONE, or CoreFrame.

The OCP supports not only basic data flow for interoperability such as simple interactions between master and slave, but also complex protocols such as out-of-order completion. A transfer across this system occurs as follows. A system initiator (OCP master) presents command, control, and possibly data to its connected slave (a bus wrapper interface module), as indicated in ① of Figure 10. The interface module sends the request across the on-chip bus system (②). Since OCP supports a bus-independent interface, the interface designer should convert the OCP request into a native bus transfer. The bus wrapper interface module on the receiver side then converts the bus transfer to a legal OCP command (③). The system target (OCP slave) receives the command and takes the requested action (④).

The OCP is flexible. Each instance of the OCP is configured based on the requirements of the connected entities. For example, system initiators may require more address bits than the system target. There are some models of how existing IPs communicate with one another. Some employ pipelining to improve bandwidth and latencies. Others use multiple-cycle access models to simplify timing analysis and reduce implementation area. In order for an IP to be OCP-compliant, the IP must include at least one OCP interface and comply with all aspects of the OCP specification.

3.2.2 VCI

Virtual socket interface alliance (VSIA) [20] and its on-chip bus (OCB) development working group also define a standard communication protocol called virtual component interface (VCI) [19, 45]. Figure 11 shows the VCI components associated

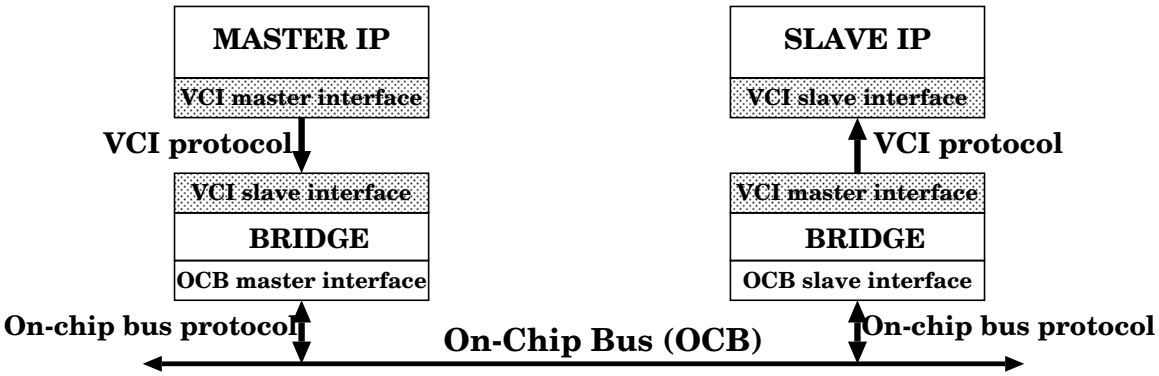


Figure 11: Integration of IPs using the VCI protocol.

with the integration of IPs using the VCI protocol. The basic structure of VCI is not much different from OCP. The OCB in Figure 11 could be any bus architecture presented in Section 3.1. The VCI protocol can be defined as a generic cycle-based, memory-mapped, point-to-point communication protocol. A master sends a request to a slave, and the slave returns a response. The basic VCI (BVCI) defines the split protocol through two handshake protocols. That is, the timing of the request and the response are fully separated. However, it does not allow out-of-order completion. The advanced VCI (AVCI) protocol offers more flexibility such as switching the order of packets. In order for an IP to be VCI-compliant, a bridge shown in Figure 11 should be available by the OCB IP provider to adapt communication between the IP and OCB.

3.2.3 Discussion

OCP and VCI are quite similar in capability and in their design concept. However, the OCP is considered a superset of the VCI [12]. While VCI focuses on data flow, OCP additionally handles control and test flows. Even though AVCI provides some advanced features, OCP 2.0 is a functional superset of all essential VCI features. The basic concept of OCP and VCI is to use a "socket" to bridge heterogeneous IPs to on-chip bus protocols such as AMBA and CoreConnect. To take advantage of this concept, IP providers should supply OCP or VCI-compliant IPs, so SoC designers

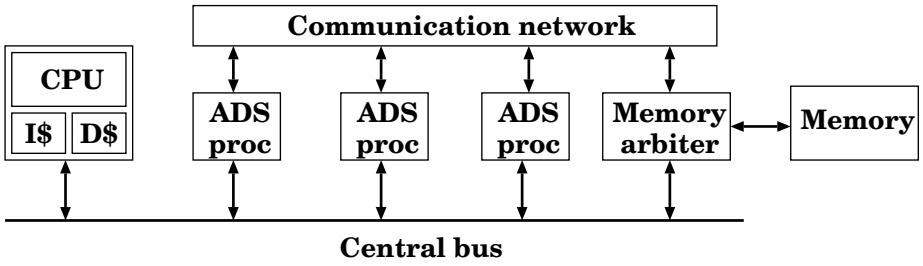
seamlessly exercise "plug and play" design practices.

The manual adaption of heterogeneous IPs to standard protocols is often a labor-intensive and time-consuming process. Some papers [46, 91, 89] proposed using algorithmic approaches to automate the interface design process. In particular, D'silva in [46] experimented with their algorithm to automatically synthesize interfaces for SoCs combined with AMBA (AHB, ASB), CoreConnect (PLB), and OCP. Their experiments focused only on read behavior.

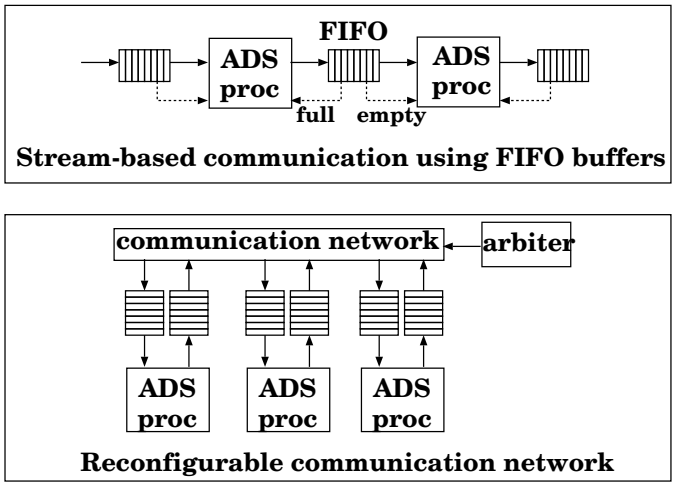
3.3 MPSoC Architecture & Methodology

Today's SoCs demand the computing power and flexibility to cope with evolving applications, leading to the integration of multiple embedded processors (homogeneous or heterogeneous) in a single chip, which is often referred to as multiprocessor system-on-chip (MPSoC). In MPSoCs, processors are chosen based on applications' demands. For instance, when an application demands MPEG and audio processing with a network support, one general embedded processor or single digital signal processor (DSP) can be insufficient in providing the computing power needed. Under this circumstance, an SoC designer would choose DSPs for video and audio processing and embedded processors for the network and housekeeping work. Such MPSoCs create new design challenges for integration and communication among heterogeneous processors.

PROPHID [71] is a design method aimed at high-performance systems with a focus on high-throughput signal processing for multimedia applications. It uses the heterogeneous MPSoC architecture template, as shown in Figure 12(a). Its development was motivated by the fact that multimedia applications require a large number of tasks on a variety of multimedia data, and many tasks such as audio processing often necessitate the support of different standards in a wide range. PROPHID exploits coarse-grain task-level parallelism, which can be extracted by drawing block



(a) Architecture template



(b) Communication networks

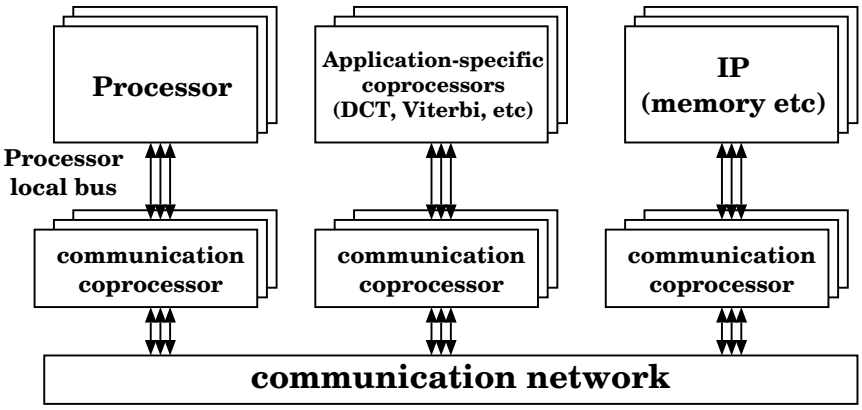
Figure 12: PROPHID heterogeneous high-performance multiprocessor architecture.

diagrams to specify applications. It enables the parallel execution of large independent tasks. In Figure 12(a), the CPU executes low-performance tasks that require a lot of programmability, while the other processors, referred to as *application domain specific (ADS)* processors, execute high-performance tasks that require only a limited amount of programmability. ADS processors are optimized in terms of speed, area, and power, and tuned towards a well-defined set of tasks. To greatly reduce explicit communication overhead for task synchronization, PROPHID employs two communication channels as shown in Figure 12(a): Central bus (shared bus) and Communication network. The central bus, where CPU and ADS processors are attached, is used for control-intensive tasks, while communication network is used for

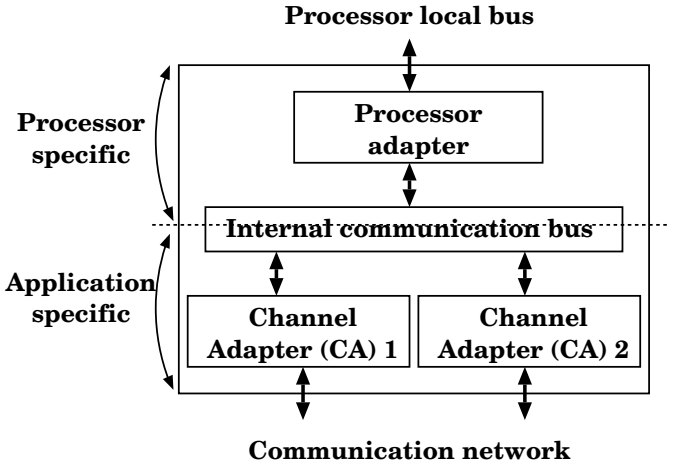
high-throughput communication among ADS processors. Figure 12(b) illustrates two communication network configurations. Taking advantage of stream-based data processing in multimedia applications, the first communication network shown at the top of Figure 12(b) is implemented using first-in first-out (FIFO) between ADS processors. FIFO enables the implicit synchronization among ADS processors by using FIFO *full* and *empty* signals. The second communication network shown at the bottom of Figure 12(b) allows the reconfigurability of the network. The reconfigurable network can range from a single bus or multiple buses to a partially or fully connected switch matrix. It allows flexibility in assigning tasks on ADS processors according to different task graphs, after the MPSoC is fabricated.

TIMA laboratory [17] in France is actively investigating the MPSoC design methodology [118, 75, 29, 119, 54, 56, 38, 117]. Lyonnard [75] *et al.* presented an approach to automatically generate application-specific architectures for heterogeneous MPSoCs. It uses the generic MPSoC architecture template [29] shown in Figure 13(a). The communication interface, referred to as *communication coprocessor*, is composed of two parts, illustrated in Figure 13(b): one specific to the processor and the other a generic depending on the number of communication channels and communication protocols used. This decomposition dissociates the CPU from the communication network, providing modularity and flexibility since other kinds of CPUs or DSP cores can be integrated in the same way. The automatic generation starts from the macro architecture specification such as connections among processors using high-level description. From the macro architecture specification, parameters for micro (physical) architecture are extracted. Then, a detailed description of the final micro-architecture is generated from pre-built libraries through a *refinement* process.

Wolf forecast the future of MPSoCs and described hardware and software challenges that MPSoC designers currently face [110]. MPSoCs have already started to enter the marketplace (for example, Intel IXP2855 [7], Philips Nexperia [10], and TI



(a) Architecture template



(b) Communication coprocessor

Figure 13: Generic MPSoC architecture template.

OMAP [11], to name a few) and are expected to be available in even greater variety over the next few years. MPSoCs require real-time and low power operation, making heterogeneous architectures more attractive over the long run. Wolf pointed out that a key area of concentration to handle both the real-time and power problem is the memory system, which needs to be predictable for data accesses. The specialized structure of the memory system can conserve energy, allowing architects to more carefully characterize the behavior of the time-critical parts. Software also plays a critical role in MPSoC design. MPSoCs are often designed to comply with

standards. Thus, a great deal of the software effort is to port the reference implementation of the standards to the platform. Since the reference implementations are written with functionality (not performance) in mind, porting the code requires the use of software analysis tools especially for trace-based analysis. In addition, the designers need to build simulation models that can make use of the traces, and there is still some work to be done to create easy-to-configure, highly accurate simulators for heterogeneous MPSoCs. From the operating system and middleware standpoints, MPSoCs demand their core functions to be implemented in a very small amount of software for performance and memory limitations. The extensive use of middleware will provide advanced features such as Internet access on top of micro-kernels, and some key functions such as interprocessor communications primitives must execute in a very small number of cycles to meet MPSoC design constraints.

CHAPTER IV

CACHE COHERENCE PROTOCOL INTEGRATION ON SHARED-BUS-BASED MPSOCS

In MPSoC designs, it is imperative to integrate processor IPs that meet the performance requirements of embedded applications. Another important design choice is the communication architecture for processor IPs. The communication among processors is inevitable for task synchronization, for example. In this chapter, we consider a shared bus as the communication architecture.

Using coherence protocols in processor IPs and integrating them, we provide the generic solutions for the efficient communication among processors on a shared-bus-based MPSoCs [100, 101, 102]. As discussed in Section 2.1, there are two main categories of snoop-based coherence protocols: invalidation-based protocols and updated-based protocols. In general, invalidation-based strategies are more robust, therefore, most vendors use a variant based on such a strategy as their default protocol [44]. We also focused on them in this contribution.

Figure 14 depicts a simplified heterogeneous MPSoC architecture based on a shared-bus, and Table 4 shows its classification in terms of the processors' cache coherence support. It shows a dual-processor platform. However, the proposed approach can be easily extended to platforms with more than two processors. Supporting coherence in PF1 and PF2 requires special hardware, and there is a limitation in the resulting coherence mechanism. This limitation is discussed in Section 4.5.2. For PF3, the cache coherence can be maintained with the proposed generic methodologies. This chapter mainly focused on PF3.

This contribution begins by studying the incompatibility problems of coherence

Table 4: Classification of MPSoC platforms according to coherence support.

Platform (PF)	Cache coherence hardware inside each processor	
	Processor 0 (P0)	Processor 1 (P1)
PF1	No	No
PF2	Yes (No)	No (Yes)
PF3	Yes	Yes

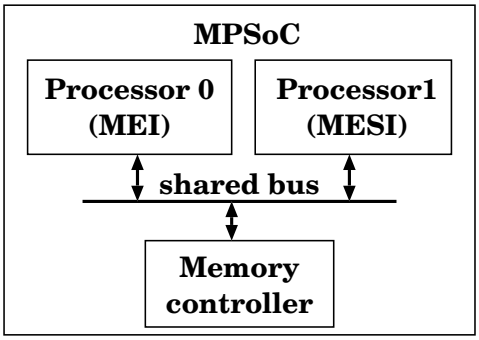


Figure 14: Shared-bus-based MPSoC

protocols when integrating heterogeneous processors in MPSoCs. Then, integration techniques are proposed to resolve incompatibility issues. To enhance the system performance, we propose two architectural features. We also discuss additional hardware and software issues in MPSoCs. To present benefits of the proposed techniques, the simulation platforms and environments are introduced, and simulation results are presented according to benchmarks.

4.1 Motivational Examples

To effectively present the problems, we discuss two examples in Table 5 according to the operation sequence. First, integrating the MEI protocol with others such as MSI or MESI requires the removal of the S state. To illustrate the problem with the S state, the example in Table 5(a) is used assuming that Processor 0 (P0) supports the MEI protocol and Processor (P1) supports the MESI protocol, with the operation

sequence ⓐ, ⓑ, ⓒ, and ⓓ executed for the same block **C**. Operation ⓐ makes the $I \rightarrow E$ state change in P1 as a result of the read. Operation ⓑ incurs the $I \rightarrow E$ transition in P0 and $E \rightarrow S$ transition in P1. Since **C** is in the E state in P0, operation ⓒ does not appear on the bus even though P1 has the same block in the S state. It invokes the $E \rightarrow M$ state transition in P0. However, the state of the block in P1 remains the same. Therefore, operation ⓓ hits on the line of the cache and accesses the stale data, which should have been invalidated during ⓒ.

The second example illustrates the integration of the MSI and MESI protocols, where the E state must be prohibited. Suppose that P0 supports the MSI protocol and P1 supports the MESI protocol, and the operations in Table 5(b) are executed for the same block **C**. Operation ⓐ invokes the $I \rightarrow S$ state change in P0. Operation ⓑ causes the $I \rightarrow E$ transition in P1, while the block's status in P0 remains unchanged because P0 does not assert the shared signal. Note that processors with the MSI protocol do not support the shared signal. Operation ⓒ then makes only the $E \rightarrow M$ transition in P1. As a result, P0 reads the stale data in operation ⓓ because of a cache hit indicated by the S state. Therefore, the E state should not be allowed in this protocol combination.

4.2 Protocol Integration Techniques

As illustrated in Section 4.1, integrating processors with different coherence protocols restricts the use of the entire protocol states. Only the states that the distinct protocols have in common are preservable with a exception of the O state. For example, when integrating two processors with MEI and MESI, the coherence protocol in a system must be MEI. To meet this requirement, we propose two integration techniques – *read-to-write conversion* and *shared-signal assertion/de-assertion*.

Table 5: Incompatibility problems and solutions.

(a) Problem and Solution with MEI and MESI

seq.	Read, Write on a block C	Without Proposed Solution		With Proposed Solution	
		C state in P0 (MEI)	**C** state in P1 (MESI)	**C** state in P0 (MEI)	**C** state in P1 (MESI)
ⓐ	P1 read	**I**	**I → E**	**I**	**I → E**
ⓑ	P0 read	**I → E**	**E → S**	**I → E**	**E → I**
ⓒ	P0 write	**E → M**	**S(Stale)**	**E → M**	**I**
ⓓ	P1 read	**I → E**	**S(Stale)**	**M → I**	**I → E**

(b) Problem and Solution with MSI and MESI

seq.	Read, Write on a block C	Without Proposed Solution		With Proposed Solution	
		C state in P0 (MSI)	**C** state in P1 (MESI)	**C** state in P0 (MSI)	**C** state in P1 (MESI)
ⓐ	P0 read	**I → S**	**I**	**I → S**	**I**
ⓑ	P1 read	**S**	**I → E**	**S**	**I → S**
ⓒ	P1 write	**S(Stale)**	**E → M**	**I**	**S → M**
ⓓ	P0 read	**S(Stale)**	**M**	**I → S**	**M → S**

4.2.1 Read-to-write Conversion

Integrating the MEI protocol with others requires the removal of the S state. The S state can be reached either by the own processor's transaction or by the remote processor's transaction. The own processor's read transaction incurs the $I \rightarrow S$ transition in its cache when the block is present in other processors' caches. This case is discussed in Section 4.2.2. The remote processor's read transaction makes the $E \rightarrow S$, $M \rightarrow S$, or $M \rightarrow O$ transitions, depending on initial states and coherence protocols.

Figure 15 depicts our proposed method to remove the S state incurred by the remote processor's transaction. Because the transition to the S state in this case occurs when the processor observes a read transaction on the bus, the technique to remove the S state is to convert a "read" operation to a "write" operation within the wrappers of snooping processors. However, the memory controller should see

the actual operation to correctly access main memory when it needs to. As such, the S state originated from the remote processor's transaction will be excluded in the controllers' state machines. When a processor observes a write transaction on a cache line in the E, M, or O state, the processor's response could be different depending on implementations. If the line is in a clean state (the E state), the processor invalidates the line. If this line is in a dirty state (the M state or the O state), the processor initiates either cache-to-cache transfer or write-back operation to memory, with invalidating the line.

The last two columns of Table 5(a) illustrate the state transitions with the proposed solution. Operation ⓑ invokes the $E \rightarrow I$ state transition in P1 because P1 observes a write operation on the bus. Operation ⓓ makes the $M \rightarrow I$ transition in P0 because a snoop-hit on the M line causes the state change to I in the MEI protocol.

Implementation cost: As shown in Table 1, a write-miss in a write-back cache initiates a bus transaction, which is generically called a bus-exclusive read (*BusRdX*). In general, the way to generate the *BusRdX* information is different depending on processors and bus protocols. Thus, we use the generic signal, *BusRdX*, for the cost estimation. Implementation requires asserting the *BusRdX* signal to snooping processors within a wrapper even in a memory read transaction by a master processor. We used Synopsys's *Design Compiler* to evaluate our implementation using 0.18μm TSMC library. The synthesized result shows that implementing the read-to-write conversion requires only two gates.

4.2.2 Shared Signal Assertion and De-assertion

As demonstrated in Section 4.1, the E state is not allowed in integrating MSI and MESI protocols. Using the MESI protocol as an example, the $I \rightarrow E$ transition occurs only when a processor initiates a read transaction on the bus, and other processors on the system do not have the same block in their caches. The sharing information

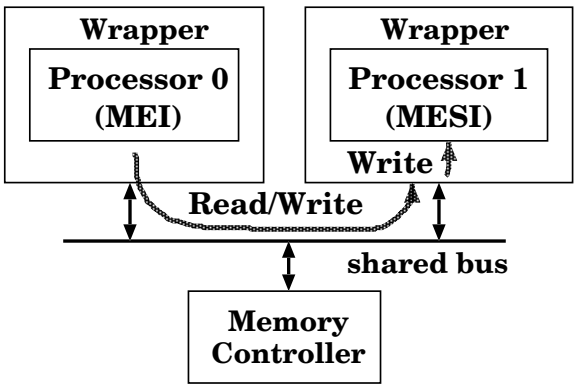
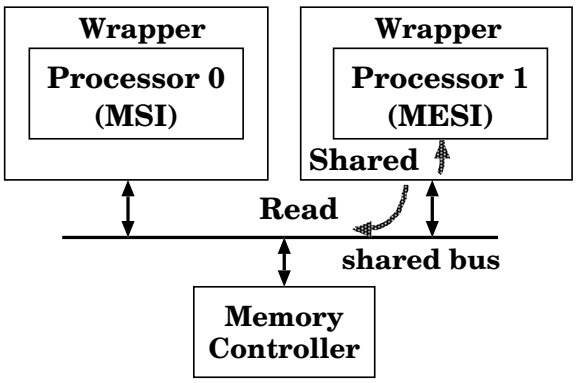


Figure 15: Read-to-write conversion to remove the S state.

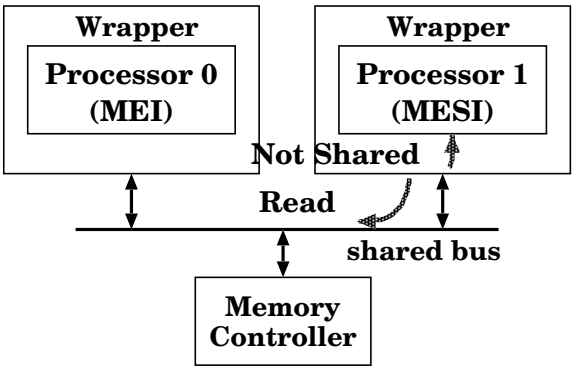
is delivered by the shared signal. Therefore, the technique to remove the E state is to assert the shared signal within the processor's wrapper, whenever a memory read transaction is initiated by its own processor. Like the read-to-write conversion, it can be implemented within the wrappers, as shown in Figure 16(a). With this technique, the operation ⓑ in Table 5(b) invokes the $I \rightarrow S$ state transition in P1, and the operation ⓓ makes the $M \rightarrow S$ transition in P1, resulting in coherent caches.

Shared signal de-assertion is also necessary to remove the S state originated from the own processor's transaction. For example, in the MESI protocol, the $I \rightarrow S$ transition occurs when its own processor initiates a memory read transaction on the bus and the shared signal is asserted in response. Because both the transaction is initiated by its own processor and the state transition occurs in itself, the read-to-write conversion can not be used in this situation. Therefore, to completely remove the S state, the shared signal de-assertion should be used in conjunction with the read-to-write conversion. As shown in Figure 16(b), the wrapper around itself de-asserts the shared signal in response to a memory read transaction to prevent the transition to the S state.

Implementation cost: The implementation of the shared signal assertion requires asserting the shared signal to its own processor whenever a read miss occurs.



(a) Shared signal assertion to remove the E state



(b) Shared signal de-assertion to remove the S state

Figure 16: Shared signal assertion and de-assertion

The synthesis result of our implementation using Verilog-HDL shows that the shared signal assertion requires 1.3 gates. It shows the same result for the shared signal de-assertion.

4.2.3 Detailed Descriptions According to Protocol Combinations

In this subsection, we discuss protocol integrations in detail for four major protocols: MEI, MSI, MESI, and MOESI. The variations include (1) MEI with MSI/ MESI/MOESI, (2) MSI with MESI/MOESI, and (3) MESI with MOESI. In the generic coherence protocols depicted in Figure 2, the cache-to-cache transfer could occur any time when snooping processors find the block in their local caches. In other words, when snooping processors have the requested block in the E, S, M, or

O states, the generic protocols allow the cache-to-cache transfer. However, we limit that the cache-to-cache transfer occurs only from M or O states, as we discussed in Section 2.1.1 and most commercial processors do.

4.2.3.1 MEI with MSI, MESI, or MOESI

The integrations with the MEI protocol do not allow the S state.

MSI Protocol: We employ the read-to-write conversion in this combination. In the MSI protocol, two transitions exist to reach the S state: (a) $I \rightarrow S$ when a local read miss occurs and (b) $M \rightarrow S$ when the processor observes a read transaction on the bus. In case (a), the S state cannot be removed since a local read miss always makes a transaction to the S state in the MSI protocol, whether the other processors share the block or not. However, even though it is in the S state, only one processor owns a specific block at any point in time due to the read-to-write conversion. With the technique, the S state changes to the I state when other processors read from or write to the same block. Therefore, despite the name, the S state is equivalent to the E state. The $M \rightarrow S$ state transition cannot occur because of the read-to-write conversion. Only the $M \rightarrow I$ transition is allowed with the operation conversion. Therefore, the resulting protocol becomes equivalent to MEI.

MESI Protocol: In this combination, we employ the shared signal de-assertion and read-to-write conversion. In the MESI protocol, there are three possible transitions to the S state: (a) $I \rightarrow S$ when a local read miss occurs and the shared signal is asserted in response (b) $E \rightarrow S$ when the processor observes a read transaction on the bus while caching the block in the E state and (c) $M \rightarrow S$ when the processor observes a read transaction on the bus while caching the block in the M state. The shared signal de-assertion prevents the transition (a). The read-to-write conversion prohibits the transitions (b) and (c). Therefore, the S state is completely removed, and the resulting protocol becomes equivalent to MEI.

MOESI Protocol: The same techniques used for the MESI protocol – shared signal de-assertion and read-to-write conversion – are applied to the MOESI protocol except the O state needs to be handled in this combination. The O state can only be reached when the processor observes a read transaction on the bus while caching the block in the M state. Nevertheless, the read-to-write conversion prevents the $M \rightarrow O$ transition. Therefore, the integrated protocol becomes equivalent to MEI.

4.2.3.2 MSI with MESI, or MOESI

The integrations with the MSI protocol do not allow the E state.

MESI Protocol: We employ the shared signal assertion in this combination. In the MESI protocol, there is one transition to the E state: $I \rightarrow E$ when a local read miss occurs and other processors are not caching the block. The shared signal assertion in a wrapper around the master processor prevents the transition to the E state. Therefore, the integrated protocol becomes equivalent to MSI.

MOESI Protocol: The same method – the shared signal assertion – is applied to the integration of MSI and MOESI protocols except the O state should be taken care of. In the MOESI protocol, the O state can be reached only from the M state accompanying cache-to-cache transfer, when the processor observes a read transaction on the bus. The O state has a similar definition to the S state with one exception. The processor with a line in the O state is responsible for updating main memory when the line is replaced. In the generic MESI protocol, main memory is updated simultaneously upon cache-to-cache transfer with the $M \rightarrow S$ transition. However, in the MOESI protocol, main memory is not updated upon cache-to-cache transfer with the $M \rightarrow O$ transition.

To combine these two protocols, the memory controller should be able to differentiate the origin of cache-to-cache transfers. If the cache-to-cache transfer is initiated

from a processor with MESI, the memory controller should update main memory simultaneously. However, if the cache-to-cache transfer starts off from a processor with MOESI, the memory controller does not have to update main memory. Nevertheless, if the memory controller always updates main memory upon cache-to-cache transfer, it still preserves the data consistency. Note that in this case, it could possibly cause the performance loss since the O state becomes the same as the S state and useless updates occur upon the evictions of the lines in the O state. Therefore, if the memory controller selectively updates main memory upon cache-to-cache transfer, this combination becomes equivalent to the MSI protocol with the O state enabled. Otherwise, this combination becomes equivalent to the MSI protocol.

4.2.3.3 MESI with MOESI

This combination does not require the integration techniques and the same discussion made in the MSI and MOESI protocol integration applies to this combination. Due to the O state, if the memory controller selectively updates main memory upon cache-to-cache transfer, this combination preserves all the protocol states. Otherwise, the MESI and MOESI combination becomes equivalent to the MESI protocol.

4.3 Architectural Features for Performance Enhancement

In the followings, we propose two architectural features to enhance the coherence performance of the integration techniques.

4.3.1 Snoop-hit Buffer

According to the generic coherence protocols, when a snoop-hit occurs on a line in the M state, the processor that originally requested the block can get the data through cache-to-cache transfer. Nonetheless, it is sometimes not feasible to do cache-to-cache transfer when integrating heterogeneous processors in MPSoCs for the following reasons. First, although some embedded processors do have coherence protocols, they

sometimes do not support cache-to-cache transfer. For example, the cache-to-cache transfer is not supported in PowerPC750's MEI protocol. Even worse, when integrating processors with no inherent coherence support, it is not feasible to transfer data through cache-to-cache transfer. Second, processors could operate at different operating frequencies because MPSoC designs are typically based on instantiating existing IPs, and pre-built IPs could be designed to operate at various clock frequencies. Moreover, the incompatibility in processors' interfaces could inhibit the cache-to-cache transfer.

In those situations, for each snoop-hit, it is necessary to do back-to-back burst main memory accesses: one for the write-back of the *M* state line and the other for the original request. These two accesses are for the same block. To reduce the memory access latency, we propose a *snoop-hit buffer*. Figure 17 illustrates a system with the snoop-hit buffer. The snoop-hit buffer has a single buffer structure, where one single cache line can be stored. It takes a snoop-hit line during a write-back transaction (① in Figure 17) and supplies the line to the requested processor (② in Figure 17). Main memory is updated simultaneously with the dirty line's buffering into the snoop-hit buffer. This simple additional hardware not only reduces the memory access latency, but also creates an opportunity for the power savings since external address, data, and control pins need not be activated for the second transaction. Once-buffered data are valid until a processor encounters any of the following conditions: the next snoop-hit, a write-miss in the same address, or a dirty line replacement of the buffered block. Since a write-miss appears as *BusRdX* on the bus in a write-back cache and a dirty line replacement appears as a write transaction on the bus, the line in the snoop-hit buffer is invalidated when detecting a snoop-hit or write-invalidation (*BusRdX* or write operation) on the bus. It means that buffered data can be accessed by all subsequent read requests from other processors until the next snoop-hit or a write-invalidation occurs.

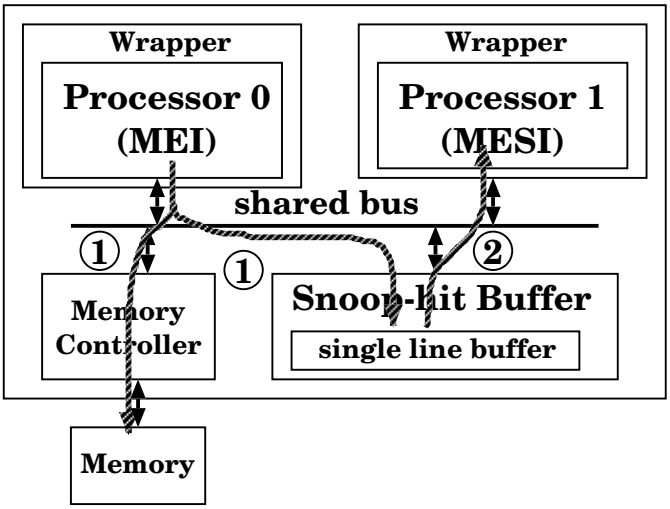


Figure 17: Snoop-hit buffer.

Performance can be further improved by employing a double buffer structure. Similar to the double buffering in a video frame buffer, a double buffer consists of a front buffer and a back buffer for keeping two cache lines. The difference from a single buffer is that main memory is not updated until the next snoop-hit occurs on a line with a different address (*sh_diff*). When the next snoop-hit occurs, the line in the front buffer is copied to the back buffer. Then, the back buffer updates main memory, and the front buffer is used for buffering a line of *sh_diff* simultaneously. The back buffer is invalidated upon finishing updating main memory. However, the front buffer is only invalidated when detecting a write-invalidation in the same address on the bus. A double buffer can remove unnecessary memory-update transactions, which could occur in the single buffer structure, when a snoop-hit is followed by other snoop-hits or write-invalidations in the same address before the *sh_diff*.

Implementation cost: Using Verilog-HDL, our implementation of the snoop-hit buffer consists of a single 32-byte line buffer, a state machine for writing to and reading from the snoop-hit buffer, and a memory-mapped register to enable the snoop-hit buffer. The synthesized result reports 2,987 gates.

Table 6: State transition percentages (SPLASH2 was executed on a 16 processor configuration, and Multiprog was executed on a 8 processor configuration).

	Percentage of the state transitions			
	I	**E**	**S**	**M**
SPLASH2	0.29%	0.76%	19.9%	78.9%
Multiprog kernel (data references)	0.14%	3.31%	30.51%	65.68%

4.3.2 Region-based Cache Coherence

Even though the integration techniques guarantee coherence among heterogeneous processors, there are potential performance losses caused by the lost protocol states (e.g., the *S* state). According to SPLASH and Multiprog simulations, as summarized in Table 6 [44], the *M* state is composed of the majority of the protocol state transitions followed by the *S* state. Note that the *M* state is always preserved in the proposed integration techniques regardless of the protocol combinations. Nevertheless, the *S* state is removed in some cases when different protocols are integrated. For example, it should be eliminated when integrating with the MEI protocol.

The shortcoming of the proposed techniques is that our techniques require processors in a system to use the minimum set of the protocol states. In a situation where SoC applications share data only among processors that have more common protocol states, our techniques become too restricted and prohibit the compatible states. To address this issue, we propose a *region-based cache coherence (RBCC)*.

Given the memory area usages from the applications, the region-based cache coherence permits the disabled states conditionally. Using an MPSoC example with four processors in Figure 18, three processors have the MESI protocol[1] and one processor has the MEI protocol. We assume that the memory area 1 is shared by all four

[1]We implemented a data cache with the MESI protocol using Verilog-HDL and ARM9TMDI core. The data cache has 8KB direct-mapped structure with a 32-byte line size

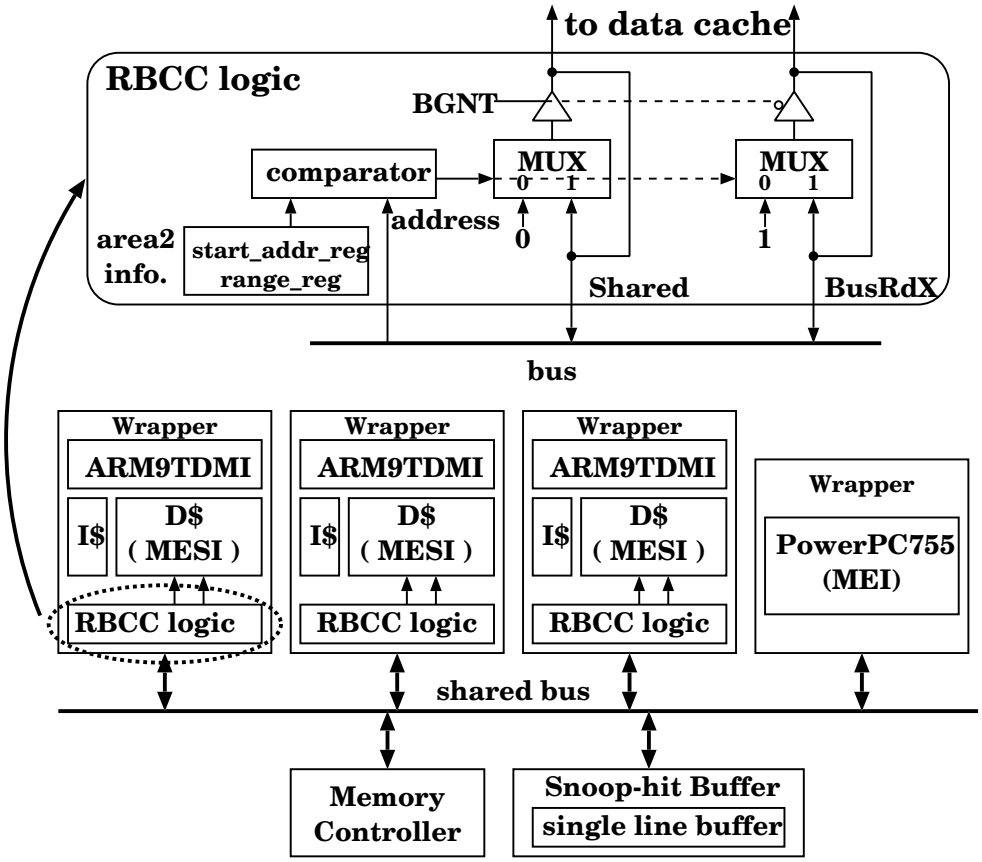


Figure 18: Region-based cache coherence.

processors and memory area 2 is shared by three processors with the MESI protocols. By implementing a comparison logic and comparing addresses generated by CPU, the RBCC logic decides on-the-fly whether to enable the *S* state or not. Using the RBCC technique, the integrated protocol becomes MESI for area 2, whereas the system-wide integrated protocol becomes MEI.

RBCC can be implemented with two memory-mapped registers, one comparator, two multiplexers, and two tri-state buffers inside wrappers. Continuing with the same example, one register (start_addr_reg) is used to keep the start address of area 2, and the other register (range_reg) has the range information as shown in Figure 18. Figure 18 also shows the shared signal and *BusRdX*. The assertion of the shared signal informs a master processor that other processor(s) is(are) caching the same block. As depicted in Figure 18, the shared signal is an input when a processor is

a bus master (that is, when the bus grant (BGNT) is asserted), and it is an output when a processor is not a bus master (that is, when when snooping). The *BusRdX* is used to request an exclusive copy of a block when a write-miss occurs. It is an input when a processor is not a bus master, and an output when a processor is a bus master. If a snoop address detected by the RBCC logic falls in the range of area 2, the RBCC logic bypasses the shared signal and *BusRdX* on the bus to processors through multiplexers, as shown in Figure 18. For area 1, the problem becomes integrating the MEI and MESI protocols. Then, we employ the shared signal de-assertion and read-to-write conversion, as discussed in Section 4.2.3.1. The shared signal de-assertion is employed by selecting "0" (de-assertion) from a multiplexer, and the read-to-write conversion is realized by selecting "1"(assertion) for the *BusRdX*. The RBCC logic in Figure 18 can be extended to include as many area register pairs as needed. It also can be easily extended to other protocol combinations such as MEI/MOESI, MESI/MOESI, etc.

Implementation cost: The implementation requires two memory-mapped registers, one comparator, and two multiplexers for each memory area. We need only two tri-state buffers no matter how many areas we choose to use. Our synthesized result reports 591 gates for one memory area.

4.4 Additional Issues in Heterogeneous MPSoCs

Integrating heterogeneous processors in SoCs incurs additional problems in hardware and software. We discuss these issues in the following subsections.

4.4.1 Synchronization Mechanism for Heterogeneous Processors

In a multiprocessor system, critical sections should be accessed in a mutually exclusive manner. To guarantee this, systems use the lock mechanism, in which processors should access lock variables atomically. Processors designed for supporting multiprocessors or multi-threaded systems inherently provide atomic instructions with

dedicated interface signals. For example, the PowerPC755 features *lwarx* and *stwcx* instructions with the *RSRV* signal, and the ARM processor supports *swp* and *swpb* instructions with the *BLOK* signal.

For these instructions to work correctly and to guarantee atomic accesses, the memory controller should support the corresponding protocol of interface signals. Even though homogeneous platforms can take advantage of these instructions, it would be infeasible to use them in a heterogeneous multiprocessor environment because the behaviors of atomic instructions are inconsistent in different processors. Software solutions such as the Dekker's algorithm [44] and the Bakery algorithm [96] are alternatives in heterogeneous environments. The Dekker's algorithm is a popular software-only mutual exclusion algorithm used in the absence of hardware support for atomic read-modify-write operations, when two processes are competing for a shared resource. The Bekery algorithm overcomes the two-process limitation in a system and provides a mutual exclusion for any number of processes. However, these software algorithms are inefficient from a performance standpoint.

The SoC Lock Cache (SoCLC) [23], a simple yet efficient hardware, would be a more attractive solution for heterogeneous environments. The SoCLC uses only 1-bit lock register for a lock, and sits on a shared-bus like the snoop-hit buffer. It uses general load and store instructions to acquire and release a lock in an atomic fashion, so SoCLC can use the same high-level code regardless of heterogeneity among processors. With the SoCLC, if a processor attempts to access a critical section, it first checks the lock register using a *load* instruction. If the lock is not in use, it returns a "0" to the processor and sets the bit value to "1". Afterward, if other processors attempt to access the critical section, the lock register returns a "1" without changing its value. As such, this mechanism guarantees the atomic access. The processor releases the lock by writing a "0" to the lock register, using a general *store* instruction.

4.4.2 Real-time Operating Systems

Embedded systems, in general, necessitate real-time properties in processing tasks, which requires the use of a real-time operating system (RTOS) in SoCs. Using RTOS simplifies the design process by splitting applications into several tasks. To provide for real-time processing, the RTOS supports multitasking, event-driven and priority-based preemptive scheduling, priority inheritance, and inter-task communications and synchronization. Especially, in heterogeneous multiprocessor platforms, inter-task communication and synchronization will impact system performance since processors' heterogeneity could lead to inefficient shared-memory management

Atalanta RTOS [104] is an embedded RTOS designed at Georgia Tech. For inter-processor communication and synchronization, Atalanta provides both message-passing and shared address space approach, whereas multiprocessor OS kernels such as real-time executive for multiprocessor systems (RTEMS) and operating system embedded (OSE) rely on message-passing. Therefore, in Atalanta, heterogeneous processors can share system objects such as semaphore, mailbox, and queue by using cacheable shared memory, taking advantage of the protocol integration techniques. The shared-memory approach provides much better use of shared memory, thereby increasing performance over that of a message-passing approach [103]. In addition, since mixed systems of RISC processors, DSP processors, and other specialized processors are assumed to be the target architectures, Atalanta's design has been tailored for heterogeneous multiprocessor platforms.

4.4.3 DMA

Many systems incorporate DMA for faster data transfer between I/O modules and memory. In general, memory-mapped I/Os are allocated in non-cacheable memory space. Therefore, DMA should not cause any coherence problem. For some unconventional systems that allow DMA to transfer data between cacheable regions, the

coherence problem can be resolved by allowing DMA controller to concede the bus mastership. Whenever a snoop hit occurs during DMA, the DMA controller yield the bus mastership to the snoop processor and reclaim it after write-back if the corresponding line is dirty. However, this DMA issue is not limited only to heterogeneous platforms, but it also applies to homogeneous platforms.

4.5 Case Studies

Here, we present two implementations using commercially available embedded processors: a PowerPC755, a write-back Enhanced Intel486, and an ARM920T. The PowerPC755 processor uses the MEI protocol, and Intel486 supports a modified MESI protocol. The ARM920T does not offer native support for cache coherence. The examples of this case study focus purely on maintaining cache coherence. Thus, we selected this combination of processors from available Seamless [79] processor models, solely to illustrate how to apply our techniques to the integration of actual heterogeneous processors.

4.5.1 PowerPC755 and Intel486 Integration

As shown in Figure 19(a), the schematic diagram illustrates the integration of a PowerPC755 and an Intel486, representing a case of the PF3. Wrappers are necessary for the protocol conversions between the processors' interfaces and the bus, in addition to the implementation of the proposed techniques (the read-to-write conversion and shared signal assertion/de-assertion). On the PowerPC755 side, the read-to-write conversion and the shared signal de-assertion are not necessary since the S state is not present in the protocol states. On the Intel486 side, however, we should remove the S state. It is done by asserting the INV (invalidation request) input signal, a cache coherency protocol pin. The Intel486 cache controller samples the signal on snoop cycles. If INV is asserted, the cache controller invalidates the addressed cache line if the line is in the E or S state. If a cache line is in the M state, the line is

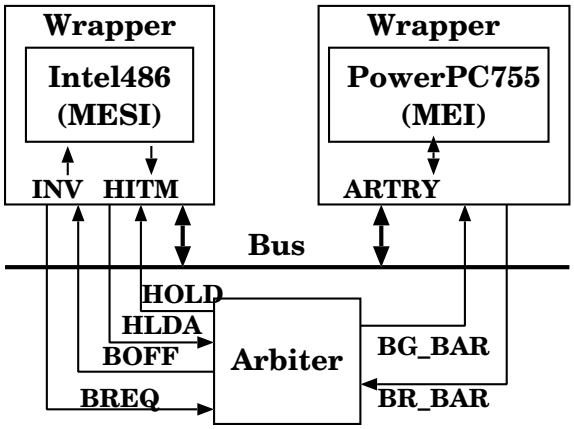
drained to main memory. Normally, *INV* is de-asserted on read snoop cycles and asserted on write snoop cycles. However, to remove the *S* state, the wrapper asserts the *INV* signal regardless of the transaction types.

In the Intel486's cache, cache lines are defined as write-back or write-through at allocation time in enhanced bus mode, depending on the WB/WT (write-back/write-through) pin status. Only write-through lines can have the *S* state, and only write-back lines can have the *E* state. Therefore, the protocol for write-through lines becomes the SI protocol while the protocol for write-back lines becomes the MEI protocol.

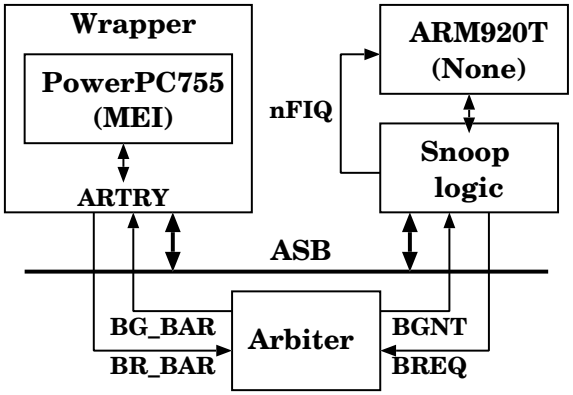
When a snoop hits on a line with the *M* state in the Intel486 cache, the *HITM* (hit to a modified line) output signal is asserted, and correspondingly, the wrapper around the PowerPC755 informs the core of a snoop hit by asserting the *ARTRY* (address retry) input signal. Then, the PowerPC755 immediately yields the bus mastership, so the Intel486 drains the modified line to main memory. When a snoop hits on a line with the *M* state in the PowerPC755's data cache, the PowerPC755 asserts the *ARTRY* output signal. The arbiter then immediately asserts *BOFF* (backoff), so Intel486 yields the bus mastership. Then, the PowerPC755 drains the modified line to main memory.

4.5.2 PowerPC755 and ARM920T Integration

Figure 19(b) shows another example of an MPSoC using a PowerPC755 and an ARM920T representing a case of PF2. The same methodology used in ARM920T is applicable to PF1. The wrapper in the Figure 19(b) converts the PowerPC's interface protocol to the ASB protocol, and vice versa. The wrapper also allows the PowerPC755 to monitor the bus transactions generated by the ARM920T. The snoop



(a) PowerPC755 and Intel486 platform



(b) PowerPC755 and ARM920T platform

Figure 19: Case study platforms for shared-bus-based MPSoCs.

logic [2] provides snooping capability for the ARM920T, which does not have any native cache coherence support. It keeps track of all the address tags of the ARM920T's data cache in a TAG content addressable memory (or TAG CAM) by monitoring bus transactions initiated by the ARM920T. When the tag of a requested address generated by the PowerPC755 matches an entry of the TAG CAM, it triggers a snoop hit to the ARM920T by asserting *nFIQ* (fast interrupt). Then, an interrupt service routine is responsible for draining the snoop-hit cache line if the line is modified or invalidating it if the line is clean.

[2]Our preliminary synthesized design using 0.18μm TSMC library shows that the snoop logic occupies 11.18% of full-custom ARM920T area, supporting 16MB shared memory

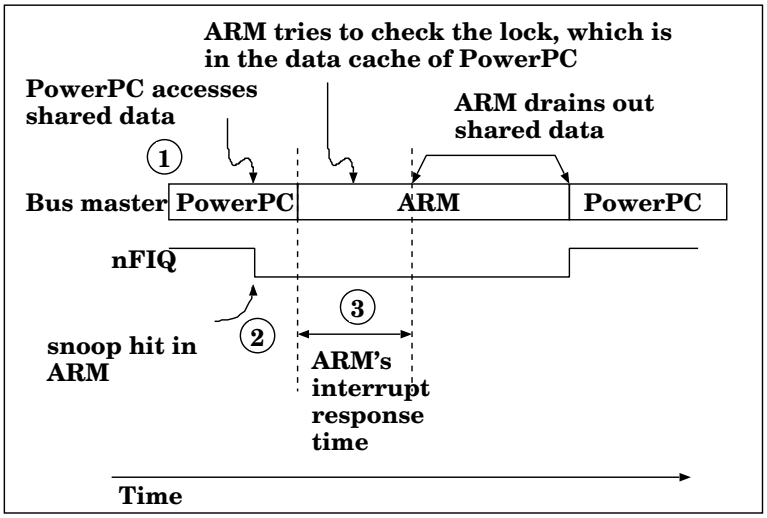


Figure 20: Hardware deadlock problem in PF1 and PF2.

Even though this mechanism can achieve the data consistency, there is one limitation when at least one of the processors does not have native cache coherence support such as in the case of ARM920T. PF1 and PF2 pertain to this category, and Figure 20 illustrates the problem. Suppose that the architecture allows lock variables and a shared data to be cached, and the shared data is currently in the ARM920T's data cache. After acquiring the lock, the PowerPC tries to access the shared data as shown in step ①. Therefore, a snoop hit occurs in the snoop logic, and the *nFIQ* is asserted as illustrated in step ②. Then, the ARM processor is supposed to drain or invalidate the addressed cache line in the interrupt service routine. However, because most commercial processors implement pipelining and precise interrupt, the ARM might or might not respond to the interrupt immediately. During the interrupt response time as shown in step ③, ARM might check the lock to see whether it has been released. Lock variables are currently in the PowerPC's data cache because the PowerPC acquired the lock lately. Therefore, PowerPC should drain the cache line storing the lock variables to main memory. However, if PowerPC gains the bus mastership, it is supposed to retry the transaction, which it did in step ①, instead of draining the lock variables. We call this situation *hardware deadlock*.

There are two solutions to preventing the hardware deadlock problem. The first option is not to cache the lock variables. The other alternative is to use the SoCLC [23] discussed Section 4.4.1. For the first solution, a software algorithm such as the Dekker's algorithm [44] or the Bakery algorithm [96], can be used as a lock mechanism for mutual exclusion even though it is inefficient from a performance standpoint. For the second solution, it needs a simple lock module sitting on a bus. Since the lock variables are not cached in either case, the hardware deadlock does not occur.

Even though we focused our discussion on synchronization using a single lock, the same problem can occur among critical sections where applications implement multiple locks in a system. For this reason, a system can have only one lock in PF1 or PF2, requiring that the program perform all shared variable operations within critical sections.

4.6 Experimental Setup

Hardware simulation demands enormous amount of time to run real applications. We experimented with the Verilog simulation of a MPEG decoding application on a two-processor platform with three small frames. Using the simulation environment listed in the Table 7, the simulation took more than three days to finish on a SUN UltraSPARC workstation, making a complete evaluation of our approach too time-consuming. Therefore, we designed a suite of micro-bench to evaluate the impact of our methodology.

Our micro-bench suite consists of a worst-case scenario (WCS), a typical-case scenario (TCS), and a best-case scenario (BCS). In these programs, one task runs on each processor. Each task intensively tries to access a critical section protected by the SoCLC. Once a task acquires the lock, it accesses shared data quantified by cache lines and modifies them before exiting the critical section. We implemented the micro-bench in a way that each task acquires the lock alternatively. It means that

the simulation assumes the worst-case situation for lock acquisition and releasing.

In addition to the micro-bench, we also modeled another benchmark, part of the Atalanta RTOS kernel. As discussed in Section 4.4.2, Atalanta is tailored for heterogeneous multiprocessor SoCs. For interprocessor communication and synchronization, Atalanta provides shared memory approach. Thus, processors that share system objects such as semaphores and mailboxes, can directly access other processors' task control blocks (TCBs). For simulations, we modeled the task insertion and deletion mechanism in the Atalanta RTOS kernel. Suppose that processor 1 and processor 2 share a semaphore **S**. Processor 1 now owns the **S**, and processor 2 is waiting for it. When processor 1 is done with the **S**, it releases the **S** and promotes the waiting tasks' TCB to the ready state by changing the state field in TCB. Processor 1 also inserts the TCB into the ready list of processor 2 and sends an interrupt to processor 2, so processor 2 can reschedule tasks and runs the highest priority task. This procedure is repeated continuously for evaluations. In the Atalanta, tasks' TCBs of each processor are connected through a doubly-linked list based on priority. Each task's TCB has 14 word-length fields including the state, priority, and two fields for the doubly-linked list. The Atalanta also maintains an array to reference the highest-priority ready tasks. The array is shared by all processors in a system.

We use a complete *software solution* as the *baseline*, in which the programmer is responsible for draining or invalidating all the used shared data before exiting critical sections. Since the critical sections are protected by the lock mechanism, users should be aware of which shared data are in use in the critical sections[3]. The Atalanta RTOS also has an option to enable the cache flush instructions at the end of each critical section, in case processors do not have native coherence support.

[3]To flush out all used shared data in the critical sections, we used "asm volatile ("mcr p15, 0, %0, c7, c14, 1"::"r"(addr))" for ARM920T and "asm volatile ("dcbf 0, %0"::"r"(addr))" for PowerPC755 recursively.

Table 7: Simulation environment.

Simulators	• Seamless CVE [79] • ModelSim [80]
Operating frequencies	• PowerPC755: 100MHz * • ARM920T: 50MHz * • ASB: 50MHz
Instruction & Data caches	Enabled
Memory access time	• Single word: 7 cycles • Burst (8 words) – 7 cycles for the 1^{st} word – 1 cycles for each subsequent word

* These low frequencies are because of the limitation of simulation models. Similar results are expected for simulations with higher operating frequencies

The integration techniques (the read-to-write conversion and shared signal assertion/de-assertion) is referred to as the *simple hardware approach*. The snoop-hit buffer in addition to a simple hardware approach is referred to as the *snoop-hit buffer approach*. The simulation environment and the hardware configurations are summarized in Table 7. Two-processor (PowerPC755 and ARM920T) and four-processor platforms (three PowerPC755 processors and one ARM920T) are used to quantify the performance. The Intel486 and PowerPC755 platform[4] should outperform the PowerPC755 and ARM920T platform because of the absence of an interrupt service routine.

We simulated and measured the performance of the simple hardware approach, the snoop-hit buffer approach, and the baseline using hardware/software co-simulations. Seamless CVE [79] and ModelSim [80] from Mentor Graphics were used as simulators. We varied the memory latency from 7-1-1-1-1-1-1-1, 13-2-2-2-2-2-2-2 .. to 97-9-9-9-9-9-9-9. The string "7-1-1-1-1-1-1-1" means 7-cycle access time for the first word and 1-cycle access time for each 7 trailing word. Note that a cache line size is eight words. The miss penalties indicated by the x-axis in Figure 21 $\sim$ Figure 30 represent the latencies taken to fetch the entire cache line from main memory.

[4]Because of the unavailability of the complete processor model of Intel486 in Seamless at the time of this study, the results of a PowerPC755 and Intel486 system are not reported.

4.7 Simulation Results

4.7.1 Performance of Integration Techniques with Snoop-hit Buffer

Figure 21 to Figure 26 show the speedup with respect to the software solution as the miss penalty increases.

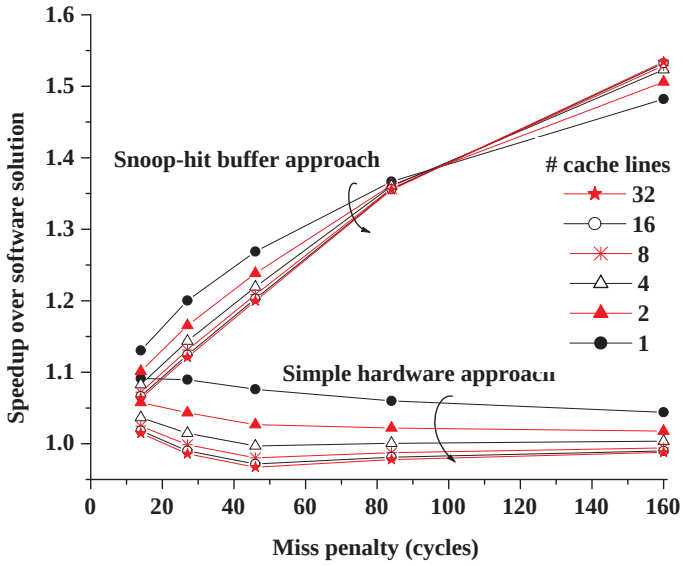


Figure 21: Worst case results on a two-processor platform.

In the WCS, each task keeps accessing the same blocks of memory alternatively. That is, a task in a processor accesses the shared blocks of memory after acquiring a lock. When the task releases the lock, a task in other processor acquires the lock and accesses the same blocks of memory. This procedure is repeated continuously. Figure 21 shows simulation results on the two-processor platform, where the simple hardware approach performs better than the pure software solution with a few exceptions. These exceptions come from cache line replacements and/or interrupt processing overheads that vary as the miss penalty changes. The simulation with the snoop-hit buffer shows at least a 6.3% performance improvement over the software solution for all WCS simulations. As the miss penalty increases, the performance of the snoop-hit buffer approach increases dramatically. The simulation result shows up

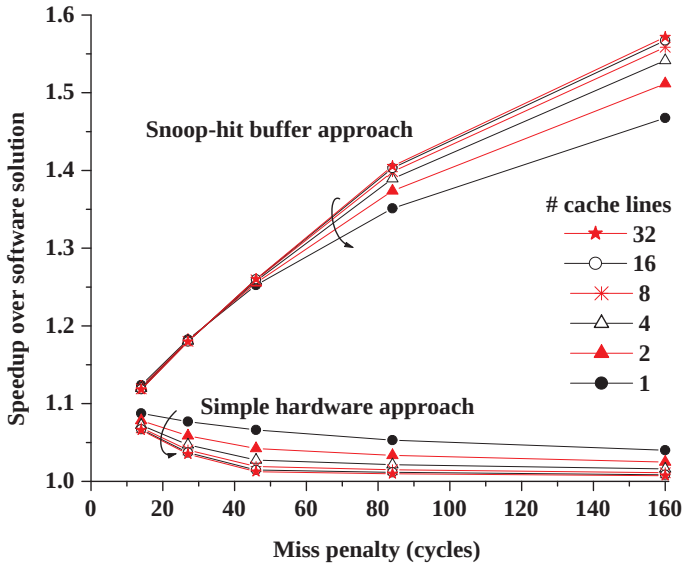


Figure 22: Worst case results on a four-processor platform.

to a 53.4% performance improvement, when the miss penalty is equal to 160 cycles and the number of accessed cache line is 32. Figure 22 shows simulation results on the four-processor platform, where even the simple hardware approach always shows better performance (at least a 0.97% improvement) with no exceptions because only one processor (ARM920T) needs the interrupt service routine. The simulation with the snoop-hit buffer approach shows an 11.8% $\sim$ 57.1% performance improvement compared to the pure software solution.

In the BCS, each processor accesses different critical sections alternatively. It means that snoop-hits do not occur with the coherence support in hardware. However, in the pure software solution, each processor should drain all the accessed shared blocks explicitly before exiting the critical sections. Figure 23 shows the simulation results on the two-processor platform, which achieved a 49.2% performance improvement with a 14-cycle miss penalty and one accessed cache line. The speedup increases as the number of accessed cache line increases and/or the miss penalty increases. The

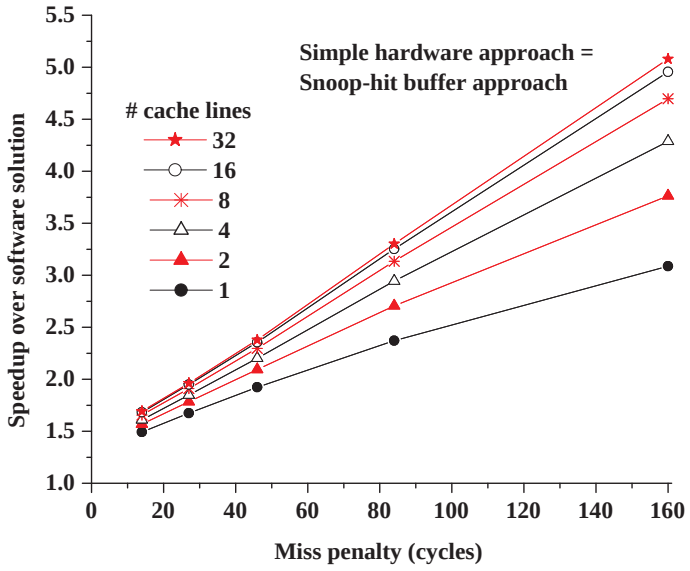


Figure 23: Best case results on a two-processor platform.

simulation with 32 cache lines shows a 407% performance improvement with an 160-cycle miss penalty. The simple hardware approach and the snoop-hit buffer approach show the same results, because snoop-hits do not occur in the BCS. Figure 24 shows the simulation results on the four-processor platform, which achieved a 51% $\sim$ 426% performance improvement.

In the TCS, each task randomly picks up shared blocks of memory among 10 blocks before entering into the critical section. Figure 25 shows the simulation results on the two-processor platform. The simple hardware approach shows a 21.7% $\sim$ 54.2% performance improvement, and the snoop-hit buffer approach shows a 24.5% $\sim$ 214% performance improvement compared to the pure software solution. Figure 26 shows the simulation results on the four-processor platform. The simple hardware approach shows a 27% $\sim$ 68.6% performance improvement, and the snoop-hit buffer approach shows a 46.4% $\sim$ 226% performance improvement over the pure software solution.

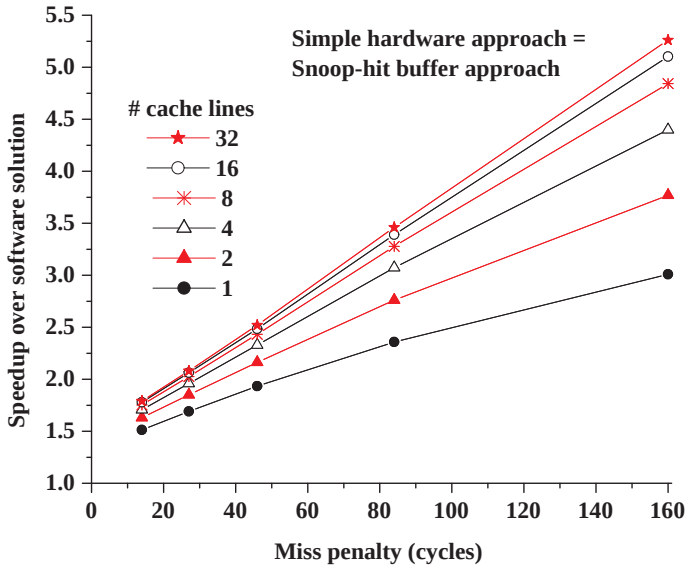


Figure 24: Best case results on a four-processor platform.

4.7.2 Performance of RBCC

We evaluated the RBCC performance with two benchmarks – Micro-bench and the Atalanta RTOS kernel model – using the hardware platform in Figure 18. Similar to the micro-bench in Section 4.7.1, the micro-bench consists of WCS, BCS, and TCS programs. One task runs on each processor, and each task accesses the same blocks of memory after acquiring the lock of the SoCLC. We use a *system without RBCC* as the *baseline*.

In the WCS, three ARM processors with the MESI protocols keep writing to the same blocks of memory while the PowerPC755 executes an idle task. Thus, the S state in the MESI protocol does not affect the performance and the RBCC shows the same performance as the baseline in Figure 27. However, with the snoop-hit buffer the performance increases dramatically because every snoop-hit takes advantage of the buffer. The simulation shows a 2.1% $\sim$ 56.9% performance improvement as the number of accessed cache line and/or miss penalty increases.

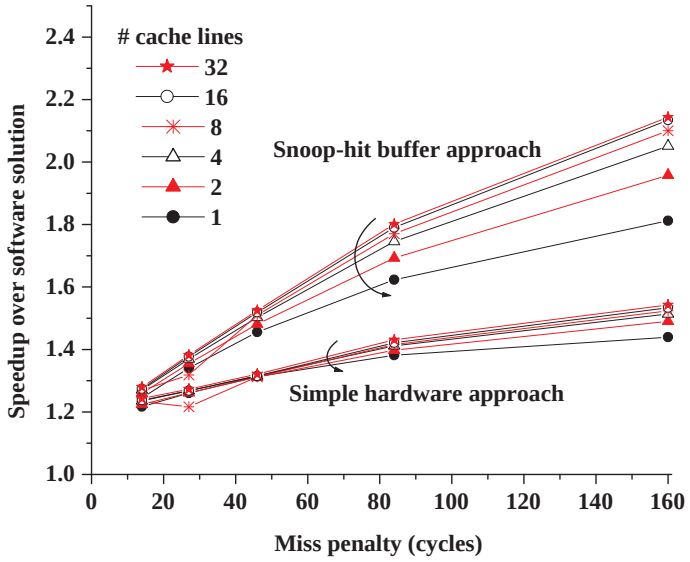


Figure 25: Typical case results on a two-processor platform.

In the BCS, three ARM processors keep reading the same blocks of memory while the PowerPC755 executes an idle task. Without the RBCC technique, the integrated coherence protocol of the platform in Figure 18 should be MEI. It means that whenever a processor reads shared blocks of memory, other processors should invalidate addressed cache lines if they are caching them. However, with RBCC, they need not invalidate the cache lines because the S state is still alive. The simulation results in Figure 28 show a 13% performance improvement with a 14-cycle miss penalty and one accessed cache line. The speedup increases as the number of accessed cache line and/or miss penalty increases. The simulation with 32 cache lines shows a 3.06x speedup over the baseline, when the miss penalty is 160 cycles. The snoop-hit buffer does not affect the performance since snoop-hits never occur in the BCS.

In the TCS, all four processors access one memory area (the MEI protocol), and three ARM processors additionally access another memory area (the MESI protocol). We modeled the even access probability of each area by processors, so the ARM processors access MEI and MESI memory areas with a 50% probability each. The

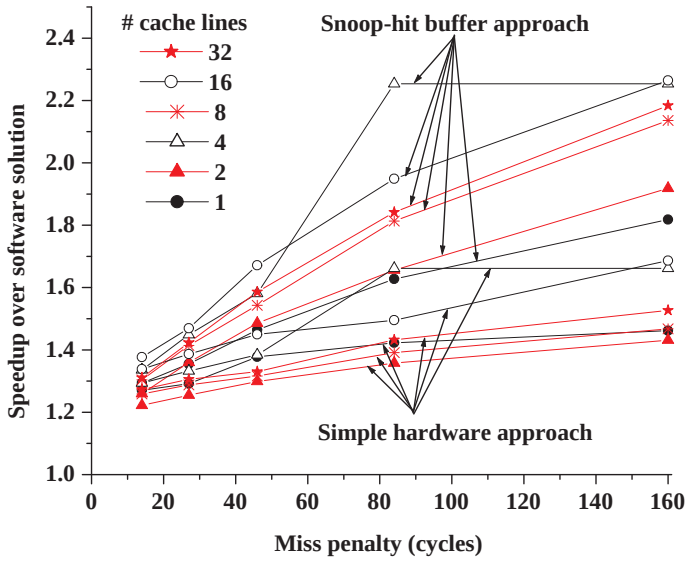


Figure 26: Typical case results on a four-processor platform.

simulation assumes that each ARM processor performs read operations 80% of the time, making a chance to use the S state around 50% ($= (0.8)^3$). Figure 29 shows the simulation results. RBCC shows a $0.5\% \sim 11.4\%$ performance improvement. The snoop-hit buffer in the baseline enhances performance by a $2.1\% \sim 19.6\%$. The RBCC with the snoop-hit buffer increases the performance from 2.7% to 36.4%.

For the RTOS kernel simulation, we used the same simulation platform in Figure 18. Figure 30 shows the performance enhancement of the RBCC as the miss penalty increases. The notation "2T-16T" means that two tasks are running on each processor in the MEI protocol area and 16 tasks are running on each processor in the MESI protocol area. A processor randomly selects two tasks to delete from and insert to TCBs. After the deletion and insertion of the tasks' TCBs from the doubly-linked lists, the processor that modified other processor's TCB generates an interrupt to the processor that owns the inserted task. Then, the interrupted processor repeats the procedure, that is, the insertion and deletion of two randomly selected tasks and the generation of an interrupt.

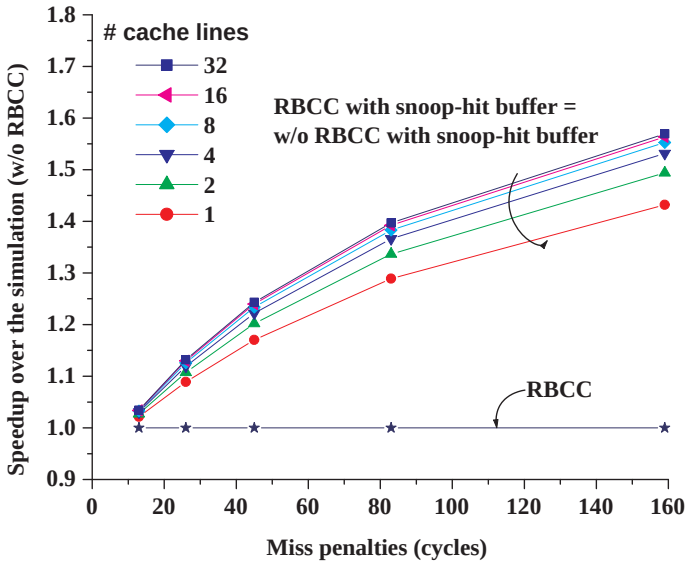


Figure 27: Worst case results of RBCC.

In the 2T-2T case in Figure 30, the RBCC shows marginal performance improvement ($0.4\% \sim 0.9\%$) over the baseline. This comes from short length of doubly linked list. Only two tasks are running on each processor in the MESI protocol area, so the list's length is two. The insertion and deletion in this short linked list demands modification of fields in both lists, leading almost no usage of the S state. Thus, the RBCC provides marginal performance improvement. This marginal improvement is due to sharing the array to reference the first ready list of tasks on each processor. However, the 2T-16T case shows an $11\% \sim 29\%$ improvement because 16 tasks are connected through a doubly-linked list of TCBs. Depending on the position of insertion and deletion, we modify the fields in only two or three TCBs. This lead to increased usage of the S state. The snoop-hit buffer in the baseline enhances the performance by a $4.4\% \sim 41.1\%$ for the 2T-2T case and by a $2.9\% \sim 26.1\%$ for the 2T-16T case. Finally, the RBCC with snoop-hit buffer enhances the performance by a $4.5\% \sim 43.0\%$ for the 2T-2T case and by $15.3\% \sim 77.0\%$ for the 2T-16T case.

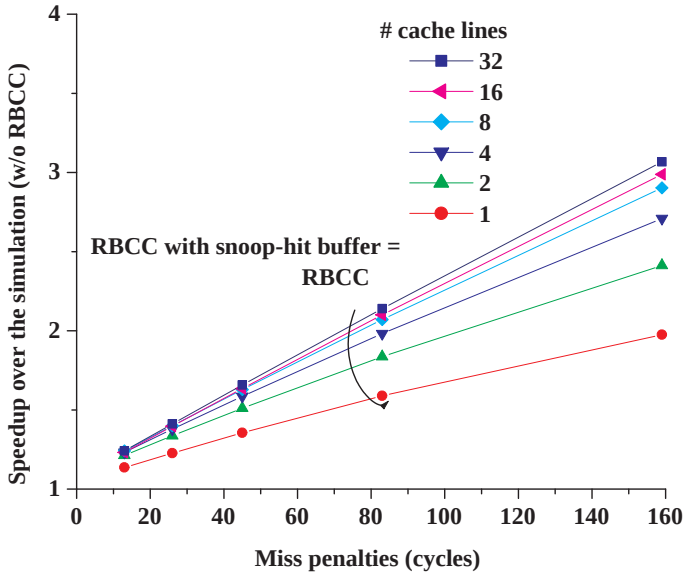


Figure 28: Best case results of RBCC.

4.8 Conclusion and Discussion

In this contribution, we provided generic solutions to integrate cache coherence protocols in heterogeneous processors for MPSoC designs. In this study, we limited our scope to shared-bus-based MPSoCs. For snoop-based and invalidation protocols, cache coherence can be guaranteed by implementing two integration techniques inside wrappers: the read-to-write conversion and the shared signal assertion/de-assertion. Depending on combinations of coherence protocols, we described generically how to integrate heterogeneous coherence protocols. In general, only the states that the distinct protocols have in common are preservable with a exception of the O state. To enhance the system performance, we proposed two architectural features: the snoop-hit buffer and the region-based cache coherence. We also presented implications and limitation of integrating processors with no native coherence support.

Using commercial embedded processors, we evaluated the performance on the two-processor and the four-processor platforms. Micro-bench simulations reported

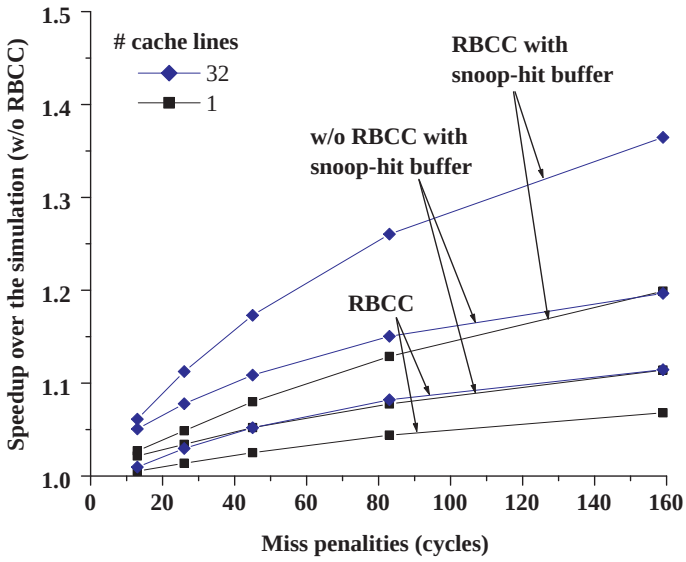


Figure 29: Typical case results of RBCC.

up to a 53.4% performance improvement over the pure software solution, even in the worst case when the miss penalty is 160 cycles. In the best case, the simulations showed a 5.26X speedup on the four-processor platform with a 160-cycle miss penalty. For the region-based cache coherence, the RTOS kernel simulations indicated a 77% performance improvement for the 2T-16T case when the miss penalty is 160 cycles.

For shared-bus-based MPSoCs, the integration techniques provide a generic solution for the incompatibility problem in communication via cache coherence protocols among heterogeneous processors. Nevertheless, in reality, the design of shared-bus-based MPSoCs demands much more engineering effort due to the following reasons.

First, heterogeneous processors could have unequal block sizes. The cache block size is the power of 2 words (2^n, where n = 1, 2, 3, ..., but typically 4 or 8 words). When integrating processors with unequal block sizes, special care should be taken for data consistency in wrappers. Suppose that the processor A's cache and the processor B's cache are based on 8 words and 4 words, respectively. When processor A, for example, generates an invalidation message on the bus, a wrapper around processor

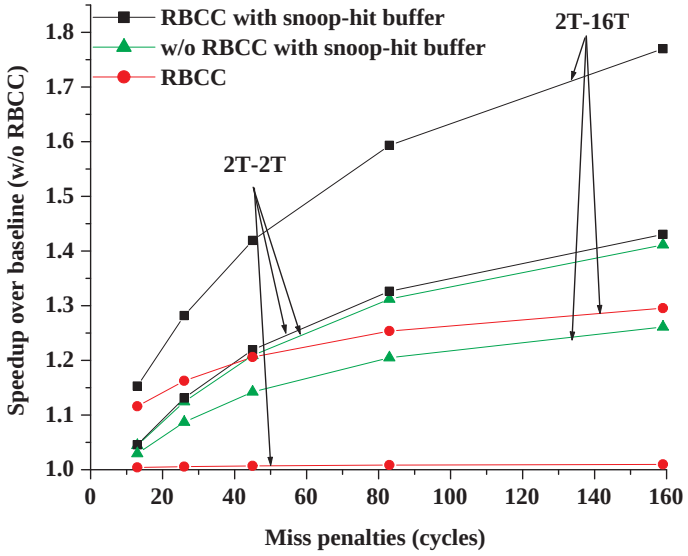


Figure 30: RTOS kernel simulation results of RBCC.

B should internally generate two invalidation messages to invalidate appropriate two blocks in the processor B' cache. Cache-to-cache transfer is rather problematic. When a snoop hits on a line in the processor B's cache, processor B is able to supply only 4 words and processor A needs 8 words to fill the line. The remaining 4 words should be supplied from either processor B if cached or main memory if not cached. Therefore, the processor B's wrapper should be able to internally generate two transactions to check two blocks.

Second, it is challenging to design the shared-bus protocol, which provides the superset functionality of different processors' interfaces. Processors' interfaces not only provide protocols for simple memory read and write transactions, but also typically deliver processor-specific information such as special exceptions, power-saving, monitoring information, and so on. Including all different protocols could make shared-bus protocol too complicated.

Third, embedded processor IPs may operate at a different range of clock frequencies. For communication between different clock domains, not only should the synchronization logic be designed, but also the wrapper design should include the

buffering mechanism for proper read and write operations.

Lastly, software-related issues play important roles in MPSoC designs. We discussed lock and RTOS problems in Section 4.4.1 and Section 4.4.2. In MPSoCs, embedded applications are mostly based on streaming data and exhibit the explicit parallelism. Therefore, the parallelization of embedded applications could be a straightforward task. However, in some rare cases, if applications do not show the explicit parallelism and need to be parallelized, it would be a challenging task since heterogeneous processors have different performance characteristics. Hardware/software co-design could resolve this issue in the early stage of MPSoC design by porting applications into processor models.

In the same context, the memory consistency model could be a problem in the heterogeneous multiprocessor environment. Depending on microprocessor vendors, several different ordering specifications have been proposed with its own mechanism for enforcing orders [44]. The purpose of these models is to overcome the performance limitation imposed by the *sequential consistency*. For example, the *total store ordering* and the *processor consistency* models relax the write-to-read program order. The *weak ordering* and the *release consistency* relax all program orders. In such systems, the *memory barrier* or *fense* instructions are used to enforce the desired ordering of memory references. Especially, in the heterogeneous multiprocessor environment, these instructions should be employed in conjunction with the synchronization operation. For example, the insertion of these instructions right after the lock acquisition (SoCLC) guarantees that all memory references before the synchronization are retired once a process acquires the lock.

CHAPTER V

CACHE COHERENCE SUPPORT ON NON-SHARED-BUS-BASED MPSOCS

5.1 Introduction

The communication architecture based on a shared-bus provides a convenient communication mechanism among processors since every processor is able to observe broadcasting messages via a shared medium. Nevertheless, when integrating heterogeneous processors based on a shared-bus architecture, the versatility provided by each processor is most likely compromised because of the difficulties discussed in Section 4.8. Moreover, SoC designers typically do not develop their shared-bus protocols due to short time-to-market and validation difficulties. Instead, they either use proven off-the-shelf protocols such as AMBA and CoreConnect, or employ standard interface protocols such as OCP-IP and VSIA. As a result, very few MPSoC vendors use a shared-bus approach. For example, commercial MPSoCs such as TI's OMAP [11] and Philip's Nexperia [10] employ multiple bus architectures due in part to fully utilize the native protocols.

The SoCs also should be inexpensive to gain competitiveness in the market. The overall cost of fabricating SoCs or ASICs highly depends on the die budget and the number of pin counts in the design. In most of the MPSoC systems, the majority of the pins are dedicated to memory interfaces. Given several address and data buses of multiple processors on an MPSoC, dozens of pins are easily consumed for each memory interface. For this reason, architects often make an effort to share or multiplex the memory interface among processors in SoC designs as long as the performance meets its requirement. For example, the C55x DSP and the ARM925T in the TI's OMAP

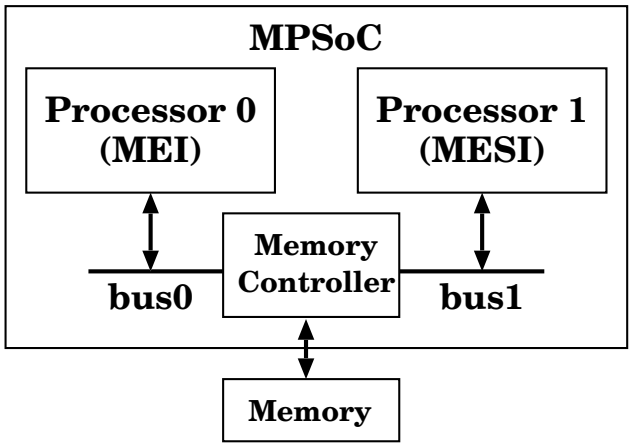


Figure 31: Multiprocessor and multiple-bus SoC architecture.

5910 processor share a common external DRAM interface in a similar way to the schematic in Figure 31.

In this contribution, we target a multiprocessor and multiple-bus (MPMB) SoC architecture, on which a memory interface is shared among multiple bus agents such as processors. Each processor uses its own private bus to access the shared memory. To simplify our subsequent discussion, we assume that only one processor is connected to each bus as depicted in Figure 31. Note that it can be easily extended to the scenario with multiple buses and multiple processors on each bus. Cache coherence among processors on each bus can be guaranteed by employing the integration techniques described in Section 4.2.

Unlike a shared-bus architecture, the communication problem in the MPMB architecture is self-explanatory because processors are physically separated by the memory controller in Figure 31. Without special communication channels, the explicit software-based synchronization must be used for the communication among processors in such architectures. Nevertheless, the software synchronization causes the performance degradation because of its inherent inefficiency.

This contribution [99] proposes low-cost techniques, which enable cache coherence capability for the efficient inter-processor communication on the MBMP architecture.

Our main focus is on PF3 (refer to Table 4) and we discuss the implication and limitation of integrating processors with no inherent coherence support in the MBMP architecture (PF1 and PF2). This chapter begins by presenting the proposed approaches. Then, we estimate the hardware overhead implementing the approaches. To present the benefits, the simulation platform and its environment are introduced, and simulation results are presented according to benchmarks.

5.2 Coherence Support

To accelerate data communication, we propose a *cache coherence-enforced memory controller* (ccMC), which wakes up the native snooping capability of processors. Then, we study two approaches in the ccMC, depending on the allowable silicon budget: *the bypass approach* and *the bookkeeping approach*. A common component in both approaches is a memory-mapped register pair. One register (start_addr_reg) is set to the starting address of a shared memory area, and the other (range_reg) specifies the size of the shared memory area. According to applications' needs, the register pair can be replicated to accommodate more discrete shared regions.

Figure 32 shows the simplified schematic of the bypass approach. The bypass approach blindly forwards a memory transaction to the opposite-side bus if the condition is met. For example, users first set the register pair with a shared area information. If the requested address of processor 1 falls into the shared region, the ccMC bypasses the memory transaction to the opposite-side bus (bus 0 in this example).

Figure 34 shows the simplified schematic of the bookkeeping approach. In addition to the register pair, the bookkeeping approach keeps track of coherence states of shared memory blocks inside the ccMC. Depending on the state information, ccMC either forwards a transaction to the opposite-side bus or sends a request directly to the main memory. The bookkeeping approach is similar to the DSM's directory-based scheme in a sense that it keeps track of the state information (the directory

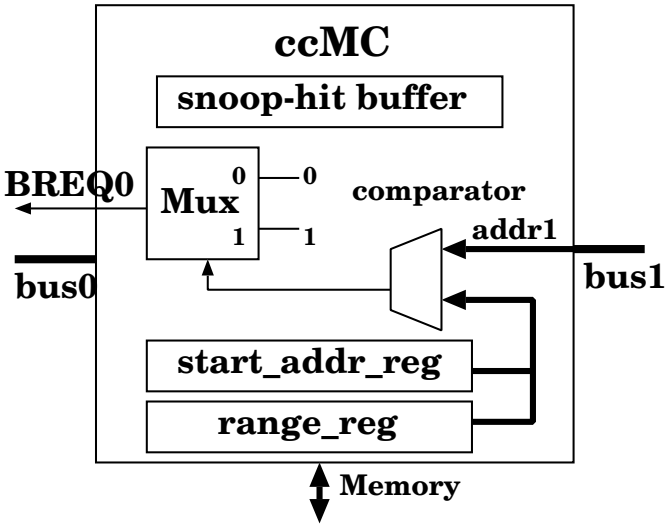


Figure 32: The bypass approach.

information in DSM) in the memory controller. However, it is different from the directory-based scheme because the goal is not to eliminate snooping, but to help snooping for improving bus utilization. Furthermore, ccMC employs a small table to cover shared memory regions based on the applications' needs.

5.2.1 Bypass Approach

The bypass approach forwards a shared memory request indiscriminately to the opposite-side bus for snooping. For example, suppose that Processor 1 (P1) in Figure 32 requests a shared memory block and misses its data cache. Consequently, this transaction appears on bus 1, and the ccMC compares the address against the register pair. Since the transaction is for the shared memory area, the ccMC requests the bus 0's mastership through the bus request (BREQ0) that is generated by the comparison match, as depicted in Figure 32. After granted the bus 0's ownership, the ccMC bypasses the transaction to bus 0. Then, Processor 0 (P0) is able to snoop the bypassed transaction. We employ the snoop-hit buffer depicted in Figure 32 to expedite data transfer between processors when a snoop-hit occurs on a line with the *M* state. Our current implementation of the snoop-hit buffer stores one cache line.

The bypass approach consumes bus bandwidth on both sides if a requested address is within the shared memory region. This overhead comes from the fact that the ccMC must claim the bus mastership of the opposite-side bus whenever a processor requests a shared data. It happens whether the other processor has the data in its cache or not. The advantage of the bypass approach is its simplicity and its small hardware overhead. As illustrated in Figure 32, it only requires two comparators, two multiplexers and one register pair.[1] The bypass approach would be useful for computation-bound applications because less memory traffic will appear on the bus.

The bypass mechanism makes separate buses effectively one single bus for shared memory regions because processors are able to observe all the traffic bound for the shared memory region. Therefore, the integrated protocols in the bypass approach become the same as the ones studied for the shared-bus architecture in Section 4.2. We summarize integrated protocols according to the combinations of four major protocols: MEI, MSI, MESI, and MOESI. We further discuss MBMP architectures where processors do not have native coherence support.

5.2.1.1 MEI with MSI, MESI, or MOESI

The integrations with the MEI protocol do not allow the S state because of the same reason illustrated in Table 5(a). We assume that the processor with the MEI protocol is on bus 1 and the processor with others is on bus 0.

MSI protocol: We employ the read-to-write conversion in the ccMC. As depicted in Figure 33, the ccMC changes a read transaction to a write when forwarding to the opposite-side bus, on which a processor with the MSI protocol resides. As explained in Section 4.2.3.1, the S state in the MSI protocol becomes equivalent to the E state with the read-to-write conversion since only one processor stores a specific block in cache at any point in time. Thus, the integrated protocol becomes equivalent to MEI.

[1] We only show the schematic diagram from bus 1 to bus 0 in this figure for brevity.

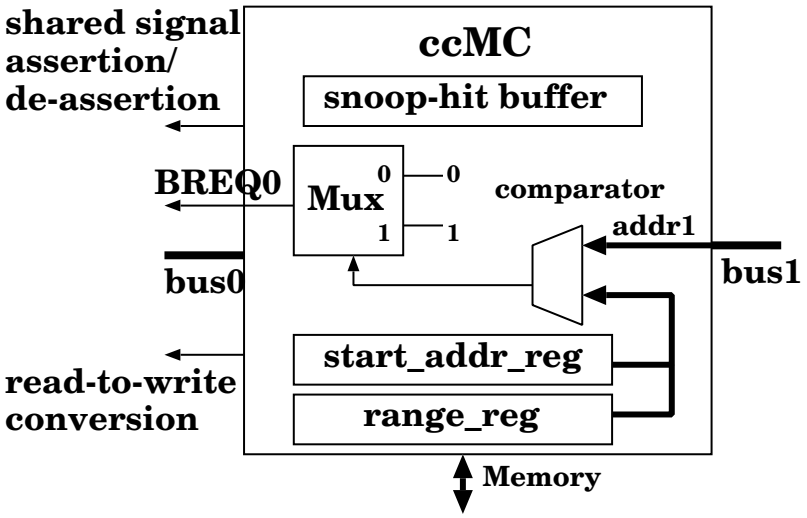


Figure 33: The bypass approach with the integration techniques.

MESI protocol: As explained in Section 4.2.3.1, the read-to-write conversion and the shared signal de-assertion are employed to remove the *S* state. The ccMC applies the read-to-write conversion when forwarding transactions to bus 0 where the processor has the MESI protocol. The ccMC de-asserts the shared signal in response to a read transaction from P0. As a result, the integrated protocol becomes equivalent to MEI.

MOESI protocol: The read-to-write conversion and the shared signal de-assertion remove the *S* state. The read-to-write conversion also eliminates the *O* state. Therefore, the integrated protocol becomes equivalent to MEI.

5.2.1.2 MSI with MESI, or MOESI

The integrations with the MSI protocol do not allow the *E* state to avoid the problem discussed in Table 5(b). We assume that the processor 1 with the MSI protocol is on bus 1 and processor 0 with others is on bus 0.

MESI protocol: The *E* state is completely removed by employing the shared signal assertion in response to a read transaction from P0. Then, the integrated protocol becomes equivalent to MSI.

MOESI protocol: The same technique – shared signal assertion – is applied to this combination. As discussed in Section 4.2.3.2, the *O* state has the same meaning as the *S* state except a line with the *O* state is responsible for updating main memory when displaced. Therefore, depending on which processor initiates cache-to-cache transfer, the ccMC decides whether to update main memory. If a processor with the *O* state supplies data via cache-to-cache transfer, the ccMC does not update main memory. In the other case, the ccMC updates main memory as the *S* state implies. Thus, the integrated protocol becomes equivalent to MSI with the *O* state enabled

5.2.1.3 MESI with MOESI

This combination preserves all protocol states. For the *O* state, the same discussion made in the MSI and MOESI combination is applied for data consistency. If a processor with the *O* state supplies data via cache-to-cache transfer, the ccMC does not update main memory. In the other case, the ccMC updates main memory as the *S* state implies. Thus, the integrated protocol becomes equivalent to MESI with the *O* state enabled

5.2.1.4 Integration with no native protocol

A data cache without native coherence support behaves like having the MEI protocol without the snooping capability. When a read miss occurs, a block is brought into the cache and the corresponding valid bit is set, which is equivalent to the *E* state. A subsequent write to the same line marks the block dirty, which is equivalent to the *M* state. A write miss sets both the valid and dirty bits when the line is brought in, which is also equivalent to the *M* state. Therefore, when integrating it with other coherence protocols, the integrated protocol becomes equivalent to MEI. However, due to the lack of the snooping capability, an interrupt is used to maintain coherence for a processor with no coherence support. This incurs the hardware deadlock problem described in Figure 20 that imposes restrictions on a system implementation.

5.2.2 Bookkeeping Approach

Figure 34 shows the bookkeeping approach. Similar to the bypass approach, the bookkeeping approach maintains one pair of registers to specify a shared memory region. Additionally, it requires a state table in the ccMC to keep track of the coherence state information for each processor. For example, when a memory transaction from bus 1 falls within the specified region, it records the coherence state of the memory transaction in the corresponding entry of the state table. Depending on the state information in the table, the ccMC decides whether to forward the transaction or not. If the state table indicates the *I*, *E*, or *S* state in the corresponding entry, [2] the ccMC does not claim the bus mastership of the opposite-side bus (bus 0 in this case) and brings the data directly from main memory. If the state table indicates the *M* state, the ccMC forwards the memory transaction to the opposite-side bus As such, it eliminates unnecessary forwarding to the opposite-side bus. For bandwidth-bound applications, the bookkeeping approach can filtrate false coherence traffic and increase the effectiveness of bus utilization. We also employ the snoop-hit buffer for the performance improvement. The bookkeeping approach is more expensive than the bypass approach in terms of hardware cost. We will estimate the cost of the hardware implementations in Section 5.3.

The bookkeeping approach does not allow the *E* state of the coherence protocols, with an exception when integrating with the MEI protocol. The issue caused by the *E* state is illustrated in Table 8. In the example, we assume that P0 and P1 support MSI and MESI protocols, respectively. There are a sequence of memory operations ⓐ, ⓑ, and ⓒ executed on the same block **C**. Operation ⓐ incurs the *I*→*E* state transition in P1 and updates the corresponding entry in the ccMC state table. The subsequent write operation ⓑ makes the *E*→*M* transition in the P1's cache and it

[2] If a processor supports cache-to-cache transfer from the *E* or *S* state, the ccMC will claim the bus mastership to initiate the transfer.

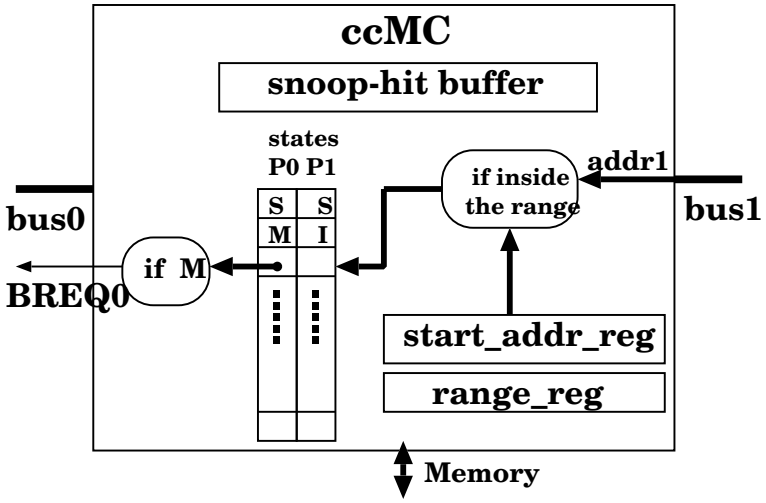


Figure 34: The bookkeeping approach.

does not appear on the bus because of a cache-hit. Thus, the corresponding update does not occur in the state table of the ccMC. The read operation ⓒ from P0 then will read a stale data from main memory, invoking the $E \rightarrow S$ transition in the ccMC table. These state transitions are illustrated in Table 8 from column 3 to column 6.

To forbid the E state, we use the shared signal assertion in the ccMC in response to a read transaction. The state transitions with this solution are illustrated in the last 4 columns of Table 8. With the shared signal assertion to P1 by the ccMC, operation ⓐ now makes the $I \rightarrow S$ transition. Then, operation ⓑ, visible to the ccMC, invokes the $S \rightarrow M$ transition in both the P1 and the ccMC state table. The operation ⓒ then receives up-to-date data from P1. It also makes the $I \rightarrow S$ transition in P0 and the $M \rightarrow S$ transition in P1. The ccMC table coherently reflects the state transitions, as shown in Table 8. The impact of eliminating the E state is rather insignificant because the E state only accounts for a very small portion of the total state transitions. [3]

Even though the E state is not allowed in general for the bookkeeping approach, the transition to the E state is inevitable in the MEI protocol. Because the S state

[3]The simulation results with the SPLASH2 show that a cache line is in the E state only in 0.76% of the time, as summarized in Table 6.

Table 8: Problem of the *E* state and solution in the bookkeeping approach.

seq.	Operation on a block C	Without shared signal assertion				With shared signal assertion			
		C state in P0 (MSI)	C state in P1 (MESI)	ccMC table		C state in P0 (MSI)	C state in P1 (MESI)	ccMC table	
				P0	P1			P0	P1
ⓐ	P1 read	**I**	**I → E**	**I**	**I → E**	**I**	**I → S**	**I**	**I → S**
ⓑ	P1 write	**I**	**E → M**	**I**	**E**	**I**	**S → M**	**I**	**S → M**
ⓒ	P0 read	**I → S**	**M**	**I → S**	**E → S**	**I → S**	**M → S**	**I → S**	**M → S**

is absent from the MEI protocol, integrating MEI with other coherence protocols requires that the ccMC employ the read-to-write conversion and/or share signal deassertion to eliminate the *S* state from the other protocols. The ccMC table in this case maintains only the *I* and *E* states because a write operation to the *E* state line does not appear on the bus. Here, the *I* state indicates the data is not in the cache, and the *E* state indicates that data is in either the unmodified state (true *E* state) or the modified state (hidden *M* state) in the cache. For each memory request, if the ccMC table indicates that the line is in the *E* state in other processor's cache, the ccMC accesses the opposite-side bus and places a write transaction on the bus. As such, a line with the *E* state becomes invalidated, or a line with the *M* state is drained to main memory. Finally, the requester gets data either from main memory or from cache-to-cache transfer depending on the state in the cache.

In the followings, we discuss integrated protocols according to the combinations of four major protocols in the bookkeeping approach. We further discuss MBMP architectures where processors do not have native coherence support.

5.2.2.1 MEI with MSI, MESI, or MOESI

The integrations with the MEI protocol do not allow the *S* state because of the same reason illustrated in Table 5(a).

MSI protocol: We employ the read-to-write conversion. As explained in Section 4.2.3.1, the read-to-write conversion effectively removes the S state because only one processor stores a specific memory block at any point in time. Therefore, the integrated protocol becomes equivalent to MEI.

MESI protocol: We employ the read-to-write conversion and the shared signal de-assertion to remove the S state. There are three transitions to the S state in MESI: $I \rightarrow S$, $E \rightarrow S$, and $M \rightarrow S$. As explained in Section 4.2.3.1, the read-to-write conversion eliminates $E \rightarrow S$ and $M \rightarrow S$ transitions, and the shared signal de-assertion removes the $I \rightarrow S$ transition. As a result, the integrated protocol becomes equivalent to MEI.

MOESI protocol: The combination also demands the read-to-write conversion and the shared signal de-assertion. These techniques not only prohibits the state transitions to the S state, but also prevents the transition to the O state because of the read-to-write conversion. Therefore, the integrated protocol becomes equivalent to MEI.

5.2.2.2 MSI with MESI, or MOESI

The integrations with the MSI protocol do not allow the E state due to the problem discussed in Table 5(b).

MESI protocol: There is only one transition to the E state in the MESI protocol: $I \rightarrow E$. By employing the shared signal assertion, the E state is completely removed from the protocol. Thus, the integrated protocol becomes equivalent to MSI.

MOESI protocol: The same technique – the shared signal assertion – is applied to this combination. As discussed in Section 4.2.3.2, the O state has the same meaning as the S state except a line with the O state is responsible for updating main memory when displaced. Depending on which processor initiates cache-to-cache transfer, the ccMC decides whether to update main memory. If a processor with the O state supplies data via cache-to-cache transfer, the ccMC does not update main memory.

In the other case, the ccMC updates main memory as the S state implies. Thus, the integrated protocol becomes equivalent to MSI with the O state enabled

5.2.2.3 MESI with MOESI

This combination prohibits the E state to avoid the problem discussed in Table 8. We employ the shared signal assertion to remove the E state. The transition to the O state is permitted as long as the ccMC has the discretion to differentiate the O and S states. As a result, the integrated protocol becomes equivalent to MSI with the O state enabled.

5.2.2.4 Integration with no native protocol

The same discussion in Section 5.2.1.4 is applied for data consistency.

5.3 Hardware Cost Evaluation

We implemented the bypass and bookkeeping approaches using Verilog-HDL, and synthesized them using Design Compiler from Synopsys with TSMC 0.18μ technology. The bypass approach uses two comparators, two multiplexers, two registers for each memory area, and a state machine to manage the forwarding. The ccMC also needs the bus master logic to control buses for each processor. The synthesis result reports 356 gates.

The bookkeeping approach introduces additional cost that mainly comes from the ccMC state table. Since the E state is not allowed as explained in Section 5.2.2, each entry in the state table needs two 2-bit registers for keeping three states (M, S, and I) for the MESI protocol and four states (M, O, S, and I) for the MOESI protocol. Table 9 summarizes the synthesized results according to the different sizes of the table. Note that all control logic overheads such as the table indexing are included in evaluating the hardware cost.

The bypass approach would consume a negligible amount of power considering the

Table 9: Hardware cost of the bookkeeping approach (cache line=32 bytes).

Shared memory area (KB)	Table size (Bytes)	Synthesized results (Gates)
1KB	16B (32×4bits)	1,337
2KB	32B (64×4bits)	3,147
4KB	64B (128×4bits)	6,106
8KB	128B (256×4bits)	11,927
16KB	256B (512×4bits)	24,467
32KB	512B (1024×4bits)	50,715

small number of gate counts. The bookkeeping approach also would consume an insignificant amount of power with small table sizes. Moreover, this power consumption overhead will be compensated by reducing power-expensive off-chip memory accesses.

5.4 Experimental Setup

Figure 35 shows the simulation platform for performance evaluation. Depending on the processor combination, we use two platforms shown in Table 10. Platform ① has a PowerPC755 with MEI and an ARM9TDMI with MESI. Its integrated protocol is MEI. Platform ② integrates an ARM9TMDI with MSI and another ARM9TMDI with MESI. Its integrated protocol is MSI. We implemented the simulation platform using Verilog-HDL and Seamless [79] processor models. The Seamless CVE and ModelSim from Mentor Graphics were used as simulators. The PowerPC755 has a 32KB data cache with the MEI protocol. For the ARM9TDMI, we implemented an 8KB data cache with the MSI and MESI protocols using Verilog-HDL. The cache line size is 32 bytes. The ARM9TDMI and the PowerPC755 operate at 50MHz and 100MHz, respectively.[4] Main memory and buses are synchronized at 50MHz. The cache miss penalty varies from 14 cycles to 45 cycles in the simulations. There are two bus agents that may request bus mastership on each bus: a processor and a

[4]This low frequency is due to the limitation of the Seamless ARM9TMDI model. However, we expect similar results at a higher frequency.

Table 10: Processor combinations in Figure 35 and integrated protocols.

	P0/Protocol	P1/Protocol	Integrated protocol
Platform ①	PowerPC755/MEI	ARM9TDMI/MESI	MEI
Platform ②	ARM9DTMI/MSI	ARM9TDMI/MESI	MSI

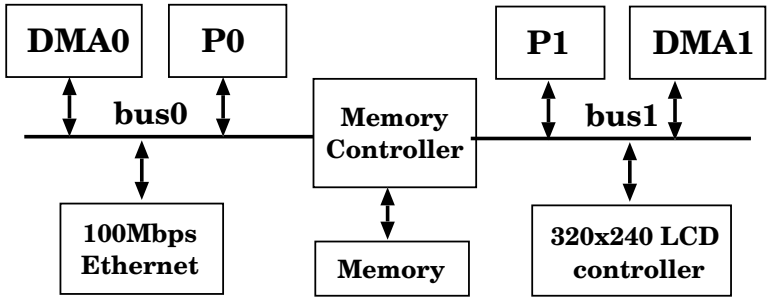


Figure 35: Simulation platform for the evaluation of the bypass and bookkeeping approaches.

DMA engine. The DMA engine handles direct data transfer from peripheral devices to local memory and vice versa. It is used to inject traffic on each bus. We modeled a 100Mbps Ethernet interface on the bus 0 and a 320×240 resolution, 30 frames/sec LCD controller on the bus 1. The DMA 0 manages data transfer from the RX buffer to local memory and from local memory to the TX buffer, while the DMA 1 transfers frame data for the display in LCD.

For the performance evaluation, we use the RTOS kernel and the micro-bench simulations. As described also in Section 4.6, the simulated RTOS kernel is the task insertion and deletion routines, which are based on a slightly modified version of the Atalanta kernel routines [104]. As the *baseline*, we use the explicit software synchronization mechanism. The micro-bench will be described in Section 5.5.2.

5.5 Simulation Results

5.5.1 Performance of the Bypass Approach

Figure 36 and Figure 37 show the simulation results with 2 tasks and 32 tasks on each processor, respectively. Without the snoop-hit buffer, platform ① and platform ②

show 2.20X and 1.64X speedups, respectively when 2 tasks are running on each processor and the miss penalty is 14 bus cycles. With the snoop-hit buffer, the speedup increases to a 2.28X in platform ① and an 1.70X in ②. The snoop-hit buffer boosts the performance because it shortens data transfer latency through the ccMC upon a snoop-hit. For 32 tasks with the same miss penalty, platform ① shows 6.65X and 6.57X speedups with and without snoop-hit buffer, respectively. Platform ② shows 4.78X and 4.71X speedups with and without snoop-hit buffer, respectively, in the same condition.

Speedup differences between two platforms come from the performance difference of the processors in the simulation platforms. Since the PowerPC755 is a two-way superscalar machine running twice faster than the ARM9TDMI, the computation time takes less in platform ①. In other words, the impact of the memory access latency becomes more conspicuous in platform ①. In the baseline, software flushes all the touched, shared blocks before exiting a critical section. This results in the invalidations of the used shared blocks that will not even be touched by the other processor. For example, TCBs' priority fields are used only for rearranging tasks' TCBs from doubly-linked lists based on tasks' priorities. Nevertheless, the software solution flushes the cache lines with the priority fields before exiting a critical section because the fields are shared among processors. Next time, those fields should be read again from main memory in the baseline, whereas it does not happen in the bypass approach (i.e., coherence support in hardware). Therefore, platform ① reports better speedup numbers than platform ②.

As shown in Figure 36 and Figure 37, the speedup with a low miss penalty is higher than the one with a high miss penalty. As the miss penalty increases, the total execution time is largely dependent on the memory access latency and the DMA traffic consumes more bandwidth equally in the bypass approach and the baseline. Since the software solution is already slow with a low miss penalty, the rate of the

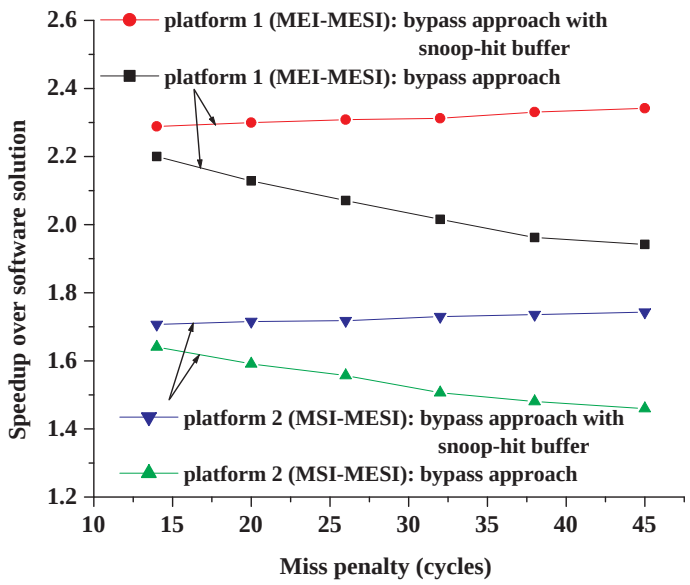


Figure 36: RTOS kernel simulation results of the bypass approach (2 tasks on each CPU).

execution time increase is faster in the bypass approach than in the baseline, As a result, the speedup with a low miss penalty is higher than the one with a high miss penalty. It becomes saturated as the miss penalty becomes large enough.

Figure 36 shows similar speedup patterns between two platforms even though the integrated protocols are different (MEI and MSI). This is a result from the shorter lengths of the doubly linked lists. Since only two tasks are running on each processor, the length of the doubly-linked list is two. The insertion and deletion of the linked list demands the modification of fields in both lists, resulting in the useless S state. On the other hand, in Figure 37, the MSI protocol enjoys the benefit of the S state because there are 32 tasks on each CPU. In this case, only two or three TCBs need to be modified depending on the position of insertion and deletion. Therefore, the speedup slope in platform ② is less steep than the one in platform ①.

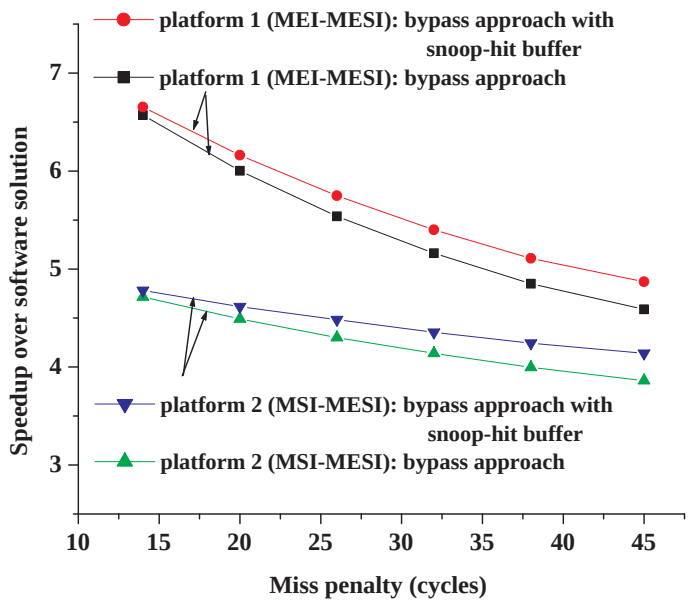


Figure 37: RTOS kernel simulation results of the bypass approach (32 tasks on each CPU).

5.5.2 Performance of the Bookkeeping Approach

In the previous RTOS simulations, the bookkeeping approach did not show any significant performance improvement over the the bypass approach, so we did not show them in Figure 36 and Figure 37. This is because the RTOS simulations use the tiny working set due to the intolerable hardware simulation time. Note that only 56 cache lines are needed for storing all the TCBs of the 32 tasks. This working set is comfortably fit into data caches. Processors do not refer to the state table in the ccMC most of the time due to cache hits. The advantage of the bookkeeping approach is shown when a processor misses its local cache and the state table indicates the I, E or S state in the other processor's cache. In this case, the requesting processor acquires data directly from main memory instead of snooping the other processor's cache. Note that the latter always occurs in the bypass approach. As explained, processors' requests hit the caches most of the time due to the small working. The task insertion and deletion mechanism incurs write operations to cache lines, resulting in

the M states in the caches and the ccMC table. As a result, the advantage of the bookkeeping approach was not clearly shown in our prior analysis.

To illustrate this advantage, we modeled a synthetic micro-bench by running one single task on each processor and having each task access the same memory blocks. Each task should acquire a lock to enter a critical section. Before exiting the critical section, all the shared blocks accessed are forcibly evicted by contrived conflict misses in the micro-bench. We use DMAs with the cycle-steal mode to inject traffic on all buses like the RTOS simulations. We studied the performance sensitivity under different bus bandwidth utilization by controlling the amount of the DMA traffic. For example, the 25% bus utilization (x-axis) in Figure 38 means that the DMA 0 and the DMA 1 both request bus masterships every 32 cycles and transfer data for 8 cycles once granted.[5]

Figure 38 shows the speedups of the bookkeeping approach over the bypass approach as the DMA's bus utilization increases from 10% to 90%. The bookkeeping approach shows up to a 10% performance improvement compared to the bypass approach. In general, the performance continues to increase until the DMAs' bus utilization reaches 70%. After that, DMAs are very likely to block the ccMC's snoop requests and the processors' requests in both approaches, resulting in the speedup declines. We also observed some outliers cases. For example, when the number of shared blocks is eight and DMAs' bus utilization is 10%, the bypass approach shows a slightly better performance. By analyzing the simulation waveforms, we found that by coincidence, DMA operations are synchronized and periodically delay processors' requests in the bookkeeping approach, but not in the bypass approach. Two other cases show the same behavior when the bus utilization is changed from 10% to 25% with two and four shared blocks.

[5]However, it does not mean that the DMAs always use 25% of the bus bandwidth because the buses might be in use by processors or ccMC's snoop requests when the DMAs request for the bus mastership.

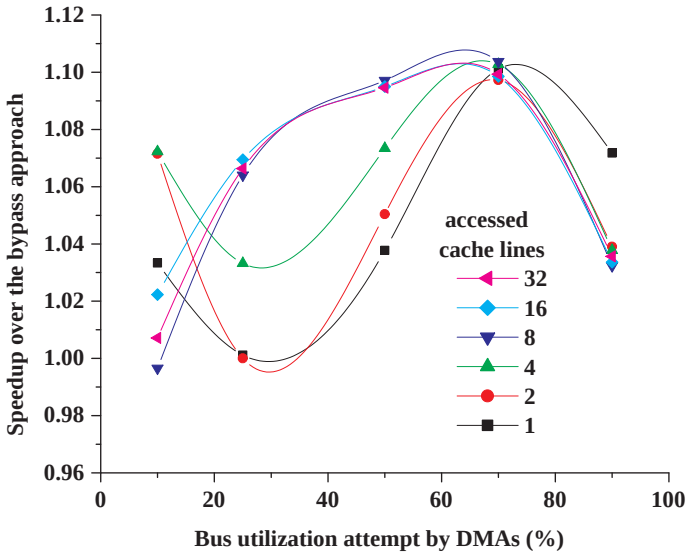


Figure 38: Simulation results of the bookkeeping approach.

5.6 Conclusion and Discussion

This contribution also provided a generic methodology to support cache coherence among heterogeneous processors in MPSoCs. This study extended our scope to non-shared-bus-based MPSoCs. For efficient communication among physically separated processors, we proposed the ccMC. In the ccMC, our techniques include the bypass approach and the bookkeeping approach depending on the allowable hardware budget. The bypass approach blindly forwards memory transactions to the opposite-side bus if the condition is met. The bookkeeping approach additionally checks cache line states for the shared region, and then filtrates unnecessary forwarding. The bypass approach is inexpensive while providing a significant speedup over the software solution. The RTOS kernel simulations report 6.65X speedup in platform ① with a low miss penalty and 32 tasks on each processor. The bookkeeping approach is comparably expensive because it demands the state table in the ccMC. The micro-bench simulations report up to a 10% performance improvement over the bypass approach when DMA traffic occupies 70% of the buses' bandwidth.

The MPMP architectures provide seamless plug-ins of processors to communication interfaces. In other words, processors are able to use native communication interfaces unlike the shared-bus-based MPSoCs. For example, the ARM and the PowerPC are able to use their own communication architectures (AMBA and CoreConnect, respectively). However, the memory controller (ccMC) design becomes complicated to provide separate interfaces for different communication architectures. There are other complications involved in engineering because of unequal cache line sizes, different operating frequencies, and software issues. Like the discussion in Section 4.8, if processors' caches have unequal line sizes, the ccMC should forward multiple memory transactions to the opposite-side bus, on which processor has a cache with a smaller line size. If buses operate at different clock frequencies, the ccMC should also take care of the clock synchronization and data buffering, as well as the protocol conversion between communication interfaces. The application parallelization and memory consistency model also complicate the design of MPSoCs as discussed in Section 4.8.

CHAPTER VI

EVALUATION OF COHERENCE TRAFFIC EFFICIENCY

Coherence protocols were evaluated in several literatures [57, 66, 88, 107, 39, 40, 26, 49, 51, 50, 48, 21, 85, 58, 44]. Traditionally, the evaluations mostly used trace-based simulations, focusing on protocols themselves. They followed state transitions of the protocols and counted the number of consequent bus transactions in snoop-based protocols, and the number of network transactions in directory-based protocols. Since those off-processor transactions play important roles in performance, the main focus of the evaluations was to devise the optimized coherence protocols that incur less off-processor transactions. As discussed in Section 2.3.1, in the coherence protocol evaluations, many detailed factors were abstracted that might effect the performance trade-offs in real systems [44]. Even though software-based simulation helps to evaluate relative effectiveness of coherence protocols, it often hinders to measure and evaluate system-wide effectiveness of coherence protocols and its traffic because the software model lacks the exact real-world modeling.

When workloads are parallelized and run natively on a symmetric multiprocessor (SMP) system, the speedup is dependent on three factors: First, how efficiently workloads are parallelized. Second, how much communication is involved among processors, which is the consequence of the first factor. Third, how efficiently the communication mechanism manages communication traffic (for example, cache-to-cache transfer between processors). While programmers make every effort to efficiently parallelize workloads, the underlying communication mechanism remains unmanageable in the software layer, and it becomes the limiting factor of the speedup as the number of processors increases. Despite the importance of the communication, it has not been

feasible to separate its contribution from the speedups collected on SMP machines. Oftentimes, because of the difficulty of the direct evaluation on real machines, software simulators [36, 76, 86, 92, 84] were developed to characterize the multiprocessor system performance. However, it is sometimes difficult to reach an unbiased conclusion in the software-based simulation because the exact real-world modeling such as I/Os is difficult. In addition, it hinders the broad range measurement of the system behavior due to the intolerable simulation time.

This contribution proposes a novel method to measure the coherence traffic efficiency on multiprocessor systems. Using an Intel server system and an FPGA, our method measured the intrinsic delay of coherence traffic. By natively executing workloads on an off-the-shelf system, this study evaluated and analyzed the impact of the communication mechanism on the overall system performance. It differs from the prior work discussed in Section 2.4. Firstly, unlike the MemorIES, HACS, and RACFCS, our method is an active emulation. Secondly, our work is based on a SMP machine whereas RPM emulates *ccNUMA* architecture. Lastly, unlike the ACE, our work implemented an L2 coherence cache. The coherence cache in FPGA not only more actively participates in FSB transactions, but also comprehensively follows all the FSB pipeline stages to enable cache-to-cache transfer for evaluation of the coherence traffic efficiency.

This study begins by introducing the coherence mechanism of Pentium®-III (P-III) processor on the front-side bus (FSB). After discussing the shortcomings of measuring the intrinsic delay of coherence traffic in multiprocessor systems, we introduce a new method to evaluate coherence traffic efficiency. We then present our equipment used to perform this study, and the evaluation metrics are discussed to quantitatively analyze collected data. After compiling collected data, we conclude our evaluations and discuss the possibilities to enhance the coherence traffic performance.

6.1 Coherence Traffic

6.1.1 Coherence Traffic in Generic Snoop-based Coherence Protocols

In the generic snoop-based coherence protocols, the coherence traffic is generated in the following circumstances.

- When a memory transaction hits on a line in the remote processor's cache
- When a processor's write operation misses its local cache
- When a processor's write operation hits on a line with the *S* state in its local cache

First, if a memory read transaction hits on a line with the *M* state in the remote processor's cache, it incurs cache-to-cache transfer accompanying the state transitions: $I \rightarrow S$ in the requesting processor's cache, $M \rightarrow S$ or $M \rightarrow O$ in the remote processor's cache. The cache-to-cache transfer is to supply data from the remote processor to the requesting processor. In the generic protocols, cache-to-cache transfer could occur anytime when the snoop hits on a line with the *E*, *S*, *M*, or *O* state for the memory read or write transaction. As discussed in Section 2.1.1, since the cache implemented in SRAM is faster than the main memory usually implemented in DRAM, the protocols expect faster transfer when transferring data through cache-to-cache transfer. However, this advantage is not necessarily present in modern bus-based SMP machines, in which intervening in another processor's cache to obtain data may be more expensive than obtaining the data from main memory [44]. Moreover, when a snoop hits on the lines with the *S* state in multiple processors, a selection algorithm is needed to determine which cache will provide the data, resulting in complicated hardware and potentially adversely affecting the performance because of the arbitration mechanism [44]. Therefore, cache-to-cache transfer typically occurs when a snoop hits on a line with the *M* or *O* state in modern multiprocessor systems.

Second, a write miss in the local cache generates the *read for ownership* transaction. It not only involves data transfer either from main memory or from cache-to-cache transfer, but also invalidates the same blocks located in remote processors' caches. Third, when a processor's write hits on a line with the S state in the local cache, it generates the invalidation traffic without incurring data transfer, which is often referred to as the *upgrade miss*.

While the state changes in caches do not result in a major performance impact, the coherence traffic such as cache-to-cache transfer and invalidation traffic plays an important role in the overall system performance, as it consumes memory bandwidth on a shared bus.

6.1.2 Coherence Traffic on Pentium-III

The P6 (Pentium®-Pro, Pentium®-II, Pentium®-III, and Celeron)-based SMP systems utilizes the FSB as a shared bus for communication. The FSB is a 7-stage pipelined bus, consisting of request1, request2, error1, error2, snoop, response, and data phases, as illustrated in Figure 39. Response and data phases are often overlapped. The FSB supports eight outstanding transactions. For cache coherence, the P-III uses the MESI protocol. Two active-low bus signals (HIT# and HITM#) are dedicated for the snooping purpose. The HIT# assertion indicates that one or more processors have the requested block in one of the clean states (E or S). The HITM# is asserted when the remote processor has the requested block in the M state, as depicted in ① of Figure 39. The snoop result of every memory transaction is driven in the snoop phase of the pipeline. Depending on the cache line status of the remote processors, the bus signal behavior and state transitions are different as shown in Table 11. Table 11 assumes that an FSB transaction hits on a line with the M, E, or S state in the snooping processor's cache. Like the generic coherence protocols, there are three kinds of coherence traffic on the P-III FSB.

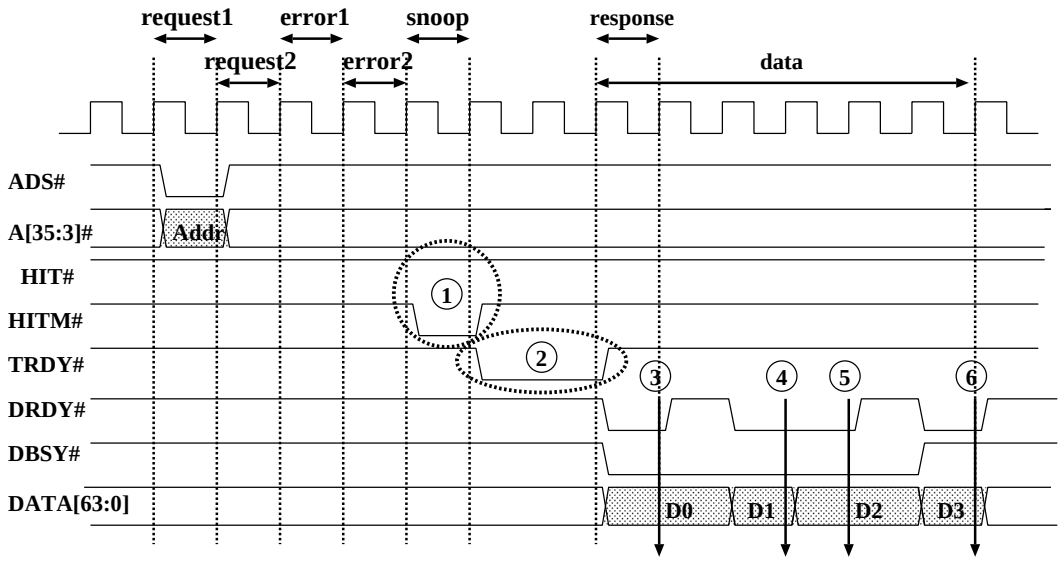


Figure 39: Timing diagram of cache-to-cache transfer on the P-III FSB.

- Cache-to-cache transfer
- Read-for-ownership
- Invalidation traffic for upgrade miss

In the P-III, cache-to-cache transfer is limited to snoop-hits on a line with the *M* state. It occurs after the HITM# is asserted, as shown in Table 11. Because of the lack of the *O* state, the P-III updates main memory simultaneously when cache-to-cache transfer occurs. To update main memory, the memory controller should be

Table 11: State transition in the cache line and bus signal behavior in the snoop phase on the P-III FSB.

FSB data traffic type	State change in request processor	State change in remote processor	FSB signal assertion		Cache-to-cache transfer
			HIT#	HITM#	
Read	I → S	M → S		√	√
		E, S → S	√		
	I → E	I → I			
Write	I → M	M → I		√	√
		E, S, I → I			

ready to accept data. Memory controller represents its readiness by asserting the TRDY# (target ready) signal, as depicted in ② of Figure 39. Afterwards, eight words (32B) of data are transferred as shown in ③, ④, ⑤, and ⑥. Note that data bus on the P-III FSB is 64-bit wide. Therefore, it takes at least four bus cycles to transfer one cache line data (32B). The number of bus cycles taken depends on the readiness of data from a processor or main memory. Figure 39 shows six cycles to transfer one cache line.

The read-for-ownership transaction takes place when a write operation misses the local cache. It generates a *full-line memory read with invalidation* on the P-III FSB. The upgrade miss takes place when a write operation hits on a line with the S state in the local cache. It initiates *0-byte memory read with invalidation* on the P-III FSB. The invalidation types are encoded in the request1 and request2 phases of the pipeline. All remote processors' caches, which may have the same line in the S state, invalidate corresponding cache lines when the invalidation transaction is observed.

6.2 Evaluation Methodology

6.2.1 Shortcomings in Multiprocessor Environment

As discussed, when workloads are parallelized and run natively on SMP systems, the speedup is dependent on three factors: Workload parallelization, consequent communication among processors, and underlying communication mechanism. Often times, programmers make use of performance monitoring to tune the performance of parallelized applications and to minimize the communication traffic. However, the underlying communication mechanism remains unmanageable in the software layer, and it could become the limiting factor of the speedup as the number of processors increases.

Despite the importance of the communication, it is not feasible to separate its contribution from the speedups collected on SMP machines. The bus architecture in the

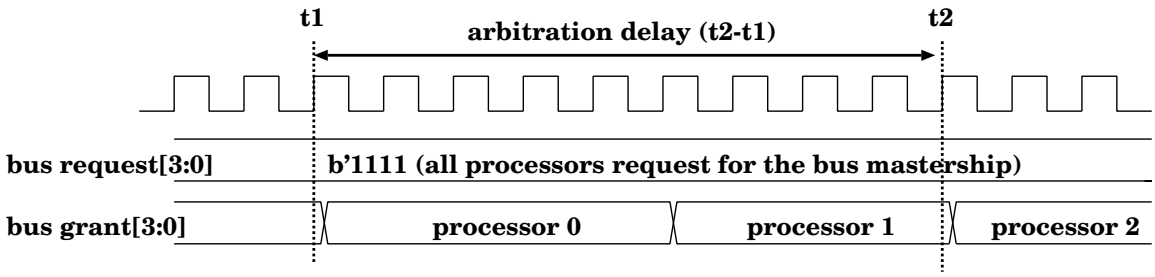


Figure 40: Bus arbitration delay in SMP systems.

snoop-based multiprocessor makes it even more infeasible to measure communication efficiency because of the arbitration delay and the pipeline stall incurred by multiple outstanding requests from processors. We discuss these shortcomings assuming a four-processor SMP platform, as shown in Figure 1.

6.2.1.1 Bus Arbitration Delay

In shared-bus-based multiprocessor systems, an arbiter mediates the bus mastership one at a time. Typically, priority-based and/or round-robin-based arbitration are used to grant the bus accesses for processors. Therefore, a processor with a low-priority or a processor recently accessed the bus will have to wait until its next turn to access the bus again. In the example of Figure 40, suppose that the processor 2's transaction incurs cache-to-cache transfer. All four processors are requesting for the bus mastership at time *t1*. Processor 0 first gets granted for the bus access, followed by Processor 1. Processor 2 is granted for the bus mastership at time *t2*. This arbitration delay elongates the processor 2's transaction by *t2* - *t1*, causing an effectively longer cache-to-cache transfer time. The arbitration delay is non-deterministic because it depends on how workloads are parallelized and when cache misses occur.

6.2.1.2 Stall in Pipelined Bus

Modern shared-bus protocols provide the pipelined bus to increase throughput. For example, the P-III FSB has a 7-stage pipeline, as illustrated in Figure 39. It is able to accommodate up to eight outstanding transactions on the bus. The pipelined

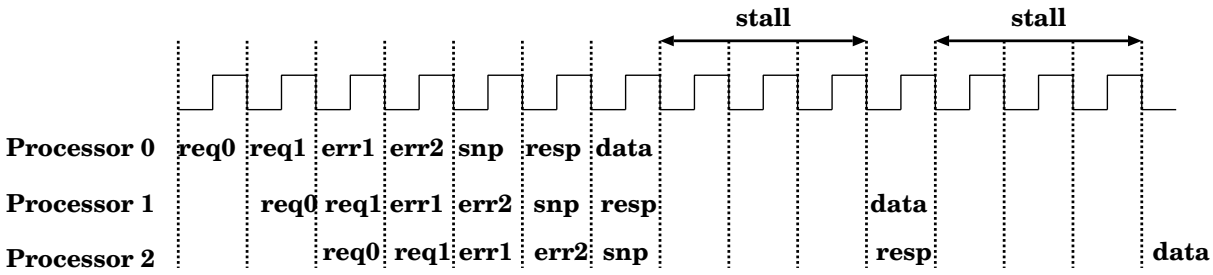


Figure 41: Delay caused by stalls in the pipelined bus.

bus also incurs the non-deterministic behavior of data transfer for bus transactions. As an example, suppose that there are three outstanding transactions from different processors on the FSB, as shown in Figure 41. We assume again that the processor 2's transaction incurs cache-to-cache transfer. Processor 0 finishes its transaction in the shortest time because there are no previous transactions. However, the processor 1's transaction is stalled by three-clock cycles, as depicted in Figure 41, because the snoop phase is overlapped with the data phase of the previous transaction. Note that the data phase requires at least four bus cycles to transfer one cache line (32B). Even worse, the processor 2's transaction is stalled by six-clock cycles because of the overlaps with the processor 0 and processor 1's transactions. This six-clock delay becomes effectively reflected on cache-to-cache transfer time. This behavior is non-deterministic because it again depends on how workloads are parallelized and consequently how processors manage cache misses. When the bus is idle, a bus transaction can be finished in the shortest cycles. However, when the bus is in full use, the 8^{th} transaction on the FSB, for example, could potentially suffer severe stalls. In the P-III FSB, the snoop phase also may be extended if it takes more time for snooping processors to determine hits or misses.

6.2.1.3 Discussion

Measuring and evaluating the intrinsic delay of coherence traffic requires eliminating non-deterministic factors such as arbitration delay and stalls in the pipelined bus.

Unfortunately, these problems persist as long as parallel workloads are running on a multiprocessor system. In the next section, we introduce our FPGA-based methodology, which is capable of eliminating these interferences and isolate the impact of coherence traffic on system performance. Note that, our methodology did not attempt to accurately model the coherence traffic of a given parallel workload running on a SMP machine. Rather, our goal is aimed to analyze and understand how the inter-processor traffic itself (cache-to-cache transfer and invalidation traffic) affects the overall performance based on coherence traffic emulated by the use of an FPGA.

6.2.2 Methodology for Intrinsic Delay Measurement

We use an Intel dual processor system, which features two P-III processors connected through the FSB. To remove discussed non-deterministic factors, one P-III has been replaced with an FPGA board. From the P-III's standpoint, we make the FPGA behave as an L2 cache[1] in a virtual processor on the FSB. For this, a cache is implemented in the FPGA. Then, our strategy to measure the efficiency of the coherence traffic is shown in Figure 42. Whenever the P-III evicts a modified cache line to main memory, the FPGA seizes the line from the FSB and saves it into the implemented cache. This is shown in ① of Figure 42. When the P-III requests the same block later, the FPGA indicates a snoop-hit by asserting the HITM# signal and provides the requested block through cache-to-cache transfer, as shown in ② of Figure 42. In other words, the FPGA is helping the P-III to run workloads by supplying data via cache-to-cache transfer. Then, we measure the execution times of the standard benchmark with and without the FPGA,

This configuration completely removes the bus arbitration delay because only one P-III is requesting for the bus mastership. In other words, the FSB is always granted for the remaining P-III. The pipeline stalls incurred by multiple processors' requests

[1]Note that P-III has an 8KB L1 and a 256KB L2 cache. The cache line size of 32 bytes.

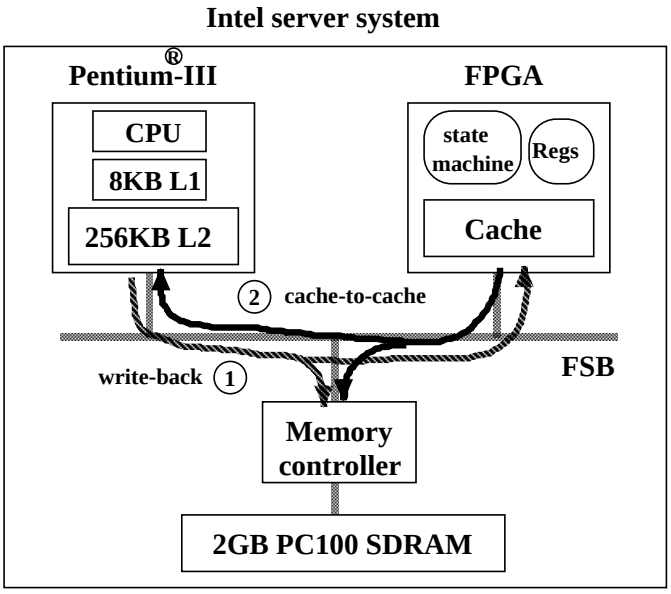


Figure 42: Evaluation methodology of coherence traffic.

are also completely removed because again, only one processor is monopolizing the bus. Even though one processor may initiate multiple transactions on the bus, it does not disturb our measurement because the same thing happens in our *baseline*. The baseline is to measure the execution times of benchmarks on one P-III without the FPGA, which will be discussed in Section 6.5. Consequently, this evaluation scheme is able to isolate impact of the intrinsic delay of the coherence traffic on system performance, and enables its efficiency evaluation by measuring and comparing execution times of benchmarks.

In this configuration, three kinds of coherence traffic are generated on the P-III FSB as if there are two P-III processors in the system. First, cache-to-cache transfer is generated when the FPGA finds the requested block in its cache. When the FPGA provides data via cache-to-cache transfer, there are two possible state transitions in the P-III's cache.

- The $I \rightarrow M$ transition if cache-to-cache transfer is incurred by a P-III's write miss
- The $I \rightarrow S$ transition if cache-to-cache transfer is incurred by a P-III's read miss

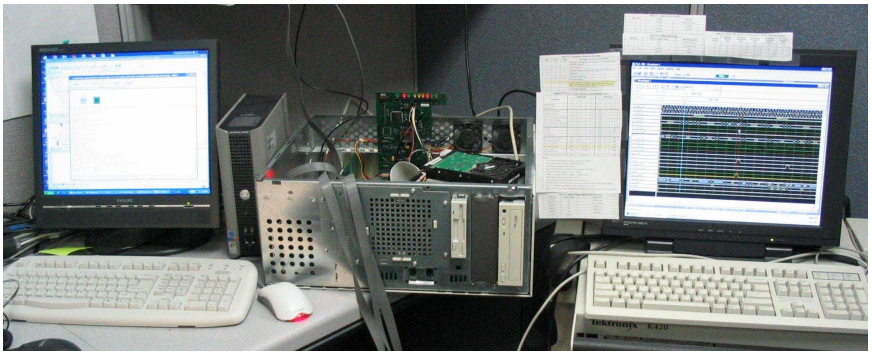
Second, the read-for-ownership transaction, which is known as the *full-line (32B) memory read with invalidation* on the P-III FSB, is generated by the internal write miss in the P-III. As mentioned, it incurs the $I \rightarrow M$ transition in the P-III, and accompanies cache-to-cache transfer if the block is found in the FPGA. Third, invalidation traffic, which is known as the *0-byte memory read with invalidation* on the P-III FSB, is generated by a subsequent P-III's write to the same block after the $I \rightarrow S$ transition.

By changing the cache size in the FPGA and measuring native execution times of workloads, we study the sensitivity of system performance on the intrinsic delay of coherence traffic. In this experiment, the more the P-III evicts replaced cache lines onto the FSB, there are better chances for the FPGA to incur coherence traffic when the P-III requests it later, leading more accurate evaluation of coherence traffic efficiency. Therefore, as long as the reasonable number of evictions occurs, the selection of the benchmark programs running on the P-III does not make any difference in evaluations.

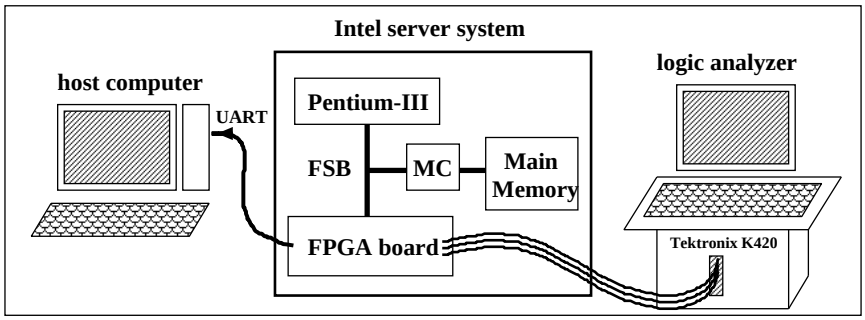
6.3 Experiment Infrastructure

Figure 45 shows the equipment setup for our experiments. There are three major components — an Intel server system, a host computer, and a logic analyzer. The Intel server system originally features two P-III processors. For this work, one processor was replaced with the FPGA board as depicted in Figure 43(b). Therefore, the P-III and the FPGA board are connected through the FSB, and the memory controller (MC) intermediates the main memory accesses. The FPGA board contains a Xilinx's Virtex-II (XC2V6000) [115], logic analyzer ports, and LEDs for debugging purposes. Each FSB signal is mapped to one Virtex-II pin. The FSB operates at 66MHz while P-III runs at 500MHz. The Intel server system also comes with 2GB main memory (4 PC100 SDRAMs).

The host computer is used for synthesizing our hardware design and programming



(a) Equipment picture



(b) Equipment schematic

Figure 43: Experiment equipment for coherence traffic efficiency measurement.

the FPGA with the generated bitstreams. The host computer also collects statistics from the FPGA board, which sends the number of events occurred every second through UART for post-processing. In this way, the disturbance to the Intel server system is completely eliminated during the execution of benchmark programs. Note that the system would be disturbed if the statistics was saved directly on the Intel server system because of the file access every second. The logic analyzer (K420) from Tektronix is used for debugging our hardware design. It is connected onto the FPGA board to probe the FSB and internal hardware signals.

6.4 Hardware Design

We used VHDL for the hardware implementation. The design is then synthesized, placed & routed, and downloaded to the FPGA fabric using Xilinx's ISE tool [113]. We used ModelSim SE [80] from Mentor Graphics to verify the hardware design.

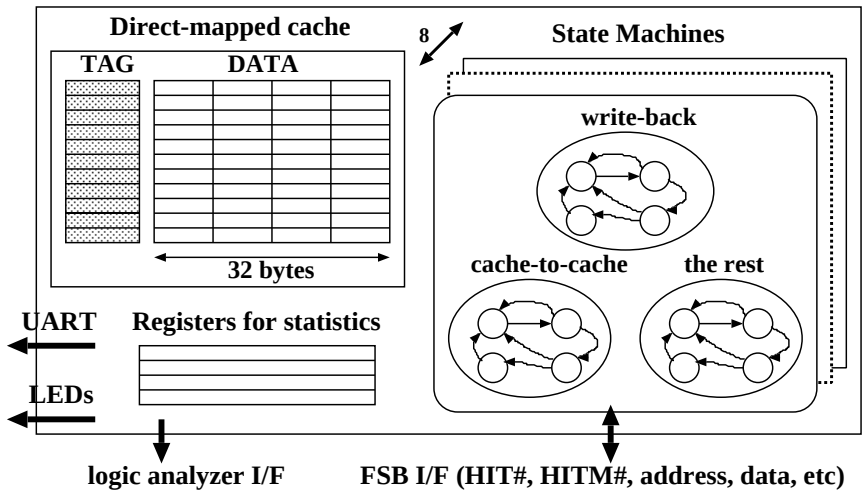


Figure 44: Hardware design schematic in the Virtex-II FPGA for efficiency evaluation.

However, because of the complexity of the FSB protocol, only certain simple functions were verified with ModelSim. After passing these simple functional tests, the design was tested directly on the Intel server system. Especially, this debugging step caused lots of system crashes until the design completely meets and follows the FSB protocol. System crashes came from supplying the wrong data from the FPGA to the P-III through cache-to-cache transfer. Figure 44 demonstrates the hardware schematic designed for the Virtex-II FPGA. It consists of several state machines, a direct-mapped cache, statistics registers, and the FSB interface. Now we detail each component.

6.4.1 State Machine

The main state machine keeps track of a bus transaction on the FSB and manages all internal and external operations. As shown in Figure 44, it is composed of three paths to perform the followings:

- To seize evicted cache lines from the FSB and store into the cache in the FPGA.
- To initiate cache-to-cache transfer that occurs when the P-III's requested block is found in the implemented cache inside the FPGA.

Table 12: BRAM usage in the Virtex-II FPGA (XC2V6000) according to the cache sizes.

Cache size	1KB∼16KB	32KB	64KB	128KB	256KB
BRAM usage	6%	12%	24%	45%	85%

- The rest that follows all other transactions on the FSB, including instruction read, I/O transactions, etc.

Since the P-III allows up to eight outstanding transactions on the bus, the FPGA should be able to track all eight transactions, concurrently. Thus, the same state machine is instantiated eight times, as shown in Figure 44.

6.4.2 Cache

To keep evicted cache lines, we implemented a direct-mapped cache. For the experiments, several versions varying from 1KB (32 cache lines) to 256KB (8192 cache lines) were designed. The cache's TAG, data, and valid bits were implemented with the dual-port block RAM (BRAM) inside the FPGA. One port is configured as the read port, and the other one is configured as the write port. Table 12 shows the BRAM usage information inside the XC2V6000 FPGA for various cache sizes. From 1KB to 16KB, the BRAM usage remains the same (6%) since the size of the basic block is fixed even when the required block is smaller than the basic block. With a 256KB cache, 85% of the BRAM is mapped to the FPGA fabric. In our implementation, the critical path is from the TAG lookup to driving the snoop result on the bus. All FSB signals are latched inside the FPGA prior to their use. Even though it exacerbated the timing budget, it is an inevitable choice for stable data processing.

6.4.3 Statistics Registers

We designed counter registers to collect statistics such as the number of cache-to-cache transfers, invalidation traffic, cache line evictions, and data read transactions

on the FSB. Whenever those events occur, the appropriate counter is incremented, and is reset to zero after sending it to the host computer. The statistics is sent every second to the host computer via the UART port. The UART is configured with 9600 baudrate.

6.4.4 FSB Interface

As explained in the cache design, the FSB signals are latched before being processed. The state machine makes state transitions depending on the latched FSB signals. Especially, when the state machine goes through the "cache-to-cache transfer" path, the FPGA actively participates in the bus transaction. Cache-to-cache transfer involves assertion of several FSB signals including HITM#, 64-bit data bus (DATA[63:0]), data-ready (DRDY#), and data-busy (DBSY#). The DRDY# and DBSY# signals are used to inform the right time to latch data by the P-III and/or by the memory controller. Figure 39 illustrates the usage of those signals. The DBSY# signal is kept asserted until all 4 quadwords are transferred. Then, when the DRDY# signal is asserted, data is available as indicated ③, ④, ⑤ in Figure 39. The last data is available when the DRDY# signal is asserted and the DBSY# signal is de-asserted, as shown in ⑥ of Figure 39. For the timing reason, our implementation allows two cycles to drive the data bus when cache-to-cache transfer occurs, meaning that the DRDY# signal is asserted every other cycle. After collecting the number of cache-to-cache transfers occurred, this effect is compensated in the efficiency calculation to be explained in Section 6.6.3.

6.5 Experiment Procedure

To measure and analyze the coherence traffic efficiency, we ran the SPEC2000 benchmark natively under Redhat Linux 2.4.20-8 on the remaining P-III of the Intel server system. Eight benchmark programs from SPEC2000 were executed for 5 times. Then, the average is calculated from the statistics gathered. Each batch run takes about

Table 13: Evaluation metrics for coherence traffic evaluation.

Metric	Unit
Number of cache-to-cache transfers	#/second
Number of increased invalidation traffic	#/second
Hit rate of coherence cache in the FPGA	Percent (%)
Execution time difference compared to baseline	second

15 hours on the Intel server system. By changing the cache size from 1KB to 256KB in the FPGA, we report and analyze the behavior of coherence traffic. The *baseline* system has a single P-III without the FPGA. As a result, all the memory transactions initiated by the P-III are serviced from main memory. In other words, cache-to-cache transfers and associated invalidation traffic never occur in the baseline. As discussed in Section 6.2, because of the nature of the experiment methodology, the benchmark selection does not effect the efficiency measurement of the coherence traffic as long as the reasonable number of eviction traffic is generated on the FSB. Table 13 summarizes the metrics used to report measured coherence traffic and evaluate its efficiency.

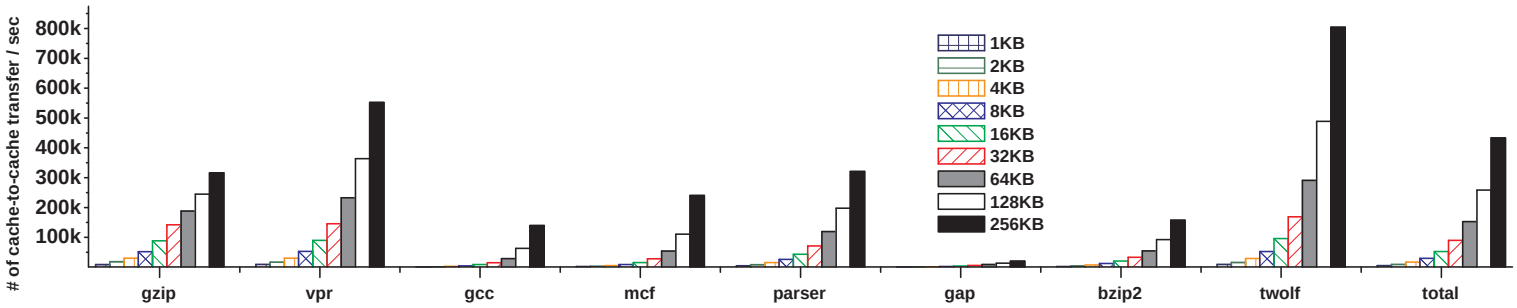
6.6 Experiment Results

6.6.1 Cache-to-cache transfer

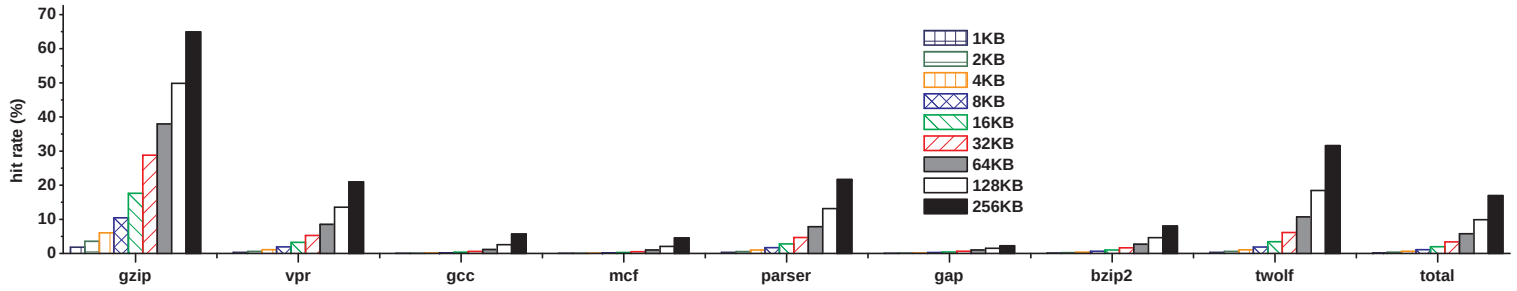
Figure 45(a) shows the average cache-to-cache transfers occurred every second, and Figure 45(b) shows hit rates of the coherence caches in the FPGA. The hit rate is calculated based on Eq (2), meaning how many times the FPGA is able to supply data when the P-III requests memory blocks.

$$hit\ rate\ (\%) = \frac{\#\ cache\text{-}to\text{-}cache\ tranfer}{\#\ data\ read\ (full\ cache\ line)\ on\ the\ FSB} \times 100 \qquad (2)$$

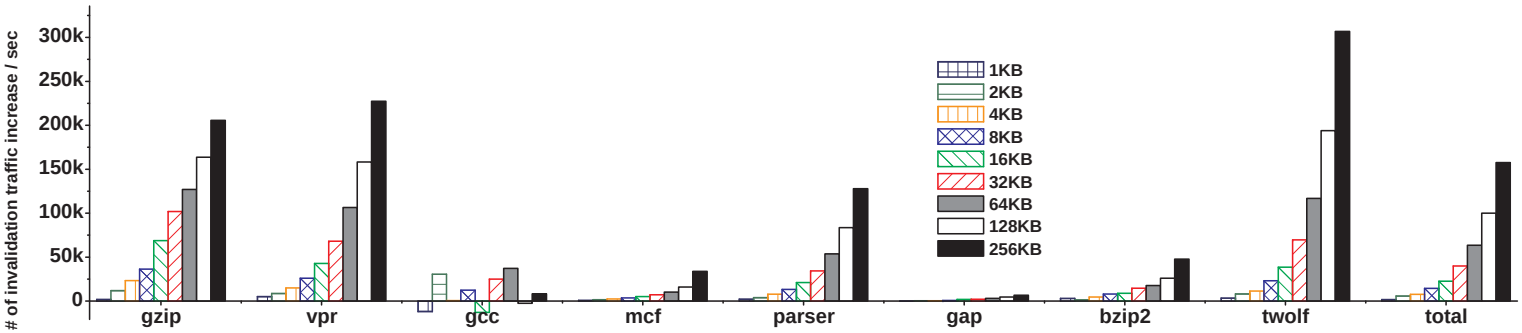
As the cache size in the FPGA increases, the number of cache-to-cache transfers also increases for all the benchmark programs. With a 256KB cache in the FPGA,



(a) Average cache-to-cache transfer per second



(b) Hit rate of coherence caches in the FPGA



(c) Average increase of invalidation traffic per second

Figure 45: Coherence traffic measurement results.

twolf shows the highest transfer frequency (804.2K/second), whereas gap shows the lowest (20.2K/second). With a 256KB cache, the hit rate can go as high as 64.89% in gzip and as low as 2.24% in gap. On average, as the cache size increases, the overall cache-to-cache transfer increases from 5.4K/second to 433.3K/second, and the hit rate varies from 0.2% to 16.9%. Memory-bound programs do not necessarily show the highest cache-to-cache transfer frequency because they might not use the evicted cache lines later and/or because conflict or capacity misses occur in the coherence cache. For example, mcf shows the comparably small number of cache-to-cache transfers. The low hit rate 4.5% even with a 256KB cache indicates that most of data requests in mcf were serviced from main memory.

6.6.2 Invalidation Traffic

Figure 45(c) shows the increased amount of invalidation traffic per second, compared to the baseline. With a 256KB cache in the FPGA, twolf again shows the highest peak (306.8K/second). On average, as the cache size increases, overall invalidation traffic increases from 1.7K/second to 157.5K/second. As explained in Section 6.1.2, invalidation traffic is incurred by two scenarios: ⓐ *0-byte memory read with invalidation*, ⓑ *full-line (32B) memory read with invalidation*. Figure 45(c) includes both traffic even though we observed that type ⓐ accounts for the major part ($> 99\%$). This indicates that SPEC2000 benchmark programs read data first and subsequently write to the same cache line, mostly generating type ⓐ traffic.

In general, Figure 45(c) shows the similar pattern to the average cache-to-cache transfers shown in Figure 45(a). This is explained as follows. When a memory read hits the cache in the FPGA, the FPGA initiates cache-to-cache transfer to supply data, causing the $I \rightarrow S$ transition in the line of P-III's cache. A subsequent write to the same line by the P-III generates type ⓐ traffic because of a upgrade miss, as the cache line is in the S state. As measured, SPEC2000 benchmark programs

tend to read data first and subsequently write to the same line. Therefore, the more cache-to-cache transfer occurs, the more likely invalidation traffic is to be generated.

The baseline system also generates invalidation traffic even though cache-to-cache transfer never occurs. This is due to cache flush instructions. When a page fault occurs, the P-III internally executes a cache flush instruction, which appears on the FSB as invalidation traffic. Depending on the Linux system services running in the background, the amount of invalidation traffic varies over time. In Figure 45(c), invalidation traffic sometimes decreases in the cases when the cache size in the FPGA is small enough, e.g., 1KB or 2KB. With small caches, the hit rate and the corresponding frequency of cache-to-cache transfer decrease significantly. For this reason, overall invalidation traffic is more sensitive to the system noise generated by the Linux system services. Especially, gcc is rather susceptible to the Linux system perturbation. The page faults caused by the large number of *malloc()* calls in gcc induce inconsistent patterns.

6.6.3 Execution Time

Figure 46 shows the increase of the overall execution time as the cache size increases. It shows all the collected data from five runs. As explained in Section 6.4, our implementation allowed two cycles to drive each data on the FSB for the timing reason. This effect was removed in Figure 46 by the linear approximation based on Eq (3), where *15.15ns* is the period of the FSB clock frequency (66MHz), *total # c-to-c* is the total number of cache-to-cache transfers, and the number *4* represents that four quadwords (32B) are transferred each time.

$$time\ (second) = measured\ time\ - (15.15ns \times 4 \times (total\ \#\ c\text{-}to\text{-}c)) \qquad (3)$$

On average, the total execution of the baseline takes 5,635 seconds (93.9 minutes).

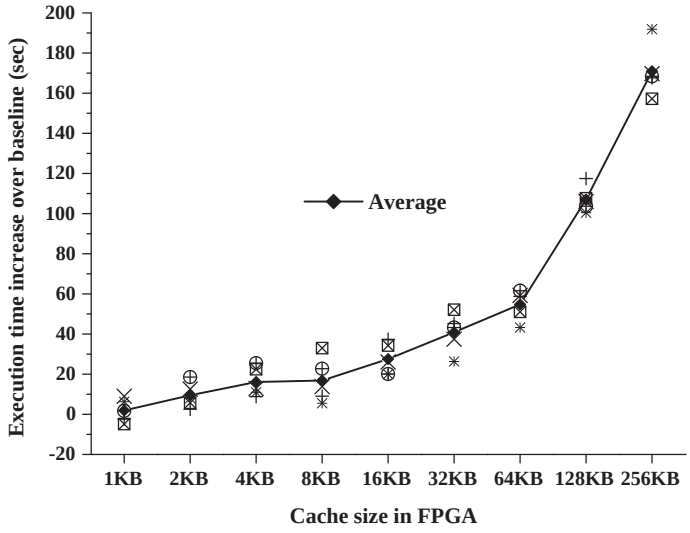


Figure 46: Execution time increase compared to the baseline (5 runs, baseline = 5635 seconds).

As shown in Figure 46, the execution time increases compared to the baseline, as the number of the coherence traffic increases. In other words, the benchmark execution assisted by the coherence traffic is more time-consuming than the one without it. With a 256KB cache, the execution time increased up to 191 seconds. There are two reasons that cause the inefficiency of coherence traffic. First, as explained in Section 6.1.2, main memory is simultaneously updated for each cache-to-cache transfer. This means that even when the P-III is ready to drive the FSB for the data transfer, it should wait until the memory controller is ready to accept data. Because of the busy schedule of the pipelined FSB, the memory controller would not promptly respond to cache-to-cache transfer requests. The second reason comes from invalidation traffic. As explained in Section 6.6.2, the increase of invalidation traffic follows a similar pattern to the number of cache-to-cache transfers. With a 256KB cache, overall invalidation traffic increased by 157.5K/second on average as shown in Figure 45(c). Even though such invalidation involves no data transfer, it still takes non-negligible amount of time since one FSB slot is needed for each invalidation.

6.6.4 Intrinsic Delay Estimation

The pipelined nature of the FSB makes it difficult to exactly breakdown the contribution of each coherence traffic to the execution times. Some coherence traffic might be injected to the FSB when the bus is idle. In this case, only pure pipeline stages are reflected into the latency. On the other hand, in some cases, coherence traffic might be delayed by long snoop and/or data phases of previous transactions when pipelined with other traffic. Therefore, our estimation is done by closely observing the FSB waveforms of coherence traffic in the logic analyzer and considering the FSB pipeline stages. By roughly estimating 5 $\sim$ 10 FSB cycles for each invalidation traffic and 10 $\sim$ 20 FSB cycles for each cache-to-cache transfer, the time spent for each coherence traffic is calculated using Eq (4).

$$\begin{aligned} &Execution\ time\ (second) = \\ &\quad (average\ occurrences/second) \times (total\ execution\ time) \\ &\quad \times (clock\ period/cycle) \times (latency\ for\ each\ traffic) \end{aligned} \tag{4}$$

$$\begin{aligned} &Average\ occurrences/second\ =\ 157.5K/second \\ &Total\ execution\ time\ = 5806\ (= 5635 + 171)\ seconds \\ &Clock\ period/cycle\ =\ 15.15\ ns/cycle \\ &Latency\ for\ each\ invalidation\ traffic\ =\ 5\ cycles \end{aligned} \tag{5}$$

For example, the time spent for invalidation traffic with a 5-cycle latency and a 256KB cache in the FPGA is calculated by plugging the numbers of Eq (5) into Eq (4), and Table 14 summarizes estimated times according to different latencies. Even with a 10-cycle latency for each invalidation traffic, it requires only 138 seconds, which is less than the average increase (171 seconds) of the total execution time. Therefore,

Table 14: Run-time estimation for each coherence traffic according to latencies (256KB cache in the FPGA).

	Coherence traffic	
	Invalidation traffic	Cache-to-cache transfer
Latencies	$5 \sim 10$ cycles	$10 \sim 20$ cycles
Times spent	$69 \sim 138$ seconds	$381 \sim 762$ seconds

the difference (33 seconds) in this calculation comes from cache-to-cache transfers unless each invalidation traffic requires more cycles. Based on this calculation, it is not unreasonable to say that cache-to-cache transfer in the experimented Intel sever system, which takes roughly $6.5\% \sim 13\%$ of the total execution time, is not as efficient as expected. In other words, cache-to-cache transfer on the P-III FSB is slower than getting data directly from main memory. [2] Even though clearly shown in Figure 46, this trend would be more conspicuous with a bigger cache size and/or high associativity cache implemented in the FPGA, as more coherence traffic is generated.

6.6.5 Opportunities for Performance Enhancement

Coherence traffic plays an important role in the performance of multiprocessor systems. In the P-III FSB, the fact that main memory should be updated simultaneously upon cache-to-cache transfer would be the main reason for the slowdown. The *O* state in the MOESI protocol is specially designed for this purpose. It allows cache-to-cache transfer without updating main memory. However, a processor with the *O* state line is responsible for updating main memory when the line is displaced. With the *O* state, the memory controller need not represent its readiness.

Another alternative is to include cache line buffers in the memory controller,

[2] It does not mean that cache-to-cache transfer is not good. Note that without cache-to-cache transfer, two memory transactions are necessary when a snoop hits on a line with the *M* state: one for writing back the modified block to main memory, the other for reading the same block from memory. Cache-to-cache transfer reduces the number of memory transactions from two to one. Therefore, it clearly has the advantage over non-cache-to-cache transfer.

similar to the snoop-hit buffer discussed in Section 4.3.1. The memory controller receives cache-to-cache transfer data in this buffer temporarily before updating main memory. As long as the buffer space is available, the memory controller is ready to accept data from the snoop-hit processor. It would enable the prompt response for faster cache-to-cache transfer. Like the snoop-hit buffer, if the memory controller is designed to have an ability to compare addresses on the FSB and supply data to a processor when hit on a buffered line, it would further reduce the memory access latency.

Invalidation traffic is also inevitable in multiprocessor systems. In the P-III FSB, the snoop phase is the 5^{th}-stage of the pipeline, and remote processors inform a master processor of the snoop results in the snoop phase. Advancing the snoop phase to an earlier stage bear a potential of reducing the latency. It could reduce the effective cache-to-cache transfer latency, too. However, it requires the faster TAG-lookup in data caches. Deep pipelined-bus and faster bus speed would also help relieve the impact of invalidation traffic even though it complicates the hardware to process requests in shorter time and to accommodate more outstanding transactions.

6.7 Conclusion and Discussion

In this contribution, we measured the intrinsic delay of coherence traffic, and analyzed its efficiency using a novel FPGA approach on a P-III-based server system. The proposed approach eliminates non-deterministic factors in measurements such as the arbitration delay and stall in the pipelined bus. Therefore, it completely isolates the impact of coherence traffic on the system performance. Our case study shows that the performance of the SPEC2000 benchmark assisted by the coherence traffic was actually degraded. The overall execution time of the benchmarks increased up to 191 seconds over 5635 seconds of the baseline with a 256KB cache implemented in the FPGA, where cache-to-cache transfer and invalidation traffic occurred 433.3K/second

and 157.5K/second on average, respectively. Performance degradation is attributed to the following reasons. First, cache-to-cache transfer in the MESI protocol requires main memory to be updated simultaneously. It often delays cache-to-cache transfer since the memory controller would not respond promptly for the update requests because of the busy schedule of the pipelined FSB. Second, in proportion to the number of cache-to-cache transfers, invalidation traffic also increased to a frequency of 157.5K/second on average. Even though invalidation traffic involves no data transfer, it still takes non-negligible amount of time since one FSB slot is needed for each invalidation.

To reduce the latency of coherence traffic, we discussed architectural possibilities. The inclusion of the O state would curtail the latency of cache-to-cache transfer because main memory need not to be updated simultaneously upon cache-to-cache transfer. Cache-line buffers in the memory controller could shorten the latency because the memory controller is able to accept data while buffer spaces are available. Advancing the snoop phase to an earlier stage could reduce both cache-to-cache transfer latency and invalidation traffic latency. Deep pipelined-bus and faster bus speed would also help relieve the impact of invalidation traffic. However, all these architectural enhancements come at the expense of additional hardware. Thus, thorough investigations are necessary to measure the trade-offs.

CHAPTER VII

CONCLUSION

This thesis made three contributions. The first two contributions addressed communication problems among heterogeneous processors in MPSoCs. The third contribution evaluated the coherence traffic efficiency with a novel methodology.

7.1 Cache Coherence Protocol Integration on Shared-bus-based MPSoCs

In this thrust, we provided a generic solution for efficient communication among processor IPs in MPSoCs. We limited our scope to the shared-bus-based MPSoCs and proposed a communication mechanism through cache coherence protocols. The coherence can be maintained with the following integration techniques: the read-to-write conversion and the shared signal assertion/de-assertion. These techniques can be implemented inside wrappers around processors with a minimal hardware cost. The integrated system-wide protocol is composed of the states that heterogeneous processors have in common, except the O state. With the generic integration techniques, SoC designers are able to utilize coherence hardware resources available in the processor IPs.

To enhance the system performance, we proposed the snoop-hit buffer. The snoop-hit buffer shortens the memory access latency by internally supplying data to the master processor when a snoop hits on a line with the M state. We also proposed the RBCC to revitalize the lost states. Depending on the shared area information among processors, the RBCC selectively enables the lost states in processors that have more states in common than the system-wide integrated protocol. We discussed the implication and limitation of integrating processors with no native coherence support.

In such an environment, the interrupt should be used to maintain coherence due to the lack of the snooping capability. The use of the interrupt incurs the hardware deadlock problem that imposes the restrictions on the way locks were implemented by the system and used by the programmer.

The micro-bench and RTOS kernel simulations were performed to measure the advantages of the integration techniques, the snoop-hit buffer, and the RBCC. In the micro-bench simulations, the integration techniques with the snoop-hit buffer showed an 11.8% $\sim$ 57.1% performance improvement over the baseline even in the worst-case. In the best-case, we could achieve a 51% $\sim$ 426% performance improvement. For the evaluation of RBCC, we simulated a part of the Atalanta RTOS kernel. The RBCC with snoop-hit buffer enhanced the performance by a 4.5% $\sim$ 43.0% for the 2T-2T case, by a 15.3% $\sim$ 77.0% for the 2T-16T case.

This outcome of this research provides a generic solution for communication among heterogeneous processors. Nevertheless, the implementation demands elaborate examination since processor IPs could have unequal block sizes and they could operate at different clock frequencies. In reality, in addition to the communication issue, the design of shared-bus-based MPSoCs requires engineering efforts due to the following reasons. It is a challenge to design a shared-bus protocol that provides superset functionalities of distinct processor interfaces. Software-related issues such as RTOS, synchronization, workload parallelization, and memory consistency models should also be taken care of for a successful implementation.

7.2 Cache Coherence Support on Non-shared-bus-based MPSoCs

This research thrust extended our scope of prior work to non-shared-bus-based MPSoCs. Compared to the shared-bus-based MPSoCs, non-shared-bus-based MPSoCs relieve an engineering challenge because processors can take advantage of their native interface protocols. However, the communication among processors is even more

problematic because communication channels are physically separated. To enable and support efficient communication via cache coherence protocols, we proposed the use of a cache coherence-enforced Memory Controller (ccMC) that bridges separate communication links. Depending on the hardware budget in MPSoCs, we proposed two implementation alternatives for the ccMC: the bypass approach and the bookkeeping approach. The bypass approach blindly forwards memory transactions to the opposite-side bus if the condition is met. The bookkeeping approach additionally checks the state information in the processors' caches through the state table in ccMC and then filtrates unnecessary forwarding.

For performance evaluation, we performed the RTOS kernel and micro-bench simulations. In RTOS kernel simulation, the bypass approach showed a 6.65X speedup in the PowerPC755 and ARM9 platforms, with a low miss penalty when running 32 tasks on each processor. In the micro-bench simulation, the bookkeeping approach achieved up to a 10% performance improvement over the bypass approach where DMA traffic occupies 70% of the bandwidth of both buses.

Similar to the shared-bus-based MPSoCs, the design of non-shared-bus-based MPSoCs requires substantial engineering efforts to elaborate the following issues: unequal block sizes, different operating clock frequencies, and software issues. In MPSoC designs, the design methodology plays a key role in reducing the time-to-market design cycle. By providing generic solutions for communication, our research contributed to addressing one of the methodology issues in MPSoCs.

7.3 Evaluation of Coherence Traffic Efficiency

In this study, we measured the intrinsic delay of coherence traffic and analyzed its efficiency using a novel FPGA approach on an Intel server system. Our methodology completely eliminated the non-deterministic factors in measurement such as bus arbitration delay and stall in the pipelined bus.

We implemented a coherence cache in an FPGA on the Intel server system. The FPGA communicated with an Intel Pentium-III processor via the front-side bus. The cache in the FPGA behaves as an L2 cache of a virtual processor on the Intel server system. The implemented cache takes evicted cache lines from the FSB and supplies data when the P-III processor requests it later. By varying the cache sizes from 1KB to 256KB, we measured the execution time of SPEC2000 benchmark and compared it with the baseline. The experiment showed that the performance assisted by the coherence traffic was actually degraded. The overall execution time of the benchmarks increased up to 191 seconds with a 256KB cache in the FPGA, where cache-to-cache transfer and invalidation traffic occurred 433.3K/sec and 157.5K/sec, respectively, on average.

Performance degradation is attributed to the following reasons: inefficient cache-to-cache transfer because of simultaneous memory update, invalidation traffic increases. To reduce the latency of coherence traffic, we discussed architectural possibilities: the inclusion of the O state, cache-line buffers in the memory controller, advancing the snoop phase to an earlier stage, deep pipelined-bus, and faster bus speed. Nevertheless, all these architectural enhancements come at the expense of additional hardware. Therefore, thorough investigations are necessary to measure and evaluate the trade-offs.

Throughout this work, we demonstrated a novel approach using FPGA to carry out architecture evaluation on a real system. There are advantages of using FPGA as an alternative to the conventional simulation-based research using high level languages. First, the entire applications and workloads can be natively executed and emulated on the off-the-shelf system. Such a simulation is very difficult to do by using a cycle-based architecture simulator due to the intolerable simulation times. Second, system activities and overheads are automatically emulated during the native execution, thereby enabling a more accurate and a broader range analysis for the

entire system behavior. In addition, our FPGA approach is highly flexible, so processor architects are able to evaluate the performance of new architecture enhancements and perform pre-silicon verification for future proliferation.

REFERENCES

[1] "AD6525 - GSM Digital Baseband Processor." http://www.analog.com/en/prod/0,2877,AD6525,00.html, November 2006.

[2] "ARC International." http://www.arc.com, November 2006.

[3] "ARM Ltd." http://www.arm.com, November 2006.

[4] "DoMiNo 8650 HDD/DVD recorder processor." http://www.lsi.com/consumer_home/products_home/standard_product_ics/dvd_recorder_combo_solutions/legacy_drm8650/index.html, November 2006.

[5] "Improv Systems." http://www.improvsys.com, November 2006.

[6] "Intel Corp. Pentium(R) Pro Family Developer's Manual, Volume 1: Specifications." http://support.intel.com/design/archives/processors/pro/docs/242690.htm, November 2006.

[7] "Intel IXP2855 Network Processor." http://www.intel.com/design/network/products/npfamily/ixp2855.htm, November 2006.

[8] "International Technology Roadmap for Semiconductors." http://public.itrs.net/, November 2006.

[9] "MIPS Technologies." http://www.mips.com, November 2006.

[10] "Nexperia mobile solutions." http://www.semiconductors.philips.com/products/nexperia/mobile/index.html, November 2006.

[11] "OMAP Platform." http://focus.ti.com/general/docs/wtbu/wtbugencontent.tsp?templateId=6123&navigationId=11988&contentId=4638, November 2006.

[12] "Open Core Protocol International Partnership." http://www.ocpip.org/home, November 2006.

[13] "OpenCores." http://www.opencores.org/, November 2006.

[14] "PARKBENCH (PARallel Kernels and BENCHmarks)." http://www.netlib.org/parkbench/, November 2006.

[15] "SoC Interconnection: Wishbone." http://www.opencores.org/projects.cgi/web/wishbone/wishbone, November 2006.

[16] "Tensilica Inc." http://www.tensilica.com, November 2006.

[17] "TIMA Laboratory." http://tima.imag.fr/, November 2006.

[18] "TPC benchmarks." http://www.tpc.org, November 2006.

[19] "Virtual Component Interface Standard. OCB 2 2.0." http://www.vsi.org/documents/vsiadocuments.htm, November 2006.

[20] "VSI Alliance." http://www.vsi.org/, November 2006.

[21] Agarwal, A., Simoni, R., Hennessy, J., and Horowitz, M., "An Evaluation of Directory Schemes for Cache Coherence," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 280–298, 1988.

[22] Agarwal, A., Bianchini, R., Chaiken, D., Johnson, K. L., Kranz, D., Kubiatowicz, J., Lim, B.-H., Mackenzie, K., , and Yeung, D., "The MIT Alewife Machine: Architecture and Performance," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 2–13, 1995.

[23] Akgul, B. and Mooney, V., "The System-on-a-Chip Lock Cache," *International Journal of Design Automation for Embedded Systems*, vol. 7, no. 1-2, pp. 139–174, 2002.

[24] AMD, "AMD-K6-III Processor Data Sheet." http://www.amd.com/us-en/Processors/TechnicalResources/0,,30_182_739_1102,00.html, November 2006.

[25] AMD, "AMD64 Architecture Programmer's Manual Volume 2: System Programming." http://www.amd.com/us-en/Processors/TechnicalResources/0,,30_182_739_7044,00.html, November 2006.

[26] Archibald, J. and loup Baer, J., "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Transactions on Computer Systems*, vol. 4, pp. 273–298, Nov. 1986.

[27] ARM, "AMBA." http://www.arm.com/products/solutions/AMBAHomePage.html, November 2006.

[28] Baer, J.-L. and Chen, T.-F., "An Effective On-chip Preloading Scheme to Reduce Data Access Penalty," in *Supercomputing '91*, 1991.

[29] Baghdadi, A., Lyonnard, D., Zergainoh, N., and Jerraya, A., "An Efficient Architecture Model for Systematic Design of Application-Specific Multiprocessor SoC," in *Proceedings of the Design Automation and Test in Europe (DATE)*, pp. 55–63, 2001.

[30] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, D., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., and

Weeratunga, S. K., "The NAS Parallel Benchmarks," *The International Journal of Supercomputer Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[31] Barroso, L. A., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, R., and Verghese, B., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 282–293, 2000.

[32] Baskett, F., Jermoluk, T., and Solomon, D., "The 4D-MP Graphics Superworkstation: Computing + Graphics = 40 MIPS + 40 MFLOPS and 100,000 Lighted Polygons per Second," in *Proceedings of the 33rd IEEE Computer Society International Conference - COMPCON 88*, pp. 468–471, Feb. 1988.

[33] Bechini, A., Foglia, P., and Prete, C. A., "Fine-Grain Design Space Exploration for a Cartographic SoC Multiprocessor," *ACM SigArch Computer Architecture News*, vol. 31, no. 1, pp. 85–92, 2003.

[34] Becker, M. K., Allen, M. S., Moore, C. R., Muhich, J. S., and Tuttle, D. P., "The PowerPC 601 Microprocessor," *IEEE MICRO*, pp. 54–68, September/October 1993.

[35] Berry, M., Chen, D., Koss, P., Kuck, D., Lo, S., Pang, Y., Pointer, L., Roloff, R., Sameh, A., Clementi, E., Chin, S., Scheider, D., Fox, G., Messina, P., Walker, D., Hsiung, C., Schwarzmeier, J., Lue, K., Orszag, S., Seidl, F., Johnson, O., Goodrum, R., and Martin, J., "The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers," *The International Journal of Supercomputer Applications*, vol. 3, no. 3, pp. 5–40, 1989.

[36] Binkert, N. L., Hallnor, E. G., and Reinhardt, S. K., "Network-Oriented Full-System Simulation using M5," in *Proceedings of the 6th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, February 2003.

[37] Callahan, D., Kennedy, K., and Porterfield, A., "Software Prefetching," in *Proceedings of ASPLOS-IV*, 1991.

[38] Cesrio, W., Baghdadi, A., Gauthier, L., Lyonnard, D., Nicolescu, G., Paviot, Y., Yoo, S., and Jerraya, A., "Component-Based Design Approach for Multicore SoCs," in *Proceedings of the 39nd Design Automation Conference*, pp. 789–794, 2002.

[39] Chaiken, D., Fields, C., Kurihara, K., and Agarwal, A., "Directory-Based Cache Coherence in Large-Scale Multiprocessors," *IEEE Computer*, pp. 49–58, June 1990.

[40] Chaiken, D., Kubiatowicz, J., and Agarwal, A., "LimitLESS Directories: A Scalable Cache Coherence Scheme," in *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 224–234, 1991.

[41] Choi, J., Dongarra, J. J., Pozo, R., and Walker, D. W., "ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers," in *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pp. 120–127, 1992.

[42] Clark, R. and Alnes, K., "An SCI Chipset and Adapter," in *Proceedings of Hot Interconnects IV*, pp. 221–235, 1996.

[43] Cordan, B., "An Efficient Bus Architecture for System-on-a-Chip Design," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, pp. 623–626, 1999.

[44] Culler, D. E., Singh, J. P., and Gupta, A., *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1999.

[45] Cyr, G., Bois, G., and Aboulhamid, M., "Generation of Processor Interface for SoC using Standard Communication Protocol," *IEE Proceedings of Computers and Digital Techniques*, vol. 151, pp. 367–376, September 2004.

[46] D'silva, V., Ramesh, S., and Sowmya, A., "Bridge Over Troubled Wrappers : Automated Interface Synthesis," in *Proceedings of the 17th International Conference on VLSI Design*, pp. 189–194, 2004.

[47] Dubois, M., Jeong, J., Song, Y. H., and Moga, A., "Rapid Hardware Prototyping on RPM-2," *IEEE Design and Test of Computers*, vol. 15, no. 3, pp. 112–118, 1998.

[48] Dubois, M., Skeppstedt, J., Ricciulli, L., Ramamurthy, K., and Stenstrom, P., "The Detection and Elimination of Useless Misses in Multiprocessors," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 88–97, 1993.

[49] Eggers, S. J. and Katz, R. H., "A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 373–392, 1988.

[50] Eggers, S. J. and Katz, R. H., "Evaluating the Performance of Four Snooping Cache Coherency Protocols," in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pp. 2–15, 1989.

[51] Eggers, S. J. and Katz, R. H., "The Effect of Sharing on the Cache and Bus Performance of Parallel Programs," in *Proceedings of the 3rd Symposium*

on Architectural Support for Programming Languages and Operating Systems, pp. 257–270, 1989.

[52] Frank, S. J., "Tightly Coupled Multiprocessor System Speeds Memory -access Times," *Electronics*, vol. 57, pp. 164–169, January 1984.

[53] Fu, J. W. C., Patel, J. H., and Janssens, B. L., "Stride Directed Prefetching in Scalar Processors," in *Proceedings of MICRO-25*, 1992.

[54] Gerin, P., Yoo, S., Nicolescu, G., and Jerraya, A. A., "Scalable and Flexible Cosimulation of SoC Designs with Heterogeneous Multi-Processor Target Architectures," pp. 63–68, 2001.

[55] Gharachorloo, K., Sharma, M., Steely, S., and Doren, S. V., "Architecture and Design of AlphaServer GS320," in *Proceedings of the 9th Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 13–24, 2000.

[56] Gharsalli, F., Meftali, S., Rousseau, F., and Jerraya, A. A., "Automatic Generation of Embedded Memory Wrapper for Multiprocessor SoC," in *Proceedings of the 39nd Design Automation Conference*, pp. 596–601, 2002.

[57] Goodman, J. R., "Using Cache Memory to Reduce Processor-Memory Traffic," in *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pp. 124–131, 1983.

[58] Grahn, H., Stenstrom, P., and Dubois, M., "Implementation and Evaluation of Update-Based Cache Protocols under Relaxed Memory Consistency Models," *Future Generation Computer Systems*, vol. 11, pp. 247–271, June 1995.

[59] Gupta, A., Weber, W.-D., and Mowry, T., "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes," in *Proceedings of the International Conference on Parallel Processing*, pp. 312–321, 1990.

[60] Hong, J., Nurvitadhi, E., and Lu, S.-L. L., "Design, Implementation, and Verification of Active Cache Emulator (ACE)," in *Proceedings of the 14th annual International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 63–72, February 2006.

[61] IBM, "CoreConnect Bus Architecture." http://www-03.ibm.com/chips/products/coreconnect/, November 2006.

[62] Intel, "Intel 64 and IA32 Architecture Software Developer's Manual." http://developer.intel.com/design/processor/manuals/253668.pdf, November 2006.

[63] Joseph, D. and Grunwald, D., "Prefetching using Markov Predictors," in *Proceedings of ISCA-24*, 1997.

[64] Jouppi, N. P., "Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers," in *Proceedings of the 17th annual International Symposium on Computer Architecture*, pp. 364–373, July 1990.

[65] Karlin, A. R., Manasse, M. S., Rudolph, L., and Sleator, D., "Competitive Snoopy Caching," *Algorithmica*, vol. 3, pp. 70–119, 1988.

[66] Katz, R. H., Eggers, S. J., Wood, D. A., Perkins, C. L., and Sheldon, R. G., "Implementing a Cache Consistency Protocol," in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pp. 276–283, 1985.

[67] Kroft, D., "Lockup-Free Instruction Fetch/Prefetch Cache Organization," in *Proceedings of ISCA-08*, May 1981.

[68] Kuskin, J., Ofelt, D., Heinrich, M., Heinlein, J., Simoni, R., Chapin, K. G. J., Nakahira, D., Baxter, J., Horowitz, M., Gupta, A., Rosenblum, M., and Hennessy, J., "The Stanford FLASH Multiprocessor," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 302–324, 1994.

[69] Lahiri, K., Raghunathan, A., and Lakshminarayana, G., "LOTTERYBUS: A New High-Performance Communication Architecture for System-on-Chip Designs," in *Proceedings of the 38nd Design Automation Conference*, pp. 15–20, 2001.

[70] Laudon, J. and Lenoski, D., "The SGI Origin: A ccNUMA Highly Scalable Server," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 241–240, 1997.

[71] Leijten, J. A., van Meerbergen, J., Timmer, A. H., and Jess, J. A., "PROPHID: A Heterogeneous Multi-Processor Architecture for Multimedia," in *Proceedings of the 1997 International Conference on Computer Design (ICCD'97)*, pp. 164–169, 1997.

[72] Lenoski, D., Laudon, J., Gharachorloo, K., Gupta, A., and Hennessy, J., "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 148–159, 1990.

[73] Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.-D., Gupta, A., Hennessy, J., Horowitz, M., and Lam, M. S., "The Stanford Dash Multiprocessor," *IEEE Computer*, pp. 63–79, March 1992.

[74] Lovett, T. and Clapp, R., "STiNG: A CC-NUMA Computer System for the Commercial Marketplace," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp. 308–317, 1996.

[75] Lyonnard, D., Yoo, S., Baghdadi, A., and Jerraya, A. A., "Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip," in *Proceedings of the 38nd Design Automation Conference*, pp. 518–523, 2001.

[76] Magnusson, P., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., and Werner, B., "Simics: A Full System Simulation Platform," *IEEE Computer*, Feb. 2002.

[77] Martin, M. M., Hill, M. D., and Wood, D. A., "Token Coherence: Decoupling Performance and Correctness," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pp. 182–193, June 2003.

[78] McCreight, E., "The Dragon Computer System: An Early Overview," tech. rep., Xerox Corporation, Sept. 1984.

[79] Mentor Graphics, "Hardware/Software Co-Verification: Seamless." http://www.mentor.com/seamless, November 2006.

[80] Mentor Graphics, "ModelSim SE." http://www.mentor.com/products/fv/digital_verification/modelsim_se/index.cfm, November 2006.

[81] Michael, M. M., Nanda, A. K., Lim, B.-H., and Scott, M. L., "Coherence Controller Architectures for SMP-Based CC-NUMA Multiprocessors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 219–228, 1997.

[82] Mowry, T. C., Lam, M. S., and Gupta, A., "Design and Evaluation of a Compiler Algorithm for Prefetching," in *Proc. of ASPLOS-V*, 1992.

[83] Nanda, A., Mak, K.-K., Sugavanam, K., Sahoo, R. K., Soundararajan, V., and Smith, T. B., "MemorIES: A Programmable, Real-Time Hardware Emulation Tool for Multiprocessor Server Design," in *Proc. of ASPLOS-9*, pp. 37–48, November 2000.

[84] Nguyen, A.-T., Michael, M., Sharma, A., and Torrellas, J., "The Augmint Multiprocessor Simulation Toolkit for Intel x86 Architectures," in *Proceedings of 1996 International Conference on Computer Desig n*, October 1996.

[85] O'Krafka, B. W. and Newton, A. R., "An Empirical Evaluation of Two Memory-Efficient Directory Methods," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 148–159, 1990.

[86] Pai, V., Ranganathan, P., and Adve, S., "RSIM Reference Manual, version 1.0," *IEEE Technical Committee on Computer Architecture Newsletter*, Fall 1997.

[87] Palmchip, "SoC Platform Architecture." http://www.palmchip.com/coreframe_downloads.asp, November 2006.

[88] Papamarcos, M. and Patel, J., "A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories," in *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pp. 348–354, 1984.

[89] Passerone, R., Rowson, J. A., and Sangiovanni-Vincentelli, A., "Automatic Synthesis of Interfaces between Incompatible Protocols," pp. 8–13, 1998.

[90] Patel, J., "Analysis of Multiprocessors with Private Cache Memories," *IEEE Transactions on Computers*, vol. 31, pp. 296–304, April 1982.

[91] Rajawat, A., Balakrishnan, M., and Kumar, A., "Interface Synthesis: Issues and Approaches," in *Proceedings of the 13th International Conference on VLSI Design*, pp. 92–97, 2000.

[92] Rosenblum, M., Herrod, S. A., Witchel, E., and Gupta, A., "Complete Computer System Simulation: The SimOS approach," *IEEE Parallel and Distributed Technology*, vol. 3, no. 4, pp. 34–43, 1995.

[93] Roth, A., Moshovos, A., and Sohi, G. S., "Dependence Based Prefetching for Linked Data Structures," in *Proceedings of ASPLOS-VIII*, 1998.

[94] Santhanam, V., Gornish, E. H., and Hsu, W.-C., "Data Prefetching on the HP PA-8000," in *Proc. of ISCA-24*, 1997.

[95] Sciuto, D., F. Salice, L. P., and Fornaciari, W., "Metrics for Design Space Exploration of Heterogeneous Multiprocessor Embedded Systems," in *CODES'02*, 2002.

[96] Silberschatz, A., Galvin, P. B., and Gagne, G., *Operating System Concepts*. Wiley, 2002.

[97] Singh, J., Weber, W.-D., and Gupta, A., "SPLASH: Stanford Parallel Applications for Shared-Memory," *Computer Architecture News*, vol. 20, pp. 4–44, March 1992.

[98] Sonics, "Sonics μNetworks Technical Overview." http://www.sonicsinc.com/, November 2006.

[99] Suh, T., Kim, D., and Lee, H.-H. S., "Cache Coherence Support for Non-Shared Bus Architecture on Heterogeneous MPSoCs," in *Proceedings of the 42nd Design Automation Conference*, pp. 553–558, June 2005.

[100] Suh, T., Lee, H.-H. S., and Blough, D. M., "Supporting Cache Coherence in Heterogeneous Multiprocessor Systems," in *Proceedings of the Design Automation and Test in Europe (DATE)*, pp. 1150–1155, Feb 2003.

[101] Suh, T., Lee, H.-H. S., and Blough, D. M., "Integrating Cache Coherence Protocols for Heterogeneous Multiprocessor Systems, Part 1," *IEEE MICRO*, pp. 33–41, July/August 2004.

[102] Suh, T., Lee, H.-H. S., and Blough, D. M., "Integrating Cache Coherence Protocols for Heterogeneous Multiprocessor Systems, Part 2," *IEEE MICRO*, pp. 70–78, September/October 2004.

[103] Sun, D.-S. and Blough, D. M., "I/O Threads: A Novel I/O Approach for System-on-a-Chip Networking," tech. rep., CERCS, Georgia Institute of Technology, 2004.

[104] Sun, D.-S., Blough, D. M., and Mooney, V., "Atalanta: A New Multiprocessor RTOS Kernel for System-on-a-Chip Applications," Tech. Rep. GIT-CC-02-19, CERCS, Georgia Institute of Technology, 2002.

[105] SUN Microsystems, "UltraSPARC IV+ User's Manual Supplement." http://www.sun.com/processors/documentation.html, November 2006.

[106] Sweazey, P. and Smith, A. J., "A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus," in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pp. 414–423, 1986.

[107] Thacker, C. P., Stewart, L. C., and Satterthwaite, E. H., "Firefly: A multiprocessor workstation. IEEE Transactions on Computers," *IEEE Transactions on Computers*, vol. 37, pp. 909–920, August 1988.

[108] Thekkath, R., Singh, A. P., Singh, J. P., and Hennessy, J., "An Application-Driven Evaluation of the Convex Exemplar SP-1200," in *Proceedings of the International Parallel Processing Symposium*, 1997.

[109] Watson, M. and Flanagan, J. K., "Simulating L3 Caches in Real Time Using Hardware Accelerated Cache Simulation (HACS): a Case Study with SPECint 2000," in *Proceedings of 14th Symp. on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 108–114, October 2002.

[110] Wolf, W., "The Future of Multiprocessor Systems-on-Chips," in *Proceedings of the 41nd Design Automation Conference*, pp. 681–685, 2004.

[111] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A., "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 24–36, 1995.

[112] Wulf, W. and McKee, S., "Hitting the Memory Wall: Implications of the Obvious," *ACM SIGArch Computer Architecture News*, vol. 23, pp. 20–24, March 1995.

[113] Xilinx, "ISE Foundation." http://www.xilinx.com/ise/logic_design_prod/foundation.htm, November 2006.

[114] Xilinx, "Virtex-4 Multi-Platform FPGA." http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/index.htm, November 2006.

[115] Xilinx, "Virtex-II Platform FPGAs." http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex_ii_platform_fpgas/index.htm, November 2006.

[116] Yeung, C., Matthews, G., Morris, J., Haverinen, A., and Zaidi, J., "Standard Bus vs. Bus Wrapper:. What is the Best Solution for Future SoC Integration?," in *Proceedings of the Design Automation and Test in Europe (DATE)*, p. 776, 2001.

[117] Yoo, S. and Jerraya, A. A., "Introduction to Hardware Abstraction Layers for SoC," in *Proceedings of the Design Automation and Test in Europe (DATE)*, 2003.

[118] Yoo, S., Nicolescu, G., Lyonnard, D., Baghdadi, A., and Jerraya, A. A., "A Generic Wrapper Architecture for Multi-Processor SoC Cosimulation and Design," in *Proceedings of the ninth international symposium on Hardware/software codesign*, pp. 195–200, 2001.

[119] Yoo, S., Youssef, M.-W., Bouchhima, A., Jerraya, A. A., and Diaz-Nava, M., "Multi-Processor SoC Design Methodology Using a Concept of Two-Layer Hardware-Dependent Software," in *Proceedings of the Design Automation and Test in Europe (DATE)*, 2004.

[120] Yoon, H.-M., Park, G.-H., Lee, K.-W., Han, T.-D., Kim, S.-D., and Yang, S.-B., "Reconfigurable Address Collector and Flying Cache Simulator," in *HPC-ASIA '97: Proceedings of the High-Performance Computing on the Information Superhighway, HPC-Asia '97*, pp. 552–556, 1997.

VITA

Taeweon Suh was born in Seoul, South Korea. He received the bachelor's degree in electrical engineering at Korea University and the master's degree in electronics engineering at Seoul National University. During his master's study, he focused on microelectronics research, in particular, the electron-beam lithography. After the graduation, he joined the ASIC center in LG Electronics Research Institute, where he was involved in the embedded processor design project. His professional career continued in Hynix Semiconductor, where he participated in a SoC design based on a Java processor. He started his long-wished Ph.D. study late after working almost seven years in industry. He joined Georgia Institute of Technology in fall 2001. During his Ph.D. study, he worked at the Microprocessor Technology Lab of Intel Corporation in Santa Clara, CA as a research intern in summer 2004. In 2005 and 2006, he also worked as a research intern at the Intel's Microprocessor Technology Lab in Hillsboro, OR. He is happily married to Soojung Lee.