

SMART MEMORIES: A RECONFIGURABLE MEMORY SYSTEM
ARCHITECTURE

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Amin Firoozshahian

December 2008

© Copyright by Amin Firoozshahian 2009
All Rights Reserved

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as dissertation for the degree of Doctor of Philosophy.

(Mark Horowitz) Principal Advisor

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as dissertation for the degree of Doctor of Philosophy.

(Christos Kozyrakis)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as dissertation for the degree of Doctor of Philosophy.

(Kunle Olukotun)

Approved for the University Committee on Graduate Studies

ABSTRACT

The move to chip level multiprocessors (CMP), where multiple processor cores are integrated on the same die, fundamentally shifts the focus and complexity of the systems towards the memory subsystem. The memory subsystem serves as the primary means for data storage, sharing and communication that processors need to perform meaningful computations. Moreover, appearance of innovative proposals for multiprocessor memory systems, such as streaming and transactional memory, diversifies the semantics requirements that need to be provided in the memory system implementation. In this dissertation we observe that while having different semantics, all major memory models in today's multiprocessors rely on very similar hardware resources and operations at the implementation level. The different memory access semantics are generated by altering how the primitive hardware operations are composed. We propose a universal memory system architecture that implements the shared resources and exports the common operations, enabling a user to implement different memory protocols by "programming" the operations that occur in the memory system. The system consists of storage elements for storing data and state information, communication channels for performing data transfers and exchanging control messages, and associated controllers which sequencing and carry out control operations. We present Smart Memories as a concrete example of such reconfigurable memory system and discuss its architecture and hardware mechanisms that provide flexibility. We also explain how protocols can be mapped to this hardware substrate by providing a simple example. Our study shows that the performance impact of the flexible hardware mechanisms are generally small, less than 20% compared to an ideal memory system, in almost all cases across three different memory models. The impact on the physical aspects of the system is more significant, consuming 60% more dynamic power and twice the area in configurable controllers compared to controllers specialized for a specific protocol.

ACKNOWLEDGMENTS

During on my long years at Stanford I had the opportunity to work and collaborate with many wonderful people and without their help and support this journey would not have been possible. My deepest and sincere thanks of course is devoted to my advisor, professor Mark Horowitz, for his guidance, advice, support, and specially his extraordinary patience and trust in me during the years. I certainly consider it a privilege to work under his supervision for all my years at Stanford. But most importantly I am grateful for his wise mentorship and his teachings, in always considering the most important questions to answer, whether it being a homework problem, a design project, or living your life.

I would also like to thank my co-advisor, professor Christos Kozyrakis, for all the helpful guidance and discussions over the course of the program. Being it the most recent architecture paper or yesterday's soccer game, I have always found his passionate and enthusiastic explanations a source of limitless inspiration. I like to thank professor Kunle Olukotun for serving in my reading and defense committee, his class lectures, and also for collaborations with his research group during the course of our project.

I want to thank professor Balaji Prabhakar for accepting to be the chair of my defense committee and for all the wonderful things I learned in his classes and during the short research collaboration we had. My thanks are also extended to professor Nick McKeown, not only for helping me when coming to Stanford, but also for his continuous support during the course of the Ph.D. program and for all that I learned from his lectures, talks and classes.

While working on the Smart Memories project I had the opportunity to collaborate with many wonderful people without whom this project would have not been possible. Most

importantly I want to thank my friend, colleague and officemate, Alex Solomatnikov, for all the long years of working shoulder to shoulder. I learned a lot from him during the years and his hard work, absolute dedication and integrity has always been the perfect example for me. I also want to thank him for all our arguments and disagreements and sometime disputes, without which this project would have not been as much fun.

I want to thank the rest of the Smart Memories team, Kyle Kelly, Megan Wachs, Wajahat Qadeer, Rehan Hameed, Stephen Richardson, Don Stark and specially Ofer Shacham for his amazing persistence and full devotedness to the project. I also would like to thank Zain Asgar and Han Chen, our backend team, whom without their hard work and dedication the test chip would have not been completed.

Finally I also want to thank my friends and family, specially Mom, Dad and my sister Yalda, for their unconditional love and support during all these years, without which I would have not been able to come to Stanford. Last but not least, I would like to thank my lovely wife Sara, for her infinite love, profound understanding and extraordinary patience during my years as a student. It would certainly not be possible to complete this journey without having her by my side.

TABLE OF CONTENTS

Abstract.....	v
Acknowledgments.....	vii
List of tables.....	xiii
List of figures.....	xv
1. Introduction.....	1
2. Background and motivation.....	5
2.1. Multicore processors and complexity of memory system.....	5
2.2. Programming models and memory access semantics.....	8
2.3. Characteristics of major memory models.....	10
2.3.1. Streaming memory systems.....	10
2.3.2. Coherent shared memory.....	12
2.3.3. Transactional memory.....	15
2.4. Commonalities between models.....	21
3. A universal architecture for memory systems.....	25
3.1. A brief review of memory system tasks.....	26
3.2. General architecture.....	29
3.2.1. Storage elements.....	31
3.2.2. Communication channels.....	32
3.2.3. Associated control logic.....	33
3.3. Controllers.....	36
3.3.1. Organization.....	36
3.3.2. Instruction set architecture.....	38
3.3.2.1. Internal State.....	38
3.3.2.2. Instructions.....	40
3.3.2.3. Address mapping modes.....	43
3.4. Sequence of operations.....	44
3.4.1. Processor interface logic.....	44

3.4.2. Highest-level controller	46
3.4.3. Lowest-level controllers.....	48
3.5. Summary	49
4. Smart Memories, a reconfigurable memory system architecture	51
4.1. Overall architecture.....	52
4.2. Processors	53
4.3. Storage elements	57
4.3.1. Reconfigurable memory mat.....	57
4.3.2. Main off-chip memory	64
4.3.3. Physical address map	65
4.4. Tile crossbar.....	68
4.5. Processor interface logic	70
4.5.1. Specifying address translation and mapping.....	72
4.5.2. Defining memory operations	76
4.5.3. Detecting access faults	78
4.5.4. Programmable request messages	79
4.5.5. Interrupt interface.....	81
4.6. Protocol controller	82
4.6.1. Organization.....	83
4.6.2. Sequencing of actions	84
4.6.3. Supported operations	85
4.6.4. Status holding registers and data buffers	91
4.7. Main memory controller	96
4.7.1. Organization.....	96
4.8. Mapping memory protocols.....	98
4.8.1. Streaming memory system.....	99
4.8.2. Shared memory system.....	101
4.8.3. Transactional memory system	102
4.9. Summary	103

5. Evaluation	107
5.1. Test chip implementation results	107
5.2. Performance overhead	110
5.2.1. Coherent shared memory	110
5.2.2. Streaming	114
5.2.3. Transactional coherence and consistency	117
5.3. Physical overhead	121
5.4. Summary	124
6. Conclusions.....	125
Appendix A: SMASH interconnection network.....	131
A.1. Inter-Quad network.....	132
A.2. Network switch architecture	133
A.3. Enforcing priorities	136
A.4. Broadcast / multi-cast capabilities	137
Appendix B: Implementing a simple protocol.....	139
B.1. Allocating resources	139
B.1.1. State and data storages	140
B.1.2. Address translation and mapping	143
B.2. Defining memory accesses.....	144
B.2.1. Accesses to local memory mats	145
B.2.2. Accesses to main memory.....	151
B.3. Communication messages	151
B.3.1. Defining communication messages.....	152
B.3.2. Specifying priorities	153
B.3.3. Programming protocol controller	155
B.3.4. Programming main memory controller	161
B.4. Summary	163
Bibliography	165

LIST OF TABLES

<i>Number</i>	<i>Page</i>
Table 2-1: Similarities between different protocol actions	22
Table 3-1: Controller instruction set (ISA)	41
Table 3-2: Functional description of ISA instructions	42
Table 4-1: Memory mat data array opcodes	59
Table 4-2: Memory mat control array opcodes	61
Table 4-3: Cache access signals generated by address slicer	75
Table 4-4: Fields of request messages to protocol controller	79
Table 4-5: Information fields in MSHR	93
Table 4-6: Information fields in USHR	94
Table 4-7: Communication messages for implemented memory models	100
Table 5-1: Test chip specifications	108
Table 5-2: Coherent shared memory benchmarks	111
Table 5-3: System parameters for coherent shared memory model	111
Table 5-4: Streaming benchmarks	115
Table 5-5: System parameters for streaming memory model	115
Table 5-6: System parameters for hardware transactional memory model	118
Table 5-7: Transactional memory benchmarks	118
Table 5-8: Performance overhead of reconfigurable controllers	120
Table 5-9: Power comparison for baseline and specialized controllers	123
Table B-1: Cache parameters for example configuration	140
Table B-2 : Processor interface operations on memory mats (cached)	146
Table B-3: Processor interface operations on memory mats (un-cached)	147
Table B-4: Protocol controller operations on tag mats	148
Table B-5: Protocol controller operations on data mats	148
Table B-6: Success/Failure conditions for LSU operations on caches	150

Table B-7: Messages between processor interface and protocol controller	152
Table B-8: Messages between protocol controller and main memory controller	153
Table B-9: Fields of messages between protocol and main memory controller	153
Table B-10: Breakdown of message handling steps in protocol controller	156
Table B-11: Breakdown of message handling steps in main memory controller	162

LIST OF FIGURES

<i>Number</i>	<i>Page</i>
Figure 2-1: SpecInt performance numbers	6
Figure 2-2: SpecFP performance numbers	7
Figure 2-3: Example streaming application, stereo depth extraction	10
Figure 3-1: High-level architecture of the memory system	30
Figure 3-2: Finding data copies by searching controller's sub-trees	35
Figure 3-3: Internal organization of a controller.....	37
Figure 3-4: Organization of processor interface	38
Figure 3-5: Information fields in SHR and data buffer entries	39
Figure 3-6: Processing a memory access in processor's interface logic.....	46
Figure 3-7: Steps for handling request/reply messages in L1 controller	47
Figure 3-8: Steps for handling request/reply messages in main memory controller	48
Figure 4-1: Smart Memories hierarchical architecture	52
Figure 4-2: Xtensa LX2 processor architecture, from [78].....	54
Figure 4-3: Internal organization of memory mat.....	58
Figure 4-4: Logical OR operation in IMCN	63
Figure 4-5: Virtual and physical address spaces.....	66
Figure 4-6: Mapping of memory mats in physical address space.....	67
Figure 4-7: Mapping of configuration registers in physical address space	68
Figure 4-8: Tile crossbar	69
Figure 4-9: Processor interface logic	71
Figure 4-10: Processor's segment table	73
Figure 4-11: An example two-way cache configuration.....	75
Figure 4-12: Inputs and outputs of the opcode translation mechanism	77
Figure 4-13: Detecting success or failure of a memory operation.....	79
Figure 4-14: Interrupt interface to processors.....	82

Figure 4-15: Internal organization of Quad’s Protocol Controller	84
Figure 4-16: Conceptual execution model of the protocol controller.....	85
Figure 4-17: MSHR structure	92
Figure 4-18: USHR structure	93
Figure 4-19: Line buffer structure.....	95
Figure 4-20: Internal organization of main memory controller	97
Figure 5-1: SMASH die plot.....	108
Figure 5-2: SMASH test chip area breakdown	109
Figure 5-3: Area breakdown for Tile and local memory controller.....	109
Figure 5-4: Performance impact in coherent shared memory model (kernels)	112
Figure 5-5: Performance impact in coherent shared memory model (applications).....	113
Figure 5-6: Breakdown of execution time (shared memory benchmarks)	114
Figure 5-7: Performance impact in streaming model	116
Figure 5-8: Performance impact in transactional memory model	119
Figure 5-9: Breakdown of execution time (TM benchmarks)	120
Figure 5-10: Area comparison for protocol controller functional units.....	122
Figure 5-11: Comparison of total area between controllers.....	123
Figure A-1: Star interconnection topology in SMASH	132
Figure A-2: Organization and connections of the network switch	133
Figure A-3: Input port of the network switch	134
Figure A-4: Output port of network switch	135
Figure A-5: Network switch scheduling logic	136
Figure B-1: Mapping and encoding of state information.....	141
Figure B-2: Example instruction cache settings	142
Figure B-3: Example data cache settings.....	143
Figure B-4: Example setting for segment table (address translation).....	144
Figure B-5: Flow of operations for processing messages in protocol controller.....	158
Figure B-6: P-Unit subroutines.....	159
Figure B-7: T-Unit subroutines (cached and un-cached parts).....	159

Figure B-8: S-Unit subroutines	160
Figure B-9: D-Unit subroutines	160
Figure B-10: N-Unit subroutines (receiver and transmitter).....	161
Figure B-11: Flow of operations for processing messages in main memory controller .	162

1. INTRODUCTION

Since the beginning days of computers, applications have always needed large amounts of fast memory. However, as the memory quantity increased, so did the application demands. With the increasing gap between the operational speed of processors and memory the only feasible way of creating an illusion of large, fast memory was by organizing it into multiple levels of hierarchy. Therefore, in addition to storing application data, optimal transfer of the data between levels of the hierarchy has also been one of the crucial tasks of the memory system and has been studied extensively in the literature.

The appearance of parallel machines, and most recently with the emergence of chip-multiprocessors, has further increased the importance of memory system design since it serves as the primary means for data communication and sharing between multiple processor cores. This communication and sharing not only increases the performance requirements of the memory, but also interacts in many ways with the memory hierarchy that was created to improve the effective performance of the memory. Additional mechanisms are required to provide a consistent view of the shared address space and guarantee orderly completion of memory accesses, in addition to performing data transfers between levels of hierarchy. These mechanisms in the memory system have to follow a specific set of rules to provide such guarantees, usually referred to as a *memory access protocol*.

Memory protocols usually are exposed to the software in the form of a *memory model*, which is the conceptual view of the shared address space and its operational semantics as seen by processors. The memory model in turn is dictated by the system's *programming model*. Besides the traditional sequential programming model for single thread processors, various programming models have been proposed by researchers to simplify the difficult task of developing parallel programs. Each programming model usually has its own view of the underlying memory and hence dictates its specific

memory access semantics. These semantics can vary from a simple, software managed memory hierarchy to very complex set of rules for providing atomicity and isolation guarantees between operations of concurrent threads and in memory system.

The distributed concurrent nature of these memory systems makes their implementation in general a very challenging and expensive task. This complexity is compounded if a machine must support more than one memory model. Interestingly, while the semantics required by various models are diverse, this dissertation will show that they have considerable similarities at the hardware implementation level. This critical observation motivates the design and development of a universal memory system architecture that can be “programmed” or “configured” after construction, in order to efficiently support implementation of existing memory models, and hopefully future ones, on the same hardware substrate.

This dissertation proposes an abstract architecture for a universal memory system, recognizing and identifying necessary resources and operations. It also proposes an abstract instruction set architecture for the operations supported by the memory system controllers for implementing memory access protocols. In order to demonstrate the feasibility and effectiveness of this approach to memory system design, the dissertation presents the design and implementation of the memory system in the Smart Memories multiprocessor, focusing on the reconfigurable controllers that implement the proposed abstract instruction set. Finally, it evaluates the performance impact of the reconfigurable mechanisms added to the memory system, as well as the physical overheads of constructing configurable controllers.

To show the commonalities between hardware model implementations, Chapter 2 reviews some of the important memory models implemented in today’s multiprocessor systems in more detail, and highlights the hardware mechanisms that are used. Using this information, Chapter 3 proposes a universal memory system architecture constructed by implementing the set of common resources and operations discussed in Chapter 2. It explains the functionality of the resources and the operations they export,

providing a set of basic, abstract operations that the memory system can support. Combining and composing these operations in different sequences implements a large class of memory protocols.

To make this design more concrete, in Chapter 4 we present the Smart Memories memory system architecture as an instance of the universal memory system. We discuss system's organization and components in detail and explain the flexible hardware mechanisms embedded within different system components to provide the discussed abstract operations. In order to provide more insight and illustrate the capabilities of the system, Appendix B discusses the details of implementing a simple coherence protocol on top of the Smart Memories hardware. It explains the steps of sketching the protocol as a set of operations on local resources and communication messages exchanged between different levels of hierarchy, and illustrates how to carry out those operations on the designated resources.

In Chapter 5 we discuss the Smart Memories test chip, SMASH, and its characteristics. We evaluate the performance impact of the reconfigurable mechanisms embedded in the architecture to provide the flexibility in composing and sequencing of the operations, as well as their effect on physical characteristics of the system, namely area and power. Finally Chapter 6 presents our conclusions and provides directions for future research.

2. BACKGROUND AND MOTIVATION

The memory subsystem is a crucial part of any computer system. In addition to managing data locality to provide the illusion of a large fast memory, it also serves as the main infrastructure for communication and data sharing in today's multiprocessors. In this chapter we discuss how the integration of more processor cores in today's CMP systems affects the memory subsystem design, as well as the implications of innovative parallel programming models on the system's memory access semantics. Next, we review the characteristics of major memory models supported in the existing multicore processors, trying to understand the underlying hardware mechanisms used in their implementation. We will see a considerable similarity between these memory systems, both in terms of low-level hardware resources and operations. The commonalities in resources and operations serve as the bases for constructing a universal memory system architecture, as presented in next chapter.

2.1. MULTICORE PROCESSORS AND COMPLEXITY OF MEMORY SYSTEM

In the past decades, number of transistors in the integrated circuits has been increasing according to Moore's Law. For microprocessor systems, this increased quantity has been successfully converted in to increased system performance, resulting in exceptional advances in the microprocessor and in general, in digital systems industry. Major reasons for this increased performance have been three-fold:

- Scaling of VLSI technology has made transistor's operational speed faster, resulting in faster clock cycles for the devices in successive generations.
- Number of pipeline stages in the modern processors has been increased, decreasing number of logic gates per pipeline stage, furthermore enabling faster clock frequencies for microprocessors.

- By using architectural techniques such as wider issue windows and out-of-order executions, modern microprocessors have been successfully extracting more and more instruction level parallelism (ILP) from the applications, hence reducing total number of clock cycles per application.

As a result of this steep performance increase, the traditional sequential programming model has remained unchanged for a long time.

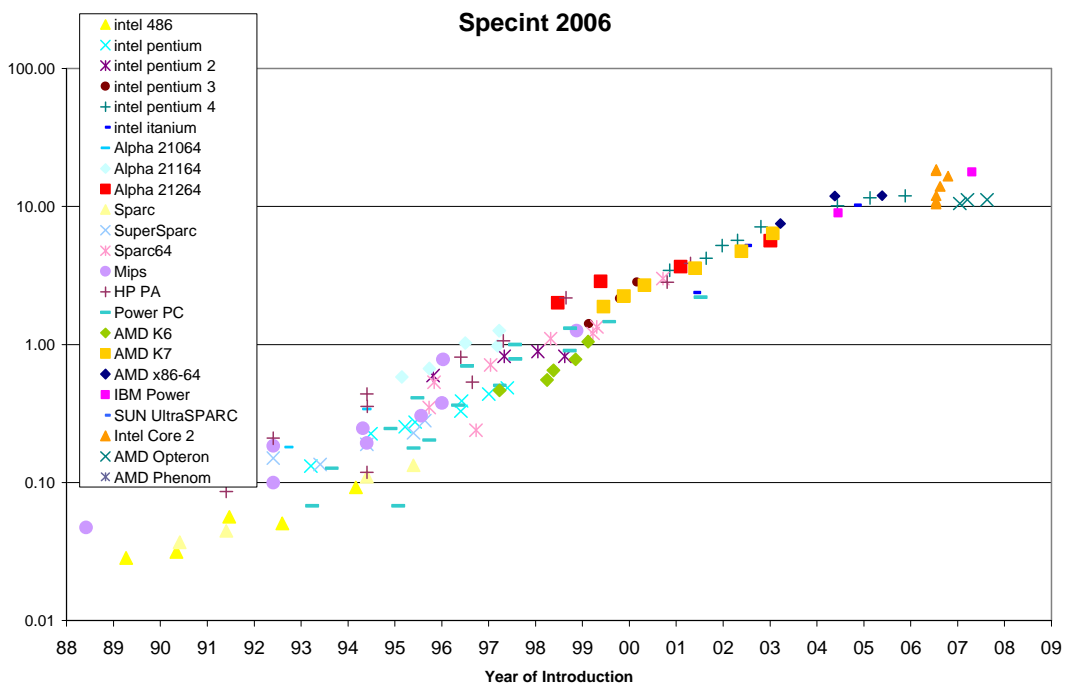


Figure 2-1: SpecInt performance numbers

However, in the recent years performance of single chip microprocessors has stopped scaling [33][34]. Figure 2-1 and Figure 2-2 display the SpecFP and SpecInt performance numbers for a various number of microprocessor families, clearly demonstrating this slowdown. There are several for this slowdown [33][34][35]: gate speeds are not increasing as fast in today's submicron fabrication technology. ILP extraction has reached its limits; there is only diminishing return in increasing the

issue width of the processor or making pipelines deeper. But most importantly, power consumption has been the major concern. Processors simply have reached their limit power consumption.

To alleviate these issues, microprocessor vendors have started integrating more than one processor core on the same die. Replicating cores is an attractive solution since it allows one to use slightly less powerful, but much more power efficient cores to get around the power wall. Such “multicore” processors have become mainstream in recent years: Intel Xeon [36] and Quad-core Itanium [37], AMD Opteron [38], Sony/Toshiba/IBM Cell [41], Sun Niagara [39] and Niagara-2 [40] are only a few examples of the multicore processors in today’s market.

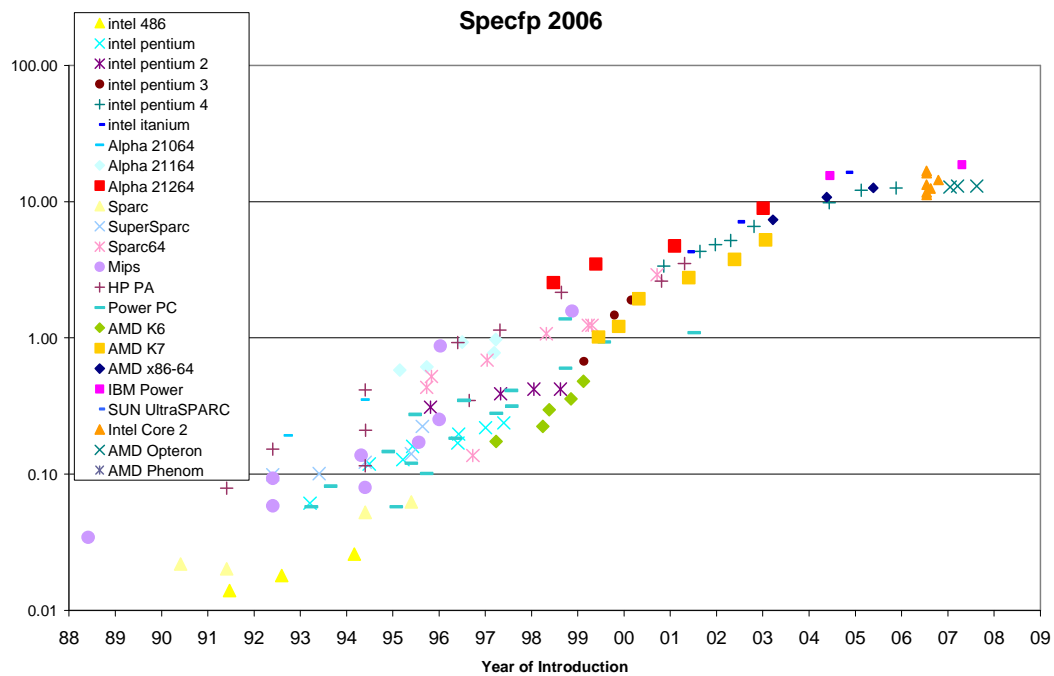


Figure 2-2: SpecFP performance numbers

While this solution is conceptually simple, replicating a number of cores, the complexity in such systems shifts towards the memory subsystem and the communication mechanisms between the processors. Processors use caches and local

memories to exploit temporal and spatial locality of the data, keeping copies close for the duration of computation. Results produced by the processors are also placed in these local memories or caches. The difficulty arises when processors need to share these results in order to cooperate in performing a meaningful computation. Data sharing involves implementing necessary communication and ordering mechanisms to keep caches and local memories consistent with each other in presence of multiple data copies. The added mechanisms usually are not trivial, both conceptually and physically, and if not carefully designed, might prove to be performance bottlenecks or in the extreme case, introduce complex design errors that render the whole system useless.

In addition, modern processors issue a large number of memory operations in order to overlap useful computation with memory accesses and tolerate long memory access latencies. Thus, in the multicore systems, the complexity of the underlying memory hierarchy increases with the number of cores; it has to accept and satisfy more and more requests as the number of cores in the system grows. Furthermore, enlarging number of processor cores increases number of local storages or caches within the system, potentially increasing number of copies of a specific data block, which further complicates the mechanisms utilized for memory coherence and consistency.

However, in spite of all the architectural complexities, the major limiting factor for multicore processor performance is the software. While system's performance potentially scales with the number of integrated cores, this performance has to be exploited by the programmer. The sequential programming model which has been dominant so far has to be replaced with explicit parallel programming models to utilize available resources, as is discussed next.

2.2. PROGRAMMING MODELS AND MEMORY ACCESS SEMANTICS

With the slowdown of single core performance and emergence of multicore processors, the task of improving application performance falls on the programmer's

shoulder. The available additional cores must be utilized by software in order to provide speedups for the running application. The traditional sequential programming model must be replaced with an explicit parallel model. Traditional parallel programming model provides the user with the abilities of creating threads that can be executed on multiple processors. POSIX threads (Pthreads) [42][43] and ANL macros [44][45] are examples of such environments. They also provide user with the low-level synchronization mechanisms such as locks, semaphores and barriers, for thread coordination. Hence, the programmer not only has to think about parallelizing his/her application, but also has to implement all coordination and orchestration activities for the concurrent threads in the application code itself, using the provided low-level mechanisms. More recent programming constructs such as OpenMP `critical` [75][76] and Java `synchronized` [74] directives allow the user to identify the critical regions of the program without worrying about the details of handling actual synchronization. However, at the lower level, these constructs also rely on the traditional locking mechanisms.

Moreover, after developing the first version of a parallel program, it is usually difficult to have it reach the desired level of performance. Oblivious coordination and coarse-grain data sharing between processor cores usually introduces unnecessary, expensive communication and serialization that reduces the performance of the running application.

In recent years, researchers have proposed innovative programming models to address the parallel programming productivity problem. Stream programming [1] and transactional memory [18][19], are among the accepted models for future multiprocessors and will be introduced and discussed in this chapter. These proposals, while being effective for some classes of applications, fail to provide a uniform and general model that can be used across application domains. More importantly, each model usually makes certain assumptions about the capabilities of the underlying hardware, specifically in defining semantics of memory accesses. Due to these differences in the requirements of the memory system, usually each of today's existing

multicore processors assumes a particular programming model and provides its specific memory access semantics. While traditional x86 architectures by Intel and AMD implement conventional coherent shared memory, more recent architectures adopt new models: IBM Cell [41] employs a stream programming model and the Sun Rock [46] supports transactional memory.

2.3. CHARACTERISTICS OF MAJOR MEMORY MODELS

2.3.1. STREAMING MEMORY SYSTEMS

The stream programming model expresses the application in terms of computational kernels communicating via data streams [1]. A stream is a sequence of similar data elements. A kernel is a compute function which performs the same operations on each data element in the stream. Data streams are passed from one kernel to the other. Each kernel consumes one or more input streams and produces one or more output streams. Expressing the program in terms of kernels and streams exposes both parallelism and communication patterns in an application. Figure 2-3 shows an example application expressed in the stream programming model.

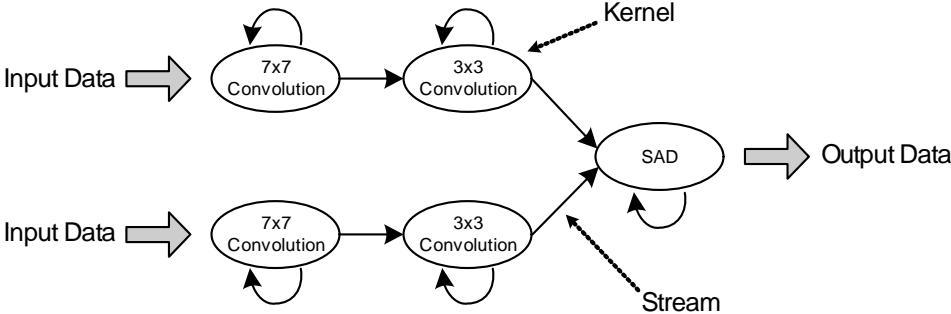


Figure 2-3: Example streaming application, stereo depth extraction

A stream programming model is mostly suitable for applications with lots of data parallelism, where operations on one data element are largely independent of other

elements. Signal processing, graphics and media applications are the most important classes of such compute-intensive applications. These applications have abundant amount of parallel computations with a relatively high ratio of compute operations to memory accesses.

Streaming applications usually have regular, statically analyzable memory access patterns, with little or no global data reuse. Most of the locality in streaming applications is in form of producer-consumer communication, where produced data stream is either passed to another compute kernel or used by the next iteration of the same kernel.

In the stream programming model, software is responsible for managing all memory references and communications between compute kernels. No implicit data sharing and copying occurs in the system. This provides the memory system with the potential of achieving better performance and energy efficiency, since programmer and compiler can orchestrate data accesses and communications with much more accuracy and efficiency. Because system behaves proactively under software control, all data transfers can be started ahead of time and before the data is actually required. Such overlapping of computation and communication/memory access (usually referred to as double buffering or Compute Transfer Parallelism, CPT [48]) leads to better latency tolerance in the streaming systems and applications. Performing memory transfers with better accuracy and variable granularity, results in more efficient usage of off-chip memory bandwidth as well as better local storage occupation.

Because of its relative simplicity and the fact that almost all aspects of the system are controlled by software, stream programming model has been mapped to a number of different architectures, including general purpose architectures (Streamware [49][50][51]) and GPUs [55]. Researchers also have proposed programming languages and run-time environments that implement stream programming model transparently, encapsulating the underlying hardware from the programmer. Examples of these systems are Stream Virtual Machine [56], StreamIt [54] and Sequoia [57][58].

However, streaming model has been demonstrated to achieve better performance on the streaming architectures, such as Imagine media processor [1][2][59][60] or Cell Broadband Engine from Toshiba/Sony/IBM [41][48][61][62].

In a streaming memory system, local memories are exposed to the software and can be addressed explicitly. In some streaming systems local memories are the only memory available to the processors for fetching operands: off-chip global memory cannot be directly accessed (such as Imagine and Cell). Hardware provides a hierarchy of storage locations and communication bandwidth to move data between levels. Data transfers between main and local memory are in the granularity of streams, which might be of arbitrary lengths. Therefore, hardware has to provide fast and efficient memory copy facilities to move data between local memories or between main memory and local memories. Such transfers are usually off-loaded to dedicated DMA engines (e.g. Stream Controllers in Imagine, Memory Flow Controllers in Cell), which support a variety of addressing modes for memory gather/scatter operations (sequential, strided, indexed, etc.), as well as queuing mechanisms for performing back-to-back transfers without the intervention from the main processor.

In general, a streaming memory system has simpler and more energy efficient hardware since it avoids complications of cache management and cache coherence protocols, but instead pushes the complexity of memory management to the software.

2.3.2. COHERENT SHARED MEMORY

While managing all the communication and data transfer in software potentially provides better performance and power efficiency, it often proves to be a burden on the programmers. Rather than performing such explicit managements, processors can rely on caches to capture temporal and spatial locality of data accesses. Since hardware transparently provides the best-effort locality management, caches are favorable for applications with dynamic control and unpredictable memory accesses which are difficult to statically analyze, such as desktop and enterprise applications.

In cache-based system, all the local storage is used to implement caches and off-chip memory is the only directly accessible storage. The granularity of data transfer between on-chip and off-chip memories is a cache block. Hardware also uses a fixed allocation and block replacement policy for transferring blocks between cache and main memory. In these systems, off-chip memory address space is shared among processors and all communication between processors is performed by reading and writing locations in the shared memory.

While this model simplifies communication, it complicates the system hardware since multiple copies of the same cache lines might be present in different caches. Therefore, in cache-based systems, hardware is also responsible for providing processors with a consistent view of the shared address space by implementing a coherency protocol. Coherence protocols ensure that copies of cache lines replicated in the system are exactly the same by defining a set of rules to be followed by hardware at the times processors attempt to read or write shared memory locations.

Coherence protocols either imply propagation of write data from one processor's cache to others (update-based protocols) or ensure that upon any modification, only one copy of the cache line exists in the whole system (invalidation-based protocols) [69]. In both cases, hardware has to locate all the current copies of the cache line to invalidate or update the data, as well as find the most up-to-date copy when satisfying a processor's read action. Depending on the scale of the system, the search for a specific cache line is either broadcasted to all possible sharers (bus-based, Symmetric Multi Processor¹ systems) or a dedicated entity in the memory system called directory, keeps the sharing information to identify possible sharers when necessary (directory-based, Distributed Shared Memory² system). Upon a write, the system sends state inquiry requests to identify sharers and invalidate copies or adjust data. Upon a read,

¹ Symmetric Multi Processor (SMP) systems are the ones in which main memory has equal distance (in term of access time) from all processors. Typical configuration of such systems has a central shared bus that connects all processors and main memory.

² Distributed Shared Memory (DSM) systems are multiprocessors in which main memory is distributed among processors. Processors are connected to each other over an interconnection network.

the same state inquiries locate and acquire the most up-to-date copy for the requesting processor. Coherent shared memory systems also have to enforce a *write serialization* property, serializing writes to the same location from different processors, by providing a serialization point. In SMP systems, the shared bus used for accessing main memory serves both as a serialization point and as a broadcast mechanism for sending state inquiry or data update requests. In DSM systems the home memory/directory controller serves as the serialization point while also identifying potential sharers, and sends explicit, point-to-point state adjustment or data update/acquire messages to all the sharers.

In shared memory systems in addition to the coherence protocol (which dictates the rules for accessing a specific memory location by all processors) hardware has to provide the set of regulations that governs ordering of memory accesses to different memory locations. More specifically, hardware should clearly identify the ordering guarantees it provides for completing memory accesses issued from different processors. This information is imperative for developing parallel software, since these rules define semantics for processors communicating via shared memory. Collection of these ordering regulations is usually referred to as system's *memory consistency model* [63]. The consistency model limits the implementation optimizations that can be made, such as overlapping and re-ordering of memory operations, because they can disturb the order of memory accesses.

The shared memory programming model relies on low-level synchronization mechanisms such as locks and barriers to provide coordination for accesses to shared data. Implementation of these mechanisms is also part of the responsibilities of the memory system. They are usually implemented by atomic read-modify-write operations on the memory locations, such as Test & Set, Compare & Swap or Load-Locked/Store-Conditional. Memory system hardware should be capable of providing necessary atomicity guarantees in performing these operations, even in the presence of interjecting accesses from other processors or actions by coherence protocol.

Given the above issues, shared memory systems usually require a set of rather complex hardware mechanisms. First of all, implementing a cache involves providing a correspondence mechanism between local storage and main memory in order to indicate which portions of the main memory currently exist in the cache. Such correspondence is commonly made by associating address tags with the cache blocks. In addition, each block should also have a state, indicating its presence, copy-back requirements and read/write permissions according to the coherence protocol. Therefore, in addition to the data storage, hardware has to provide extra space for keeping the associated tags and state information.

Cache management and maintenance of coherence and consistency model is usually off-loaded to cache/coherence controllers in the shared memory systems. These controllers integrate all necessary facilities in one place: they monitor and update state information associated with cache lines, initiate and carry out coherence actions on behalf of the processor, provide the necessary ordering between memory accesses, and include the necessary data transfer mechanisms to move cache blocks between caches or between cache and main memory. In addition, they might also be equipped with prefetch engines which recognize and detect streams of cache misses and initiate data transfers prior to the processor's data access.

Locality management, coherence, synchronization and memory consistency model are strongly related in the context of a shared memory system. As a result, while these systems simplify the task of programmer by providing best-effort locality and communication management behind the scene, they are often times more complex and challenging to design and verify than streaming memory systems.

2.3.3. TRANSACTIONAL MEMORY

Speculation has proven to be a useful technique for extracting better performance. Out-of-order execution, branch prediction, value prediction [3][4][5], etc. are all examples of speculative execution techniques commonly used in modern processors.

The common base for all these techniques is to speculatively predict the outcome of an operation before the operation is completed and launch the following operations using this value. At the time where the actual result of the operation is known, if it is recognized that the speculation was incorrect, all the speculatively executed operations are cancelled and execution is resumed with the actual result.

Speculative execution is also used as a relatively simple method for parallelizing sequential applications [6]. Thread Level Speculation (TLS) speculatively executes sections of an application concurrently as threads running on different processors. The concurrent execution does not consider the logical dependence between code segments. The underlying memory system hardware tracks such dependencies and recognizes any dependence violation at run time. In case of a violation, hardware automatically re-executes the dependent sections after the results from their logically earlier sections are produced. Parallel threads in sequential applications are created from iterations of the loops or procedure calls [7].

In addition to speeding up sequential applications, TLS can also be used to speed up traditional parallel programs that use locks and barriers for synchronization. In such systems, a thread continues to execute the critical region of the application speculatively, assuming that it has successfully acquired any necessary locks protecting the region [8]. When a collision is detected between two threads that have entered the same critical region, the system rolls back the executed critical region and re-executes it after acquiring necessary locks. This optimistic concurrency extraction helps to remove the penalty of conservative synchronization and exploit parallelism whenever possible.

Many architectures for thread-level speculative systems have been proposed: Multiscalar project [9][11], Stanford Hydra [7][12][13], CMU's STAMPede [14][15] as well as others [16][17]. These systems buffer speculative results in the memory system for two main reasons: first, they speculate over large sections of the code where register file is not large enough for storing the speculative results. Second,

hardware can relatively easily track all the dependences and detect dependence violations by observing loads and stores from different threads.

More recently, TLS proposals have been evolved from a simple speedup mechanism into Transactional Memory (TM), an innovative programming model for developing parallel application [18][19]. This programming model finds its roots in the Data Base Management Systems (DBMS) [20] where all operations in the shared database are performed as atomic transactions. By definition, a transaction is a sequence of operations that appear to be executed atomically and instantaneously. Specifically, transactions in the TM systems have three major properties [19]:

- **Atomicity:** Operations within a transaction are either all completed successfully or none of them is executed. Hence, the transaction either *commits* as a whole or *aborts* without any visible side effect.
- **Consistency:** Each transaction starts its operations with a consistent view of the shared data and leaves the system in a consistent state after completion. Consistency is defined with respect to the specific application and structure and semantics of its shared data.
- **Isolation:** Transaction executes in such a way that it does not have any effect on the concurrently running transactions. Particularly, this property implies that all of the modifications of a transaction are hidden from other transactions within the system and are made visible only after commit.

The isolation property of the transactions also implies that they are serializable; for a system running concurrent transactions, the produced result should be the same as produced by *one* execution in which are all transactions run serially.

With these powerful abstraction mechanisms, transactional memory claims to provide a new paradigm to increase parallel programming productivity. Programming within a transaction is much simpler since programmer writes sequential code and is only

concerned with correctness of results within a transaction's scope. Transactional semantics are provided by system hardware or runtime software and programmer does not need to be concerned with their implementation. This facilitates development of parallel programs by shifting programmer's focus on optimizing the parallel software rather than "getting it right" at first place.

TM is most useful for applications with irregular synchronization and low probability of contention, where the dependences cannot be statically analyzed and predicted by the compiler or programmer. For such applications, TM allows parallelization by enabling optimistic concurrency: potentially dependent transactions are executed concurrently and are only rolled-back and re-executed if there is true dependence. This provides a better execution performance in contrast to conservative synchronization in traditional shared memory model. Delegating all the correctness issues to hardware enables compiler or programmer to only identify potentially parallel sections of the application without being concerned about the details of coordination and synchronization of their parallel execution.

There have been many implementations of the transactional memory proposed by researchers. These implementations are usually categorized in three classes. Software Transactional Memory or STM systems [21][22][23] implement transactions purely in software and a runtime system, without requiring any modifications to the underlying hardware. While STM systems are easier to develop and maintain a great degree of flexibility in terms of transaction sizes or different operational policies, their performance is poor compared to hardware TM systems due to runtime overheads for tracking transaction read/write sets and managing commit/undo logs.

Hardware Transactional Memory (HTM) systems directly implement transactional semantics in the hardware. LogTM [24][25][26], Transactional Coherence and Consistency (TCC) [27][28] and UTM/LTM [29] are example implementations of HTM systems. While achieving better performance compared to STM systems, HTM systems usually are limited by fixed amount of hardware resources available for

tracking transactions, e.g. limited buffering space for a transaction's modifications. HTM systems therefore cannot handle arbitrarily large transactions and fall back to software mechanisms when a transaction overflows hardware structures. In such situations they usually suffer from the same performance penalties as STM systems.

Hybrid transactional memory systems (HyTM) rely on a few modifications in the underlying hardware system in order to support transactions effectively, but implement most of the system in software. [30][31][32] are examples of these systems.

Hardware implementations of transactional memory, like TLS systems, rely heavily on memory system to provide the key capabilities:

- **Tracking:** The memory system has to provide mechanisms to keep track of transactions' read and write sets. These sets are the memory locations that are read or written by a transaction, and are used for detecting dependencies between the transactions to decide when a transaction commits or aborts. The memory system hardware maintains these sets by associating meta-data or state information with the memory locations touched by each transaction. Tracking can be performed at different granularities, such as cache line or memory word, depending on the system.
- **Buffering/Logging:** All speculative results produced by a transaction should be buffered somewhere inside the memory system and kept hidden from other transactions. The memory system has to propagate these changes to the architecturally visible state only when a transaction successfully commits. Most of the HTM systems use the processor's cache for buffering a transaction's write set, since it can be accessed very fast and is private, hence the modifications can be kept isolated from other transactions. Alternatively, if the updates are done in place, undo logs for the modified locations should be kept elsewhere in the memory so that the effects of the transaction can be rolled back if it aborts.

- Detecting conflicts: The memory system has to detect any potential conflict between any two running transactions in the system. This task is accomplished by cross checking a transaction's write set against other transactions read and write sets. A conflict is detected if both transactions modify same memory location or a transaction modifies a memory location that is previously read by another transaction. Conflict detection can happen early (eagerly) [19] when memory locations are accessed or late (lazily), when a transaction is intended to commit its modifications.
- Committing/Aborting: Committing a transaction's modifications can be performed eagerly, by propagating all the modifications at commit time to main memory and other transactions [27] or lazily, by allowing them to remain local and be discovered by the underlying sharing mechanism (e.g. coherence protocol) when they are needed. In case of aborting a transaction, all the speculative modifications should be discarded, without any side effects. If updates are done in place, the locations should be overwritten with their previous values extracted from the undo log.

Given the above roles, in HTM systems the memory subsystem hardware has to provide extra storage for the necessary state information as well as buffering space for speculative modifications or alternatively undo logs. It also has to provide the necessary facilities for detecting accesses to shared memory locations, very similar to the coherence mechanisms in the shared memory systems. In fact, some implementations of the HTM rely on existing coherence protocols for detecting conflicting accesses [24]. In addition, hardware has to have functionality for keeping intermediate changes of a transaction isolated from other transactions and atomically make them visible at commit time or completely discard them at abort time. Therefore, in general, the implementation of the memory system hardware for HTM is more complicated than shared memory systems since it has delicate interactions with the system software.

2.4. COMMONALITIES BETWEEN MODELS

When considering all the memory models discussed above, one can observe similarities between them, most importantly requiring similar resources for implementing the desired functionality. First and foremost, all models have a hierarchy of storage elements: data storage for storing user data and state storage for keeping associated meta-data along with it. They utilize communication resources (channels and message send/receive engines) for data transfers between storages and coordination of accesses to shared data. Lastly, in all the models there is a set of external logic entities or controller agents for implementing access protocol and providing assistance in completing processors' memory references. This logic usually serves as request generator and/or performs control, sequencing and scheduling operations in order to execute protocol actions. DMA engines in streaming memory system, cache/coherence controllers and prefetch engines in shared memory systems and cache/commit controllers in HTM systems are instances of these external control agents.

Furthermore, the operations performed on these common resources are also very similar. One can recognize such similarity at two levels: at the high level, many protocol actions that implement the discussed memory models have the same conceptual functionality. Table 2-1 lists a few of these actions, specifying their memory model and specific protocol, indicating which other actions they resemble. For example, a DMA transfer between the local memories of the two processors is very much like a cache to cache transfer performed in any invalidation based coherence protocol: while there are extra actions for checking and writing the state information, both of the operations essentially copy data from one physical location to another. As another example, the committing of modifications of a transaction in the TCC HTM is very much like a scattered DMA operation in stream programming model: source addresses are read from an auxiliary structure (FIFO associated with the cache in TCC, or index memory in streaming), data elements are read from the source memory (L1 cache in TCC and local memory in streaming) and are scattered to main

memory as well as other caches or local stores. Other examples are the commit operation which update the word in the destination cache exactly the same way as an update-based coherence protocol.

#	Model	Protocol	Action	Similar to
1	Streaming		DMA block read (main mem. to local mem.)	5
2			DMA block write (local mem. to main mem.)	6
3			DMA transfer from one local mem to another	7
4			DMA indexed scatter	10
5	Coherent Shared Memory	Any	Cache refill	1
6		WB caches	Write-back (cache spill)	2
7		Invalidation based	Cache to cache transfer	3
8		Invalidation based	Snoop, coherence downgrade/invalidate	11, 12
9		Update based	Updating word in destination caches	10
10	HTM	TCC	Commit - updating data in other caches and main mem.	4, 9
11		TCC	Conflict detection (lazy) - checking for violation in destination cache upon commit	8
12		LogTM	Conflict detection (eager) - checking for violation upon receiving coherence request	8

Table 2-1: Similarities between different protocol actions

At a lower level, the primitive memory operations that are combined to form the protocol actions are the same in all of the above models. These primitive operations can be categorized into five different classes, as described below:

1. Data/State read and write – Accessing data and state storages for performing data transfers, state inquiries and updates, according to the specific protocol action
2. Communication – Sending and receiving request/reply messages over available communication infrastructure
3. Ordering – Guaranteeing a specific order between requests from the same or different processors, according to the specific protocol or memory consistency model

4. Tracking – Keeping track of the outstanding requests in the system so that each request can be completed after the corresponding reply is received. This is also necessary for enforcing ordering between different requests
5. Interpretation of state information – The major differentiating factor among memory models; indicates how the state associated with data is interpreted and the flow of control is changed according to the specific interpretation

These operations are essentially the basic blocks for composing protocol actions. One can describe the activities occurring in the memory system hardware upon receiving any protocol request/reply message as a composition of the above operations in the appropriate sequence. Given the common set of resources and their associated primitive operations as well as the strong similarities observed in the composition of operations to form protocol actions, the interesting challenge is to construct a universal memory system that can be “programmed” to implement a given memory model.

Having a programmable memory system not only allows executing applications developed for different memory models on the same hardware substrate, but also allows the user to tailor the memory system to the specific needs of the application, potentially achieving better performance. Also, the late binding of actual memory protocol to the system hardware makes it possible to fix implementation errors by changing the memory system “program”, potentially avoiding expensive fixes in the underlying hardware and costly chip re-spins.

Considering this common ground between different memory models discussed in this chapter, the following chapter presents our proposal for the universal memory system architecture. We explain system’s resources and operations in more details and express the primitive operations discussed in this chapter as an instruction set architecture for the controlling agents in the memory system.

3. A UNIVERSAL ARCHITECTURE FOR MEMORY SYSTEMS

After reviewing the major memory systems used in today's multicore processors in the previous chapter and recognizing common resources and operations in their implementation, in this chapter we propose a universal memory system architecture which enables the realization of different classes of memory models on the same set of hardware resources.

Executing a processor's memory access instruction involves performing a set of actions in the memory system hardware. A "*memory model*" defines the set of requirements that should be satisfied by the memory system after executing each memory access instructions. A "*memory protocol*" expresses the set of rules that should be followed by the hardware when executing a memory access instruction, so that the semantics requirements of the memory model are fulfilled.

The design philosophy of the universal memory system is very similar to the concept of reduced instruction set (RISC) architectures for microprocessors; instead of providing a fixed sequence of actions in the hardware that conforms to a specific memory model (or protocol), a universal memory system provides a set of basic, primitive memory operations as well as flexible means for combining and sequencing these operations. The flexibility enables one to develop or adopt a memory model that is best suited for a specific application and implement it in hardware by "*programming*" or "*configuring*" the underlying resources.

In order to construct such a generic model, we first have to distinguish the major tasks of the memory system and recognize the necessary hardware resources. The next step is defining a comprehensive set of operations on these resources, and the final step is to provide mechanisms that allow meaningful composition and coordination of operations in order to implement the desired memory protocol. Note that in our

discussion we concentrate on the *functional characteristics* of the memory system and operations that it performs internally, rather than on its quantitative characteristics, such as size of memories or available bandwidth of the communication channels.

3.1. A BRIEF REVIEW OF MEMORY SYSTEM TASKS

The primary task of the memory system is to store application data. Processors view the memory as a linear array of storage locations where each location is identified by a unique address. Applications require a large, fast memory. However, in today's VLSI fabrication technology, as the size of the memory increases so does its access time. In reality, the only economically feasible approach to provide an illusion of large, fast memory is by organizing it as a hierarchy of locations: small, fast memories closer to the processors and larger but slower memories farther from processors.

When running an application, the data should be brought into the closest memory (also referred to as local, level1, or L1 memories or caches) in order for the processor to operate on it faster. Therefore, *one of the crucial tasks of the memory system is to transfer data between levels of the hierarchy in order to bring it closer to the processor.* Transfer involves copying the desired data from larger, slower memories that are located farther from processor to smaller, faster memories closer to processor, and copying it back to the main storage after processing finishes. Data transfers also might copy data from a processor's private memory to another processor's private memory, when the two processors are sharing data or communicating. In order to exploit spatial locality of the data accesses and amortize the overhead associated with the transfer, such data copy operations usually involve a few adjacent memory words, referred to as a data block, or in the systems with caches, a cache line.

Data transfer operations can be explicitly initiated by the software via executing memory copy instruction, or implicitly by hardware, when a memory access cannot be satisfied in local memory, for example after detecting a cache miss. In cache based systems, the hardware allocation policy decides where to place cache lines in caches at

different levels of hierarchy and establishes a correspondence between the locations in the cache and main memory. In addition, the hardware has to decide whether a cache line should be copied back when being replaced or can simply be overwritten. In order to facilitate such decisions, cache based systems associate *meta-data* or *state information* with cache lines to establish their correspondence with locations in main memory, express their validity, and whether they need to be copied back on replacement. This state information is inquired, observed, and updated by memory system hardware when executing memory access instructions.

In most cache based systems, processors are unaware of data transfers and state adjustments that occur inside memory system, and simply view the memory as a linear storage array. However, in order to assist the hardware and achieve better performance, modern processors often include instructions for explicitly initiating data transfers and adjusting state information in their caches at various levels of hierarchy. Most common examples of such instructions are prefetch instructions, instructions for locking cache lines or explicitly invalidating and/or writing them back.

Furthermore, in shared memory multiprocessor systems, where all processors view the same linear memory array, multiple copies of the same data block might exist in the caches of different processors. In such settings, it is the responsibility of the memory system hardware to provide a coherent view of the underlying array of addresses despite the fact that multiple copies of the same address might be present. As mentioned in the previous chapter, this coherent view is provided by following a certain set of predefined rules when accessing a memory location, commonly known as a "*coherence protocol*". Invalidation-based coherence protocols have dominated shared memory multiprocessor systems. In these systems, the state information of the cache line is extended to contain access permissions: whether cache line data can be read or written by the processor. When executing Store instructions, hardware guarantees that the only copy of the cache line is with the writing processor and when executing Load instructions, hardware finds the most up-to-date copy of the cache line to read the data from.

As discussed above, the associated state information plays an essential role in guiding hardware and helping in making correct decision about data transfer and data access. Therefore, another major task of memory system is to provide mechanisms for storing, inquiring, interpreting and adjusting the state information associated with data as well as finding potential data copies. State updates can be initiated when processors access memory locations (e.g. cache misses), by explicit processor instructions (e.g. invalidation or ownership prefetch instruction), or by following the set of rules dictated by memory access protocol (e.g. coherence actions).

In addition to the coherence protocol, which imposes specific rules for establishing order between memory accesses to the same addresses, a shared memory system has to provide the user with a series of regulations that govern the order of completion of memory operations issued to different memory locations. These rules, commonly known as *memory consistency model*, provide a base for programmers and compiler writers to reason about correctness of the developed program or generated machine code. Consistency model dictates semantics of concurrent execution of memory accesses issued by different processors in a multiprocessor system and specifies how processors can synchronize their communication via accesses to shared memory. Many consistency models have been proposed and utilized by modern multiprocessor systems over the past years [63].

As part of the consistency model, modern processors have explicit instructions for enforcing order between the accesses they issue to memory. These instructions are usually known as *memory barriers* or *memory fences*. Execution of such instructions involves preventing a processor from issuing any new memory operation until all previously issued memory operations (from the same processor) are completed. Hence the third major task of the memory system is providing the ordering guarantees dictated by the consistency model, coherence protocol and memory barrier instructions.

Given these three important tasks of the memory system, the rest of the chapter discusses a universal memory system architecture that not only provides the necessary means for efficiently fulfilling these tasks, but also offers adaptability in supporting memory semantics of various programming models.

3.2. GENERAL ARCHITECTURE

Figure 3-1 shows high-level logical organization of the universal memory system. It consists of distinct memory elements arranged in levels of hierarchy, connected by communication channels. There are three main elements in the memory system: memories as storage locations, their associated controllers, and communication channels connecting the controllers together. In actual implementation, elements might be organized and grouped differently, however the logical view of any implementation is similar to Figure 3-1. Note that in this figure we assume processors are located at the top and main memory at the bottom. Memories and controllers closer to the processors hence are referred to as higher-level memories or controllers and the ones farther from processor are referred to as lower level ones.

The execution model of the system is based on exchanging messages between the different components. Operations start by processors emitting memory instructions to their corresponding Load/Store Unit (LSU). At each level of hierarchy, controllers receive and decode messages, then execute a set of operations to handle the received message. Executed operations might include accesses to the local memory as well as composing and sending new messages to other controllers. The combined result of the operations executed by all controllers involved, results in our desired outcome, satisfying a processor's memory request in compliance with the system's memory model.

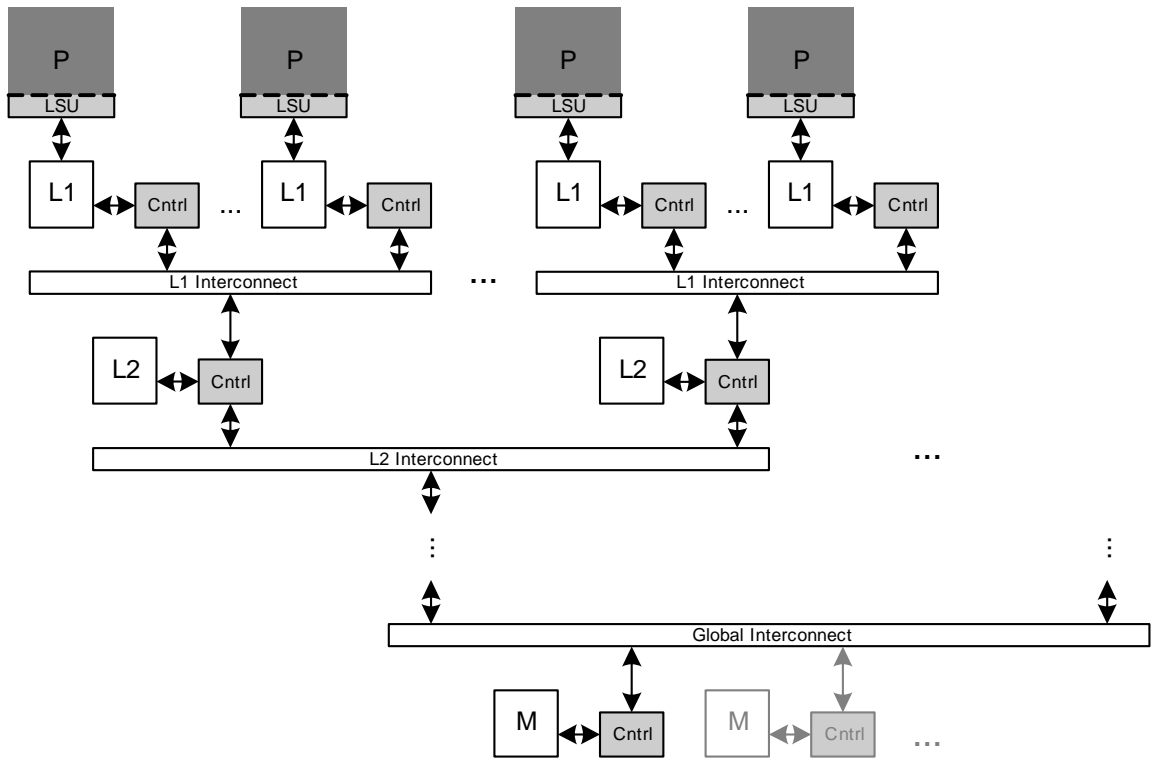


Figure 3-1: High-level architecture of the memory system

Four major categories of messages are recognized in the system:

Data Transfer Requests: Data transfers involve copying a block of data from one memory location to another. Data transfer messages usually travel downwards (towards main memory) in the memory hierarchy, attempting to read/write data blocks from/to larger, slower memories to faster smaller ones. They might also copy data between memories at the same level of hierarchy. Transfer requests can be short messages that attempt to acquire a data block for the local memory, such as cache misses and DMA gather requests, or long messages writing a data block to a remote memory such as write-backs and DMA scatter requests.

Data Transfer Replies: Transfer replies are either long messages carrying requested data block, such as cache refills or short acknowledgement messages indicating that data copy operation is completed (e.g. write-back acknowledgements).

State Inquiry/Update Requests: The purpose of these messages is to query and adjust the state information associated with a data blocks. They are usually sent by a controller to the controllers in the same or higher level and travel upward in the hierarchy, where data copies are located. These messages are short, containing no data, but depending on the state information they acquire, their corresponding reply might contain data in addition to the acquired state information. Most common examples of such messages are coherence requests or bus snoops requests.

State Inquiry/Update Replies: Reply messages for state inquiries contain the state information of the target data block. They also might bring back the data portion of the target block depending on the state in which they find it. Examples are replies to coherence messages that carry data and/or ownership information.

In the following, we describe the memory system resources and the capabilities that they should provide in more details.

3.2.1. STORAGE ELEMENTS

Memories at each level of hierarchy must not only store the application data, but also keep the state information that system associates with data. Our logical model does not make any specific assumptions about organization of the memories at each level, such as granularity of the data storage (word, byte, etc.), size of the memory, number of banks, or even number of state bits associated (However we assume that there are enough state bits available to implement the desired memory model). The only requirement is that all the storage locations have unique addresses across the system and are addressable by each and every processor. If processors only use main memory addresses (e.g. when local memories are used as caches), then at each level of the hierarchy controllers convert the processor generated address to the unique physical address of the local memory they are associated with before attempting to access the local memory.

Memories at each level of the hierarchy should support the basic read and write operations on the data and state information they store. As it will be discussed later in this chapter, data accesses in the memories are usually preceded by accesses to their associated state information. This is due to the fact that state information oftentimes protects the data by encoding necessary access permissions. Before attempting the data access, processors and controllers must check the state information to ensure that they have the required permissions. Therefore, as an optimization, the memories can overlap data and state accesses, provided that the data access is conditioned on having correct state information. This necessitates support for conditional operations on data in the memories, as well as the basic means for propagating and exchanging state information between them. Given such optimizations, sequential operations on the state and data can be converted into concurrent operations, reducing the latency of the overall memory access time which is particularly advantageous for L1 memories due to the frequent processors accesses. The next chapter presents an architecture of a basic storage element which enables conditional operations and exchange of the necessary state information, mostly based on the work by Ken Mai et. al. [71][70].

3.2.2. COMMUNICATION CHANNELS

Communication channels are used for exchanging messages and moving data between different memories in the system. In some systems in addition to the memory hierarchy there also exists a bandwidth hierarchy in the memory system where the available bandwidth decreases as traveling downward in the hierarchy [1][2].

In practical systems communication channels might be implemented in many ways: as shared busses or a type of interconnection network with point-to-point connections. In our model we do not assume any particular structure for the channels or any specific latency/bandwidth assumptions associated with communication mechanisms. However, we require the communication infrastructure to satisfy two requirements:

1. Lossless channels: we assume that any communication channel that establishes a connection does not drop exchanged messages; at the abstract level, communications are assumed to be lossless. It is the sole responsibility of the underlying channel implementation to either guarantee delivery of messages or recover from failures by using retransmissions or any other recovery technique.
2. Point-to point ordering: We do not require any of the channels to be completely ordered, however, we assume that point-to-point communication between any two entities on a channel are ordered. That is, no reordering of messages occurs in a point-to-point connection between source and destination. If the underlying channel provides virtualization facilities and communication occurs over virtual channels, the assumption is that communications between any two points over any virtual channel is ordered. This assumption simplifies satisfying the ordering requirements that a memory consistency model might place on the memory system hardware³.

3.2.3. ASSOCIATED CONTROL LOGIC

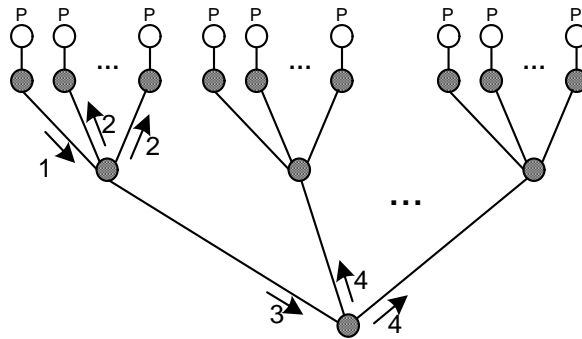
We assume that at each level of the hierarchy there is an associated controlling agent that executes the necessary operation to satisfying a processor's memory request. While memories and communication channels are considered passive resource, controllers are active resources of memory system, issuing operations that utilize the passive resources. The processor's interface to the memory system, the Load/Store unit, is considered to be the top-level control logic, communicating directly with processor's data-path. The following summarizes major tasks of the controllers in the universal memory system architecture:

- **Address mapping/translation:** Controllers, including LSU, map an effective address generated by the processor to the address of physical location(s) in the

³ A common technique for providing such ordering over an unordered physical interconnect is using timestamps or sequence numbers, similar to TCP protocol or timestamp snooping [77]

local memory, where the requested data might reside. The most common example of such mapping occurs in set-associative cache structures, where control logic extracts the cache set index from the received address and accesses all the ways in the set to see whether data is available or not. In addition to this mapping phase, a translation operation might also occur (typically only in the processor interface logic) which converts the effective memory address from virtual address space to the system-wide, physical address space. Other controllers map this physical address into addresses in the appropriate locations in their associated local memory.

- **Buffering and scheduling:** Controllers schedule and perform all data read/write operations from/to the memories at each level of the hierarchy. They take all necessary actions for buffering data and sending/receiving it over the communication channels when data transfer is required.
- **Message composition/decomposition:** Control agents are also responsible for generating, sending, receiving and decoding messages used for requesting and transmitting data blocks and/or associated state information.
- **Finding data copies:** When it comes to finding copies of replicated data blocks and performing state adjustments, each controller is responsible for finding copies and updating state information in its own sub-tree. The sub-tree of a controller contains memory associated with it and all higher-level memories that are connected to this controller. (Figure 3-2). Controller can locate copies either by broadcasting inquiry messages to nodes in its sub-tree or by keeping the sharing information internally as done by directory controllers in DSM systems.



Messages 2 and 4 are state inquiry messages looking for copies, resulted from data transfer messages 1 and 3

Figure 3-2: Finding data copies by searching controller's sub-trees

- Tracking and ordering:** Controllers, including the processor's interface logic, keep necessary tracking information about memory requests they receive and are currently processing. This information is used for completing requests after receiving corresponding replies. Keeping this information is also essential for enforcing any ordering constraint dictated by the memory consistency model or coherence protocol.

Controllers are the operating agents in the memory system; while memories and communication paths provide means for storing and moving data, the actual operations for reading/writing as well as sending/receiving data and state information are performed by the system controllers. The next section describes the general architecture of these controllers and elaborates on the operations they should be capable of performing. Afterwards, we discuss how these basic operations could be combined for handling protocol actions and request/reply messages. Since every memory protocol at the implementation level is decomposed into a set of primitive operations, a user can map a wide variety of memory protocols on this universal model by appropriately defining protocol messages and sequence of operations each must perform.

3.3. CONTROLLERS

The above mentioned tasks for controllers can be decomposed into a set of basic operations on the memory resources. This section explains the general architecture of memory system controllers, the state maintained within them, and the set of abstract operations they provide. These abstract operations either affect the internal controller state or operate on the local memories and communication channels. The architectural state of the controllers and the set of operations effectively defines an Instruction Set Architecture (ISA). The next section explains how these instructions are put together in order to handle protocol actions and request/reply messages.

3.3.1. ORGANIZATION

Figure 3-3 shows the internal organization of a controller. It has interfaces to the communication paths and memory, a set of internal status holding registers to keep tracking information of memory requests as well as data buffers for temporarily storing data blocks. The memory interface has an address mapping block that is used for accessing local memory. All the interfaces can access the internal data buffers in order to read/write data. A sequencing mechanism coordinates all the actions within the controller, including receiving incoming and sending outgoing messages, managing tracking information in the status holding registers, issuing local memory accesses and interpreting the collected state information.

The communication interfaces are used for composing outgoing messages and decoding incoming ones. They should contain the necessary flow control mechanisms to stall further communication when the interface runs out of the buffer space. However, the utilized flow control mechanism should independently control requests and replies, to avoid circular buffer dependency and deadlock [69].

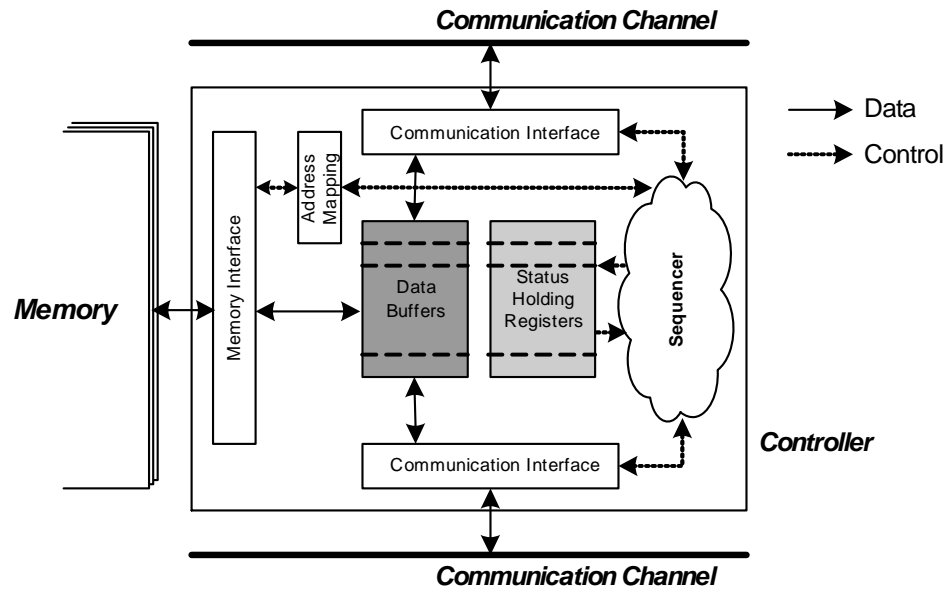


Figure 3-3: Internal organization of a controller

A set of internal status holding registers hold the tracking information of the requests that are currently being handled by controller or the requests that are waiting for a reply from lower levels of the hierarchy. For each request message that is received, the controller allocates a register and records the necessary tracking information. This information is retrieved and used for completing the processing when a corresponding reply is received. It is also used for enforcing any necessary ordering between memory requests. We do not assume any specific mechanism for associating requests and replies. This association can be realized by tagging the requests and reply messages or by guaranteeing that requests are processed in order, which allows controllers to use a simple in-order queue structure for storing and retrieving tracking information.

Controller operations are triggered by an incoming message; it is received and decoded at one of the communication interfaces and then is passed to the central sequencing logic. The sequencer executes (or schedules) the necessary operations for handling the message which depends on the type of message received. The execution model of the controller is assumed to be sequential; each operation is logically completed by the controller before moving to the next one in the sequence.

The main memory controller at the bottom level of the hierarchy has the same organization, as other system controllers with the exception that it only has a single communication interface and channel. The processor interface logic however, has a slightly different organization (Figure 3-4). It does not require data buffers, since there are no block transfers from/to processor's data path. However, its address mapping and translation logic is more sophisticated and contains mechanisms for converting addresses from virtual space to physical space (e.g. Translation Look-aside Buffers or TLBs). However its controlling logic is generally much simpler and is integrated with the processor's pipeline.

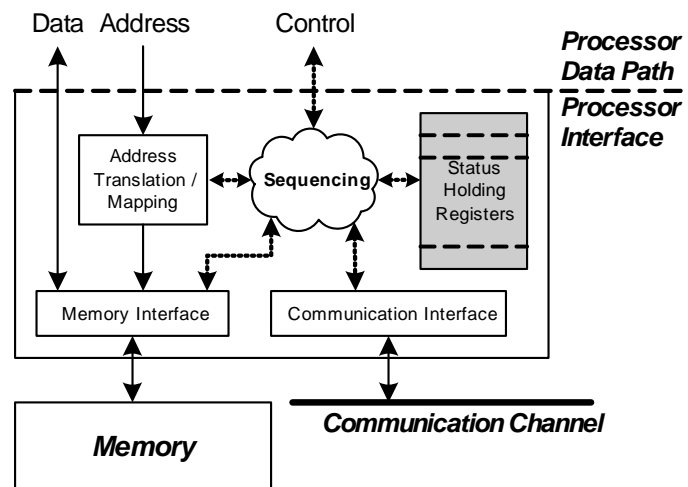


Figure 3-4: Organization of processor interface

3.3.2. INSTRUCTION SET ARCHITECTURE

Upon receiving a message, controller executes a sequence of “instructions” that perform certain operations on the memory system resources in order to process the received message, very much like executing an interrupt handler in a processor.

3.3.2.1. Internal State

The internal state of the controller consists of tracking information about outstanding memory requests, blocks of data being transferred by the controller, and a few

information fields for managing controller operations. Status holding registers keep the necessary tracking information about the received requests. Figure 3-5 shows the necessary information fields that should be kept within each register. These fields are:

- Valid: Indicates that this register contains information of a valid request
- Source address: address of the location in which the data should be fetched from, can be a local or global address depending on the type of the request
- Destination address: address of the destination where data should be written into, can be a local or global address depending on the type of the request
- Type: Type of the request
- Requestor: Identifies the source of the request, e.g. processor ID/Port ID
- Data Buffer index: The index of associated data buffer. Alternatively each status holding register can be statically associated with a data buffer and use the same index.



Figure 3-5: Information fields in SHR and data buffer entries

Since each memory request can potentially involve a data transfer, each SHR entry should have access to a temporary data buffer. Data buffers contain the data words of the memory blocks that are being read or written by controller. Potentially there is a valid bit per each data element (word or byte) to identify whether that element is valid or not. Information fields for a Data Buffer entry includes (Figure 3-5):

- Valid: Indicates that this entry is allocated and associated with an SHR
- Data *i*: *i*th data word within data block

- State i : State information associated with i th data word
- V_i : Valid indicator for data element i .

In addition to SHRs and data buffers, controller needs three separate registers for storing the result of its instructions. These registers are:

- *Accumulator (AC)*: Temporary location for storing the data or state information that controller is currently working on
- *Result_Flag (RF)*: Stores the result of the executed instruction, for example a state comparison instruction or SHR allocation instruction.
- *SHR_Index (S)*: Stores the index of a SHR entry. It can be the next available SHR entry, index of the entry indicated by received message (reply messages), or index of the matching entry when executing a SHR lookup instruction.

The usages of these special registers are discussed in the next subsection.

3.3.2.2. Instructions

Controllers perform a set of primitive operations on their internal state as well as local memory and communication resources. These operations are performed by executing corresponding “memory instructions”. Controller instructions are divided into five categories. Data and state access instructions are performed on the local memory addresses using the memory interface. Send/receive operations are executed by the communication interfaces. Instructions related to internal state and control flow are performed by the central sequencing logic. While the exact syntax and semantics of each operation/instruction depends on the actual implementation of the controller, a summary of the instructions is listed in Table 3-1.

Table 3-2 describes the effect of these instructions on the internal controller state. In this table, SHR and DB indicate the status holding register and data buffer structures. Req means input request to the controller and L is the size of a data block. Note that

when writing a word into data buffer, both state and data fields of the word are written and V flag is set to one.

Category	Instruction	Description
Data	Word Read	Reads a word from local memory into data buffer or <i>Accumulator</i>
	Word Write	Writes a word to from data buffer or <i>Accumulator</i> to local memory
	Block Read	Reads a data block from local memory to a data buffer entry
	Block Write	Writes a data block from a data buffer entry to local memory
State	State Read	Reads state information associated with data into <i>Accumulator</i>
	State Write	Writes state information associated with data with <i>Accumulator</i> contents
Tracking Info / Internal State	Load AC	Loads an immediate value into the <i>Accumulator</i>
	Compare	Compares <i>Accumulator</i> contents with a predefined bit pattern. Adjusts <i>Result Flag</i> accordingly
	SHR Allocate	Allocates next available SHR entry by setting its Valid bit to one and storing its index in the <i>SHR Index</i> register. If there is no available entry, adjusts the <i>Result Flag</i> to indicate that allocation was not successful.
	SHR Write	Writes different fields of the SHR by a request's tracking information
	SHR Search	Searches SHR structure to find an entry with matching fields (typically memory address or requestor). Adjusts <i>Result Flag</i> accordingly If a matching entry is found, stores the index of it in the <i>SHR Index</i> register
	SHR Free	Releases a status holding register by setting its Valid bit to zero
	DB Allocate	Allocates next available data buffer entry by setting its Valid bit to one and storing its index in the appropriate field in the SHR entry. If there is no available entry, adjusts the <i>Result Flag</i> to indicate that allocation was not successful.
	DB Free	Releases a status holding register by setting its Valid bit to zero
Flow Control	Branch if	Checks the <i>Result Flag</i> and changes flow of control depending on its status
Comm.	Send	Sends a message on a given communication interface

Table 3-1: Controller instruction set (ISA)

Message receive is another basic operations performed by communication interfaces. However it is not performed as result of executing an instruction in the controller. Receivers accept messages and pass them over to the central sequencing logic for processing without relying on any specific receive instruction. Like the interrupt handling in a normal processor, where receiving an interrupt causes the processor to jump to the beginning of the interrupt handler, receiving a message causes execution of a sequence of instructions in the controller which form the appropriate message handler. If the receiver detects that the message is a reply, it loads the index of the SHR entry corresponding to the request into the *SHR_Index* register before passing the message to the sequencing logic.

Instruction	Operation
Word Read	DB[SHR[S].DBIndex][Req.Address%L] <- Mem[Req.Address] Or: AC <- Mem[Req.Address].Data
Word Write	Mem[Req.Address] <- DB[SHR[S].DBIndex][Req.Address%L] Or: Mem[Req.Address].Data <- AC
Block Read	DB[SHR[S].DBIndex][L-1:0] <- Mem[((Req.Address/L)*L)+L-1:Req.Address/L)*L]
Block Write	Mem[((Req.Address/L)*L)+L-1:Req.Address/L)*L] <- DB[SHR[S].DBIndex][L-1:0]
State Read	AC <- Mem[Req.Address].State
State Write	Mem[Req.Address].State <- AC
Load AC	AC <- Immediate
Compare	RF <- (AC == Immediate)
SHR Allocate	S <- next available entry SHR[S].Valid <- 1 RF <- available ? 0 : 1
SHR Write	SHR[S] <- Req
SHR Search	S <- match entry RF <- match ? 1 : 0
SHR Free	SHR[S].Valid <- 0
DB Allocate	SHR[S].DBIndex <- next available entry DB[SHR[S].DBIndex].Valid <- 1
DB Free	DB[SHR[S].DBIndex].Valid <- 0
Branch if	if (RF) execute target instruction
Receive	if (reply message) S <- Req.SHR_Index

Table 3-2: Functional description of ISA instructions

3.3.2.3. Address mapping modes

When executing a memory access instruction, processors usually generate effective addresses in their *virtual* address space. This address is translated in the processor's memory interface into the *physical* address before it is used by memory system. In cache-based system, the resulting address is the address of the physical location in the *main memory* (since it is the only addressable memory) and still might not be directly useable for accessing local memories (which are used as caches). When memories are arranged as caches, physical address is first sliced into a <tag, index, offset> triplet and the resulting subfields are used to identify actual physical location(s) to be accessed in the cache. We refer to the process of converting target address into the address of physical location(s) where data might be found as *address mapping*.

Address translation from virtual to physical address space usually only occurs in the processor interface. The complexity of this step varies from a simple identity mapping (where physical address is the same as effective virtual address) all the way to paging and hierarchies of translation look-aside buffers with different granularity of page sizes.

The second step of the mapping, which is common to all controllers including the processor interface logic, is converting the address into the address of the locations in the local memories. Since size and structure of memories in each level of the hierarchy is different this conversion is potentially different for each level. Complexity of this conversion might also vary; it can be any thing from masking most significant bits of the address to performing a full associative lookup on the local memory to find the matching address.

Each controller needs to support more than one mapping function at the same time. For example, a cache lookup operation involves searching all the ways of a cache while a cache refill operation only involves accessing a single way of the set-associative cache. The concept of address mapping is very similar to generation of the effective addresses in the processor using a set of predefined addressing mode with

memory instructions. Addressing modes in the processor specify how the effective address is generated based on contents of a registers and an immediate value. Mapping modes specify which physical location(s) in the local memory are accessed based on the processor generated address. We assume all data and state read/write operations support three mapping modes:

- Direct: Treats received address as the absolute address of the local memory
- Cache: Received address is decomposed into <tag, index, offset> and all ways of the cache in which the target address might reside are accessed
- Cache way: Received address is decomposed and used for accessing the cache, but instead of all the ways, a specific way of the cache is accessed

3.4. SEQUENCE OF OPERATIONS

This section describes the commonly observed processing patterns when handling protocol actions in the controllers and processor's interface logic, and describes how primitive operations are combined in order to process incoming request/reply messages.

3.4.1. PROCESSOR INTERFACE LOGIC

Execution of any memory access instruction in the processor interface logic involves taking the following (logical) steps (Figure 3-6):

1. Ordering: The first step is to enforce any ordering requirements dictated by the memory protocol (e.g. memory consistency model), between memory accesses issued from the same processor. This involves searching the status holding registers and determining if there is another memory request that has been initiated but not completed. If the request cannot be issued at this time due to an ordering regulation, processor will be stalled until the collision is cleared.

2. Address translation and mapping: The second step is to determine which physical locations in the memory might contain the data requested by processor, including translation from virtual to physical address space.
3. State access: The next step is checking the state information associated with memory location (if any) to determine whether requested data is present or accessible. For example, when executing a Store instruction, the interface logic has to find out whether a specific cache line is present in the cache or if it has the appropriate permissions before attempting to write data.
4. Data access: If the state information indicates that memory access can be carried out, the physical location containing data is accessed and actual data read/wire operation is performed.
5. Storing tracking information: If the data is not present in the local memory or cache or if its state information indicates that the operation cannot be carried out, a request message should be sent to the L1 controller to ask for assistance in completing memory access. But before sending the actual request, necessary tracking information is stored internally such that the request can be completed after reply is received. This information includes the address of interested, destination register inside processor, write data and type of the operation.
6. Sending request: After storing necessary information about the memory access instruction, a request message is generated and sent to the L1 controller to ask for assistance. This might involve fetching the requested data from the lower level memory or adjusting the state such that processor's operation can be completed.

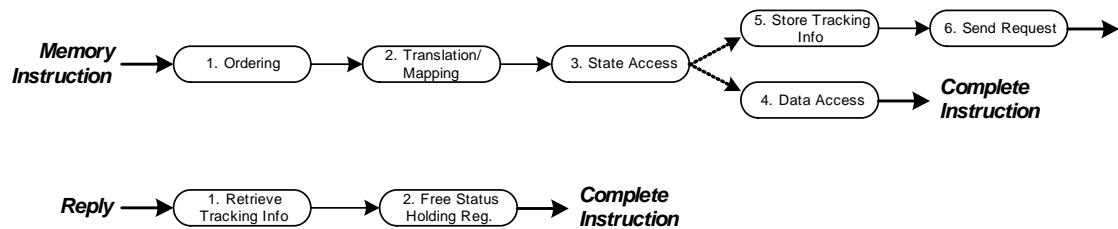


Figure 3-6: Processing a memory access in processor's interface logic

Steps for processing the reply are usually much simpler and involve returning data to the processor's destination register, as well as releasing any status holding register used for storing tracking information.

Note that not all of the above steps are necessarily required for every memory access instruction. For example, a prefetch instruction does not have a Data Access step, or if the local memory of the processor is not organized as a cache memory the State Access step might be completely omitted.

In order to reduce execution latency of memory access instruction and hence increase the performance, the above steps might potentially be overlapped: translation from virtual to physical address can take place in parallel with accessing the state information, a very common technique in systems with virtually-indexed, physically-tagged caches [35]. Another common example is execution of Load instructions in the L1 cache by overlapping tag comparison with data read in the cache line and simply discarding read data if the tag comparison fails.

3.4.2. HIGHEST-LEVEL CONTROLLER

The highest-level or L1 controller directly communicates with processor's memory interface logic. When processor cannot complete a memory access instruction it notifies the L1 controller and asks for assistance in gathering required data. L1 controller also receives request messages from other controllers at the same or lower level of the hierarchy. These can be data transfer requests or requests that search for

and update the state information of a specific data block in the associated local memory. Each class of requests is handled by taking a common set of steps inside the controller, very similar to the processor interface logic, as shown in Figure 3-7. However there are a few minor differences, as described next.

The ordering step in the L1 controller orders the received request not only with respect to the previous processor requests, but also with respect to requests received from other controllers. Also, if the L1 controller is shared between processors, the received request is ordered with respect to requests from other processors. After performing the necessary steps, if a processor request is not completed successfully it is forward downward in the hierarchy. Reply messages and request messages from other controllers and are always successfully completed. Note that request and reply messages perform the state and data accesses in reverse order; requests have to access state first, since state information guards data. Reply messages have to update data before adjusting the state and making it visible to the processors.

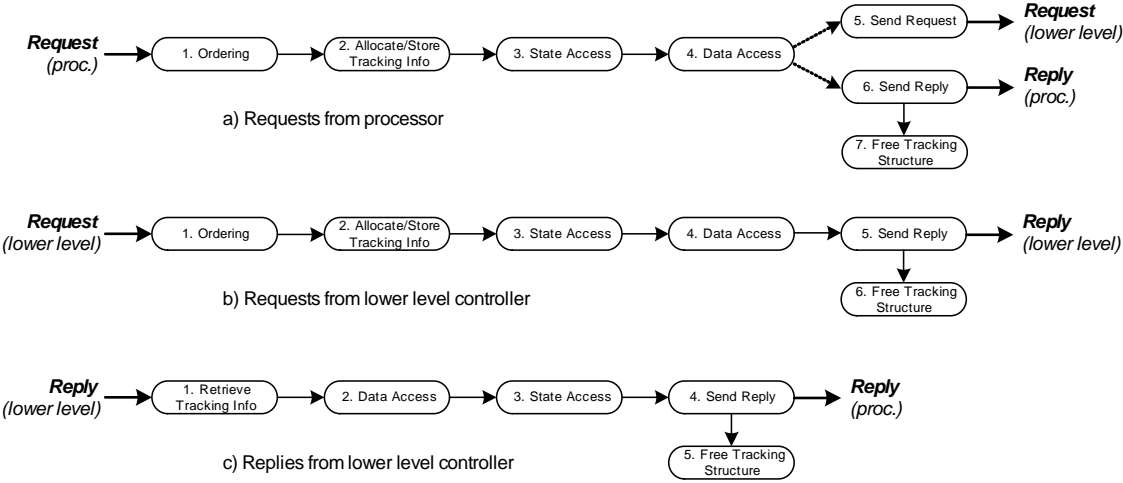


Figure 3-7: Steps for handling request/reply messages in L1 controller

Figure 3-7 illustrate the *logical* sequence of steps in handling each message type. In practice, controllers might overlap and parallelize the steps (e.g. pipelining) in order to

increase performance. Also, depending on the memory model being mapped and functionality of the received message, some of the above steps might be unnecessary and hence omitted from the sequence.

3.4.3. LOWEST-LEVEL CONTROLLERS

The lowest level controller is the controller associated with the main memory. A system might have multiple controllers at this level if main memory is distributed and/or organized in separate banks. In such cases each controller is responsible only for a subset of memory addresses. Controllers at this level receive data transfer requests from higher-level controllers to read or write a data block. Like any other system controller, main memory controllers are responsible for finding copies of data blocks when multiple copies of data exist in the system, by sending messages to controllers at the higher levels.

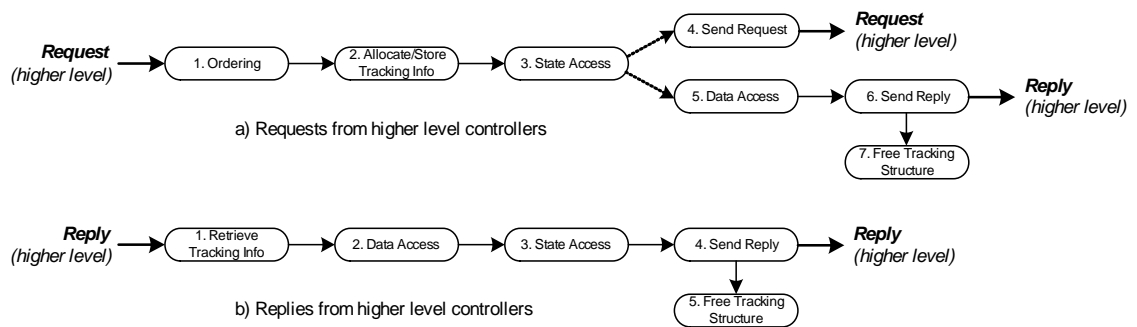


Figure 3-8: Steps for handling request/reply messages in main memory controller

Figure 3-8 shows the processing steps for handling request and reply messages in the main memory controller. Steps of operations are the same as discussed for L1 controller.

3.5. SUMMARY

In this chapter we proposed a universal architecture for a memory system in a multiprocessor setting. This system provides a set of basic, primitive memory operations as well as a flexible means for combining and sequencing these operations in the system controllers. A user can define her/his desired memory access semantics, design a memory protocol that implements the desired model, and map the operations and communications of the protocol on top of the available system resources.

In the universal memory system, we recognize three types of resources in the memory hierarchy: storage elements, communication paths and control agents that utilize them. The architecture defines a set of basic operations and state registers for system controllers in terms of an abstract ISA. Controllers sequentially execute these operations after receiving request or reply messages, very much the same way as a processor executes instructions. The entire system operates by exchanging messages between controllers at different levels of hierarchy.

The next chapter presents Smart Memories, a reconfigurable memory system architecture, as a realization of the universal memory architecture discussed in this chapter.

4. SMART MEMORIES, A RECONFIGURABLE MEMORY SYSTEM ARCHITECTURE

This chapter presents the Smart Memories architecture, an example implementation of a universal memory system. Smart Memories [70] is a modular reconfigurable architecture that instantiates common resources and implements the basic, primitive operations of the memory system discussed in the previous chapter. It allows a user to implement a memory access protocol by allocating resources and defining the processing steps for protocol requests/replies via composing a sequences of basic memory operations. These sequences of operations, called handler subroutines, are executed by controllers in different parts of the system. The collaborative result of their execution leads to completion of the desired protocol action.

The purpose of this chapter is to illustrate the implementation of the basic operations and sequences discussed earlier. The first section of the chapter presents an overview of the Smart Memories architecture and introduces the major components and their role in the memory system. Section 4.2. briefly explains the processor elements used in the Smart Memories architecture. Section 4.3. discusses main data and state storage elements, the operations they support, and explains how they are incorporated into the physical address space of the system while Section 4.4. presents the interconnect infrastructure for connecting the local memories to processors. Sections 4.5. to 4.7. discuss controller agents: Section 4.5. describes processor interface logic and its operations, covering address translation and mapping functions, accesses to data and state storage, detecting access faults and sending request messages. Section 4.6. explains the organization and operations of the local memory controller (L1 or protocol controller), and its flexible mechanisms for composing and sequencing primitive operations. Section 4.7. briefly discusses the architecture and operations of the main memory controller. Discussion about the system's interconnection network, its properties and capabilities is postponed to Appendix A.

4.1. OVERALL ARCHITECTURE

Figure 4-1 shows the overall architecture of the Smart Memories system. The system consists of units called “*Tiles*”. Each Tile contains two processor cores and a shared processor interface logic, 16 blocks of local memory, and a crossbar interconnect which connects the processors to local memories. Tiles are grouped in groups of four to form “*Quads*”. Tiles within the Quad share a local memory controller, also referred to as the protocol controller. The shared controller provides the Quads with a generic network interface which allows communication with other Quads and off-chip memory controllers via a mesh-like network.

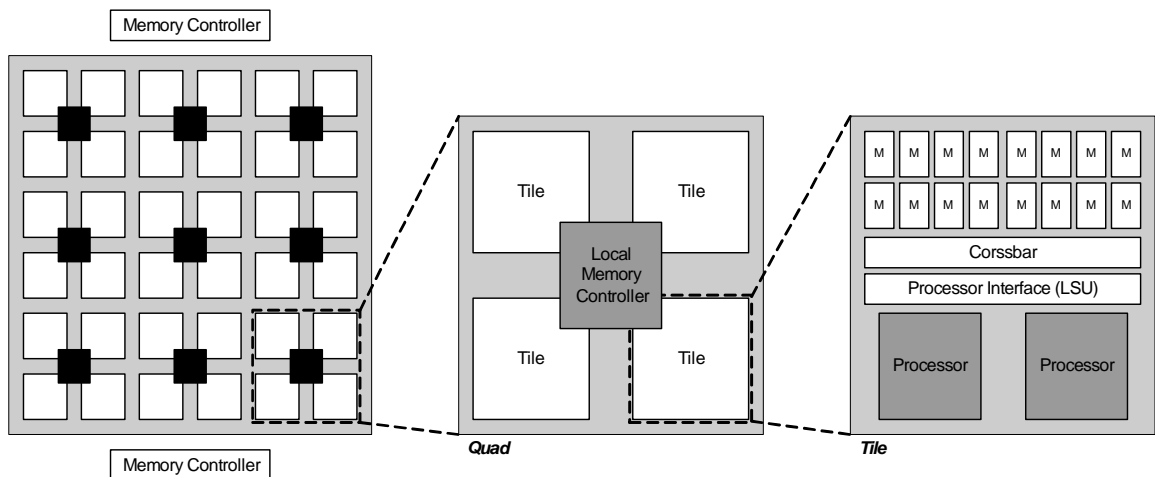


Figure 4-1: Smart Memories hierarchical architecture

As shown in the figure, there are two levels of the hierarchy in the memory system: the first level is comprised of local memories inside the Tiles and the shared protocol controller in the Quad. Second level consists of the off-chip memory and associated memory controllers. The system is capable of having multiple off-chip memory modules and memory controllers.

All the communication between processors and main memory is performed through the protocol and main memory controllers by exchanging messages over the network.

The network is capable of carrying short messages with no data or single data word and long message with blocks of data. Maximum size of the data block within the message is the same as maximum cache line size supported by the system.

The protocol controller in each Quad receives and handles request messages from the processor interface logic in each Tile. After taking any necessary actions locally, if the request is not satisfied it is forwarded to the main memory controller responsible for the target address. The main memory controller enforces global properties of the memory protocol by sending requests to and collecting replies from other Quads when needed, in addition to accessing the main memory.

4.2. PROCESSORS

Smart Memories uses Xtensa LX2 processor cores from Tensilica as the basic processing units in the Tiles. Xtensa LX2 is a 32-bit RISC machine with a 7-stage pipeline and two cycle memory access latency. Tensilica processors can be configured for dedicated application/environments in two major ways [66][67]:

- User can choose between many available optional features provided by Tensilica such as MAC units, FPU, VLIW instruction issue, JTAG interface etc.
- User can add additional architectural registers, register files, interfaces, execution units and custom instructions using Tensilica Instruction Extension (TIE) language. Additional features are not only added to the final processor RTL but also are seamlessly integrated with the rest of the software tool chain such as instruction set simulator, assembler, compiler and debugger.

Figure 4-2 displays the architecture of the Xtensa LX2 processor core. Our specific processor configuration includes a 32-bit integer multiplier and divider units, a 32-bit single precision floating point unit, On-Chip Debug (OCD) and JTAG interfaces, the instruction trace port and a 3-way FLIX/VLIW instruction issue using variable instruction encoding [68].

In order to integrate the Xtensa LX2 core in the Smart Memories architecture we had to extend the existing memory interfaces using the TIE language. We added an extra memory interface port to the processor, called TIE port, similar to the existing processor instruction and data ports. This port issues a 6-bit “TIE opcode” to the memory interface logic which indicates what memory operation it intends to perform. Our configurable processor interface logic (Load/Store Unit) receives accesses from all three processor ports (instruction / data / TIE) and returns necessary replies after completing the issued memory access instruction.

Using the TIE language we also added a few special memory access instructions to the processor’s instruction set. These instructions have specific memory accesses semantics and are briefly described below.

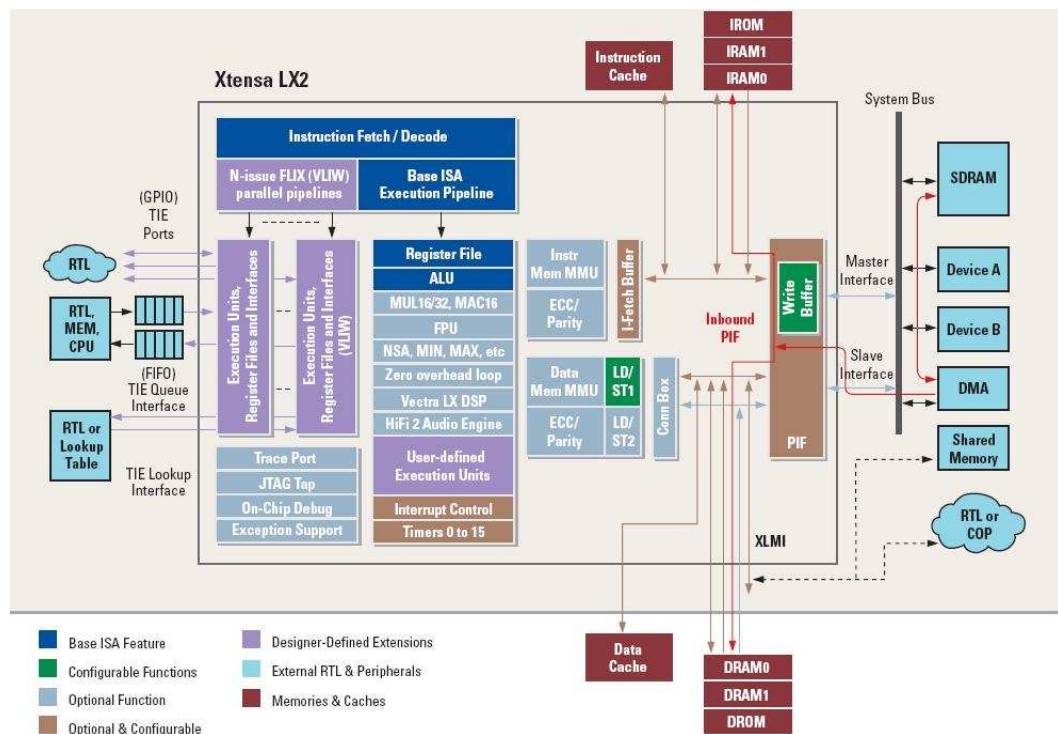


Figure 4-2: Xtensa LX2 processor architecture, from [78]

Synchronized Load (Sync Load): Treats a meta-data bit associated with data word as a Full/Empty indicator and stalls the processor if this bit is zero (associated word is “Empty”). If successful, turns off the bit to indicate that the data word is consumed.

Synchronized Store (Sync Store): Opposite of the above instruction; attempts to write the data word and stalls the processor if Full/Empty bit indicates that location is “Full”. When successful, sets the Full/Empty bit to one to indicate that there is a valid data word available.

Future Load: Same as Sync Load above, but does not consume the data word (leaves Full/Empty bit as one when successful).

Reset Load: Resets Full/Empty bit to zero and returns the data word to the processor regardless of current status of Full/Empty indicator.

Set Store: Sets Full/Empty bit to one and writes data word regardless of current status of Full/Empty indicator.

Meta Load: Reads the value of meta-data (control) bits associated with the data word. These bits are described in the next section.

Meta Store: Writes the value of meta-data bits.

Raw Load: Special Load instruction which skips the address translation step in the processor interface logic and treats processor issued address as physical address rather than virtual.

Raw Store: Same as Raw Load instruction for writing data.

Raw Meta Load: Same as Raw Load instruction for reading meta-data bits.

Raw Meta Store: Same as Raw Meta Load instruction for writing meta-data bits.

FIFO Load: Reads data word from a memory mat that is configured as a FIFO; FIFO status register in the interface logic is updated with FIFO status information, i.e. whether FIFO was empty. Memory mats and their operations are discussed in the next subsection.

FIFO Store: Writes a data word to a memory mat that is configured as FIFO; FIFO status register is updated with FIFO status information, i.e. whether FIFO was full.

Safe Load: Reads a data word from the memory address but ignores virtual to physical address translation errors if encountered.

Memory Barrier: Memory fence instruction that stalls the processor until all outstanding memory accesses are completed.

Hard Interrupt Acknowledgement: Signal to the memory system that a hard interrupt was received by the processor; is used only inside interrupt handler code.

Mat Gang Write: Does a column-wise write operation on one of the meta-data columns in the memory mat (described further in the following section).

Conditional Mat Gang Write: Conditional column-wise write operation on one of the meta-data columns in the memory mat.

Cache Gang Write: Same as Mat Gang Write, but is issued to all memory mats forming the cache structure (Section 4.5.1. and Appendix B.1.1. describe how to set up a cache structure using memory mats).

Conditional Cache Gang Write: Same as Conditional Mat Gang Write but is issued to all memory mats forming the cache structure.

4.3. STORAGE ELEMENTS

The most important resources in the memory system are the storage elements or memories themselves. Memories are used for storing application or user data and associated state information. There are two distinct storage structures in the Smart Memories architecture: the memory blocks within Tiles, called *memory mats* for the local or L1 memories, and the main memory located outside of the chip (referred to as *off-chip memory*).

4.3.1. RECONFIGURABLE MEMORY MAT

Memory mat is the basic element of storage in the Smart Memories system. It is an array of 1024 words where each data word has 32 bits and is augmented by 6 additional bits of meta-data or control information. Internal organization of the memory mat is depicted in Figure 4-3. A mat consists of data array (1024×32), meta-data or control array (1024×6), pointer logic and Read-Modify-Write (RMW) logic which provides atomic update the meta-data information. Adding the RMW logic simplifies the manipulation of the state information associated with data: instead of having controllers to read the state information and write new values, the internal RMW logic performs necessary updates on the state information. Atomicity of this operation further simplifies the state updates and allows effective implementation of atomic memory accesses instructions such as Test & Set. Each of the components in the memory mat is capable of performing a few independent operations and is individually controlled by an external opcode signal.

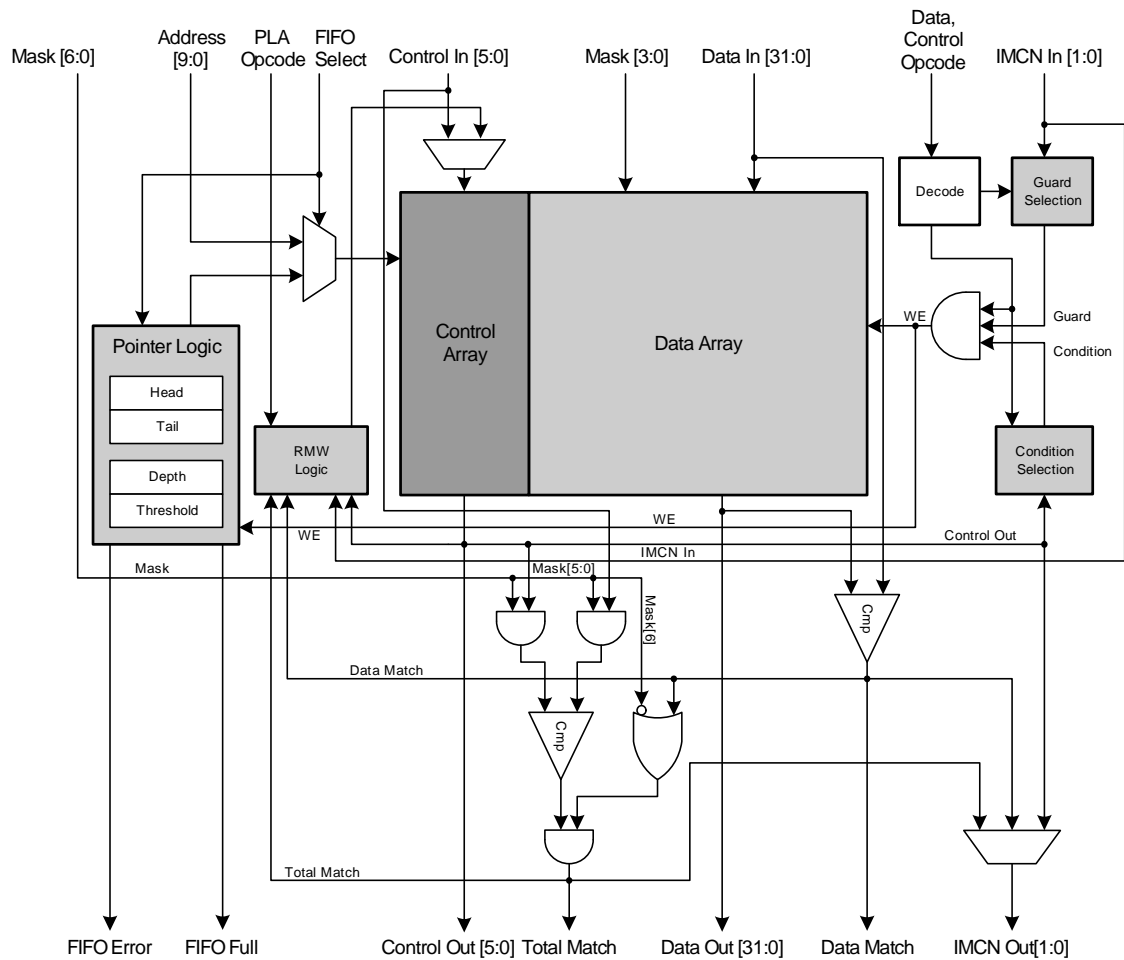


Figure 4-3: Internal organization of memory mat

The main data array supports read, write and compare operations, using an internal 32-bit comparator. It also supports byte-writes using a four bit input mask which specifies which bytes should be written into the array. Compare operations behave similar to read operations: they read the addressed word in the data array and send out the value on the *Data Out* output, while they use the internal comparator to compare this data with the *Data In* input and generate a *Data Match* signal as result of the comparison.

In addition to simple write operations, the data array supports conditional and guarded write operations, where the write is performed only if the *guard* and/or *condition* signals are activated. This an optimization to eliminate the branch operations in the

controllers that access the mat: instead of reading and comparing the state information and accessing the data based on the result, controllers can issue a conditional data access concurrent with the state access, where the data access is conditioned on having the desired state. As it will be described in the rest of this section, these operations are particularly useful when memory mats are used for implementing cache structures. A three bit *Data Opcode* input dictates the operation of the data array, according to Table 4-1. *Guard* and *condition* signals can be configured in each memory mat separately and are discussed later.

Operation	Opcode	Description
NOP	0	No operation, idle
Unused	1	Not used, similar to NOP
Read	2	Read accessed word
Compare	3	Read accessed word and compare it to input data
Guarded Write	4	Write accessed word if <i>guard</i> signal is active
Guarded Conditional Write	5	Write accessed word if both <i>guard</i> and <i>condition</i> signals are active
Unguarded Write	6	Write accessed word
Conditional Write	7	Write accessed word if <i>condition</i> signal is active

Table 4-1: Memory mat data array opcodes

The mat's control array is a dual ported memory block that can supports atomic read-modify-write operations. A read-modify-write access takes two cycles to complete: read and modify operations occur in the first cycle while the write operation occurs in the second cycle. The first port of the array is used for carrying out the external access, while second port is used by the read-modify-write logic to update contents. An internal forwarding logic forwards the updated contents to the *Control Out* output when the same word is accessed in back to back cycles.

The control array supports read, write, compare, read-modify-write and compare-modify-write operations. In addition, it receives the same *guard* and *condition* signals as data array and supports guarded and conditional write and read-modify-write operations. When performing compares, the content of the addressed location is compared with the *Control In* input and result is reported by *Total Match* output. An

external *Mask* signal controls the bits that participate in the comparison. Note that the *Data Match* signal from data array also participates in generating the final comparison result, but it can be masked out using the *Mask* input.

Bits 2-0 of the control array support a special addressing mode. These bits are capable of flash-setting or flash-clearing a whole column in a single cycle. In addition, bit 2 of the array can be conditionally flash-set or flash-cleared based on the value stored in column 1: bit 2 of every entry is set to one or zero if corresponding bit 1 in the same entry is set to one. These operations, described in Section 4.2. as gang-write and conditional-gang-write instructions, are particularly useful for flash clearing a cache structure or conditionally clearing a transaction's read and write sets after detection a violation [27]. A four bit *Control Opcode* input specifies the operation of the control array, as listed in Table 4-2.

When performing read-modify-write and compare-modify-write operations the updated values of the control bits are supplied by RMW logic within the mat. This logic is implemented as a lookup table with 64 entries⁴. The input signals to the lookup table can be selected from values of the six output control bits, the data match and total match signals generated by comparators, two control signals from inter-mat communication network (described later), and an external four bit opcode signal called *PLA Opcode*. Conceptually the *PLA Opcode* serves as command input for the RMW logic and specifies how the output values are generated after receiving appropriate inputs. These values are written back to the control array in the next clock cycle.

⁴ Total number of inputs to the RMW logic is 13 bits. A multiplexer chooses 6 bits from the input signals for addressing the lookup table. Select signals for the multiplexer are derived from a configuration register.

Operation	Opcode	Description
NOP	0	No operation, idle
Unused	1	Not used, similar to NOP
Unguarded Read-Modify-Write	2	Read accessed location and write back updated contents from RMW logic
Guarded Compare-Modify-Write	3	Read accessed location, compare it to input, write back updated contents if <i>guard</i> signal is active
Read	4	Read accessed location
Compare	5	Read accessed location and compare to input data
Guarded Read-Modify-Write	6	Read accessed location and write back updated contents if <i>guard</i> signal is active
Guarded Conditional Read-Modify-Write	7	Read accessed location and write back updated contents if <i>guard</i> and <i>condition</i> signals are active
Guarded Write	8	Write accessed location if <i>guard</i> signal is active
Guarded Conditional Write	9	Write accessed location if both <i>guard</i> and <i>condition</i> signals are active
Unguarded Write	10	Write accessed location
Unused	11	Not used, similar to NOP
Gang Write	12	Write specified column (2-0) with given data
Conditional Gang Write	13	Write column 2 with data if corresponding bit in column 1 is one
Unguarded Conditional Write	14	Write accessed location if <i>condition</i> signal is active
Unused	15	Not used, similar to NOP

Table 4-2: Memory mat control array opcodes

Each memory mat is also equipped with a pair of head/tail pointers which make memory mat suitable for implementing hardware FIFOs. An external control signal, *FIFO select*, enables the FIFO behavior by selecting the address source for the data and control arrays (Figure 4-3). Head and tail pointers are incremented when mat is accessed in FIFO mode: read and compare operations increment the head pointer while write operations increment the tail pointer. When performing guarded and conditional write operations to the FIFO, the tail pointer is incremented only if guard or condition signals are active.

Pointer logic has a configuration register that specifies the depth of the FIFO. When the depth of the FIFO reaches this register, next write operation (which writes a full FIFO) will cause the *FIFO Error* signal to become active. The same output signal is activated when a read tries to access an empty FIFO. In addition to the depth register, there is a user controller threshold register which sets up the FIFO warning threshold; if the depth of the FIFO reaches this threshold, a *FIFO Full* signal is asserted to inform the user that FIFO is becoming full.

Each mat can send and receive two bits over an *inter-mat communication network* (IMCN). This network is a fast path for exchanging control and state information between memory mats to implement composite storage structures such as caches. For example, when memory mats are used as caches, IMCN propagates hit/miss information from mats storing line tags to data storage mats so that they can take appropriate action. *IMCN_out* outputs of the mat are controlled by separate configuration registers and can be selected to be either one of the six control bits (of the location accessed in the current cycle), or the results of the comparison operations (data match or total match signals). *IMCN_in* inputs are used by the RMW logic in generating new values for control bits or used as *guard* signals inside the mat.

IMCN can perform a logical OR operation on the control signals collected from memory mats before feeding them back. This allows the control information from more than one source mat to be combined before being passed to destination mats. As an example, consider an implementation of a two-way set-associative cache that implements an LRU replacement policy. The logical OR of the *Total Match* signals from the tag storage mats is the cache hit/miss indicator. This hit/miss indicator is fed back to the tag storage mats using IMCN to update the LRU information.

IMCN allows the contributing mats of a logical OR operation to be specified via a configuration register. For each destination memory mat, the user can define which source mats should participate in the logical OR operation. These settings are defined separately for each IMCN bit, resulting in total of 32 mask registers in the IMCN.

Figure 4-4 shows how the logical OR operation is controlled when determining *IMCN_in* signals for a memory mats.

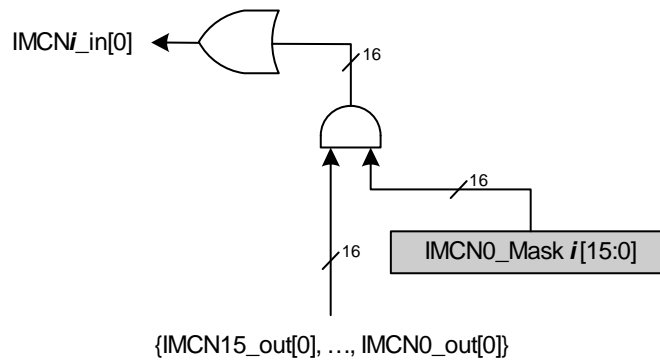


Figure 4-4: Logical OR operation in IMCN

The *guard* signal in the mat can be selected to be any logic function of the two input *IMCN_in* bits. In the above example of a cache, assuming that the hit/miss information is broadcasted on the *IMCN_in[0]*, the guard signal is selected to be equivalent to *IMCN_in[0]*. The *condition* bit, controlled by a separate configuration register, can be selected to be any of the control bits read from control array. An example of using *condition* bit is implementing special type of store operations which treats a meta-data bit as a Full/Empty indicator for the word. This special “Synchronize Store” (SyncStore) operation writes the data word only if associated Full/Empty control bit indicates that location is empty. Data array uses a conditional write operation to implement SyncStore, by setting the condition to be $Full/Empty == 1'b0$. If the *condition* is not evaluated to true the write operation is discarded.

In summary, even though data mats serve as basic storage units for data and state information, they support a rich set of logical operations on data and state bits, which allows optimizing and overlapping of data and state accesses from processors and local memory controller. Having a dedicated network for exchanging control information allows mats to be used for implementing composite memory structures such as caches, where control information should be sent from one set of storage

elements to others. Details of how the memory mats are set up for implementing a variety of cache structures are discussed more in Appendix B.

4.3.2. MAIN OFF-CHIP MEMORY

Off-chip memory serves as the main storage for application data and is controlled and operated on by the off-chip memory controllers. Smart Memories supports multiple off-chip memory modules, each one controlled by its dedicated memory controller. System can be configured to have one, two, four, or eight separate memory controllers. When there is more than one memory controller present in the system, the addresses are interleaved between different controllers. System supports interleaving factors of 16, 32, 64 or 128 bytes. Note that the interleaving factor should be at least the same size as cache lines (if system implements caches).

Off-chip memory is viewed as an array of 32-bit words similar to memory mats, but each word is associated with only four control bits. These four control bits map to bits 3-0 of control array in memory mats. In other words, when simply copying the words from local memory mats to main memory, the four least significant bits of the control array are saved in the four control bits and bits 6-5 are lost. When copying data from off-chip memory to local memory mats, bits 6 and 5 in the destination memory mat are written with zero. Similar to memory mats, the control bits in the off-chip memory are used for storing state information associated with the memory word.

Main memory supports basic read and write operations, including byte writes. It also can read and write four associated control bits along with or separate from the main data word. However, unlike memory mats, there is no support for comparison or read-modify-write operations. All of the accesses to the off-chip memory are handled by the associated memory controller which interprets the received opcode field and accesses appropriate bits in the memory accordingly.

4.3.3. PHYSICAL ADDRESS MAP

Since some memory models like streaming require the local memories be addressable and exposed to the software, all the storage locations in the Smart Memories system are mapped to a global physical address space which is shared by all processors. This address space includes off-chip memory, all memory mats in the Tiles, and all the configuration locations that control hardware. When accessing a memory location, processors issue operations to a virtual address space. Translation between the virtual address space and physical address space occurs in the processor interface logic, as will be described later.

Figure 4-5 shows system's virtual and physical address spaces. Total size of both address spaces is 4GB and they are divided into 16 segments. Processors do not generate any accesses to segment 0-3 of the virtual address space. Segments 4-7 of this address space are dedicated to instruction code while segment 8-15 are used for application data. Segments 0 and 1 of the physical address space are reserved segments. Segment two contains all of the system's configuration locations while segment 3 contains all the Tile memory mats. Main (off-chip) memory is mapped to segments 4-15. A segment table in the processor interface logic translates addresses from virtual space to physical space by simply replacing the four most-significant bits of the address.

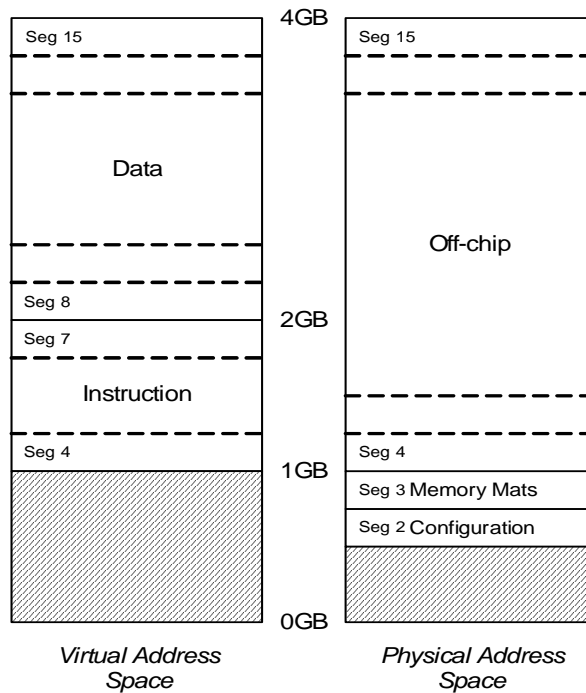


Figure 4-5: Virtual and physical address spaces

Memory mat addresses in segment 3 start from mat 0 in Tile 0 and Quad 0 and continue by going to next Tile and Quad, as shows in Figure 4-6. Note that total size of the existing memory mats in the system is usually much smaller than a whole segment (256 MB). In such cases, the upper section of segment 3 will be empty. Figure 4-7 shows how all the configuration registers are mapped in to physical address space. Segment 2 starts by memory mat, configuration registers, followed by Tile, local memory controller, and main memory controller configuration registers. The address map can contain up to 64 Quads. However, there are usually much fewer Quads present in a typical system configuration. In such systems, segment 2 of the address space will not be contiguous and accesses to locations for the non-existing Quads will cause undefined behavior.

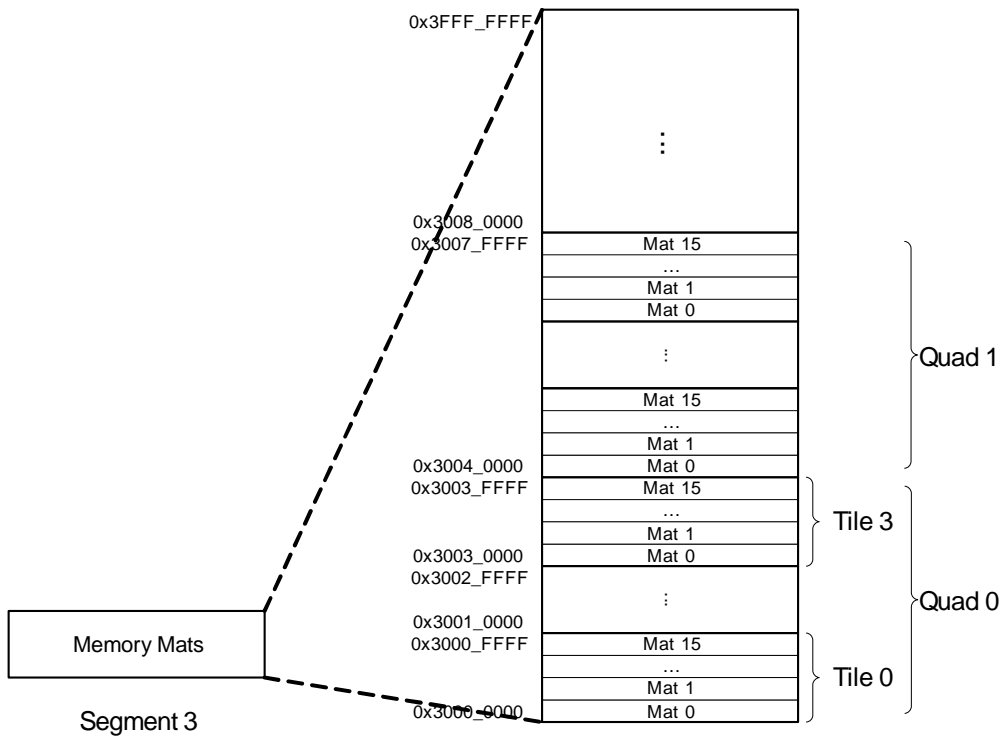


Figure 4-6: Mapping of memory mats in physical address space

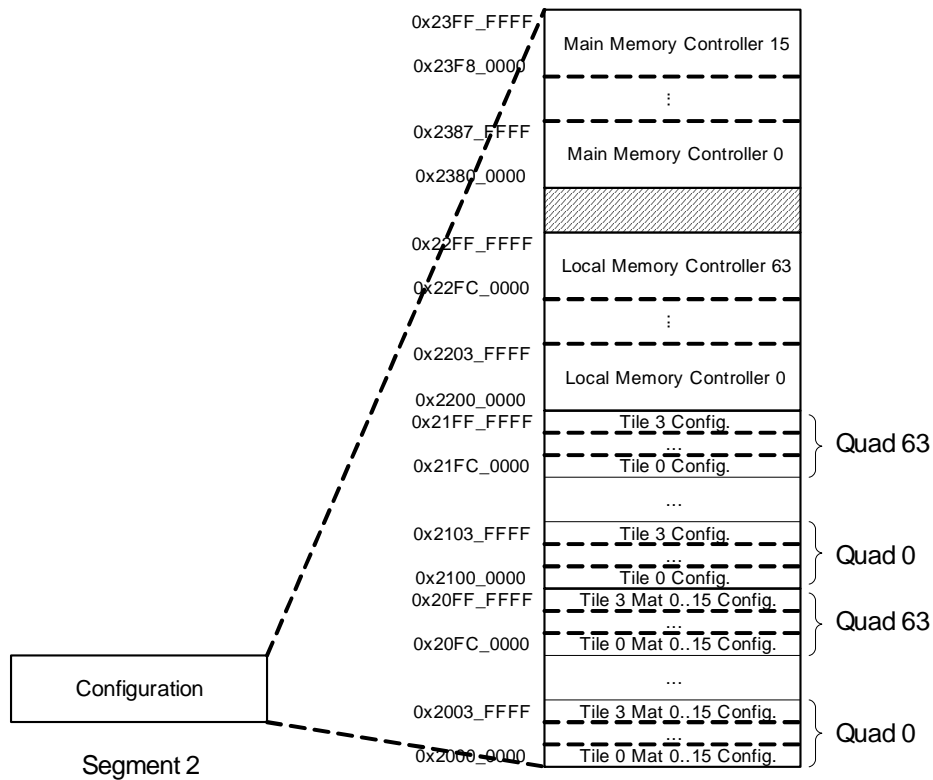


Figure 4-7: Mapping of configuration registers in physical address space

4.4. TILE CROSSBAR

There are two levels of interconnect in the Smart Memories system: the Tile crossbar connects the memory mats to the processor interface logic and shared protocol controller, while at next level, a generic network connects Quads to each other and to off-chip memory controllers (Figure 4-1). Both of these interconnect mechanisms satisfy the requirements explained in the previous chapter: they do not drop communicated messages and preserve ordering between the two end points.

The Tile crossbar performs arbitration between different sources that attempt to access memory mats and has a built-in multi-casting capability that can propagate control signals to a combination of memory mats specified by a mask. Figure 4-8 shows the

interface signals of the crossbar. Each processor has two distinct ports to the crossbar (instruction and data) and the protocol controller also has two separate ports. These ports are routed through the crossbar to 16 memory mats and a Tile configuration storage block.

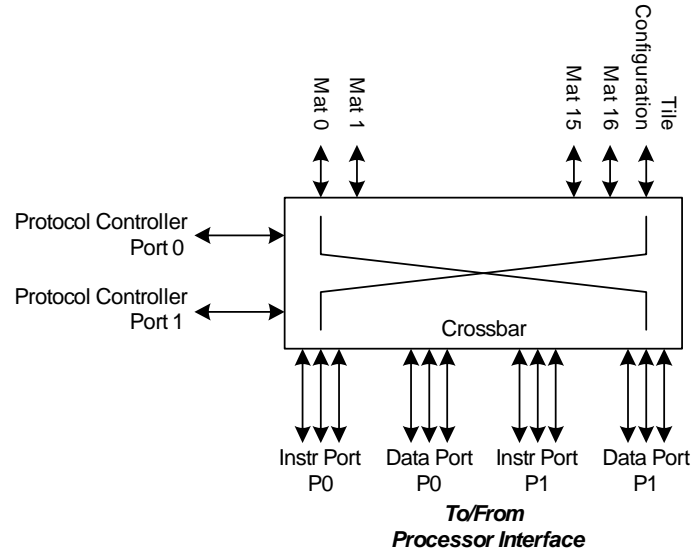


Figure 4-8: Tile crossbar

Each processor port can potentially access three distinct set of mats simultaneously. When memory mats implement a cache structure, a processor's access to the cache has to be routed to both tag and data storage mats. If the cache has more than one way, there will be a set of mats storing tags (tag mats) and another set storing data portion of the cache line (data mats). When implementing more complex storage structures, in addition to the data and tag, auxiliary storage might be required to keep other pieces of information. Hence, a third set of mats might be accessed to store or retrieve the auxiliary information from each processor port⁵. Supporting three parallel accesses allows processor interface logic and protocol controller to overlap accesses to state and data storages. Parallel mat accesses and conditional operations on data and state

⁵ For example, a TCC cache [27] uses a FIFO structure to store the addresses of a transaction's write set. The addresses are written to the FIFO in parallel with accessing the cache.

bits inside the mat allows overlapping of logically sequential operation and data and state information, hence reducing total number of clock cycles required for completing memory accesses and improving the overall memory system performance.

Tile crossbar also acts as an arbiter between different sources when they want to access the same memory mat. If there is a collision between the two processors' accesses, the crossbar stalls one of them. The protocol controller is assumed to have a higher priority for accessing memory mats and will stall colliding processors. Unlike processors, the protocol controller ports can only access a single set of mats and the set of mats they access are always disjoint. Therefore, they never collide with each other and crossbar does not perform any arbitration between them.

4.5. PROCESSOR INTERFACE LOGIC

The processor interface logic or Load/Store Unit (LSU) translates the processor's memory access instruction into memory mat operations, detects success or failure of a memory mat accesses and in case of failures, asks protocol controller for assistance in completing processor's instruction. In addition, it also translates the request's virtual address into the system's physical address and identifies which memory mats the access should be routed to.

Figure 4-9 illustrates the input/output signals for processor interface logic. Each section of the interface is connected to instruction, data and TIE port of the processor to receive the memory access instructions. The instruction port issues accesses to instruction address space while the data port and the TIE port access the data portion of the address space. The data port issues simple Load/Store instruction to memory while the TIE port issues more sophisticated instruction such as synchronized accesses or prefetch operations⁶. Data and TIE ports to the processor are 32-bits wide and

⁶ In general, all custom memory instructions added to processor core are issued from TIE port.

processor can only activate one of these ports at each cycle. The instruction port is 64-bit wide and can be active along with the data or TIE port.

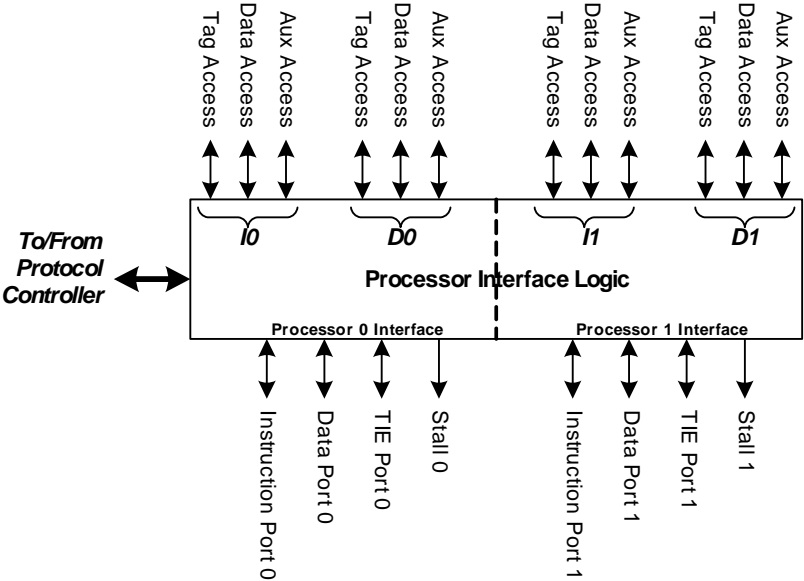


Figure 4-9: Processor interface logic

The processor interface logic is connected to Tile crossbar with a set of four ports, corresponding to instruction and data/TIE ports for each processor. As discussed earlier, each crossbar port contains a set of three mat access ports. The processor interface logic also has a port for communicating with the shared protocol controller in the Quad. This port is shared between the instruction and data ports of both processors (I0, D0, I1 and D1) and is used for sending request messages and receiving replies from protocol controller. There is an internal arbitration logic that selects the next available request message for sending to protocol controller.

As discussed in the previous chapter, in the universal memory system architecture, the processor interface logic is viewed as the top-level controller which is closely integrated with the processor’s data path. The rest of this section describes how the

processor interface logic implements address translation, memory access and communication with L1 controller, mentioned earlier in this chapter.

4.5.1. SPECIFYING ADDRESS TRANSLATION AND MAPPING

Converting a processor's virtual address to its physical location(s) has two steps: translation and mapping. The first step is to translate the address from virtual to physical address space, which is performed by using a segment table and simply remapping the virtual segment number to a physical segment number by replacing four most-significant bits of the address. The translation keeps the rest of the address bits (segment offset) the same. A virtual segment can be mapped either to off-chip or on-chip memory, but not to the configuration or reserved segments.

When a segment is mapped to on-chip memory mats (segment 3), a segment base parameter specifies which memory mat the segment starts from. The base is expressed in form of Quad ID / Tile ID / Mat ID. Since the size of the on-chip memory is much less than a virtual segment size, a segment size parameter restricts the range of the offset portion. If the offset exceeds the specified segment size, processor interface logic throws an exception at the issuing processor. Both of the base and size parameters are expressed in number of memory mats. Therefore a segment always starts at the starting address of a memory mat and the size of it can only be an integer multiple of mat size (4KB). Since memory mats are mapped contiguously in the address space, a segment can be mapped to any contiguous number of mats in any Quad/Tile. Each processor has its own segment table. Figure 4-10 shows the structure of segment table. Since processors never issue any memory accesses to virtual segments 0-3, these segments are omitted from segment table and are not implemented.

The segment table also has a few additional features. First, it provides the system with a simple protection mechanism. Each segment has separate Read (R) or Write (W) permission bits. If a processor attempts a read or write operation without having

necessary permission it receives an exception. Segment table also specifies whether the accessed memory segment is cached or not (C bit). Caching is only applied to off-chip memory segments; caching any part of the on-chip memories is not allowed. The On-Tile (OT) bit, if active, forces a memory access to an un-cached segment to be routed to local Tile memories by ignoring the address bits that identify destination Quad and Tile.

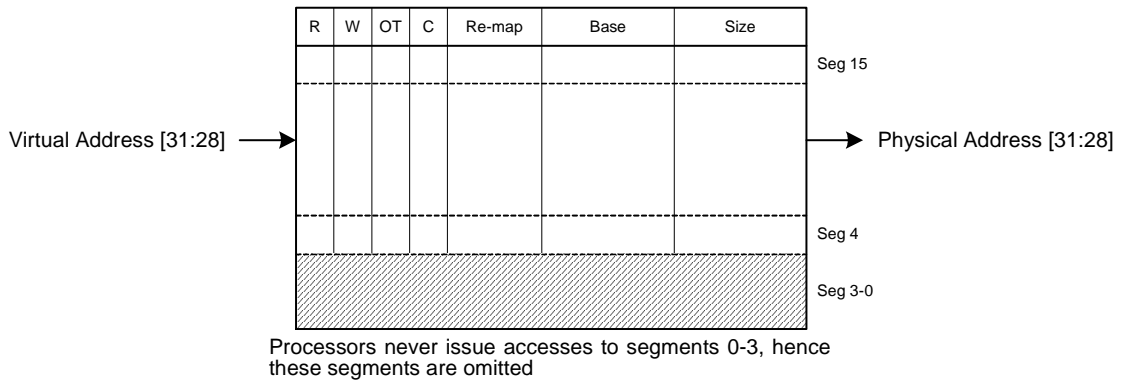


Figure 4-10: Processor's segment table

As mentioned earlier, segment 2 of the physical address space is dedicated to configuration locations. No virtual segment can be mapped to this physical segment. It is only accessible by special memory operations, RawLoad/RawStore. These operations ignore the segment table and directly access the physical memory. In other words, the processor generated memory address is treated as the actual physical address, which can be the off-chip memory address or address of a memory mat in the system (depending on the segment number).

After translating a virtual address to a physical address, a second step determines which physical location(s) should be accessed to complete the memory access instruction. The mapping depends on the addressing mode of the memory operation issued by processor, which is specified either by the segment table (by Cached/On-Tile bits) or by the TIE opcode of the memory operation.

If the accessed memory segment is un-cached, the physical location accessed is identified by the physical address. If this address lies in the Tile's address region or if it is forced to go to on-Tile memory mats by setting OT bit in the segment table, the access is sent to the target memory mat specified by physical address. If the address lies in a memory mat in a different Tile or Quad, a help request is composed and sent to that Quad's protocol controller to access the location on behalf of the processor. The protocol controller sends back a reply to the interface logic after completing the access.

If the accessed memory segment is cached, the configuration of the cache dictates which memory mat(s) should be accessed. A set of cache configuration registers specify the following cache parameters:

- Number of ways (maximum is 4 ways)
- Tag mats: which mat stores the address tags (for each way)
- Data mats: which mats store the cache line data (for each way)
- Cache line size: can be 16, 32, 64 or 128 bytes
- Number of data mats in each cache way: can be 1, 2, 4 or 8 mats

Provided this information, the interface logic can correctly slice the physical address to the exact addresses for both tag and data mats and identify which memory mats to route the access to. Figure 4-11 shows an example cache configuration, with two ways, a 16-byte line size, and two data mats per each cache way. Total size of the cache is 16KB. Memory mats 0 and 3 store cache tags and memory mats 1, 2, 4 and 5 store cache line data.

The address slicer inside the processor interface uses the cache configuration information in order to generate the necessary signals for accessing cache. Table 4-3 lists these signals.

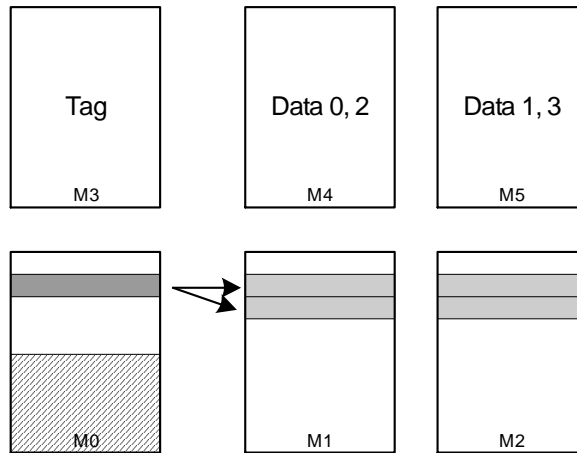


Figure 4-11: An example two-way cache configuration

Parameter	Description
Tag mat mask	Identifies memory mats storing cache tags, used by crossbar for routing accesses to tag mats
Data mat mask	Identifies memory mats storing cache line data, used by crossbar for routing accesses to data mats
Tag mat index	Used as address input for tag mats
Data mat index	Used as address input for data mats
Tags	Actual cache tags, used as data input for tag mats (for comparison)

Table 4-3: Cache access signals generated by address slicer

In addition to the segment table, the TIE opcode of the memory access instruction issued by processor also might implicitly specify or affect the address mapping mode. The TIE opcodes that have such effects are listed below:

RawLoad / RawStore: These opcodes completely bypass the segment table and translation and are sent to the address specified by processor. In other words, the processor generated address is considered as physical address for these opcodes and no translation takes place. The mapping mode for these instructions is direct mapping hence they never go to a cache.

FIFOLoad / FIFOStore: These opcodes should always be used for a virtual segment that is mapped to on-chip memory. Segment table produces a mat number according the base memory mat for the segment. This mat is then accessed as a FIFO: FIFO select control signal is activated and mat's input address is discarded.

Cache control instructions (DIWB, DIWBI): These instructions use the cache way mapping mode and explicitly specify the cache way and index that they access. Therefore, the access is not routed to all ways of the cache (if more than one). Instead it only goes to the way specified by the instruction.

4.5.2. DEFINING MEMORY OPERATIONS

The processor interface logic defines the semantics of processor's memory instructions by specifying how these instructions should be carried out and what are their associated success/failure conditions. If a memory access instruction fails for any reason, the interface logic either throws an exception back at the processor or sends a request message to protocol controller asking for assistance in completing the access.

The processor issues a TIE opcode for each memory access instruction, which specifies the type of the instruction. The processor interface converts this opcode into actual operations that memory mats must perform on their internal data and control arrays. For each TIE opcode issued by processor, the interface logic generates data, control and PLA opcodes for all sets of memory mats that should be accessed and specifies the operations performed on data and associated state (control bits), as well as how the state information should be updated if necessary. This mechanism is referred to as opcode translation. For each memory access instruction, potentially three sets of mats can be accesses (tag, data, auxiliary). Therefore, the opcode translation mechanism specifies necessary control signals for each one of these sets. The crossbar routes the generated control signals to all the memory mats within each set using its multi-cast capability.

Necessary inputs for the opcode translation mechanism are the TIE opcode from processor and the *Cached* and *On-Tile* bits from segment table. Each set of outputs consists of Data, Control and PLA opcodes, as well as Control In, Mask and FIFO select signals (Figure 4-12). These signals along with the mat mask and mat index signals generated by the address translation and mapping logic provide all necessary signals for accessing memory mats.

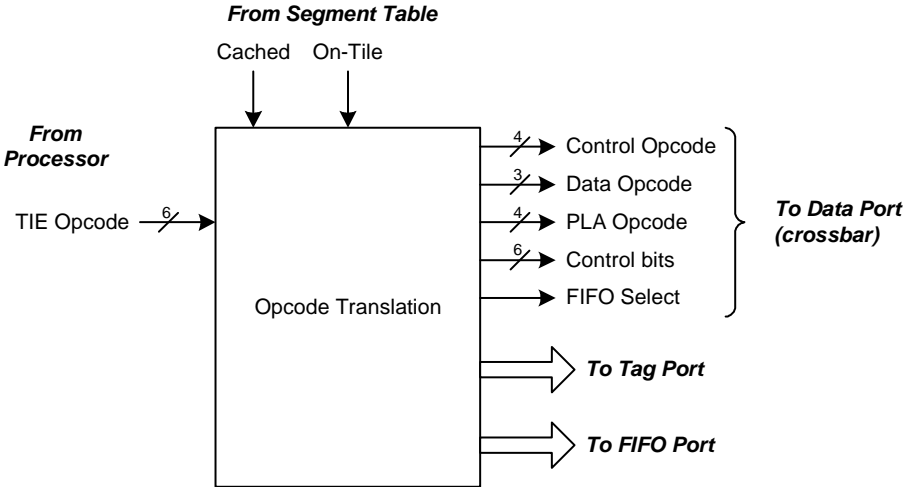


Figure 4-12: Inputs and outputs of the opcode translation mechanism

Logically, the translation mechanism is an array of configuration registers indexed using processor issued TIE opcode. Depending on the TIE opcode and *Cached* and *On-Tile* bits from segment table, each of the tag/data/aux accesses might be enabled or disabled. For example, when a memory access goes to an un-cached memory segment only the data access is activated, or when a cached segment is accessed, both tag and data access are enabled. By modifying the contents of this table, a user can change the operational semantics of each and every one of the processor memory access instructions. Additionally, since the table receives the necessary information about the configuration of the local memory mats from the segment table, cached and un-cached accesses that use the same TIE opcode can be altered independently.

4.5.3. DETECTING ACCESS FAULTS

After sending control signals to the target memory mats, the crossbar returns the responses back to processor interface logic, which analyzes the received signals and determines whether the memory operation was successful or not. The signals returned to the interface logic are *Total Match*, *Data Match*, *Control Out*, *FIFO Full* and *FIFO Error* outputs of the accessed set of memory mats. If the access is routed to more than one mat, the crossbar aggregates control signals from each set of accessed mats and returns it back to the processor interface. In doing the aggregation, crossbar returns the logical OR of the *Total Match* and *Data Match* signals for each set of the accessed memory mats. The logical OR of *Total Match* output from all tag mats serves as the hit/miss indicator when a cache structure is accessed.

Similar to opcode translation registers, a set of success or failure conditions are defined for each one of the processor's memory access instruction. These conditions are expressed as a set of bit vectors for each set of the accessed memory mats. The processor interface logic compares the returned control signals against these pre-defined bit vectors and determines whether memory access was accomplished successfully or not.

Logically this mechanism can be viewed as a content addressable table (Figure 4-13), which receives the TIE opcode from processor, *Cached* and *On-Tile* bits from segment table and the returned bit vectors from memory mats and produces a success/failure result. In addition, it also indicates whether processor should be stalled or not, whether a request message has to be sent to local memory controller, and the type of the request message.

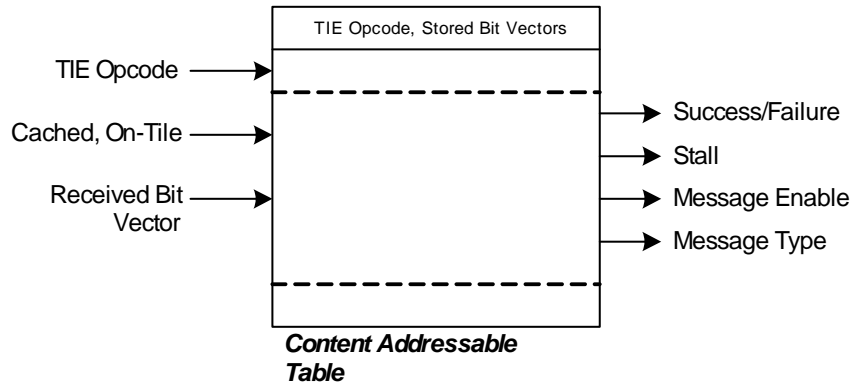


Figure 4-13: Detecting success or failure of a memory operation

4.5.4. PROGRAMMABLE REQUEST MESSAGES

If a memory access instruction is not successfully completed, the processor interface sends a request message to the protocol controller in the Quad to ask for assistance in completing the access. Table 4-4 lists the information fields that are forwarded to the protocol controller in the request message. The same mechanism that defines success/failure conditions generates the message type and the message enable signals.

Field	Description
Type	Identifies type of the request
Opcode	TIE opcode issued by processor as part of memory access instruction
Sender ID	Tile, processor and port ID of the sender
Address	Physical address of the memory location being accessed
Data	Write data, if memory access was a write
Byte Mask	A 4-bit mask, which indicates which bytes should be written
Tag Info	Information collected from tag mats if request is for servicing a cache. Includes <i>Total Match</i> , <i>Data Match</i> and <i>Control Out</i> from each way of the cache (32 bits total)
Blocking	Indicates whether processor is stalled for the memory access or not

Table 4-4: Fields of request messages to protocol controller

The type of the message and blocking indicator are extracted from the content addressable table that detects the success or failure of the operation. The tag

information is supplied by crossbar, and the rest of the fields are extracted from the original memory access instruction issued by processor. After composing, the message is placed in a FIFO structure that implements the status holding registers. The head of the FIFO participates in the arbitration for accessing a port to the protocol controller and is sent to the protocol controller after winning the arbitration. The request sits in the FIFO structure until the processor interface receives an acknowledgement signal from the protocol controller. The acknowledgement indicates that controller has received and registered the request and its processing is started processing.

In order to avoid having a large number of status holding registers inside the processor interface logic, non-blocking memory access instructions (e.g. a Store instructions) are taken out of the status holding registers after they are received and accepted by the protocol controller. The processor interface then proceeds with sending the next request message to the protocol controller. However, even though the status holding register is released without waiting for the reply, a counter keeps track of the number of outstanding requests sent to protocol controller. This counter is incremented after sending a request message and is decremented after receiving a reply message. It allows the processor interface logic to enforce ordering regulations that only require knowledge about number of outstanding requests, e.g. memory fences, but since complete information about non-blocking requests are not maintained, not all the memory orderings are possible to enforce⁷.

For blocking memory access instructions (e.g. ones that need to return a result to processor) access fault detection mechanism should stall the processor after detecting the failure. In that case the information of the request is kept in the FIFO structure until the actual reply is received. Usually all the read accesses from the processor are blocking operations. Among the write instructions, FIFO Store, Sync Store and Set Store are defined as blocking operations, while the rest of write accesses are treated as

⁷ Due to this limitation, the memory consistency model in the resulting architecture can only support weak ordering and sequential consistency

non-blocking. A configuration register allows the user to control blocking or non-blocking property of each type of the write operations separately.

4.5.5. INTERRUPT INTERFACE

The processor interface logic has an interrupt interface to each of the processors in the Tile (Figure 4-14). Each processor receives a 16-bit active high interrupt signal, which allows the processor interface logic to independently issue any combination of interrupts to any of the processors.

Interrupts are generated in two different situations: First when an error occurs during execution of a memory access instruction. For example, if the segment offset exceeds the segment size (when a virtual segment is mapped to on-chip memory), or when processor does not have the necessary permission to access the segment. In such cases the processor interface kills the memory access instruction and generates an interrupt for processor.

The second situation is when the memory system cannot handle a memory operation on its own and needs to run a handler code on the processor in order to complete a memory access. Such situations are usually encountered when implementing complicated memory models such as transactional memory. For example, when a transaction encounters a data dependency violation, or if it overflows its local write buffer an interrupt is generated for the processor to run handler code and resolve the situation in software. Such interrupts are programmable and are requested by the protocol controller.

When sending an interrupt to processor, the protocol controller can select between *hard* or *soft* interrupts; while soft interrupts are essentially normal interrupt requests, hard interrupts force the receiving processor out of stall if processor is waiting on a memory operation. When receiving hard interrupts, the processor interface logic unstalls the processor and immediately passes the interrupt signal to it. The only exception is processor stalls due to instruction fetch; if processor is stalled on an

instruction fetch, the interface logic waits for the fetch reply in order to un-stall the processor and then passes the interrupt.

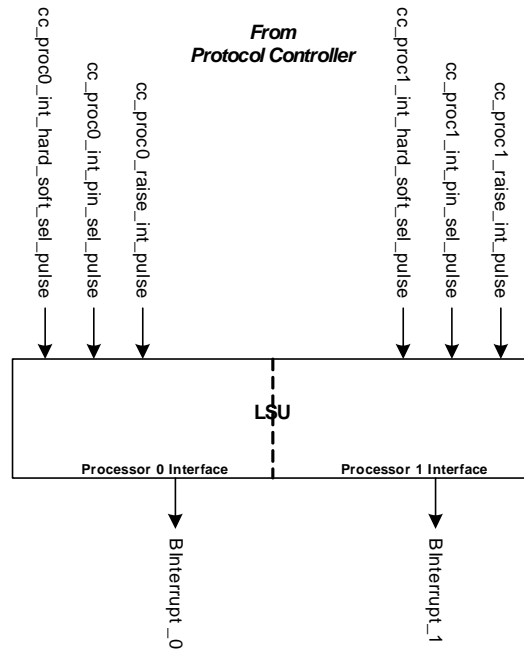


Figure 4-14: Interrupt interface to processors

4.6. PROTOCOL CONTROLLER

The protocol controller in Smart Memories implements the L1 controller in the universal memory system. The only slight difference with the abstract view shown in Figure 3-1 is that the controller is shared between all the processors in a Quad. It implements the memory ISA that discussed in the previous chapter. Conceptually, each request message when received invokes a “subroutine” that executes a series of basic operations. After completing the execution of the handler subroutine, either the request message is serviced and the appropriate reply is sent back to the sending processor’s interface logic or it is forwarded to the next level controller for completing the request.

In this section we describe the organization of the protocol controller and its interfaces to the Tiles and outside world. We elaborate on how it implements the abstract ISA discussed in the previous chapter by presenting the structure of the status holding registers and the embedded functional units which implement the basic memory operations. We also explain how the controller is programmed and how a sequence of basic operations can be put together to handle an incoming request or reply message.

4.6.1. ORGANIZATION

Figure 4-15 illustrates the internal organization of the protocol controller. The execution core of the controller consists of three major units: tracking and serialization (T-Unit), state update (S-Unit), and data movement (D-Unit). All basic memory operations are implemented by these three units except the communication primitives, which are implemented in processor and network interfaces. The tracking and data movement units have dedicated storage structures: *Status Holding Registers* for storing request tracking information and *Data Buffers (Line Buffers)* for storing blocks of data. In addition, the controller is equipped with eight independent DMA channels which essentially are programmable request generator engines, as well as a dedicated interrupt unit which is responsible for sending interrupt requests to processors. Communication with the processor interface logic in each Tile is handled by the processor interface unit. This unit receives request messages from Tiles and sends back replies when the sequence of operations in the controller is completed. Communication over the network is handled by network interface unit.

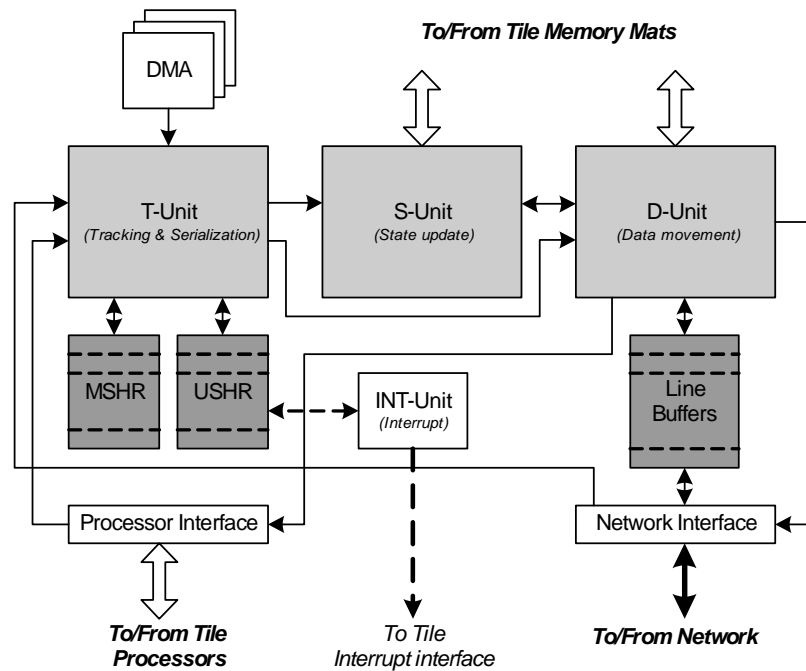


Figure 4-15: Internal organization of Quad's Protocol Controller

The state update and data movement units have interfaces to access the Tiles memory mats. These interfaces are connected to the Tile crossbar as shown in Figure 4-8. The S-Unit interface is 32-bits wide while the D-Unit interface is 64-bits wide and can access two adjacent memory mats in parallel. It also supports 32-bit accesses to a single memory mat.

4.6.2. SEQUENCING OF ACTIONS

The conceptual programming model of the controller is set of subroutine calls, triggered by an input message. Each subroutine composes a few basic operations and is executed by one of the internal functional units. After executing its own subroutine, each functional unit invokes another subroutine in the next functional unit by passing an appropriate request to it. Functional units use a *type* field when invoking a subroutine. This field essentially is the name of the function to be performed and determines the operations to execute. A sequential execution semantic is maintained within each subroutine.

Figure 4-16 depicts a conceptual execution model in the controller. Request message *foo* invokes subroutine *foo* in the processor interface unit. This subroutine calls subroutine *A* in T-Unit, which calls subroutine *B* in S-Unit and so on so forth. The right side of the figure shows operations in the subroutine *I* of the D-Unit. Calls to other subroutines are placed at the end and, as shown in the figure, two or more calls to different units can be made concurrently at the end of a subroutine. The lower part of the figure shows the internal steps of a call inside the controller. In this example, processing of the message ends after the N-Unit sends a request message *bar* to the main memory controller. The protocol controller then waits for the reply to this request message from the memory controller and completes the processing by executing another set of subroutines after receiving this reply.

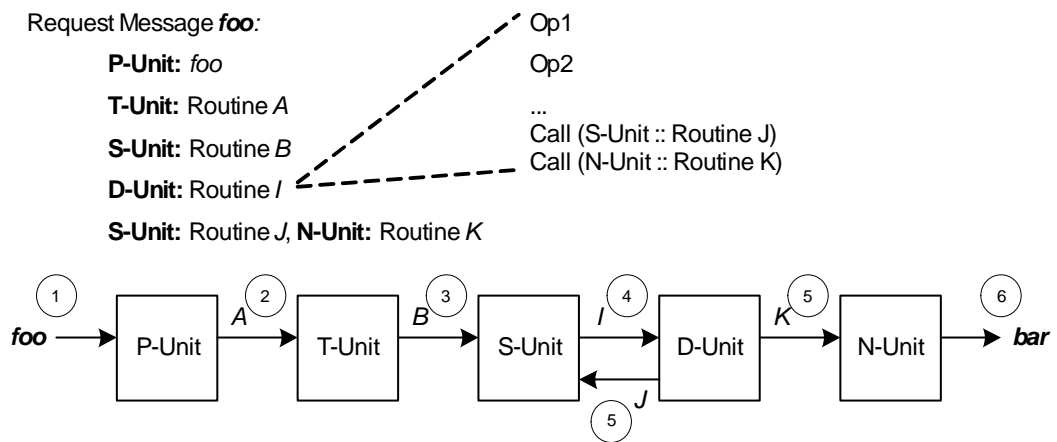


Figure 4-16: Conceptual execution model of the protocol controller

4.6.3. SUPPORTED OPERATIONS

The operations within each unit are controlled by an internal configuration (or program) memory. Similar operations in the controller's ISA mentioned in the previous chapter are grouped and mapped to a specific functional unit. The following

explains the grouping and the set of operations implemented by each of the internal functional units.

Tracking Unit (T-Unit)

The tracking unit serves as the entry port to the execution core of the controller. All request/reply messages that are received by the controller from processors, network or internal DMA channels are passed to the T-Unit. It implements the operations in the ISA that are related to management of internal data structures: allocation of the SHR and data buffer entries, storing or retrieving the tracking information for the input requests and replies, performing lookups in the SHR structure and enforcing any serialization properties that might be required by the memory protocol.

The T-Unit consists of two independent parallel sections, CT (T-Unit, Cached) and UT (T-Unit, Un-cached). The CT section handles memory requests that need a form of serialization or ordering. Specifically when an input request needs to be checked against already outstanding memory requests, such as cache misses or coherence requests, it is handled by CT. In contrast, the UT is used for handling memory requests that only need to store and retrieve their tracking information and do not obey any specific ordering requirements. Such requests can be completed out of the issuing order for performance reasons. DMA requests from DMA channels or un-cached accesses from processors are examples of the latter group.⁸

Each of the CT and UT sections has its associated status holding registers for storing tracking information of the received requests. CT uses *Miss Status Holding Registers* (MSHR) and UT uses *Un-cached request Status Holding Registers* (USHR) for this purpose. The major difference between the two structures is that MSHR provides an associative lookup operation to check the address and sender of the received request against already outstanding requests in a single cycle, while USHR only provides read/write operations. In addition to the tracking information, these structures also

⁸ If out of order completion of processor's un-cached requests is allowed by the implemented memory consistency model

keep the internal status of the outstanding requests, which is checked and updated by different functional units as the request is passed from one unit to the next.

After receiving an input request, the CT evaluates certain conditions by performing a lookup operation on its associated status holding registers. The provided state information includes whether:

- There is another request to the same memory address present in MSHR and the index of the matching register (if a match is found)
- The new request can be merged with existing one (if a match is found) In other words, are the two requests to the same address of the same type
- There is another request from the same processor present in the MSHR
- There are any available registers in the MSHR
- There are any available data buffers for the input request

After collecting this information, the CT proceeds to execute the operations specified by the requests handling subroutine. Operations that are mapped to CT fall into three major categories:

Request acceptance: A set of CT operations is used to decide whether to accept an input request or not. Acceptance operations can evaluate any combination of state bits mentioned above and decide either to accept or reject the input request. If a request is not accepted, it is supplied again by the issuing unit (P-Unit, N-Unit or DMA channel) and is retried in the next clock cycle.

Storing and retrieving tracking information: These operations manage the Miss Status Holding Registers by allocating registers, writing tracking information into an allocated register, or retrieving the tracking information of a request using the received register index.

Handling data storage: In addition to the status holding registers, the CT manages the line buffer structure which is used as temporary storage for data blocks. It implements the necessary operations for allocating and writing data into data buffers.

The UT section of the T-Unit operates more or less the same way as CT. The operations supported by UT are:

Request acceptance: The UT provides operations for checking the availability of the Un-cached request Status Holding Registers. These operations are used to ensure that there are available USHR entries before attempting to write tracking information into the USHR.

Storing and retrieving tracking information: Similarly the UT provides operations for allocating and managing USHR entries and writing/reading the necessary tracking information about an input request.

A common set of operations supported by both the CT and UT sections of the T-Unit is the ability to invoke another subroutine in the next functional unit. Such invocation is performed by passing a *type* field along with the parameters of the received request. CT can invoke a subroutine in S-Unit and D-Unit, while in addition to these two units UT can invoke a subroutine in N-Unit, P-Unit and any one of the DMA channels.

State update unit (S-Unit)

This unit provides operations to access the Tile memory mats in order to read, write or update state information. The S-Unit operates on the state information associated with a block of data, such as tags and line state information in a cache structure. State information associated with individual data words are accessed and operated on by D-Unit. The S-Unit has a dedicated port to each Tile's memory mats and similar to processor's interface logic, can access any number of memory mats in parallel using a bit mask. In particular, it supports four mapping modes when accessing Tile memory mats:

Direct: Access goes to a single memory mat, specified by the received memory address

Cache: Access goes to all tag mats in a defined cache structure. It can be an instruction or data cache of either processor in the Tile

Cache way: A single tag mat in the specified cache way is accessed

FIFO: A predefined FIFO mat is accessed

S-Unit supports the following operation:

Memory mat accesses: The S-Unit provides all necessary signals for the accessed memory mats and always reads the state information back from the accessed mats. It can update the state information using a plain write operation in the mat's control array or by using read-modify-write logic in the mat itself. Furthermore, the S-Unit can send a memory mat access either to a single Tile or all the four Tiles simultaneously. Simultaneous accesses to all Tiles are useful when looking for specific cache blocks in all the Tiles (when implementing a shared memory model) or invalidating/downgrading them upon receiving a cache miss or a coherence request.

Flow control: After the memory mats are accessed, the state information collected from all the accessed mats (meta-data bits) is returned back to the S-Unit. Flow control operations compare the received bit vector against a set of pre-defined bit vectors and invoke an appropriate subroutine in the next functional unit. One can think of this operation as a `case` statement in high-level programming languages, where an expression is compared against a set of labels and the action defined by the matching label is executed. In our case, labels are pre-defined bit vectors and the action is a subroutine invocation in a specific functional unit.

Data movement unit (D-Unit)

The D-Unit is a data movement engine which moves blocks of data between the Tile memory mats and line buffers inside the protocol controller. It has a dedicated 64-bit port to each Tile's crossbar and can access a 64-bit word (two adjacent memory mats in parallel) for faster block transfers. It supports the following operations:

Data block accesses: The D-Unit can transfer a block of memory from Tile memory mats to the line buffer inside controller or vice versa. All necessary signals for the memory mats are generated by D-Unit. Supported addressing modes for these accesses are *cache way* or *direct*. In addition to single block read and block write operations, the D-Unit supports transfer operations where a block is read from one Tile and is written into another one. The operation is staged through the line buffer to minimize the transfer latency.

Data word accesses: The D-Unit also can access a single memory mat in a specified Tile. Similar to S-Unit accesses, all the necessary control signals for the memory mat are generated. The D-Unit can read, write or update the state information in addition to reading and writing the data word. Supported addressing modes for such accesses are *direct* and *cache way*.

Flow control operations: D-Unit can read and compare the state information associated with an individual data word against a set of pre-defined bit vectors. Depending on the comparison results, it can invoke a subroutine in the next functional unit (the same as S-Unit).

Network interface unit (N-Unit)

The network interface provides communication primitives to talk with other Quads or main memory controllers. It consists of separate transmit and receive sections which operate independently. The receiver receives messages, decodes them and passes each message to the T-Unit. When the incoming message carries a data block, the receiver places the received data in a line buffer entry before passing the message to T-Unit.

The transmitter receives requests from internal functional units to send desired messages to an outside entity. It is capable of sending short messages, which at most carry one data word, or long messages that contain a whole data block. When sending a data block, the transmitter reads the data from specified internal line buffer entry. A user can program the transmitter to send either short or long messages, adjust the information fields that are sent in the message header (each message type might require different information to be included in the message), and whether to release the status holding register occupied by the request after sending the message (reply messages to other controllers should release the register after they are sent). In addition, a user can select which virtual channel to use.

Processor interface unit (P-Unit)

The processor interface unit is a very simple interface logic that consists of two parts: the front-end and the back-end. The front-end of the P-Unit acts as a receiver, which receives requests from processors in the Tiles, decides whether a request should be passed to CT or UT section in the T-Unit, and arbitrates between received requests to determine which request should be passed on. For each received processor request, a user can program whether the request should be passed to CT or UT in T-Unit and what subroutine in CT or UT should be invoked by the message. In other words, P-Unit only supports *call* operations to pass input request to T-Unit. The back-end simply passes the replies generated by internal functional units back to the originating processors.

4.6.4. STATUS HOLDING REGISTERS AND DATA BUFFERS

As discussed in the previous chapter, controllers in the memory system should have internal registers for keeping tracking information of the outstanding requests. The protocol controller has status holding registers (MSHR and USHR) for storing the tracking information of cached and un-cached requests respectively, as well as temporary data storage (line buffers).

As was mentioned earlier, the status holding registers are divided into MSHR and USHR structures and are managed by T-Unit. MSHR is used for storing requests that need a form of ordering and serialization (e.g. cache miss requests) and information stored in it are used by CT section in the T-Unit. Information stored in each MSHR entry is divided into two categories: request tracking information and request status information (Figure 4-17).

The tracking information holds different parameters of the request while status information shows the current status of the request in the system and how the system should handle it at each stage. Table 4-5 describes each of these information fields.

The MSHR has separate read and write ports and supports read and write operation on each entry using separate indices. In addition, it has an associative lookup port based on the Address and Requestor fields and can detect any entry that has a valid request to the same address or is from the same requestor. The lookup port reports the result of the matching back to CT in the *Result Flag*.

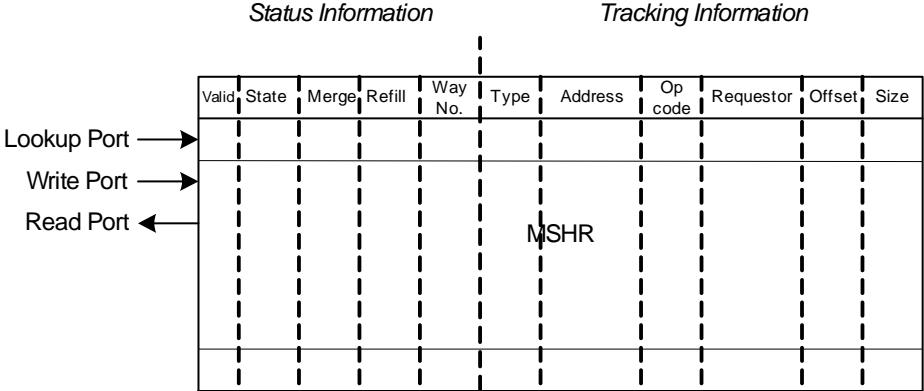


Figure 4-17: MSHR structure

Field	Description
Address	Stores the address of the memory request, which can be the address of a memory word or a memory block, such as cache line address
Opcode	TIE opcode of the memory request issued by processor
Type	Type of the input request passed to CT (name of the subroutine invoked in CT)
Requestor	Specifies source of the request, Tile, processor and port ID (instruction/data)
Offset	If request is a cache miss, indicates the offset within the cache line
Size	Size of the memory block, if request is for a block of memory
Valid	Indicates whether this register contains a valid request
Refill	For cache miss requests specifies whether requesting cache should be refilled or not
State	ACTIVE or OUTSTANDING. A request is in active state if it is currently being processed inside controller. A request is in outstanding state if it is waiting for a reply from other Quads or main memory controllers
Way number	For cache misses only, specifies the way of the cache which data should be refilled
Merge	An optimizing flag that says whether later requests to the same memory address can be merged with this request or not

Table 4-5: Information fields in MSHR

The USHR is a similar structure operated by UT and only supports simple read and write operations. It stores information about un-cached memory access request from processor or DMA channels (Figure 4-18). Table 4-6 lists and explains the information fields stored in the USHR.

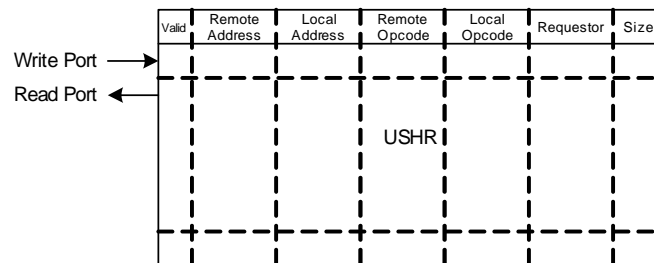


Figure 4-18: USHR structure

Field	Description
Remote Address	Address of the memory location to be accesses (in other Quads or main memory)
Local Address	For DMA requests, specifies the local address for the DMA transfer ⁹
Remote Opcode	TIE opcode for accessing remote memory location
Local Opcode	TIE opcode for accessing local memory location
Size	Size of the memory block, if request involves moving a block of data
Requestor	Identity of the requesting entity (Tile, Processor and Port ID for processor, and channel number for DMAs)
Valid	Flag that indicates whether the register contains a valid request

Table 4-6: Information fields in USHR

Registers in both MSHR and USHR structures are divided into two separate pools of entries. The *outgoing* pool consists of entries which store tracking information for requests generated in the Quad by processors or DMA channels. The *incoming* pool is the set of entries used for storing tracking information of the requests received from other Quads and main memory controllers. Allocation of registers to pools is independently controlled for each structure via configuration registers in the T-Unit. The (*Outgoing*) register allows the size of the pools to be adjusted by user when configuring the system and allocating system resources. For each of the MSHR/USHR structures, register indices between 0 and *Outgoing-1* form the outgoing pool and the rest form the incoming pool.

While the tracking information is stored in MSHR and USHR structures, data blocks are stored in a different line buffer structure associated with the data movement unit. Even though the structure is physically associated with the data movement unit, it is allocated and managed by the T-Unit along with the MSHR and USHR structures.

A line buffer entry consists of 8, 32-bit data words (total of 32 bytes). Each data word has 6 bits of meta-data or control information, similar to memory mats. These additional bits facilitate the movements of meta-data information between different

mats when such transfers are necessary. In addition, each byte within the line buffer has a byte valid bit which indicate that the location contains valid data (Figure 4-19).

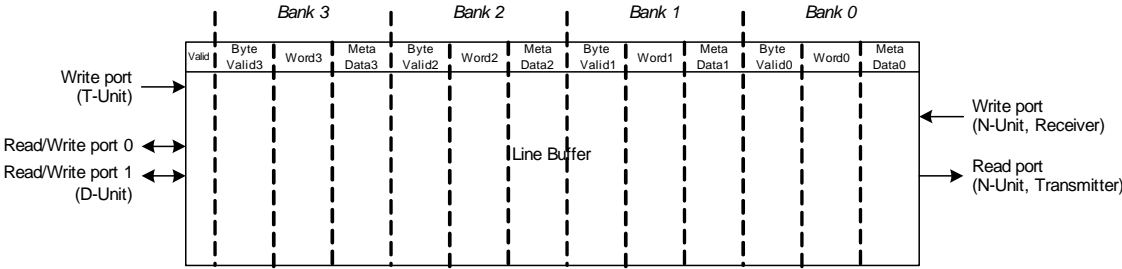


Figure 4-19: Line buffer structure

The line buffer is accessed mainly by the data movement engine and the network interface, since these are the major units involved in a data transfer operation. The tracking unit also has a write port into the line buffer which is used for placing a processor’s write data when receiving write requests from processor, e.g. a Store miss request. Using the byte valid bits, the line buffer later can combine write data with the rest of the cache line when it is received from main memory.

As mentioned before, functional units inside protocol controller communicate by passing requests and invoking subroutines. Each functional unit internally has a configuration (or program) memory which stores the subroutines for all the request types it might receive. The configuration sets the operations that each functional unit has to execute after receiving an input request and also specifies parameters for each operation, such as the addressing mode for accessing memory mats or virtual channel number for a network message that has to be sent.

The configuration memories of the protocol controller are mapped to the segment 2 in the physical address space and are accessible from processors by issuing RawLoad and RawStore instructions. In addition, the controller provides a configuration

⁹ A DMA transfer always moves data between a memory location inside the Quad (local) and a memory location in other Quads or in off-chip memory (remote)

interface which through the Quad's JTAG controller allows a user to access all the configuration memories by issuing JTAG read/write operations even while the system is operating.

4.7. MAIN MEMORY CONTROLLER

The main memory controller is the controller connecting Quads to the main, off-chip memory. In addition to serving as the interface to main memory, it implements the same basic memory operations mentioned in the previous chapter, with a structure similar to the Quad's protocol controller. When the system is configured with more than one memory controller, the addresses are interleaved among different controllers. Therefore all controllers are shared among all Quads in the system and act as lowest level convergence/serialization point for memory requests they receive. The execution model of the main memory controller is also the same as Quad's protocol controller. An incoming message triggers a set of operations in the network receiver and is then passed from one functional unit to the other, until all necessary operations are completed and a reply is returned to the originating Quad.

4.7.1. ORGANIZATION

Figure 4-20 shows the internal organization of the main memory controller. Similar to protocol controller, related operations are mapped to the same functional units inside the controller: the C-Req unit manages the status holding registers, performs serialization operations, and generates necessary requests for Quads to inquire or updated state information. The C-Rep unit gathers replies, composes the resulting state information and decides how to proceed depending on the results. U-Req/Rep unit handles requests that only need to access main memory without any serialization or state update operations. There is a dedicated functional unit for implementing a fine-grain synchronization protocol. This unit operates rather independent of the rest of the controller. It has its own tracking structure and all of its supported operations are related to managing and searching this storage structure.

The main memory controller communicates with Quads by exchanging messages over the communication network. It uses the same network interface logic as the protocol controller with separate transmitter and receiver sections.

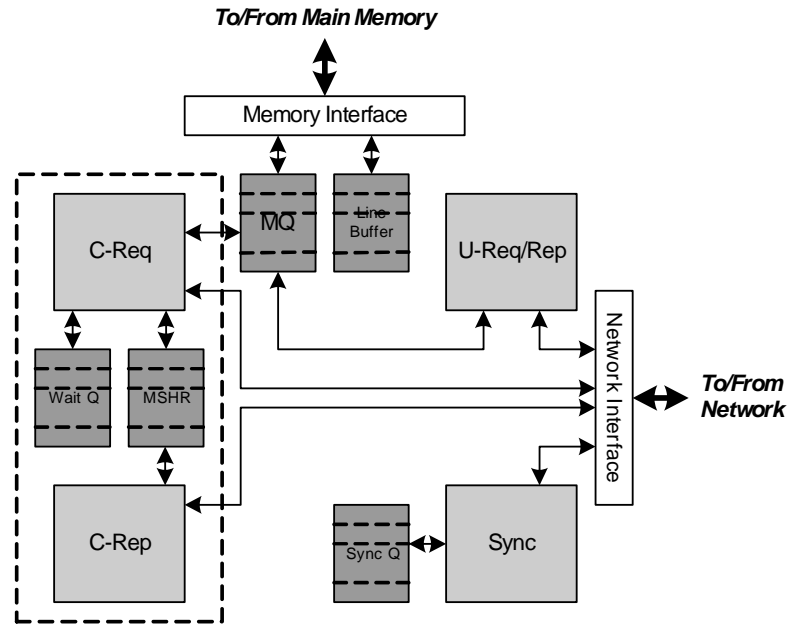


Figure 4-20: Internal organization of main memory controller

A dedicated memory interface unit performs accesses to main memory. Functional units that need to access main memory use a memory queue structure (MQ) which drives this interface unit. Memory requests received by controller specify a global physical memory address. Therefore, when memory is interleaved among more than one controller, the memory interface unit makes necessary adjustments to the address in order to access the correct data word or block in the associated memory bank. The details of the operations and structure of status holding registers in memory controller are more or less the same as Quad's protocol controller and hence are not discussed here.

4.8. MAPPING MEMORY PROTOCOLS

After describing the architecture of the memory system and the flexible mechanisms embedded in different components, this section discusses the necessary steps for implementing memory protocols using the hardware. Specifically it explains how to provide the semantics requirements for a shared memory system, a streaming system and a transactional memory system. While discussion in this chapter focuses on the high-level usage of the flexible constructs, the details of configuring the hardware structures and values that should be written to the configuration registers to provide a specific functionality are described in Appendix B, as an example of implementing a simple coherence protocol.

Necessary steps for mapping a protocol are:

1. Defining and associating state information: If the desired memory protocol requires to have state information associated with data words or blocks, the first step is to determine what this information is and how it is mapped and stored in the local and main memory.
2. Allocating resources: This step essentially determines the configuration of the memory mats in the Tile and how they are structured by setting up the configuration registers in the processor interface logic. The address translation mechanism which converts virtual addresses to physical addresses also is configured at this step.
3. Defining memory operations: The next step determines the processor and controller operations on the local and main memory, and defines the necessary accesses that should be issued to memory mats. It also defines the success/failure condition for each memory operation. The opcode translation table (Figure 4-12) and content addressable table in Figure 4-13 in the processor interface logic, as well as configuration memories in the S-Unit and D-Unit of the protocol controller which access Tile memory mats, are populated at this step.

4. Defining and handling communication messages: The final step specifies the messages that are exchanged between processor interface login and Quad's protocol controller, as well as between the protocol controller and main memory controller. For each received message, the controllers should also be programmed to carry out necessary operations. This involves developing appropriate subroutines for each of the controller functional units and connecting them to each other by making appropriate invocations.

4.8.1. STREAMING MEMORY SYSTEM

A stream memory system has the simplest hardware requirements among the implemented models. When implementing streaming memory system, Tile local memory mats serve as the storage for kernel code and local stores for streams. There is no state information associated with data words or blocks. Segment table maps desired segments of the virtual address space into memory mats in the Tiles.

Processors issue Load and Store operations which are translated by the opcode translation mechanism to read and write accesses on the target mat's data array. The protocol controller serves the DMA requests that are generated by the DMA channels by reading and writing data blocks in different Tiles. The DMA channels are configured by processors via writes to their control registers. All operations on the Tile memory mats by processors and protocol controller are successful.

Table 4-7 lists the communication messages that are exchanged between different components for all three implemented memory models. Processors access memory mats in other Tiles by sending a request message to the protocol controller. DMA channels support sequential, strided and indexed gather/scatter operations. They issue index read requests to acquire the address of the next data element and then generate necessary gather/scatter requests to move the data blocks. Gather/Scatter replies are sent by the main memory controllers or protocol controllers in the other Quads after processing of the request message is completed.

Model	Source	Message	Description
Shared Memory (MESI coherence)	Proc.	Cache Miss	Read/Write miss request from a processor
		Upgrade Miss	Upgrade miss request (request for ownership)
		Prefetch	Prefetch for read or write from a processor
		Cache Control	Invalidate/Writeback a specific cache line
	Main Mem Cntrl	Coherence Request	Read, Read-Exclusive or Invalidate request for specific cache line
		Refill	Returns cache line data to be refilled
Upgrade		Returns cache line ownership (no data)	
Streaming	Proc.	Un-cached Access	Direct access of a memory in another Tile
	DMA Channel	Index Read	Read of index memory (indexed transfers)
		DMA Gather	Request for gathering data from another Quad or main memory
		DMA Scatter	Request for scattering data to another Quad or main memory
	Main Mem Cntrl / Another Quad	Gather Reply	Reply for a gather request, contains actual data
		Scatter Reply	Acknowledgement for a previous scatter
		Un-cached Reply	Reply for direct memory access from processor
		Net Gather	Gather request from another Quad's DMA
		Net Scatter	Scatter request from another Quad's DMA
TM (TCC)	Proc.	Cache Miss	Read/Write miss request from a processor
		FIFO Full	Address FIFO full indicator, overflow occurred
	DMA Channel	FIFO Read	Read store address from FIFO
		Commit Read	Read committed data from source cache
		Commit Write	Write committed data to other caches
	Main Mem Cntrl	Refill	Returns cache line data to be refilled
		Net Commit	Committed data word from another Quad's transaction

Table 4-7: Communication messages for implemented memory models

The protocol controller handles gather/scatter messages by first storing the tracking information of the request in the USHR (since no specific ordering between requests are required), and invoking the appropriate subroutine in D-Unit or N-Unit. For example, for scatter requests from a DMA channel, first D-Unit reads the data block from the source memory mat in the Tile into the line buffer. Then it invokes appropriate subroutine in the N-Unit to read the data from line buffer entry and send it to the destination Quad or main memory controller as a scatter request. When the scatter reply is received, it is passed to the T-Unit which de-allocates the USHR entry after retrieving the tracking information and acknowledging the DMA channel.

4.8.2. SHARED MEMORY SYSTEM

When implementing a shared memory system, the local memory mats in the Tiles are used for implementing instruction and data caches for processors. One mat per cache way stores the address tags while the other mats store the cache line data, as shown in Figure 4-11. Data array in the tag mats store the address tags while the control array stores the cache line state: Valid, Modified and Shared/Exclusive bits for MESI protocol. Configuration registers in the processor interface specify the exact configuration of the caches in terms of size, number of ways, and line size. Segment table maps the segments in the virtual address space into the caches by setting the C bit.

Processors access caches using Load and Store instructions. Each instruction is converted into a tag comparison operation on the tag mats and a data read/write operation on the data mats. Crossbar routes these accesses to appropriate mats. Tag mats compare the address tags and cache line state and generate a hit/miss signal (*Total Match* output of the comparator). This indicator is sent back from each way to the processor and is also sent over the IMCN to the associated data mats. For Store instructions, the write operation on the data mat is guarded by this signal, so that the write is discarded if there is no hit in the specific cache way.

The processor interface logic collects the state information extracted from each way of the cache and determines the cache misses and upgrade misses. It sends request messages to the protocol controller to refill the appropriate cache.

The protocol controller receives cache and upgrade miss requests from the processor interface logic and coherence requests from the main memory controller. As the first step for serving the request, the T-Unit looks up the MSHR structure, serializing the request against already outstanding ones. After ordering the request appropriately, an MSHR entry is allocated and the tracking information of the request is stored. Upon receiving a cache refill, it retrieves the information about the cache miss such that the data can be placed in the right location in the cache. S-Unit snoops the state of the

cache line in the other Tile caches to enforce coherence and check for the possibility of cache-to-cache transfers. It also writes the new tags after refilling a new cache line in the cache. The D-Unit extracts the evicted cache line from the cache, performs cache-to-cache transfers by reading the cache line from one Tile's cache into another's and writes the new cache line into the cache upon receiving a cache refill message.

The main memory controller fetches the cache lines or writes the lines received from Quads back to main memory. It also serializes requests from different Quads and sends coherence requests to enforce the coherence properties among the Quads. Appendix B describes the details of implementing a simple MESI coherence protocol for a single Quad system.

4.8.3. TRANSACTIONAL MEMORY SYSTEM

Smart Memories implements Transactional Coherence and Consistency (TCC) [27] protocol as its transactional memory model. When implementing TCC, memory mats implement the instruction and data caches for processor, very similar to the shared memory model. The data cache of the processor is augmented with a FIFO that stores the addresses of the transaction's write set. The control array in the tag mats encode the cache line state as Valid, Speculatively Read and Speculatively Modified. Speculation indicators are used to avoid eviction of speculative cache lines since all necessary dependency tracking information is stored in the cache. The control array of the data mats in the cache are used to associate same Speculatively Read (SR) and Speculatively Modified (SM) flags for each data word. These bits essentially mark the transaction's read and write sets in the cache and are used to detect conflicts between transactions. Configuration of the cache and setup of the segment table in the processor interface logic is similar to the shared memory model.

Processors once again issue Load and Store operations to the cache, but the opcode translation mechanism issues necessary opcodes for the control array to appropriately adjust the status of SR and SM bit associated with the data words. The hit/miss

indicator of each cache way is also forwarded to the FIFO mat to avoid placing address of the stores that miss in the cache in the FIFO. In addition to detecting cache misses, processor interface logic also monitors the address FIFO and notifies protocol controller when it becomes full.

Protocol controller receives cache miss requests from processors and commit requests from either a local DMA channel or another Quad's committing transaction. Cache misses are serviced by fetching the data from main memory and no snooping and coherence action happens in the controller. Transaction commits are handled by DMA channels similar to performing an indexed DMA operation: The address of a word is extracted from the FIFO by S-Unit, data word is read from the committing transaction's cache and is written into the cache of other Tiles by D-Unit. The word is also sent to main memory controller to be written into main memory. When writing the committed word in a cache, D-Unit checks the SR bit of the word that is being written. If the SR bit is set, a violation is detected between the two transactions and an interrupt is sent to the violated processor. T-Unit appropriately serializes commits against outstanding cache misses and stores and retrieves tracking information of the cache miss requests.

A major differentiating factor for the TCC implementation on the Smart Memories is that transactions' arbitration for acquiring commit token occurs in software, by accessing synchronization variables that are stored in shared local memory. Also, one processor in each Tile is reserved for handling asynchronous events, such as overflow of the hardware structures (cache and address FIFO), and transaction violation. This processor does not execute the code for the actual transaction and runs the necessary software handlers for resolving exceptional situations.

4.9. SUMMARY

In this chapter we explained Smart Memories, a scalable reconfigurable architecture which implements a universal memory model described in the previous chapter. We

presented the different components in the system, including processors, Tile memory mats and interconnect structure as well as memory system controllers. We discussed the internal organization of the controllers, their internal status holding registers and how the basic memory operations mentioned in the previous chapter as controller ISA are mapped to their internal functional units. We also described how these resources and operations are used in order to map specific memory protocols on the hardware. Appendix B provides more insight about implementing protocols by illustrating the details of mapping a simple coherence protocol on the hardware.

In general, while the hardware implementation of the ISA operations in the controllers are not difficult, the challenge mostly is in providing a micro-architecture which provides sufficient level of concurrency in processing request. Specifically in the case of Smart Memories, since the Quad's protocol controller is shared between eight processors, it potentially can become a bottleneck if it cannot provide the necessary throughput. Grouping related operations into separate functional units and passing requests from one unit to other allowed us to divide a handler routine into smaller subroutines carried out by each functional unit independently and hence provide a macro-level pipeline for processing input messages. Successive memory requests hence can be pipelined across different functional units to increase processing throughput. Also, concurrent subroutine calls by a functional unit as shown in Figure 4-16 allows overlapping different operations of the same memory request, increasing the level of concurrency and reducing the processing time.

As described, currently the Smart Memories system implements three different memory models: coherent shared memory, streaming and transactional memory. However it is possible to map other protocols that implement the same or even different memory models using the same hardware resources. An interesting experiment with this system is to create a comprehensive library of different memory protocols that system users can choose from. This involves developing the necessary hardware configurations as well as software interfaces, such as libraries and runtime systems that applications need for execution. Having such a comprehensive collection

not only allows the user to simply try and choose the best memory protocol that matches the desired programming model, but also allows direct comparison between performance and power characteristics of the application when employing different protocols or even different memory models.

The next chapter describes the results of implementing a single Quad of the Smart Memories architecture in the context of SMASH test chip. It also presents our evaluation of the architecture and the impact that the embedded reconfigurable mechanisms have on the over all system performance. We also try to estimate the power and area overhead that these mechanisms introduce in the Quad's protocol controller.

The interconnection mechanism between Quads and memory controllers in the Smart Memories architecture is assumed to be a mesh-like network. The infrastructure should satisfy the requirements mentioned in the previous chapter: being lossless and preserving point-to-point ordering. In order to connect multiple SMASH chips we developed a star topology and a central switch which allows connecting up to four Quads and four memory controllers. A detailed description of the system interconnect and central switch are discussed in Appendix A.

5. EVALUATION

The previous chapter explained the Smart Memories architecture and implementation of the basic operations in the memory system. This chapter describes the implementation results of the Smart Memories test chip, SMASH. It also evaluates the impact of embedding reconfigurable features on the performance, area and power of the resulting system.

5.1. TEST CHIP IMPLEMENTATION RESULTS

The SMASH test chip contains a complete Quad of the Smart Memories architecture, including four Tiles and associated protocol controller. There are 8 processor cores and total of 256KB of local memory in the test chip. The main memory controller and interconnecting logic are mapped on an external FPGA.

The test chip is fabricated in ST Microelectronics 90 nm technology (ST90nmGP) with worst-case clock cycle time of 5.5 ns (180MHz). Die dimensions are 7.77mm × 7.77mm (60.5mm²) Figure 5-1 shows a plot of the die and Table 5-1 summarizes the specifications of the test chip.

Figure 5-2 shows breakdown of the area for different modules in the test chip, including Tiles, protocol controller, I/O pads and routing channels. Figure 5-3 shows breakdown of the area for a single Tile and for functional units inside protocol controller. As illustrated, most of the Tile area is taken by memory mats, since we used regular flip-flops for implementing the gang-writable and conditional-gang-writable meta-data bits in the control array.

Technology	ST 90nm-GP (General Purpose)
Supply voltage	1.0 V
I/O voltage	2.5 V
Dimensions	7.77mm × 7.77mm
Total Area	60.5 mm ² (core size 51.7 mm ²)
Clock cycle time	5.5 ns (181MHz)
Nominal power (estimate)	1320 mw (300mw for Tile, 120 mw in protocol controller)
Number of transistors	55M
Number of Gates	2.9 M (600K in each Tile, 500K in protocol controller)
Number of memory macros	128 (32 per Tile)
Signal pins	202
Power pins	187 (93 VDD, 94 VSS)

Table 5-1: Test chip specifications

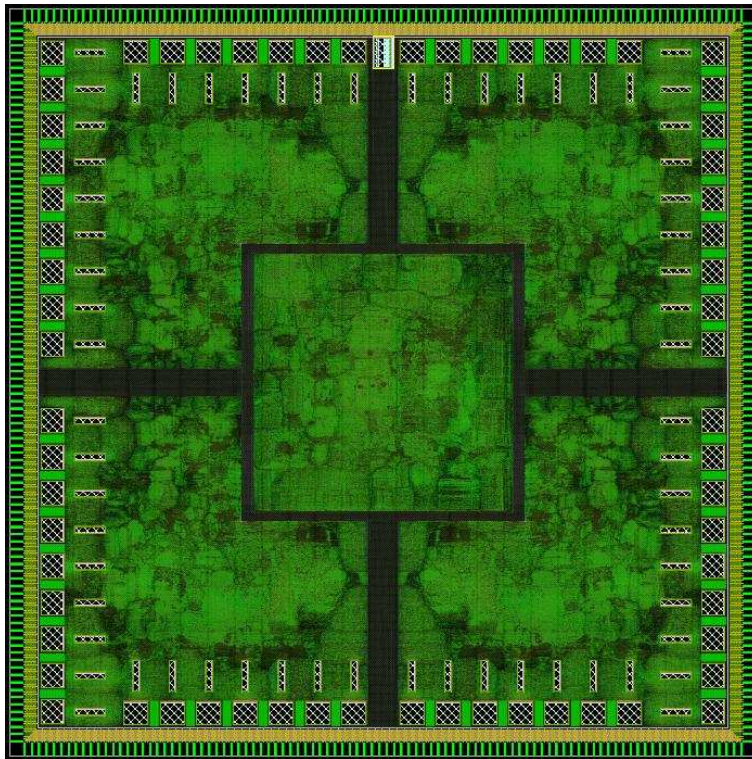


Figure 5-1: SMASH die plot

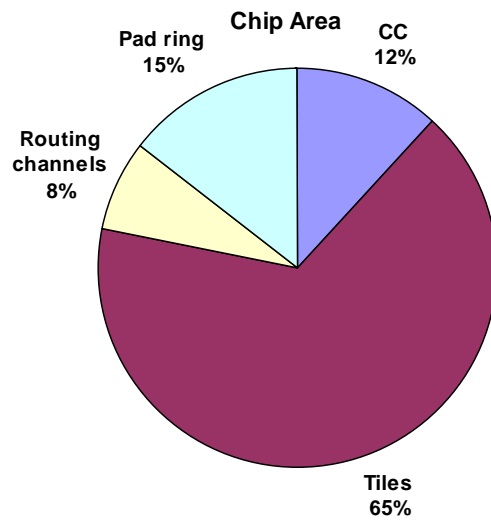


Figure 5-2: SMASH test chip area breakdown

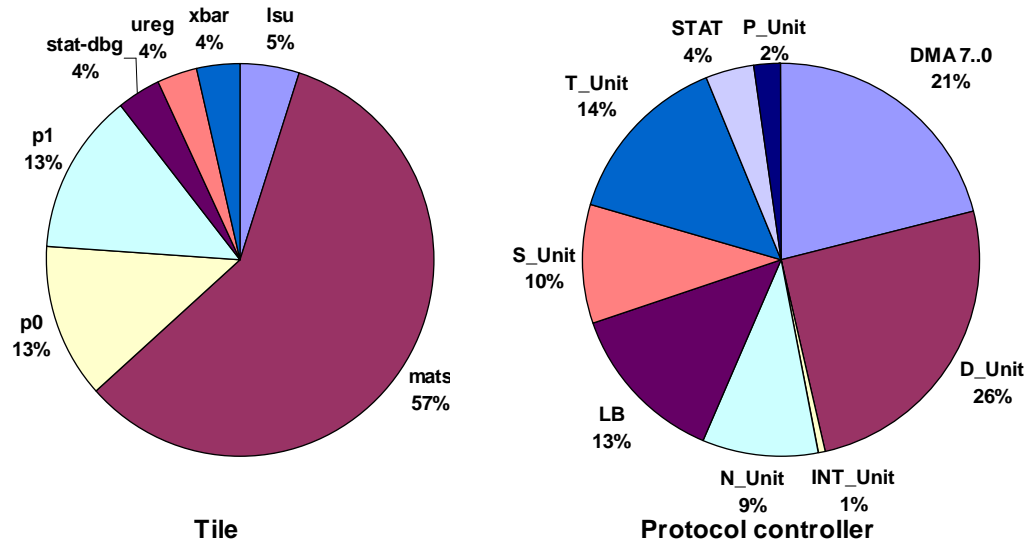


Figure 5-3: Area breakdown for Tile and local memory controller

5.2. PERFORMANCE OVERHEAD

In the Smart Memories system, local memory access times fit within the two-cycle latency of the processor pipeline. This includes the traversal time in the processor interface, crossbar and reconfiguration logic in the memory mats. Therefore, in our performance evaluations, we focused on the performance impact of the flexible mechanisms embedded in the controllers rather than the Tile’s local memory subsystem.

In order to evaluate this performance impact, we back-annotated the Smart Memories functional simulator with the latency numbers extracted from the actual controller RTL. Then we created “ideal” controllers, where the overhead of the internal controller actions in executing protocol operations is set to zero. In other words, the operations that occur inside controller such as invoking a subroutine in a functional unit, lookup and write of status holding registers, message composition and decomposition in the interfaces, etc. will not incur any latency in the “ideal” controller. However, the latency of operations performed by controller on the other resources such as accesses to the local memory, communication over interconnect, data transfers, etc. are accurately accounted for. This provides an “upper bound” estimate on the performance of a controller. We then compare results gathered from simulating our back-annotated controller model with this upper bound. This experiment is performed for three major memory models mapped to the Smart Memories hardware: a shared memory system using hierarchical MESI coherence protocols, a streaming memory system, and a hardware transactional memory system implementing Transactional Coherence and Consistency (TCC).

5.2.1. COHERENT SHARED MEMORY

We used a few kernels and applications from SPLASH-2 suite [47] parallelized using ANL macros and an MPEG2 encoder application to evaluate the coherent shared memory system. Table 5-2 describes these benchmarks and their corresponding problem sizes.

Table 5-3 shows the details of our coherent shared memory system. For the MPEG2 video encoder application we used a 16KB instruction cache instead. In order to ensure that system is not bandwidth limited and the overhead of reconfigurability is not hidden by the latency incurred due to insufficient memory bandwidth, we assumed two separate memory controllers per each Quad and added L2 cache banks between the memory controller and off-chip memory to further improve main memory bandwidth and latency. In the resulting system, each L2 bank caches only the addresses mapped to the corresponding memory controller and does not need any coherence mechanisms, but is shared by all the processors in the system.

App.	Problem Size	Description
FFT	2^{16} data points	Complex 1-D Fast Fourier Transform
LU	512×512 matrix 16×16 block	Dense matrix LU factorization
Radix	2^{20} keys, radix=1024	Integer radix sort
Cholesky	tk15.O	Blocked sparse matrix factorization
Barnes	16K particles	Barnes-Hut hierarchical N-body method
MP3D	30K particles	Rarefield fluid flow simulation
FMM	16K particles	N-body adaptive fast multi-pole method
Mpg2enc	10 CIF frames (foreman)	MPEG2 video encoder

Table 5-2: Coherent shared memory benchmarks

I-cache	8KB, 2-way associative, 32B line size, 1 port (per processor)
D-cache	16KB, 2-way associative, 32B line size, 1 port (per processor)
Local Memory	None
Protocol controller	28 MSHRs (24 for processor requests, 4 for coherence requests)
L2-cache (unified)	4MB, 4-way, 32B line size, 10 cycle access latency, banked among main memory controllers
Switch latency	5 cycles
Memory controller	2 controllers per Quad, 32 MSHRs each
Main memory	100 cycle access latency

Table 5-3: System parameters for coherent shared memory model

Figure 5-4 and Figure 5-5 depict the speedups we have achieved by both using the idealized controllers (dashed line) and our real controllers (solid line) and compares them to the linear speedup. Average overhead across all benchmarks is slightly greater

than 15%. As the system scales beyond a single Quad (more than 8 processors) the difference between ideal and real controllers becomes more visible. The reason is due to more controllers getting involved in providing coherence across multiple Quads, and hence latency of the controllers actions affect the overall latency of servicing cache misses.

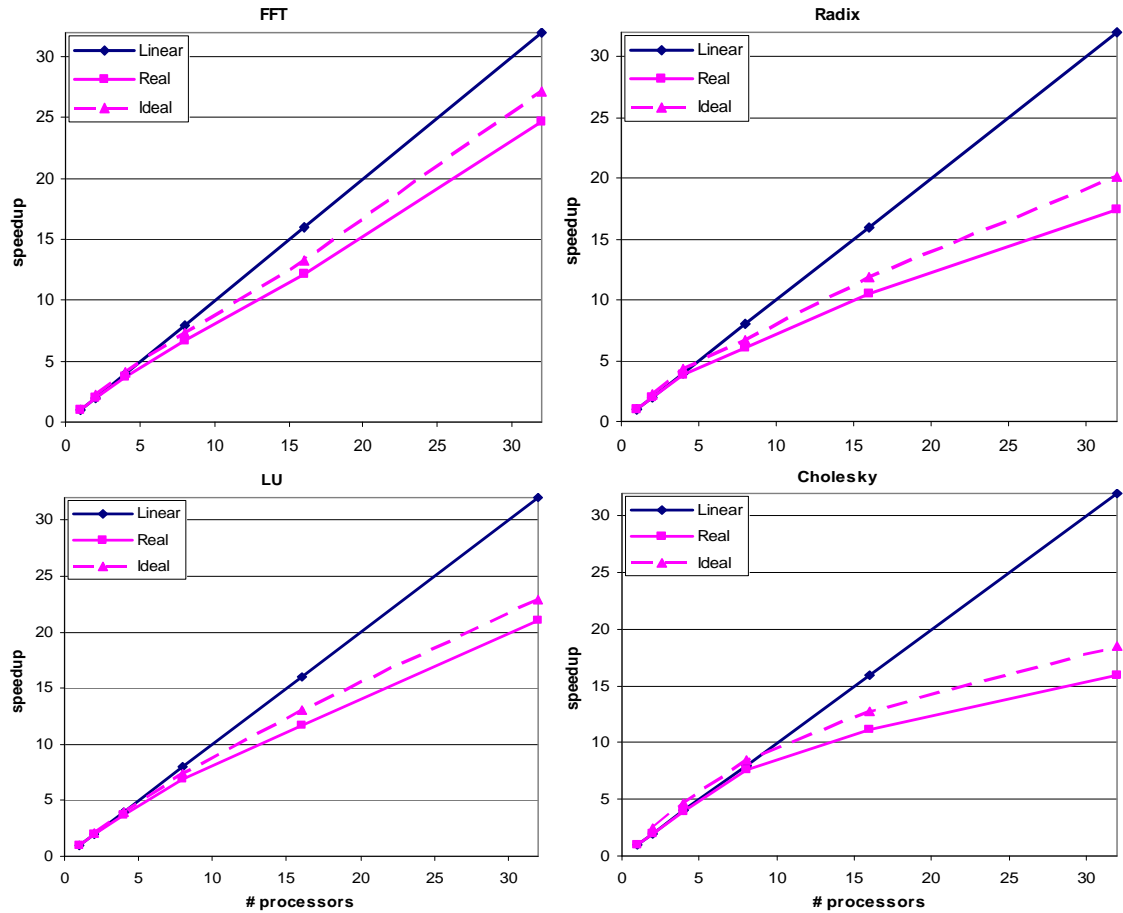


Figure 5-4: Performance impact in coherent shared memory model (kernels)

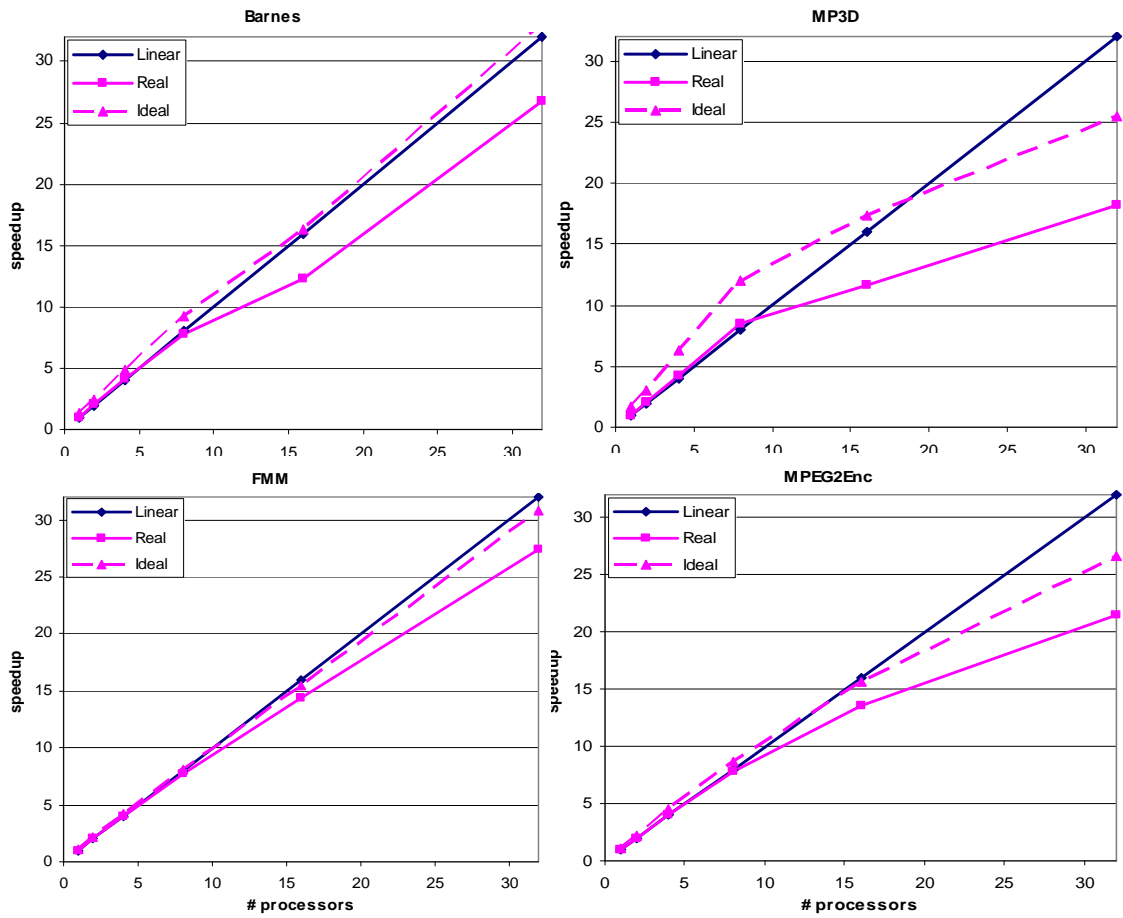


Figure 5-5: Performance impact in coherent shared memory model (applications)

Figure 5-6 shows the breakdown of execution cycles of the parallel section of the three best-case kernels and two worst-case applications in a system with 32 processors, illustrating the effect of the controller latency on the processor stall time. Execution times are normalized to the execution time when using real controllers. The effect of the increased latencies becomes more visible when there are a lot of reads and updates to the shared data which are handled by the controllers without access to the next level of the hierarchy (for example MP3D). In contrast, when most of the cache misses are serviced by fetching the data from main memory or L2 cache, internal latency of the controller is effectively hidden by the long latency of the L2 or main memory access.

Note controller's latency also affects the cycles processors spend on synchronization. This is due to the fact that synchronization accesses are considered modifying operations and cause coherence actions in order to acquire ownership of the cache lines. Ideal controllers effectively complete coherence actions faster and therefore cause processors to spend less time on synchronization stalls.

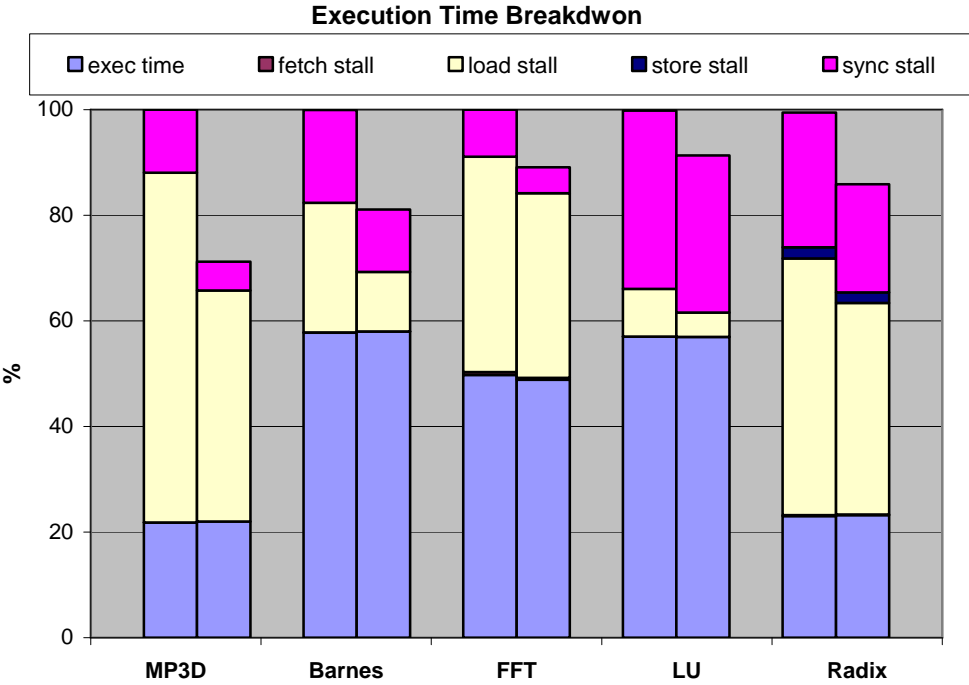


Figure 5-6: Breakdown of execution time (shared memory benchmarks)

5.2.2. STREAMING

Table 5-4 lists the applications we used to evaluate the impact of reconfigurability in streaming model and Table 5-5 shows the details of them memory system. Each processor has an instruction cache and a small private data cache for storing runtime variables and stack. There are 20KB of private local memory per processor. There is an additional 4KB shared local memory for all the processors in the system, used for storing synchronization variables. For stereo depth extraction and MPEG2 video

encoder application we used 24KB of local memory instead and in the configuration used for MPEG2 video encoder, the two Tile processors share an 8K data cache instead of having separate data caches. The protocol controller contains 8 DMA channels with each processor having its own dedicated channel. L2 cache and off-chip memory have the same parameters as before.

App.	Problem Size	Description
179.art	SPEC reference data set	Image recognition
Bitonic	2^{19} 32-bit keys	Bitonic sort
Merge	2^{19} 32-bit keys	Merge sort
Depth	352x288 CIF image pair	Stereo depth extraction
Mpg2enc	10 CIF frames (foreman)	MPEG2 video encode

Table 5-4: Streaming benchmarks

I-cache	8KB, 1-way associative, 32B line size, 1 port (per processor)
D-cache	4KB, 1-way associative, 32B line size, 1 port (per processor)
Local Memory	20KB per processor, 4KB shared between all processors
Protocol controller	28 MSHRs (24 for processor requests, 4 for coherence requests), 8 DMA channels (one per processor)
L2-cache (unified)	4MB, 4-way, 32B line size, 10 cycle access latency, banked among main memory controllers
Memory controller	2 controllers per Quad, 32 MSHRs each
Main memory	100 cycle access latency

Table 5-5: System parameters for streaming memory model

Figure 5-7 shows the scaling of the streaming applications, comparing the performance of the system with upper bound limit. Worst-case overhead imposed in this case (MPEG2 video encode) is less than 14%. Due to the latency tolerance nature of the streaming applications and overlapping of computation with the data transfer, streaming applications are much closer to the upper bound limit. Also, since the application data is managed explicitly by software and hardware does not perform any implicit state manipulation operations (unlike coherent shared memory system), latency of controller actions does not impose a visible overhead for these benchmarks.

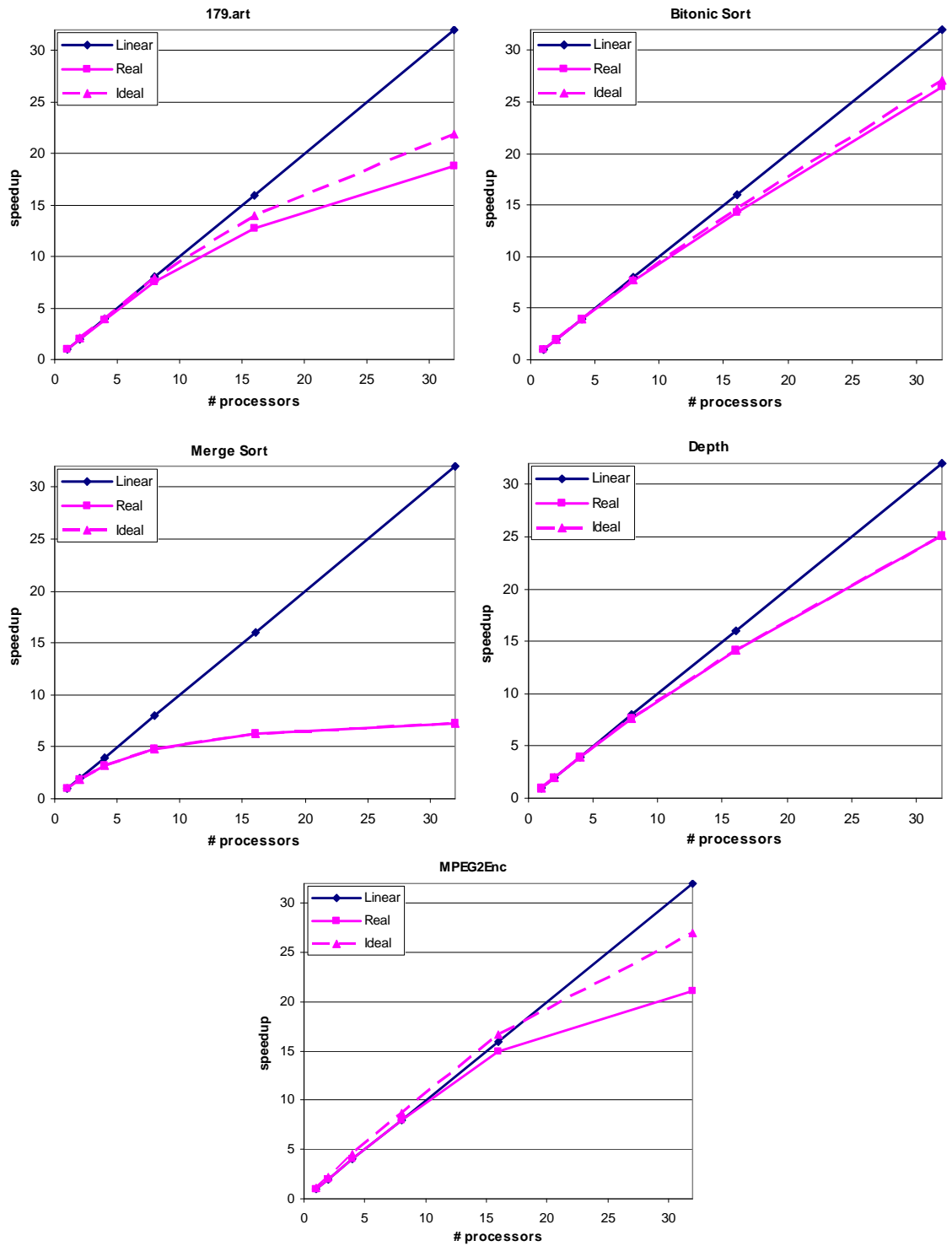


Figure 5-7: Performance impact in streaming model

5.2.3. TRANSACTIONAL COHERENCE AND CONSISTENCY

As mentioned in section 4.8. the transactional memory model that is mapped on the Smart Memories hardware is transactional coherence and consistency (TCC) [27]. Table 5-6 shows the details of the memory system used for evaluating TCC benchmarks. In TCC mode one of the processors in the Tile is used as support processor to handle asynchronous events such as cache or address FIFO overflows and transaction violation. This processor does not run a separate transaction. Therefore, local caches are shared between the two processors. In the normal operational mode, the support processor is stalled, waiting to start execution of necessary handlers if required. In case of a cache or address FIFO overflow, the main processor is stalled and support processors starts execution; hence there is not much collision between the two processors for accessing L1 caches.

The L1 data cache also has a 1K entry Store Address FIFO, which keeps the addresses of the words written during the transaction. The FIFO suppresses duplicate writes such that if a single word is written multiple times by the transaction, the address is only stored once in the FIFO. Similar to the streaming system configuration, there is a 4KB local memory that is used for keeping synchronization variables. This memory is shared by all the processors in the system. The protocol controller has 24 status holding registers for storing outstanding memory requests and 4 DMA channels, one per each Tile used as commit controllers to broadcast transaction's write set to other caches and main memory. The rest of the memory system is similar to other memory models.

Table 5-7 lists the applications used for evaluating transactional memory model. In these applications, we have separated the address space into "TCC coherent" and "TCC buffered" regions. The coherent space is the shared data between the transactions; the part of the transaction's write set that is in the coherent space is broadcasted to other transactions at commit time and is used for violation detection. On the other hand, the TCC buffered space is the transaction's private data and is not shared with other transactions. The part of the transaction's write set that is mapped to

buffered space is not broadcasted at commit time and is kept in the cache. It is committed lazily to the main memory upon evicting the cache lines. However, this part is discarded from the cache, similar to the coherent addresses, when a data dependence violation is detected.

Separation of the transaction's shared and private data achieves better utilization of the store address FIFO associated with data cache (since TCC buffered writes are not placed in this FIFO) and helps in reducing overflows by filling up the FIFO. It also reduces number of committed words by the transaction shortening the commit period, where transactions are serialized against each other.

I-cache	16KB, 2-way associative, 32B line size, 1 port (shared between two processors)
D-cache	32KB, 4-way associative, 32B line size, 1K entry Store Address FIFO, 1 port (shared between two processors)
Local Memory	4KB, shared between all processors
Protocol controller	24 MSHRs (for processor requests), 4 DMA channels (one per Tile)
L2-cache (unified)	4MB, 4-way, 32B line size, 10 cycle access latency, banked among main memory controllers
Memory controller	2 controllers per Quad, 32 MSHRs each
Main memory	100 cycle access latency

Table 5-6: System parameters for hardware transactional memory model

App.	Problem Size	Description
Barnes	8K particles	N-Body application
MP3D	30K particles	Particle simulator

Table 5-7: Transactional memory benchmarks

Figure 5-8 shows the scaling performance of the two benchmarks for transactional memory model. While Barnes has a few writes to shared data and hence a few violations, MP3D performs a lot of writes and transactions encounter a considerable number of violations. Worst-case performance impact of the reconfigurable controllers however is relatively small (slightly less than 20% for MP3D) compared to the ideal controller.

Figure 5-9 shows breakdown of the execution time for worst case of the two benchmarks (16 processor case for Barnes and 8 processor case of MP3D). Note that in this case the execution time is measured only for the processors that actually execute the transaction and not for the support processors. Unlike the shared memory model, controllers do not perform any coherence actions in this case and fetch cache lines directly from main memory, successfully masking the internal controller latencies. However, the load stalls for MP3D is decreased by almost 13% in case of ideal controllers, decreasing the total number of execution cycles. This decrease is due to the large number of data cache misses in MP3D (almost 34% miss rate compared to Barnes with almost 3% miss rate), which puts more pressure on the controller and increases the dependence on the controller latencies.

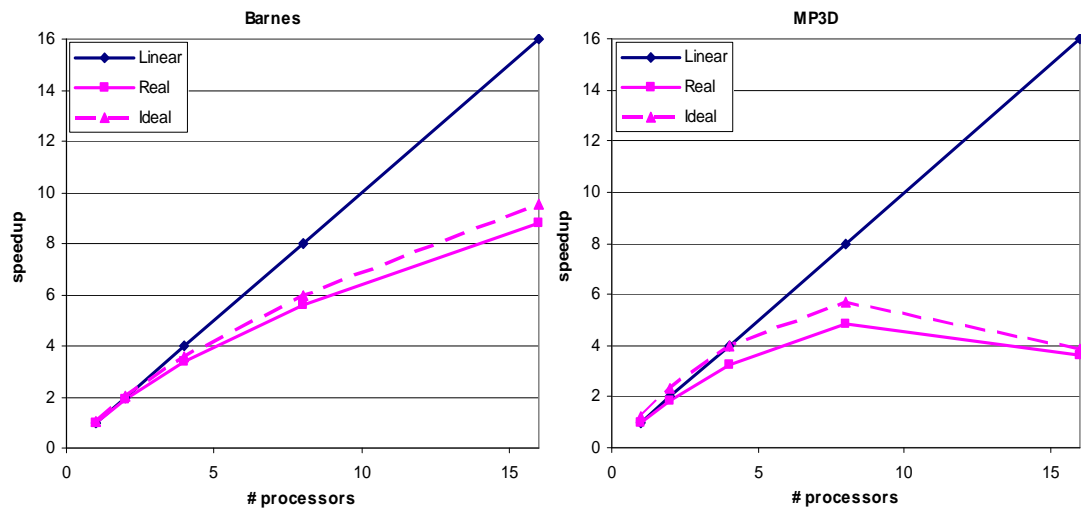


Figure 5-8: Performance impact in transactional memory model

Table 5-8 summarizes the performance impact for all benchmarks in different models. Given that the upper bound corresponds to an idealized controller with zero cycles for its internal actions, the overall performance impact of reconfigurable controllers would be even less compared to any realistic controller for each of the memory models.

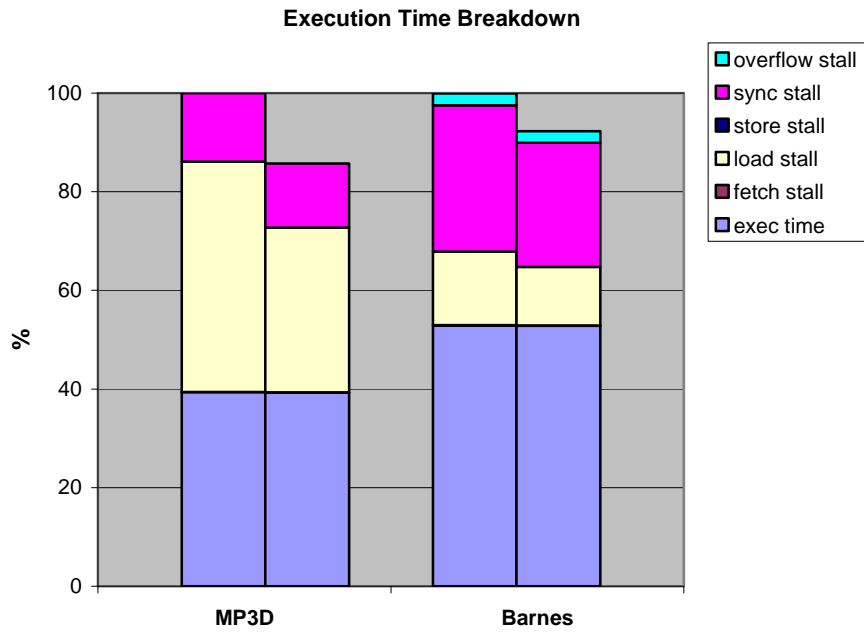


Figure 5-9: Breakdown of execution time (TM benchmarks)

Model	Application	Overhead %	Average %
Coherence	FFT	10.63	15.1
	Radix	13.71	
	LU	8.65	
	Cholesky	18.4	
	Barnes	24.11	
	MP3D	48.38	
	FMM	6.92	
	MPEG2Enc	14.43	
Streaming	179.art	7.49	1.42
	Bitonic sort	1.87	
	Merge sort	0.5	
	Depth	0.06	
	MPEG2Enc	13.97	
Transactions	Barnes	8.82	12.8
	MP3D	19.78	
Overall average			6.72

Table 5-8: Performance overhead of reconfigurable controllers

5.3. PHYSICAL OVERHEAD

In addition to the performance overhead, incorporating reconfigurable mechanisms in a design also affects its physical aspects such as timing, area and power. While a precise evaluation of the physical impact of the reconfigurability is a difficult task (since it requires comparing the reconfigurable system with a specific, non-reconfigurable one), we performed a series of simple experiments to estimate this impact in our system.

Our focus was on the area and power overhead induced by the reconfigurable protocol controller. In these experiments, we tailored the protocol controller to a specific memory protocol by initializing and fixing all internal configuration memories to the operations required by the specific protocol and converted the memories into constant values. Our synthesis tool then removed the memories and propagated the constant values into the logic, eliminating unnecessary logic and creating an “instance” of the controller tailored to that specific memory protocol.

Figure 5-10 shows the area for each of the functional unit in the protocol controller and compares it with specific controller instances created to support coherence (CC), streaming (STR) and transactional memory (TCC) protocols. Internal resources such as number of entries in MSHR/USHR structures, number of line buffers and virtual channel buffers for network messages are kept exactly the same for the baseline and specialized instances. However, DMA channels are not used in the coherence controller and therefore are omitted altogether. Also, in our transactional memory protocol, only one processor in each Tile runs the main transaction, thus only four DMA channels are used in the TCC controller. In the streaming controller each processor has its own dedicated DMA channel, same as the baseline controller, but configuration of the DMA channels are fixed to only provide gather/scatter operations.

Aside from number of DMA channels, since the internal resources are kept the same for all controllers, most of the area reduction (both combinational and non-combinational) comes from simplifying and removing the flexibility in the major

functional units, namely D-Unit, S-Unit and T-Unit. The major reason for such substantial decrease in the area is the fact that in our implementation, all configuration memories were constructed using flip-flops. This simple, but inefficient way of building memories not only uses more transistors to store configuration information, but also consumes a lot of routing resources to connect the flops to output multiplexers, as well as connecting them to the system clock. In the case of specialized controllers, flops and their routing resources are removed during synthesis, reducing both combinational and non-combinational area. Note that since the MSHR and USHR structures are accounted as part of the T-Unit, the area reduction in T-Unit is not as much as the other two units.

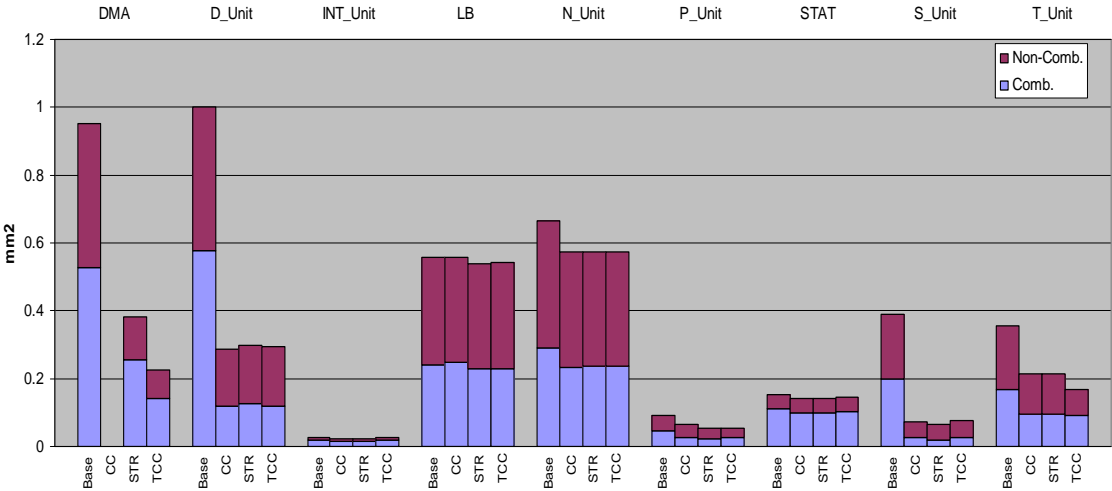


Figure 5-10: Area comparison for protocol controller functional units

Figure 5-11 compares the total combinational and non-combinational area of the baseline controller with each specialized instance. As illustrated, almost half of the area savings is achieved by removing the configuration memories (non-combinational logic). The combinational area savings come from two separate sources: first, since some of the configuration memories in the functional units are organized as TCAMs, eliminating the configuration storage also saves the area consumed by TCAM

comparators. The rest of the savings in combinational area is achieved by propagating the constant values and optimizing combinational logic of functional unit itself.

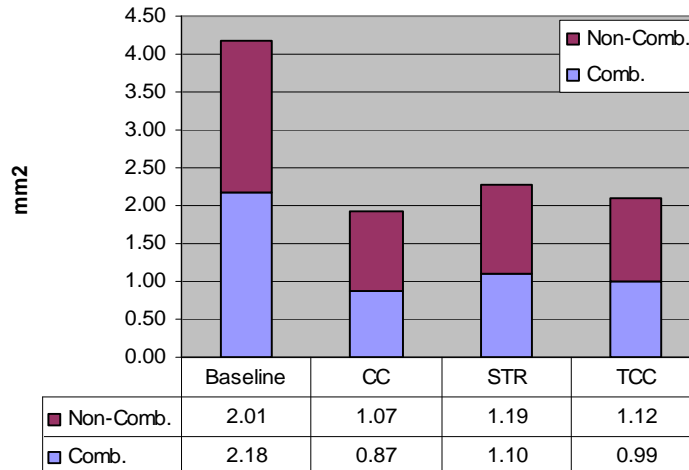


Figure 5-11: Comparison of total area between controllers

Table 5-9 lists the estimated dynamic and worst-case leakage power consumption for the baseline as well as specialized controllers, reported by our synthesis tool. While not accurate, these estimates provide an insight about the power overhead of embedded reconfigurable mechanism. Most of the increase in the power of the baseline controller is due to leakage in the configuration memory structures, but it also has a higher dynamic power. Most of the excess dynamic power is consumed by the read ports of the configuration memories and TCAMs, which are accessed every clock cycle, and comparators in the TCAM structures.

Power (mW)	Baseline	CC	STR	TCC
Dynamic	170	101	115	111
Leakage	450	189	230	214
Total	620	290	345	325

Table 5-9: Power comparison for baseline and specialized controllers

5.4. SUMMARY

While Smart Memories allows mapping of different memory protocols, the flexible mechanisms added to provide reconfigurability impact both the performance and physical characteristics of the system. Our studies show that while the performance overhead induced by these mechanisms is modest, less than 20% in most of the cases, the increase in the system's area and power is not negligible. However, most of this increase is resulted from our poor implementation of system's configuration storage, using flip-flops for implementing memories and TCAMs. These inefficiencies can be removed and a better implementation of the system can be achieved by using memory macros and custom structures, as shown by Mai et al for the case of a reconfigurable memory mat [71].

6. CONCLUSIONS

The emergence of multicore architectures places an increased emphasis on the design of the memory system since it implements the communication and data sharing between processor cores. The paradigm shift from traditional sequential programming to explicitly parallel programs introduces a major productivity challenge in developing parallel software. Researchers have tried to address this problem by proposing innovative models such as programming with streams and transactional memory. Deployment of any multicore processor hence involves the adaptation of such a parallel programming model, which usually places a set of strict requirements on the functionality of the memory system. Therefore, the existing multicore architectures usually are optimized for, if not restricted to, realizing and implementing a single parallel programming model.

In this dissertation, we observe that the basic hardware operations and resources employed for implementing different memory models in today's multicore architectures are the same, with the only differentiating factor being in the combination and sequencing of these primitive operations. This observation is supported by studies comparing the performance of the memory systems in modern multiprocessors [28][64]. These studies demonstrate that different memory systems can achieve similar levels of performance, given intelligent management of the resources, since they intrinsically rely on the same operations at the implementation level. For example while a stream memory model relies on the programmer's knowledge for orchestrating communications and data movements, coherent shared memory systems try to imitate the same level of intelligence by employing sophisticated coherence controllers and prefetch engines that automate data communication and transfer and eliminate the programmers effort for handling such transfers explicitly.

Based on the above observations, we propose a universal memory system architecture that extends the notion of “programmability” from the processor core to the memory system hardware. The programmability not only enables supporting multiple programming models on the same hardware substrate, but also allows a user to tailor the underlying memory system to the application of interest and potentially achieve better levels of performance and efficiency. We identify the necessary hardware resources, namely storage elements, communication channels and their associated controllers as operating agents in the universal memory system. We propose a set of basic operations and state registers for the controllers and describe how the request and reply messages in the memory system can be processed in each controller by combining these basic operations.

With this framework in place, we present Smart Memories, a reconfigurable memory system architecture as a first implementation. Our performance study shows that the overhead of the added flexibility in the system is small, less than 20% slowdown in clock cycles compared to an idealized controller for almost all cases across three different memory models. However, our simple but inefficient way of implementing storage structures for system’s configuration induces significant area and power overheads.

While creating a better implementation of this reconfigurable multicore architecture is a very interesting engineering task, a more attractive challenge is to understand how much and what kind of reconfigurability is useful for patching a system after construction. In practical systems, a specialized architecture always performs better and has less physical cost than a reconfigurable one. Hence the major advantage of reconfigurable architectures is in being able to alter the functionality of the system after implementation, in order to fix design errors or integrate new functions. The key question therefore is whether one can achieve the same advantage by integrating small amounts of flexibility into a specialized system, which allows patching potential design errors or modifying and upgrading the system’s functionality.

Another challenge is while Smart Memories architecture provides a large degree of freedom in configuring and using hardware resources, one can always find or develop protocols that cannot be mapped on this system. This problem can be due to resource constraints, such as insufficient hardware resources, or requiring special functional units that are not provided in the system. Alternatively it can be due to absence of support for the information fields that a protocol requires to communicate between different components and necessary operations to act upon them. For example, while Smart Memories supports all necessary mechanisms for implementing a Token Coherence protocol [65], such as token counting, un-ordered communication for exchanging transient requests, serialization operations for persistent requests, etc., it lacks the watchdog mechanisms that are used to trigger persistent requests and avoid starvation. Another example is STAMPede's TLS protocol [14], where coherence messages carry epoch numbers, a processor's speculation degree, in order to decide whether to acquire a cache line and invalidate the owner or not.

An attractive approach for alleviating this problem, as well as addressing the design efficiency issues, is pushing the abstract framework discussed in this dissertation into the memory system design phase. The designer can then construct a memory system by means of allocating necessary resources and implementing desired protocols by composing the basic memory operations at the design stage. The design framework provides the user with necessary resources, mechanisms and operations to choose from, which can be considered as programming a *virtual* memory system. When realizing the actual implementation, the design tools can identify and analyze utilized resources in the virtual system and optimize away the unused flexibility, which leads to a much more efficient system implementation. Such design framework can also be augmented with additional resources and state information, as well as hardware mechanisms and operations, which allows implementing specialized memory system protocols. This eliminates designer's concern about the physical and performance cost of the reconfigurable mechanisms to control resources, as well as unused system resources and operations that might consume area and power.

Smart Memories provides a basis for constructing such an extensible memory system design framework. Separation of the data path and control in processor interface logic and memory system controllers, microcode based implementation of the control logic, and use of standard interfaces for inter-module communication, makes it easier for the design to be automatically extended. New functional elements can be added to data paths while their controlling bits can be added to configuration memories. Configuration memories can also be extended both to accommodate new control bits as well as more entries for new operations. Existing interfaces can be augmented with new information fields that are added to interfaces and routed between different components and modules. Such system extensions have been successfully realized for commercial reconfigurable processors such as Tensilica [66][67][68]. Providing the same extensibility for designing memory systems is naturally the next logical step in providing a higher level of abstraction for computer system design.

An important element in designing such a framework is providing suitable interfaces for memory system designer to express a memory system specification. Implementing memory protocols requires decomposing protocol actions into logically distinct operations carried out by separate system components, as well as communication messages exchanged between them. Due to this distributed and decentralized nature, protocol design is a challenging and error-prone task. Developing a simple language for describing memory protocols in form of a “memory system program” expressed in form of the ISA instructions, raises the level of abstraction in protocol design and helps in detecting logical protocol errors. A compilation framework can then analyze and generate necessary control signals for the virtual memory system, which is then used by the design tools to eliminate unnecessary hardware components and construct the desired memory system. The compiler can also apply optimization techniques to memory programs, such as fusing commonly encountered sequences of operations into a single operation, as well as verify the generated control information to reveal inconsistencies or conflicts in using available hardware resources.

Such language and compilation framework would be even beneficial for the current Smart Memories system and SMASH test chip. The current method of configuring the system requires manual development of the bit patterns uploaded into different memory system components. The compiler which could read in the protocol description and generate necessary bit patterns to for uploading into different components, would greatly simplifies the task of system configuration and eliminate many errors.

APPENDIX A: SMASH INTERCONNECTION NETWORK

In the general Smart Memories system Quads and memory controllers are connected via a mesh like network. This requires each Quad to have a network router in order to route packets received from its neighbors to their appropriate destinations. In our implementation of Smart Memories and the SMASH test chip, we simplified all the network architecture by settling on a star topology, as shown in Figure A-1. In this topology, all the connections are made by a central switch and Quads do not have the routing capabilities. They simply send and receive packets to/from the switch and only have to properly identify the destination for each message. This topology supports maximum of four Quads and four memory controllers. In addition, one can create a simplified version of the system by directly connecting a single Quad to a single memory controller without any additional interfacing. This allows creating a minimum system with reduced communication overhead. However, the network switch is required if the user needs more than one Quad or one memory controller to be enabled. In this appendix we describe the properties of our implemented interconnect mechanism and explain the internal architecture of the network switch and its capabilities.

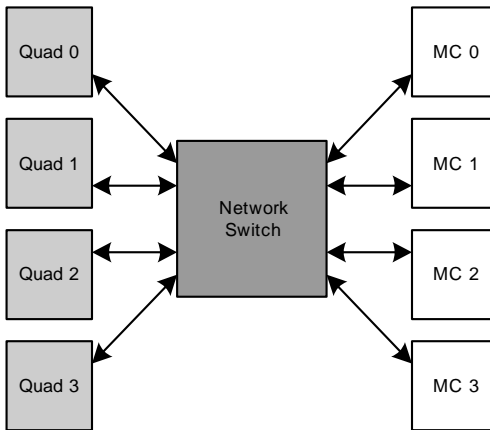


Figure A-1: Star interconnection topology in SMASH

A.1. INTER-QUAD NETWORK

Inter-Quad network in the SMASH system is organized as a star. All communications between Quads or between a Quad and a main memory controller are routed through a central switch referred to as the Network Switch. Communication between the switch and each Quad or memory controller is full-duplex using dedicated transmit and receive channels. Each physical communication channel is virtualized into separate virtual channels, with each virtual channel having its own dedicated buffering space at the receiving end.

Packets are divided into units of transmission called *flits*. Each flit contains 72 bits of information and is transferred from source to destination in a single clock cycle. The system uses a credit based flow control mechanism; whenever a flit is consumed at the destination by routing it (in the switch) or passing to execution core (in Quads or memory controller), a credit is sent for the source entity. Credit counting mechanism at sources ensures that they do not attempt sending a packet unless there is enough buffering space (credit) at the destination to buffer the whole packet.

The clock rate of the communication on the network, or I/O Clock, can be adjusted to be equal to, 1/2, 1/4 or 1/8 of the system clock. Each Quad and memory controller

receives a two-bit static control signal which dictates the ratio between system and I/O clocks.

A.2. NETWORK SWITCH ARCHITECTURE

The network switch is an 8×8 input-queued switch connecting four Quads to four memory controllers. Quads are connected to ports 0-3 of the switch and memory controllers are connected to ports 4-7 (Figure A-2). The switch fabric is an 8×8 crossbar controlled by a scheduler. The switch operates at system's I/O Clock speed.

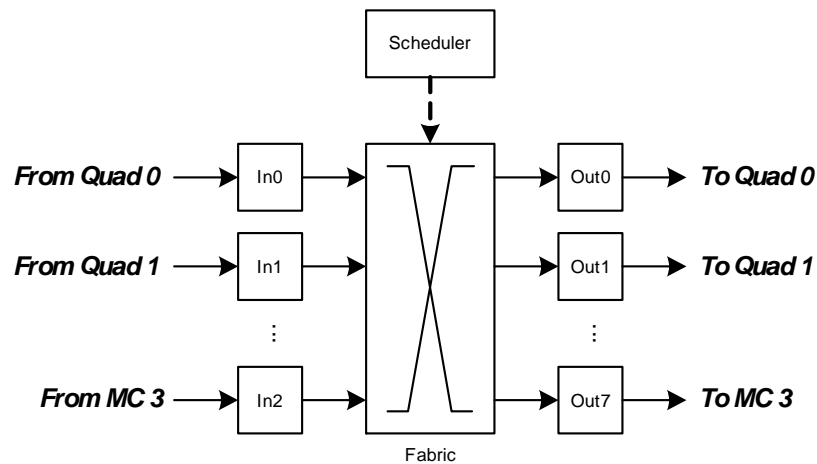


Figure A-2: Organization and connections of the network switch

Each input port in the switch has eight separate virtual channel buffers to store packets on each virtual channel separate from others (Figure A-3). Whenever a virtual channel buffer becomes full, credit based flow control mechanism causes back pressure on the source, preventing it from sending more packets. At each clock cycle, each virtual channel in an input port sends requests to scheduler asking for specific outputs. The scheduler sends back a grant signal and an output channel number in response. The input port then extracts the head flit from the buffer and sends it to the designated output port.

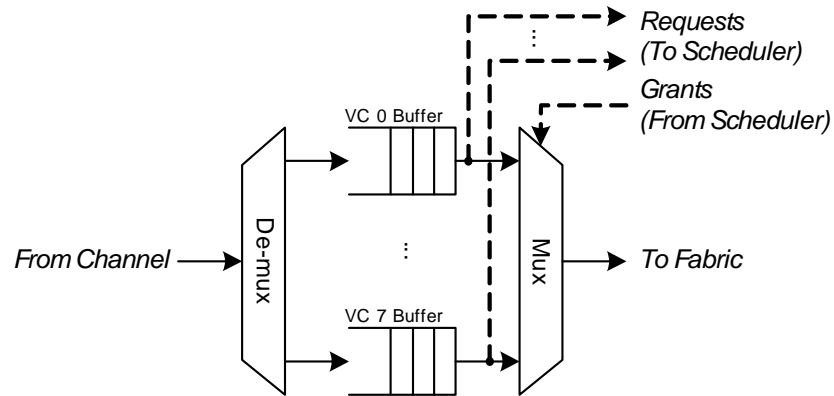


Figure A-3: Input port of the network switch

Each output port (Figure A-4) has a buffer for a single flit per virtual channel, as well as the credit counters for downstream destination. The output port receives flits from the fabric whenever the scheduler indicates there is an incoming flit for this port. The scheduler also specifies the virtual channel on which the flit is traveling. The output port puts the flit in the buffer and sends it to the destination whenever there is credit on the specified virtual channel. If there is not enough credit for sending the flit the output port signals the scheduler that its buffer is full and it cannot accept any more flits. This stops the scheduler from granting requests to this output on the specific virtual channel, stalling the input ports' virtual channel buffer.

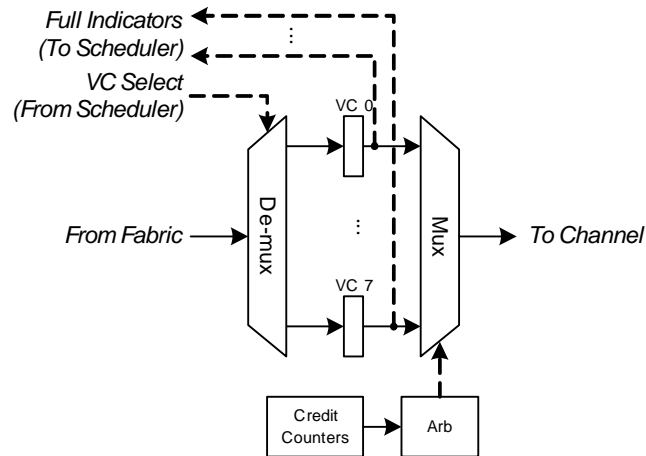


Figure A-4: Output port of network switch

The scheduler determines the connections between each pair of input/output ports at any given cycle. It receives eight request vectors from each input port, one per virtual channel. The request vector indicates which output ports the input is making a request for. It also receives the full indicators for each virtual channel from all output ports. The scheduler then generates a match matrix, which indicates which input/output pairs should be connected at that clock cycle. It also specifies the virtual channel number of each connection.

In order to perform the scheduling decisions, the scheduler logic internally runs eight concurrent iSLIP schedulers [72], and combines their output match results. Each iSLIP scheduler receives requests related to a single virtual channel and produces a match matrix according to that virtual channel. Match matrices from all schedulers are then combined according to the priorities specified for virtual channels, as shows in Figure A-5.

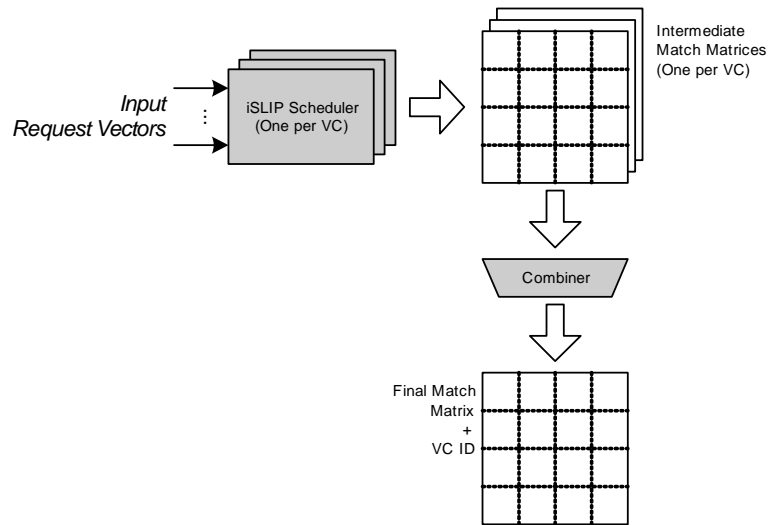


Figure A-5: Network switch scheduling logic

A.3. ENFORCING PRIORITIES

The network can prioritize traffic sent on one virtual channel over the others. Such prioritization is essential when the same physical network is used to carry different types of traffic, since reply messages should always have priority over request messages to avoid deadlock in the system [69]. The system provides a very flexible way of defining priorities: for each virtual channel, an 8-bit mask specifies the other channels that have priority over it. In other words, a one bit in position i of the mask for channel j indicates that traffic on virtual channel j can be blocked by traffic on virtual channel i .

When no priority relation is established between two virtual channels, system utilizes a fair, round robin arbitration when serving requests from these two channels. Priorities are enforced in all arbitration points in the system: when a message sending request is passed to transmitter, at the network switch and in the receivers, when a received message is to be passed to the execution core. Each entity (Quad, memory controller and network switch) has its own set of priority mask registers. These

registers should be configured with the same values in order to guarantee correct prioritization over the communication channels.

A.4. BROADCAST / MULTI-CAST CAPABILITIES

The network switch in the SMASH system provides basic broadcast/multi-cast capabilities. These capabilities are very useful when implementing memory models that need to send inquiries or updates to all entities in the system. For example, when updating state of the cache line in an invalidation based coherence protocols or updating a memory word in update based coherence protocols, messages have to be sent to all the Quads that (might) contain the specific word.

The switch supports a limited form of multicasting a message to multiple destinations. Note that the switch is aware of the fact that Quads are always connected to ports 0-3 and memory controllers to ports 4-7. This information is leveraged by the switch in order to generate messages for desired destination. Each packet has a three bit multicast field in the header which specifies which destinations the message should be sent to, if packets should be sent to more than one destination. These bits are:

- Bit [0] – Quad broadcast: Indicates that message should be broadcasted to all the Quads in the system (ports 0-3).
- Bit [1] – memory controller broadcast: Indicates that message should be broadcasted to all memory controllers in the system (ports 4-7).
- Bit [2] – Except destination: When this bit is set, switch does the broadcasting to Quads or memory controllers, but does not send the message to the entity specified in the destination field of the message. This is particularly useful when a message should be sent to all Quads (or memory controllers) except one; For example, when broadcasting a coherence request in serving a cache miss, memory controller wants to enquire state of the cache line in all Quads except the one that originated

the cache miss. This bit allows the custom multicasting that is commonly used in memory protocols.

APPENDIX B: IMPLEMENTING A SIMPLE PROTOCOL

Chapter 4 described the Smart Memories architecture as an example implementation of the universal memory system. This appendix explains how the embedded reconfiguration features are used for implementing a memory protocol by presenting a simple example. We consider a system with only a single Quad and a single memory controller and explain the necessary steps for configuring the system to implement caches and a MESI coherence protocol between Quad processors.

Configuration process is divided in to three major steps: the first step is to allocate necessary memory resources, including defining and associating necessary state information with cache lines as well as allocating necessary storage structures for data and state information. Second step involves defining memory operations that can be performed on memory locations by processors as well as protocol controller. The definitions include update of the state information (if necessary) as well as success and failure conditions for each memory access. Last step is defining communication messages between different controllers and programming controllers at each level to handle defined messages such that requests are served and properties of the MESI coherence protocol are enforced appropriately. The following sections elaborate on these three steps.

B.1. ALLOCATING RESOURCES

The first step of the configuration process is to allocate necessary system resources. These resources are mainly storage structures used for storing data and state information. In addition, one should also specify address translation and mapping mechanisms in the processor interface logic and controllers.

B.1.1. STATE AND DATA STORAGES

In our example of shared memory system with MESI coherence protocol [69], we assume processors have instruction and data caches with parameters as Table B-1. Memory mats in the Tile are used for storing both data and state information of the cache, including tags and cache line state. Before attempting to allocate storage for state information, we have to specify what state information is required and how it should be associated with the user data.

Cache	Size	Ways	Line Size	Data Mats (per way)	Tag Mats (per way)	Total Mats
Data	32KB	2	32B	4	1	10
Instruction	8KB	1	32B	2	1	3

Table B-1: Cache parameters for example configuration

Mapping state information

Since we need to have identifying tags for each cache line, we store these tags in a separate memory mat per cache way. Tags are stored in the data array of the memory mat¹⁰, which supports a comparison operation. Each processor access to the cache sends a compare operation to the data array of the tag mat comparing the address tags with the stored tags. It treats the result of comparison (*Total Match* output of the mat) as hit/miss signal. Since each cache in our configuration has two ways, two tag mats are required (one per each cache way) and multicast mechanism of the Tile crossbar sends the tag comparison request to both of these memory mats.

In our simple MESI coherence protocol, each cache line has four main states: *Modified*, *Exclusive*, *Shared* and *Invalid*. Figure B-1 shows how these states are mapped into the control (meta-data) bits in memory mats. In addition to these four states, we need an intermediate state, *Reserved*, which indicates that location is reserved for the incoming cache line. The state bits are stored in the control array of the tag mat and are accessed along with the tags in the data array. Tag mats do both

¹⁰ Data array in the mat is 32-bits wide and hence it has enough bits for storing the tag bits extracted from a 32-bit address. Unused bits are filled with zero.

tag and state comparison on each processor access and if either of the tag or line state does not match the desired value, *Total Match* output of the mat will be inactive, indicating a miss in the specific cache way.

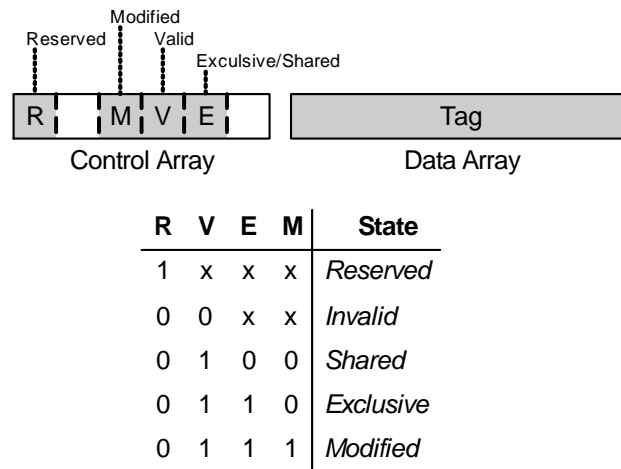


Figure B-1: Mapping and encoding of state information

Allocating memory mats

After determining how to map the state information to memory mats, we need to allocate mats for storing cache line data and state. This is simply done by programming the cache configuration registers inside the processor interface logic, as described in section 4.5.1. Figure B-2 and Figure B-3 show the configuration of the instruction and data caches in the processor interface logic. Note that these caches are shared between the two processors therefore values loaded into the configuration registers are the same for both processors.

In addition, the following configuration registers are also programmed:

- The *IMCN* output of tag mats in both caches is set to be the *Total Match* signal. This transmits hit/miss indication from tag mat in each cache way to corresponding data mats.

- The *guard* signal for guarded operations in data mats is set to be *IMCN* input. This way, modifying operations in the data mats (i.e. writes) are guarded by hit/miss indicator in corresponding tags and hence are discarded if the corresponding tag mat reports a miss.

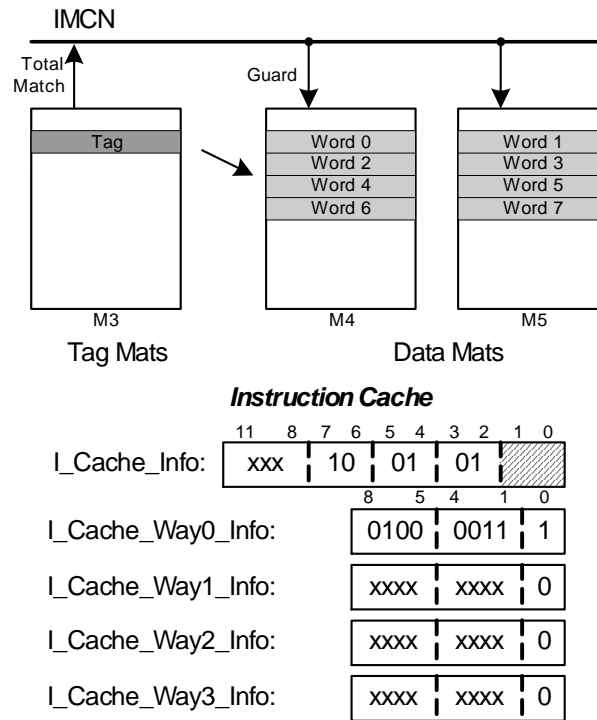


Figure B-2: Example instruction cache settings

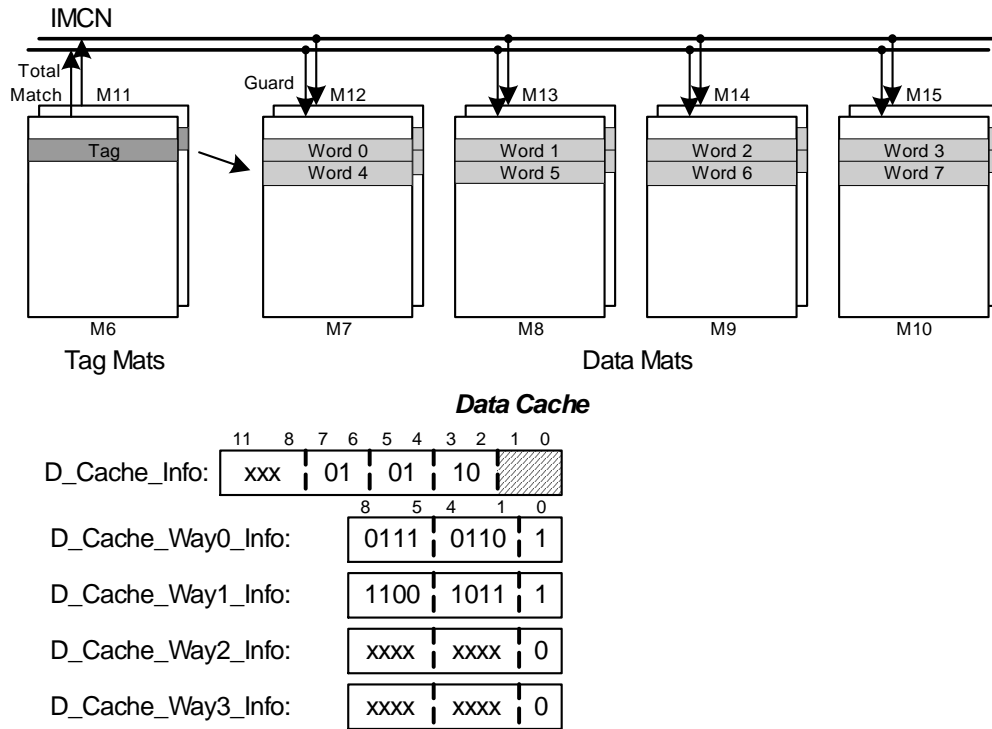


Figure B-3: Example data cache settings

B.1.2. ADDRESS TRANSLATION AND MAPPING

As discussed earlier, obtaining physical address of memory location(s) to access for processor’s memory access instructions involves two steps: translation from virtual to physical address space and slicing generated physical address to obtain tags and indices for memory mats. Second step of the mapping is done by setting up cache configuration registers discussed above. These registers provide necessary control signals for the address slicing logic inside the processor interface, which generates tags and indices for accessing memory mats. First step of the mapping, the translation, is performed by segment table.

Figure B-4 shows an example configuration of the segment table. All instruction segments (4-7) are mapped to off-chip memory segments 11-14 and are accessed via instruction cache. Data segments 8-13 in virtual address space are also mapped to off-chip memory (segments 4-9) and are set to be accessed via data cache. Segment 15

contains the I/O region; it has an identity mapping and is accessed via un-cached memory access instructions (*Cached* bit is set to zero). Segment 14 is mapped to memory mats 0-2 inside the Tile (Segment 3 in physical address space). All instruction segments have read only permissions while all data segments have read/write permission.

R	W	OT	C	Re-map	Base	Size	
1	1	0	0	15	x	x	Seg 15
1	1	1	0	3	0	3	Seg 14
1	1	x	1	9	x	x	Seg 13
1	1	x	1	8	x	x	Seg 12
1	1	x	1	7	x	x	Seg 11
1	1	x	1	6	x	x	Seg 10
1	1	x	1	5	x	x	Seg 9
1	1	x	1	4	x	x	Seg 8
1	0	x	1	14	x	x	Seg 7
1	0	x	1	13	x	x	Seg 6
1	0	x	1	12	x	x	Seg 5
1	0	x	1	11	x	x	Seg 4

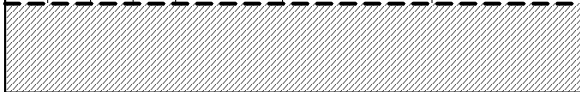


Figure B-4: Example setting for segment table (address translation)

B.2. DEFINING MEMORY ACCESSES

After associating the state information with data and allocating necessary memory mats, next step in configuring the system is defining accesses to local and main memories. Local memory mats are accessed by processor interface logic and protocol controller while off-chip memory is accessed only by main memory controller. There is division of the tasks between these three entities:

- Processor interface logic carries out processor accesses to memory mats, such as load/store instructions or any other memory instruction that might be issued by processor (e.g. prefetch instructions).

- Protocol controller performs cache refills, write-backs and handles coherence operations such as snooping caches and updating state information. It also communicates with main memory controller to write-back or fetch cache lines.
- Main memory controller receives write-backs from Quad and writes data to main memory or reads data from main memory and sends it back to Quad's protocol controller.

The following subsections describe how necessary accesses are defined for memory mats and off-chip memory.

B.2.1. ACCESSES TO LOCAL MEMORY MATS

Processor interface accesses local memories when processor issues a memory access instruction to its cache or local memory. Protocol controller accesses them when it receives a request from a Tile processor that involves reading/writing data or adjusting state of cache lines. In our simple example, we assume that processor can only issue Load, Store, Prefetch for Read and Prefetch for Write instructions to memory. Loads and Stores might access the cache, go directly to off-chip memory (segment 15) or access local memory directly (segment 14 which is mapped to memory mats 0-2). We also know that protocol controller has to implement MESI coherence protocol and therefore it has to snoop and adjust the state of cache line in Tiles when it receives cache miss requests.

Table B-2 shows the configuration of processor interface for assumed processor accesses to caches. The configuration table defines operations for both tag mats and data mats in each cache (instruction fetch is treated the same as Load). Load and Store instructions access tag and data mats at the same time. Prefetch instructions check the status of cache line by only accessing tag mats. For each access, operations on data array and control array are specified. When RMW logic in the mat is used to update line state, the figure also shows how the state bits are updated.

TIE Opcode	Tag Mats					Data Mats				
	Data Op	Cntr Op	PLA Op	Cntr Bits	Mask	Data Op	Cntr Op	PLA Op	Cntr Bits	Mask
Load	Cmp	Cmp	NOP	0xx1xx	1100100	Read	NOP	NOP	6'bx	7'bx
Store	Cmp	CMW	M←1 if TM	0xx11x	1100110	Guard Write	NOP	NOP	6'bx	7'bx
Prefetch Read	Cmp	Cmp	NOP	0xx1xx	1100100	NOP	NOP	NOP	6'bx	7'bx
Prefetch Write	Cmp	Cmp	NOP	0xx11x	1100110	NOP	NOP	NOP	6'bx	7'bx

Table B-2 : Processor interface operations on memory mats (cached)

Load instruction compares the cache tags and ensures that cache line is not in *Invalid* or *Reserved* state by comparing *V* bit with 1'b1 and *R* bit with 1'b0. The other control bits are ignored since the *Mask* input disables comparison operation for them. In data mats, Load instruction reads the data from data array and does not do any operation on control array.

Store instruction performs the same tag compare operations on the data array and control array of the tag mat but it also checks the *E* bit to ensure that it has write permission (*Exclusive* or *Modified* states). Instead of normal compare, it uses a Compare-Modify-Write operation on the control array to set the *M* bit if *Total Match (TM)* is activated which converts the line to *Modified* state in case of hit. On the data mats, Store uses a Guarded Write operation to write data word only if the *guard* signal is active. *Guard* is set to be the IMCN input which propagates *Total Match* signal from tag mat to data mats. Similar to Load instruction, no operation is defined for control array in data mat.

Prefetch operations only access the tag mat in order to compare the tags and line state. Prefetch for Read checks only the *V* and *R* bits to ensure that line is in a valid state and is not reserved. Prefetch for Write also checks the *E* bit to see whether cache has ownership of the line or not.

In our example setting of segment table, there are two segments that are marked as uncached. The first segment is segment 15 which is mapped to off-chip memory and

second segment is segment 14 mapped to memory mats 0-2 in the Tile. For accesses that go to segment 15, LSU does not access the local memory mats and instead sends a request message to protocol controller to read/write the memory address of interest. For accesses to segment 14 (un-cached, on-Tile), it accesses the target mat the same was as data mats in the caches. Table B-3 lists un-cached operation of LSU on the memory mats.

Segment	TIE Opcode	Data Op	Cntrl Op	PLA Op	Cntrl Bits	Mask	Comment
14	Load	Read	NOP	NOP	6'bx	7'bx	--
14	Store	Unguard Write	NOP	NOP	6'bx	7'bx	--
15	Load	NOP	NOP	NOP	6'bx	7'bx	Message to controller
15	Store	NOP	NOP	NOP	6'bx	7'bx	Message to controller

Table B-3: Processor interface operations on memory mats (un-cached)

Protocol controller is responsible for servicing cache misses by writing back evicted cache lines and refilling new lines into the cache. In addition, it has to enforce the coherence properties and adjust the line states in all of the Quad caches according to MESI protocol. Hence, we can define the following accesses for protocol controller:

- *Eviction*: Put a cache line in the reserved state by turning on the *R* bit
- *Write-back*: Read cache line tags and data and send it to main memory controller
- *Line read*: Read data portion of the cache line from cache, used when doing a cache-to-cache transfer between to Tiles
- *Refill*: Write cache line tags and data when requested line received from memory controller or found in another Tile's cache
- *Search (Snoop)*: Read cache line state and atomically updates it (using Read-Modify-Write operations) to comply with the MESI protocol

Table B-4 and Table B-5 present the details of the protocol controller accesses to tag and data mats. There is a major difference between accesses from processor interface logic and protocol: While processor interface accesses tag and data mats concurrently when carrying out a memory instruction, protocol controller accesses tag and data separately when processing a request. The reason is that controller has separate functional units for accessing data and line state information, and each has its own dedicated port to Tile memory mats. Therefore, tag and data accesses for any give request inside controller are carried out at different times, since request is passed from one functional unit to the other. Note that controller still might issue concurrent accesses to tag and data mats at the same cycle but these accesses will correspond to different requests.

Operation	Tag Mats				
	Data Op	Control Op	PLA Op	Control Bits	Mask
Eviction	Read	Unguarded Write	NOP	100000	7'bx
Tag Read	Read	NOP	NOP	6'bx	7'bx
Tag Write	Unguarded Write	Unguarded Write	NOP	M: 001110 E: 000110 S: 000100	7'bx
Snoop-Read	Comp	Comp	E, M←0 if TM	0xx11x	7'bx
Snoop-ReadEx	Comp	Comp	V←0 if TM	0xx11x	7'bx

Table B-4: Protocol controller operations on tag mats

Operation	Data Mats				
	Data Op	Control Op	PLA Op	Control Bits	Mask
Read	Read	NOP	NOP	6'bx	7'bx
Write	Unguarded Write	NOP	NOP	6'bx	7'bx

Table B-5: Protocol controller operations on data mats

For cache line evictions, controller writes the state bits in the tag mats and sets the *R* bit to one. This indicates that line is in *Reserved* state and there is a refill pending. For write-backs, controller reads the tags as well as data using Read and Tag Read operations. It then sends the extracted cache line to memory controller. When there is

a possibility to service a cache miss by doing a cache-to-cache transfer, controller reads the cache line from another Tile's cache and refills it in the destination cache. This is similar to the write-back operation, but no tag read is required. When doing Tag Writes, controller writes both tag and state into the tag mat using unguarded writes. Exact value of the control bits depends on the state in which controller refills the cache line. As part of the refill, controller also writes the data portion of the cache line into data mats using unguarded write operations on the data array. For snoops, controller uses the Read-Modify-Write logic in the tag mats to update the state bits. Two types of snoops are possible: "Read Exclusive" invalidates the cache line by setting the *V* bit to zero, while "Read" only degrades the cache line by setting the *E* and *M* bits to zero.

FAILURE CONDITIONS AND REQUEST MESSAGES

Part of defining the memory accesses is specifying when a memory access is successful. As discussed before, when a memory access fails a request message is sent by processor interface to protocol controller, asking for assistance. When defining accesses to local memory mats by processor interface logic and protocol controller, user has to define corresponding success/failure conditions for each access. In addition, we have to specify whether a request message has to be sent and what is the type of the request for each failure condition. For processor interface accesses specifically we also have to indicate whether issuing processor has to be stalled or not.

Table B-6 shows processor interface settings that define success or failure conditions and message types that are sent to protocol controller, in case that specific failure condition is encountered. Note that these conditions are defined for accesses to caches only. Un-cached accesses to memory mats (segment 14) are always successful. Un-cached accesses to off-chip memory (segment 15) are always unsuccessful and result in sending a message to protocol controller. The table only shows returned information from two ways of the cache, since in our configuration a cache has at most two ways.

Prefetch for Read and Prefetch for Write operations have the same conditions as Load and Store and therefore are not shown in the table.

TIE Opcode	Way 0			Way 1			Success?	Stll	Msg	Type
	TM	DM	Cntrl	TM	DM	Cntrl				
Load	1	1	0xx1xx	x	x	xxxxxx	Y	N	N	--
Load	x	x	xxxxxx	1	1	0xx1xx	Y	N	N	--
Load	0	x	xxxxxx	0	x	xxxxxx	N	Y	Y	Cache Miss
Store	1	1	0xx11x	x	x	xxxxxx	Y	N	N	--
Store	0	1	0xx10x	x	x	xxxxxx	N	N	Y	Upgrade Miss
Store	x	x	xxxxxx	1	1	0xx11x	Y	N	N	--
Store	x	x	xxxxxx	0	1	0xx10x	N	N	Y	Upgrade Miss
Store	0	x	xxxxxx	0	x	xxxxxx	N	N	Y	Cache Miss

Table B-6: Success/Failure conditions for LSU operations on caches

In the table above, Way0 and Way1 are state bit vectors returned from tag mats in ways 0 and 1 of the cache (for instruction cache there is no way1, therefore only information returned from way0 is considered). *TM* stands for *Total Match* output (comparison result for both data and control arrays in the mat), *DM* indicates *Data Match* output (comparison result for mat's data array) and *Control* are control bits read from control array. *Success* column indicates the result of the access if that specific bit pattern in encountered, *Stall* indicates whether processor has to be stalled or not, *Msg* says whether a message should be sent to protocol controller and *Type* specifies the message type.

Note that table is searched from bottom to top and content of the last matching entry is taken as output. Therefore, entries in the table are implicitly prioritized: each entry has priority over the entries lower to it. For example, by looking the last bottom two entries, one can notice that last entry for Store opcode (cache miss) covers the previous one (upgrade miss). In other words, the state bit vector in the entry with upgrade miss is a special case (subset) of the state bit vector for the cache miss case. However, since the table is searched from bottom to top, the output will be the last

matching entry when an upgrade misses is encountered (*Data Match* and *V* bit are active, but *E* bit is not, or in other words, line is valid and tags are matching, but there is no ownership).

Also note that the *Stall* column defines store accesses to be non-blocking, meaning that even when a store fails either due to a cache miss or upgrade miss, processor is not stalled and keeps executing later instructions. This is because processor interface keeps necessary information about the failed Store instruction and can complete it without stalling the processor. However, for Load instructions since processor needs the data word to load into the target register the access cannot be completed without processor being stalled.

In our simple example, memory mat accesses defined for protocol controller are always successful and therefore there is no need to define such condition table for protocol controller accesses.

B.2.2. ACCESSES TO MAIN MEMORY

Main memory in this example is either accessed via instruction or data caches to refill a cache line or by direct, un-cached accesses that go to segment 15. For cache accesses, main memory controller has to supply cache lines to be refilled into caches, or it receives cache lines that are being written back from caches. This involves reading and writing blocks of memory. For un-cached accesses only a single word in the memory is read or written at a time. Therefore, all main memory controllers have to provide is simple read/write accesses to main memory addresses. Main memory controller can then perform a series of such read/write accesses on successive addresses to do block read/writes.

B.3. COMMUNICATION MESSAGES

Third and last step in the system configuration process is to define protocol messages that are exchanged between levels of hierarchy and specify how they are handled at

each level. We mentioned what messages are sent from processor interface logic to protocol controller when a local memory access fails. This section elaborates on these messages as well as messages exchanged between protocol controller and main memory controller. It also specifies the details of the operations within each controller to handle messages.

B.3.1. DEFINING COMMUNICATION MESSAGES

Table B-7 lists all the request/reply messages exchanged between processor interface and protocol controller. It also explains the purpose of each message and conditions in which it is sent. Information fields of these messages were described in section 4.5.4. Protocol controller receives request messages from all processors in the Quad and sends a reply back for each message it receives. The table also lists the possible replies from controller to processors. Note that there is no type field for the reply messages, the table only indicates whether controller sends back data to or just an acknowledgement about requested operation being completed.

Message Type	Direction	Description
Cache Miss	LSU → Controller	Cache line is not present in the cache
Upgrade Miss	LSU → Controller	Cache line is present, but cache does not have ownership to
Un-cached Access	LSU → Controller	Direct accesses to off-chip memory
Reply Data	Controller → LSU	Returns data word to processor (Loads)
Reply Ack.	Controller → LSU	Returns acknowledgement indicating requested operation is complete (Stores and Prefetches)

Table B-7: Messages between processor interface and protocol controller

Similarly, communications messages exchanged between local and main memory controller are listed in Table B-8. Information carried by each message is listed and described in Table B-9.

Message Type	Direction	Description
Cache Miss	Local → Main	Sent when cache line is not present in the cache and needs to be fetched from main memory
Write-back	Local → Main	Sent when cache line is in <i>Modified</i> state and main memory copy has to be updated
Un-cached Request	Local → Main	Sent for direct accesses to off-chip memory
Refill	Main → Local	Returns requested data cache line
Un-cached Reply	Main → Local	Returns requested data word that is read from off-chip memory or Store acknowledgement

Table B-8: Messages between protocol controller and main memory controller

Field	Description
Source ID	ID of sender entity
Destination ID	ID of receiver entity
Type	Type of the message
Address	Address of word or cache line of interest
Requestor	Tile ID, processor ID and port ID of the requesting processor
Opcode	TIE opcode issued by processor
Data	A single data word (for un-cached requests) or a data block (for cached requests)
Byte Mask	For un-cached Stores, identifies which bytes should be written to main memory
Size	Size of the data block, if message carries a data block
SHR Index	Index of the status holding register (MSHR/USHR) that contains request's information. Used for retrieving the tracking information when reply is received
Line State	State in which line should be refilled in cache

Table B-9: Fields of messages between protocol and main memory controller

B.3.2. SPECIFYING PRIORITIES

While messages between processor interface and protocol controller are exchanged on the dedicated channel between them, messages between protocol controller and main memory controller are exchanged over the general interconnection network. Since this interconnect is used by all Quads and memory controllers in the system, at times it can

potentially be congested, blocked or un-accessible for sending packets. Furthermore, oblivious usage of available (virtual) channels might create circular dependency between messages waiting in system buffers and hence create a deadlock.

General strategy for avoiding deadlock in lossless interconnection networks is to separate messages into requests and replies. By definition [69][73] a reply is a message that will not generate another message as a result. Requests, however, are messages that might generate other messages when processed. Systems usually guarantee deadlock free communication by reserving enough buffering space for replies and limiting number of request messages that can be generated.

Smart Memories architecture uses similar strategy by assigning requests and replies to different virtual channels. Assigning virtual channel numbers to messages and setting up priorities between channels is one of the user's responsibilities when configuring the system. In our simple example, we assign virtual channel 1 (VC1) for carrying replies and virtual channel 2 (VC2) for carrying requests. Hence, cache miss and un-cached access requests are assigned to VC2, while write-back, refill and un-cached reply messages are assigned to VC1. Note that write-back is considered as a reply by this definition, since it does not generate any other message when being processed. Priorities for the virtual channels are adjusted by setting configuration registers in protocol controller and main memory controller network interfaces as well as the central network switch.

System relies on the back pressure mechanism provided by the flow control scheme in order to limit number of outstanding request messages, as describe in Appendix A. Whenever the network buffers of the request virtual channel are filled up, controllers will not be able to generate and send further requests. However, they are guaranteed to process messages on other virtual channels, the reply virtual channel in our example, such that there is no circular dependency between requests and replies and system always is guaranteed to make forward progress.

B.3.3. PROGRAMMING PROTOCOL CONTROLLER

For each communication message received by protocol controller, user has to define necessary processing steps in handling it. Protocol controller receives messages from processor interface logic in each Tile and main memory controller. As discussed earlier, the execution model of the controller is by defining and linking subroutines for each of the relevant functional units. Each input message invokes a chain of subroutine executions that complete its handling. In this section we describe how communication messages are handled inside protocol controller, what is the series of subroutines invokes for each message, and how the subroutines are defined for internal functional units.

First thing is to define the processing steps for each input message and determine which functional unit is responsible for executing that step. Table B-10 breaks down the processing steps required for each message received by protocol controller and identifies functional units that should participate in handling it.

In example system protocol controller accomplishes two major tasks: one is supporting caches by performing cache refills, doing write-backs and enforcing coherence protocol properties. Another task is handling un-cached accesses to off-chip memory. Tracking information for requests that are related to the above tasks is kept separately. Coherence protocol imposes serialization requirements on requests that for the same cache lines. More specifically, writes to the same location have to be serialized. This implies that local controller has to compare the line address of the incoming cache miss requests against already outstanding cache misses and serialize them if they target the same cache line. This task is accomplished by the tracking unit (T-Unit, cached section), using associative search capabilities of the MSHR structure. In contrast to the cache miss requests, there is no such serialization requirement on un-cached memory accesses; the only requirement is to store appropriate tracking information such that a reply can be sent back to requesting processor after un-cached access is completed. Hence controller can store tracking information for un-cached accesses in USHR.

Message	Unit	Operations
Cache Miss (from processor interface)	P-Unit	- Receive/decode message, pass to T-Unit, cached - Pass reply data/Ack to processor, release MSHR entry
	T-Unit	- Serialize against outstanding cache/upgrade misses - Allocate MSHR and line buffer entries, store tracking information in MSHR
	S-Unit	- Perform cache line eviction in the source cache - Snoop other Tile's caches and update cache line state - Read cache tags in source cache if write-back is necessary
	D-Unit	- Do a cache to cache transfer (if possible) - Read cache line data if write-back is necessary
	N-Unit	- Send cache miss message to main memory controller - Send write-back message to main memory controller, if necessary
Upgrade Miss (from processor interface)	P-Unit	- Receive/decode message, pass to T-Unit, cached - Pass reply Ack to processor, release MSHR entry
	T-Unit	- Serialize against outstanding cache/upgrade misses - Allocate MSHR and line buffer entries, store tracking information in MSHR
	S-Unit	- Snoop other Tile's caches and update cache line state - Change cache line state to <i>Modified</i> in source cache
	D-Unit	- Write data word into source cache
Un-cached Access (from processor interface)	P-Unit	- Receive/decode message, pass to T-Unit, un-cached
	N-Unit	- Send un-cached access message to main memory controller
Refill (from main memory controller)	P-Unit	- Pass reply data/Ack to processor, release MSHR entry
	T-Unit	- Retrieve tracking information from MSHR
	S-Unit	- Write tags, adjust cache line state in source cache
	D-Unit	- Write data portion of line in source cache
	N-Unit	- Receive/decode message, pass to T-Unit cached
Un-cached Reply (from main memory controller)	P-Unit	- Pass data/Ack to processor, release USHR entry
	T-Unit	- Retrieve tracking information from USHR
	N-Unit	- Receive/decode message, pass to T-Unit un-cached

Table B-10: Breakdown of message handling steps in protocol controller

After performing appropriate serialization and storing the tracking information, cache miss request is passed to S-Unit. S-Unit manipulates the state information in Tile caches: for cache miss requests it evicts cache lines by putting them in *Reserved* state and snoops other caches to adjust the cache line state and see whether a cache-to-cache transfer is possible. For refill operations, it writes cache tags and adjusts cache line state. S-Unit is not used for handling un-cached memory accesses since there is not state information to be operated on.

D-Unit handles all data access operations: it reads evicted cache lines from source cache if write-back is necessary, refills new lines into the caches when they are received from main memory controller, and potentially does a cache-to-cache transfer from one Tile to another.

N-Unit sends request messages to main memory controller and receives and decodes reply messages. Similarly P-Unit receives request messages from e processor interface, decodes them and passes them to appropriate part of the T-Unit. It also sends back replies (data or acknowledgement) to processor interface logic in Tiles.

Figure B-5 shows the flow of operations inside protocol controller for each one of the above messages. It also shows the subroutines that are called in each unit to perform a processing step. After executing a subroutine in a functional unit, request might be passed to one unit or more depending on the conditions that are evaluated. For example in case of a cache miss, if S-Unit finds a valid copy of a cache line in another Tile's cache, it performs a cache-to-cache transfer otherwise it sends the miss request to main memory controller. Solid lines in the figure represent the calls that are always made; dotted lines indicate that only one of the calls is made.

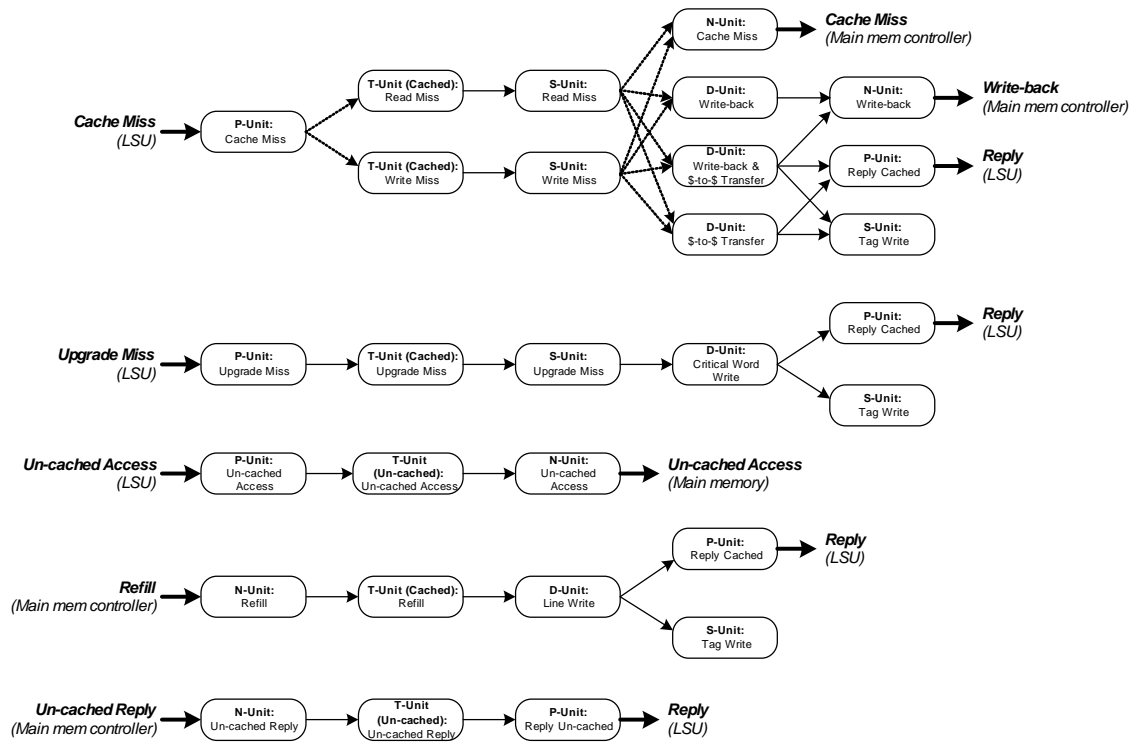


Figure B-5: Flow of operations for processing messages in protocol controller

Figure B-6 to Figure B-10 list subroutines for each one of the functional units in the controller in as a pseudo-code. Each subroutine might call one or more subroutines in other functional units after it completes all of its operations. Note that parameters of the input message such as memory address, write data, requestor, index in MSHR/USHR structures, etc. are passed along with the each subroutine invocation.

<p>P-Unit:</p> <p><u>Cache Miss:</u> if (TIE Opcode is READ) Call T-Unit(cached)::Read Miss else Call T-Unit(cached)::Write Miss</p> <p><u>Upgrade Miss:</u> Call T-Unit(cached)::Upgrade Miss</p>	<p><u>Un-cached Access:</u> Call T-Unit(un-cached)::Un-cached Access</p> <p><u>Reply Cached:</u> Send reply to processor Release MSHR entry</p> <p><u>Reply Un-cached:</u> Send reply to processor Release USHR entry</p>
--	--

Figure B-6: P-Unit subroutines

<p>T-Unit (cached):</p> <p><u>Read Miss:</u> MSHR Lookup (Address) if (exists request to same Address) Do not accept if (No available entry in MSHR) Do not accept if (No available entry in Line buffer) Do not accept Allocate MSHR entry Allocate Line buffer entry Store tracking information in MSHR Call S-Unit::Read Miss</p> <p><u>Upgrade Miss:</u> MSHR Lookup (Address) if (exists request to same Address) Do not accept if (No available entry in MSHR) Do not accept Allocate MSHR entry Store tracking information to MSHR Call S-Unit::Upgrade Miss</p>	<p><u>Write Miss:</u> MSHR Lookup (Address) if (exists request to same Address) Do not accept if (No available entry in MSHR) Do not accept if (No available entry in Line buffer) Do not accept Allocate MSHR entry Allocate Line buffer entry Store tracking information to MSHR Store write data in Line buffer Call S-Unit::Write Miss</p> <p><u>Refill:</u> Retrieve tracking information from MSHR Call D-Unit::Line Write</p>
<p>T-Unit (un-cached):</p> <p><u>Un-cached Access:</u> if (No available entry in USHR) Do not accept Allocate USHR entry Store tracking information in USHR Call N-Unit::Un-cached Access</p>	<p><u>Un-cached Reply:</u> Retrieve tracking information from USHR Call P-Unit::Reply</p>

Figure B-7: T-Unit subroutines (cached and un-cached parts)

<p>S-Unit:</p> <p><u>Read Miss:</u> Send <i>Evict</i> to requesting cache Send <i>Snoop-Read</i> to other caches if (found in other caches) if (Evicted line is Modified) Call D-Unit::<i>Write-back</i> & <i>\$-To-\$</i> transfer else Call D-Unit::<i>\$-To-\$</i> transfer else if (Evicted line is Modified) Call D-Unit::<i>Write-back</i> Call N-Unit::<i>Cache Miss</i></p> <p><u>Write Miss:</u> Send <i>Evict</i> to requesting cache Send <i>Snoop-ReadEx</i> to other caches if (found in other caches) if (Evicted line is Modified) Call D-Unit::<i>Write-back</i> & <i>\$-To-\$</i> transfer else Call D-Unit::<i>\$-To-\$</i> transfer else if (Evicted line is Modified) Call D-Unit::<i>Write-back</i> Call N-Unit::<i>Cache Miss</i></p>	<p><u>Upgrade Miss:</u> Send <i>Snoop-ReadEx</i> to other caches Call D-Unit::<i>Critical Word Write</i></p> <p><u>Tag Write:</u> Send <i>Refill</i> to source cache</p>
--	---

Figure B-8: S-Unit subroutines

<p>D-Unit:</p> <p><u>Write-back:</u> for (i=0 to Size-1) Send <i>Line Read</i> to requesting cache Write word into <i>Line buffer</i> Call N-Unit::<i>Write-back</i></p> <p><u>Write-back & \$-to-\$ transfer:</u> for (i=0 to Size-1) Send <i>Read</i> to requesting cache Write word into <i>Line buffer</i> Call N-Unit::<i>Write-back</i> for (i=0 to Size-1) Send <i>Read</i> to source cache Write word into <i>Line buffer</i> for (i=0 to Size-1) Read word from <i>Line buffer</i> Send <i>Write</i> to requesting cache Call S-Unit::<i>Tag Write</i> Call P-Unit::<i>Reply</i></p>	<p><u>\$-to-\$ transfer:</u> for (i=0 to Size-1) Send <i>Read</i> to source cache Write word into <i>Line buffer</i> for (i=0 to Size-1) Read word from <i>Line buffer</i> Send <i>Write</i> to requesting cache Call S-Unit::<i>Tag Write</i> Call P-Unit::<i>Reply</i></p> <p><u>Line Write:</u> for (i=0 to Size-1) Read word from <i>Line buffer</i> Send <i>Write</i> to requesting cache Call S-Unit::<i>Tag Write</i> Call P-Unit::<i>Reply</i></p> <p><u>Critical Word Write:</u> Read word from <i>Line buffer</i> Send <i>Write</i> to requesting cache Call S-Unit::<i>Tag Write</i> Call P-Unit::<i>Reply</i></p>
--	--

Figure B-9: D-Unit subroutines

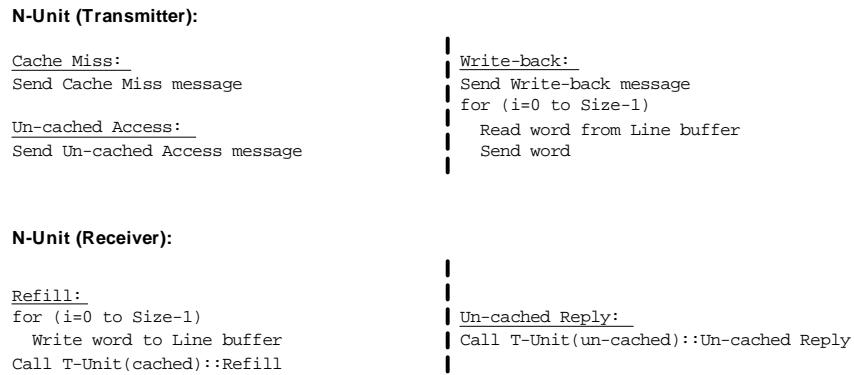


Figure B-10: N-Unit subroutines (receiver and transmitter)

B.3.4. PROGRAMMING MAIN MEMORY CONTROLLER

Programming main memory controller is very similar to programming protocol controller. Table B-11 shows the breakdown of steps in handling input messages to main memory controller and Figure B-11 shows the flow of operations.

Message	Unit	Operation description
Cache Miss	Network Intf.	- Receive/decode message, pass to C-Req - Send reply message back to protocol controller
	C-Req	- Allocate MSHR and memory queue entries, store tracking information in MSHR - Issue memory read request to memory queue
	C-Rep	- Initiate reply process when memory access is complete - Release MSHR entry
	Memory Intf.	- Read cache line from memory
Write-back	Network Intf.	- Receive/decode message, pass to C-Req
	C_Req	- Allocate MSHR and memory queue entries, store tracking information in MSHR - Issue memory write request to memory queue
	C-Rep	- Release MSHR entry when memory access is complete
	Memory Intf.	- Write cache line to memory
Un-cached Access	Network Intf.	- Receive/decode message, pass to U-Req/Rep - Send reply message back to protocol controller
	U-Req/Rep	- Allocate memory queue entry, issue memory access to memory queue -Initiate reply process when memory access is complete
	Memory Intf.	- Read/Write word from/to memory

Table B-11: Breakdown of message handling steps in main memory controller

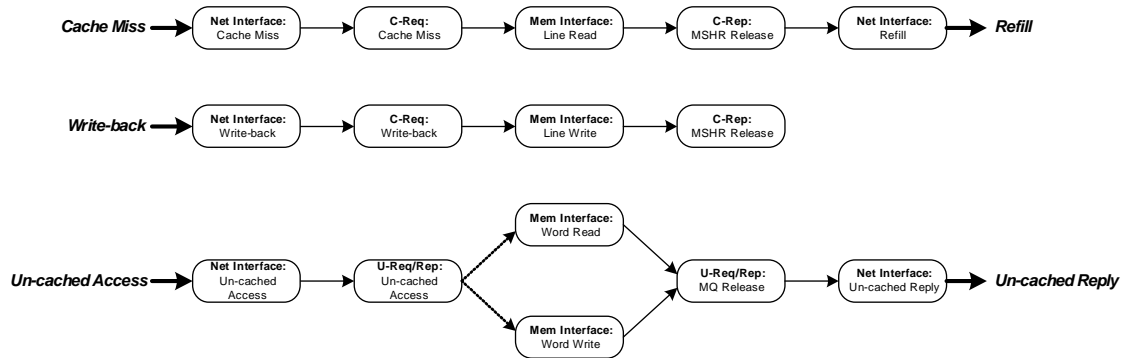


Figure B-11: Flow of operations for processing messages in main memory controller

B.4. SUMMARY

As illustrated by the simple example of coherent shared memory model, there are three major steps in implementing a memory protocol on the Smart Memories hardware platform. In first step, user should allocated resources for storing data and associated state information. This involves defining state information that should be associated with data, allocating physical storage locations where data and state information are stored, and defining translation/mapping functions that produces address of physical location(s) corresponding to processor's virtual addresses.

Second step is defining accesses for local and off-chip memories (on both data and state information), specifying success/failure condition for each memory access, and defining request messages that should be issued if an access fails. These conditions are specified in terms of bit vectors that are compared with state bits returned from memory mats. Following this step, user has to define all communication messages between all levels of hierarchy, specify their priorities when traveling on interconnect, and program protocol controller and main memory controller to appropriately handle each and every message. Controllers employ a simple step by step processing method that involves defining subroutines for functional units and then chaining appropriate subroutines together to handle a specific input message.

While our example is very simplistic it shows all the necessary steps of the system configuration. Smart Memories is capable of implementing a variety of memory models, including coherent shared memory, streaming and transactional coherence and consistency (TCC). The system by no means is limited to these currently implemented protocols: When implementing shared memory models, system can support various coherence protocols such as MSI, MESI or MOESI or updated based protocols on both single Quad and multi-Quad configurations. It is capable of supporting hybrid memory models, for example combining streaming and caches; using caches simplifies accesses to instruction code and runtime stack, while streaming operations and DMA accesses are used for accessing application data. Even though Transactional

Coherence and Consistency [27] has been chosen to be primary hardware transactional memory protocol, user can implement other HTM protocols such as LogTM [24] or change various parameters in a transactional memory system, such as granularity of conflict detection between transactions (word vs. cache line), system commit policy (eager vs. lazy) and conflict detection policy (optimistic vs. pessimistic).

BIBLIOGRAPHY

- [1] Khailany, B., Dally, W.J., Rixner, S., Kapasi, U.J., Mattson, P., Namkoong, J., Owens, J.D., Towles, B., and Chang, A., Imagine: Media Processing with Streams, IEEE Micro, Vol. 21, Issue 2, pp. 35-46, March/April 2001.
- [2] Ahn, J.H., Dally, W.J., Khailany, B., Kapasi, U.J., Das, A., Evaluating the Imagine Stream Architecture, Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA-31), pp. 14-25, Munich, Germany, June 2004.
- [3] Lipasti, M., Wilkerson, C., Shen, J.P., Value Locality and Load Value Prediction, Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII), pp.138-147, Cambridge, MA, October 1996.
- [4] Lipasti, M., Shen, J.P., Exceeding the Dataflow Limit via Value Prediction, Proceedings of the 29th International Symposium on Microarchitecture (MICRO-29), pp. 226-237, Paris, France, December 1996.
- [5] Rychlik, B., Faistl, J., Krug, B., Shen, J.P., Efficacy and Performance Impact of Value Prediction, Proceedings of the 7th International Conference on Parallel Architectures and Compilation Techniques (PACT'98), pp. 148-154, Paris, France, October 1998.
- [6] Prabhu, M., Olukotun, K., Using Thread-Level Speculation to Simplify Manual Parallelization, Proceedings of the 9th Symposium on Principles and Practice of Parallel Programming (PPoPP'03), pp. 1-12, San Diego, CA, 2003.
- [7] Hammond, L., Willey, M., Olukotun, K., Data Speculation Support for a Chip Multiprocessor, Proceedings of the 8th Conference on Architectural Support for

Programming Languages and Operating Systems (ASPLOS-VIII), pp. 58-69, San Jose, CA, October 1998.

- [8] Rajwar, R., Goodman, J., Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution, Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO-34), pp. 294-305, Austin, TX, 2001.
- [9] Sohi, G., Breach, S., Vijaykumar, T.N., Multiscalar processors, Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA-22), pp. 414-425, Italy, 1995.
- [10] Franklin, M., Sohi, G., ARB: A Hardware Mechanism for Dynamic Reordering of Memory References, IEEE Transactions on Computers, Vol. 45, No. 5, pp. 552-571, May 1996.
- [11] Gopal, S., Vijaykumar, T.N., Smith, J.E., Sohi, G., Speculative Versioning Cache, IEEE Transactions on Parallel and Distributed Systems, Vol. 12, Issue 12, pp. 1305-1317, 2001.
- [12] Hammond, L., Hubbert, B., Siu, M., Prabhu, M., Chen, M., Olukotun, K., The Stanford Hydra CMP, IEEE Micro, Vol. 20, Issue 2, pp. 71-84, March/April 2000.
- [13] Olukotun, K., Hammond, L., Willey, M., Improving the Performance of Speculatively Parallel Applications on the Hydra CMP, Proceedings of the 13th International Conference on Supercomputing, pp. 21-30, Rhodes, Greece, 1999.

- [14] Steffan, J.G., Colohan, C.B., Zhai, A., Mowry, T.C., A Scalable Approach to Thread-Level Speculation, Proceedings of the 27th International Symposium on Computer Architecture (ISCA-27), pp. 1-12, Vancouver, Canada, June 2000.
- [15] Steffan, J.G., Colohan, C.B., Zhai, A., Mowry, T.C., Improving Value Communication for Thread-Level Speculation, Proceedings of 8th International Symposium on High-Performance Computer Architecture (HPCA-8), pp. 65-75, Cambridge, MA, February 2002.
- [16] Zhang, Y., Rauchwerger, L., Torrellas, J., Hardware for Speculative Parallelization of Partially-Parallel Loops in DSM Multiprocessors, Proceedings of 5th International Symposium on High-Performance Computer Architecture (HPCA-5), pp. 135-139, Orlando, FL, January 1999.
- [17] Cintra, M., Martinez, J.F., Torrellas, J., Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors, Proceedings of 27th International Symposium on Computer Architecture (ISCA-27), pp. 13-24, Vancouver, Canada, June 2000.
- [18] Herlihy, M., Moss, J.E.B., Transactional Memory: Architectural Support for Lock-Free Data Structures, Proceedings of 20th International Symposium on Computer Architecture (ISCA-20), pp. 289-300, San Diego, CA, 1993.
- [19] Larus, J.R., Rajwar, R., Transactional Memory, Morgan & Claypool, 2007.
- [20] Ramakrishnan, R., Gehrke, J., Database Management Systems, New York: McGraw-Hill, 2000.

- [21] Shavit, N., Touitou, D., Software Transactional Memory, Proceedings of 14th Symposium on Principles of Distributed Computing (PODC14), pp. 204-213, Ottawa, Canada, August 1995.
- [22] Harris, T., Fraser, K., Language Support for Lightweight Transactions, Proceedings of 18th Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA-18), pp. 388-402, Anaheim, CA, October 2003.
- [23] Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N., Software Transactional Memory for Dynamic-Sized Data Structures, Proceedings of 22nd Symposium on Principles of Distributed Computing, Boston, MA, July 2003.
- [24] Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., Wood, D.A., LogTM: Log-Based Transactional Memory, Proceedings of 12th International Symposium on High-Performance Computing (HPCA-12), pp. 254-265, Austin, TX, February 2006.
- [25] Moravan, M.J., Bobba, J., Moore, K.E., Yen, L., Hill, M.D., Liblit, B., Swift, M.M., Wood, D.A., Supporting Nested Transactional Memory in LogTM, Proceedings of 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII), pp. 359-370, San Jose, CA, October 2006.
- [26] Yen, L., Bobba, J., Marty M.R., Moore, K.E., Volos, H., Hill, M.D., Swift, M.M., Wood, D.A., LogTM-SE: Decoupling Hardware Transactional Memory from Caches, Proceedings of 13th International Symposium on High-

Performance Computer Architecture (HPCA-13), pp. 261-272, Phoenix, AZ, February 2007.

- [27] Hammond, L., Wong, V., Chen, M., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M., Wijaya, H., Kozyrakis, C., Olukotun, K., Transactional Memory Coherence and Consistency, Proceedings of 31st International Symposium on Computer Architecture (ISCA-31), p. 102, Munich, Germany, June 2004.
- [28] McDonald, A., Chung, J., Chafi, H., Minh, C.C., Carlstrom, B.D., Hammond, L., Kozyrakis, C., Olukotun, K., Characterization of TCC on Chip Multiprocessors, Proceedings of 14th International Conference on Parallel Architecture and Compilation Techniques (PACT'05), pp. 63-74, September 2005.
- [29] Ananian, C.S., Asanovic, K., Kuszmaul, B.C., Leiserson, C.E., Lie, S., Unbounded Transactional Memory, Proceedings of 11th International Symposium on High-Performance Computer Architecture (HPCA-11), pp. 316-327, San Francisco, CA, February 2005.
- [30] Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D., Hybrid Transactional Memory, Proceedings of 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII), pp. 336-346, San Jose, CA, October 2006.
- [31] Kumar, S., Chu, M., Hughes, C.J., Kundu, P., Nguyen, A., Hybrid Transactional Memory, Proceedings of 11th Symposium on Principles and Practices of Parallel Programming (PPoPP'06), pp. 209-220, New York, NY, March 2006.

- [32] Saha, B., Adl-Tabatabai, A.R., Jacobson, Q., Architectural Support for Software Transactional Memory, Proceedings of 39th International Symposium on Microarchitecture (MICRO-39), pp. 185-196, Orlando, FL, December 2006.
- [33] Horowitz, M., Dally, W., How Scaling Will Change Processor Architecture, Digest of Technical Papers, IEEE International Solid-State Circuits Conference (ISSCC), pp. 132-133, San Francisco, CA, 2004.
- [34] Agarwal, V., Hrishikesh, M.S., Keckler, S.W., Burger, D., Clock Rate versus IPC: The end of the Road for Conventional Microarchitectures, Proceedings of 27th International Symposium on Computer Architecture (ISCA-27), pp. 248-259, Vancouver, Canada, June 2000.
- [35] Patterson, D.A., Hennessy, J.L., Computer Architecture: A Quantitative Approach (fourth edition), Morgan Kaufmann, 2006.
- [36] Rusu, S., Tam, S., Muljono, H., Ayers, D., Chang, J., Cherkauer, B., Stinson, J., Benoit, J., Varada, R., Leung, J., Limaye, R. D., Vora, S., A 65-nm Dual-Core Multithreaded Xeon® Processor With 16-MB L3 Cache, IEEE Journal of Solid-State Circuits, Vol. 42, Issue 1, pp. 17-25, January 2007.
- [37] Stackhouse, B., Cherkauer, B., Gowan, M., Gronowski, P., Lyles, C., A 65nm 2-Billion-Transistor Quad-Core Itanium® Processor, Digest of Technical Papers, IEEE International Solid-State Circuits Conference (ISSCC), pp. 92-93, San Francisco, CA, February 2008.
- [38] Dorsey, J., Searles, S., Ciraula, M., Johnson, S., Bujanos, N., Wu, D., Braganza, M., Meyers, S., Fang, E., Kumar, R., An Integrated Quad-Core

- Opteran™ Processor, Digest of Technical Papers, IEEE International Solid-State Circuits Conference (ISSCC), pp. 102-103, San Francisco, CA, February 2007.
- [39] Kongetira, P., Aingaran, K., Olukotun, K., Niagara: A 32-Way Multithreaded Sparc Processor, IEEE Micro, Vol. 25, Issue 2, pp. 21-29, March/April 2005.
- [40] Nawathe, U.G., Hassan, M., Yen, K.C., Kumar, A., Ramachandran, A., Greenhill, D., Implementation of an 8-Core, 64-Thread, Power-Efficient SPARC Server on a Chip, IEEE Journal of Solid-State Circuits, Vol. 43, Issue 1, pp. 6-20, January 2008.
- [41] Pham, D.C., Aipperspach, T., Boerstler, D., Bolliger, M., Chaudhry, R., Cox, D., Harvey, P., Harvey, P.M., Hofstee, H.P., Johns, C., Kahle, J., Kameyama, A., Keaty, J., Masubuchi, Y., Pham, M., Pille, J., Posluszny, S., Riley, M., Stasiak, D.L., Suzuoki, M., Takahashi, O., Warnock, J., Weitzel, S., Wendel, D., Yazawa, K., Overview of the Architecture, Circuit Design, and Physical Implementation of a First-Generation Cell Processor, IEEE Journal of Solid-State Circuits, Vol. 41, Issue 1, pp. 179-196, January 2006.
- [42] Lewis, B., Berg, D. J., Multithreaded Programming with Pthreads, Prentice Hall, 1998.
- [43] Nichols, B., Buttlar, D., Farrell, J.P., Pthreads Programming, O'Reilly 1996.
- [44] Lusk, E.L., Overbeek, R.A., Use of Monitors in FORTRAN: A Tutorial on the Barrier, Self-scheduling DO-Loop, and Askfor Monitors, Tech. Report No. ANL-84-51, Rev. 1, Argonne National Laboratory, June 1987.

- [45] Lusk, E.L., Overbeek, R.A., Portable Programs for Parallel Processors, Holt, Rinehart and Winston Inc., 1987.
- [46] Tremblay, M., Chaudhry, S., A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARC® Processor, Digest of Technical Papers, IEEE International Solid-State Circuits Conference (ISSCC), pp. 82-83, San Francisco, CA, February 2008.
- [47] Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A., The SPLASH-2 Programs: Characterization and Methodological Considerations, Proceedings of 22nd International Symposium on Computer Architecture (ISCA-22), pp. 24-36, Santa Margherita Ligure, Italy, June 1995.
- [48] Gschwind, M., The Cell Broadband Engine: Exploiting Multiple Levels of Parallelism in a Chip Multiprocessor, International Journal of Parallel Programming, Vol. 35, No. 3., pp. 233-262, June 2007.
- [49] Gummaraju, J., Erez, M., Coburn, J., Rosenblum, M., Dally, W.J., Architectural Support for Stream Execution Model on General-Purpose Processors, 16th International Conference on Parallel Architectures and Compilation Techniques (PACT'07), pp. 3-12, Brasov, Romania, September 2007.
- [50] Gummaraju, J., Coburn, J., Turner, Y., Rosenblum, M., Streamware: Programming General-Purpose Multicore Processors Using Streams, Proceedings of 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIII), pp. 297-307, Seattle, WA, March 2008.

- [51] Gummaraju, J., Rosenblum, M., Stream Programming on General-Purpose Processors, Proceedings of 38th International Symposium on Microarchitecture (MICRO-38), pp. 343-354, Barcelona, Spain, November 2005.
- [52] Taylor, M.B., Kim, J., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffman, H., Johnson, P., Lee, J.-W., Lee, W., Ma, A., Saraf, A., Seneski, M., Shnidman, N., Strumpfen, V., Frank, M., Amarasinghe, S., Agarwal, A., The Raw Microprocessor A Computational Fabric for Software Circuits and General Purpose Programs, IEEE Micro, Vol. 22, Issue 2., pp. 25-35, March/April 2002.
- [53] Taylor, M.B., Psota, J., Saraf, A., Shnidman, N., Strumpfen, V., Frank, M., Amarasinghe, S., Agarwal, A., Lee, W., Miller, J., Wentzlaff, D., Bratt, I., Greenwald, B., Hoffmann, H., Johnson, P., Kim, J., Evaluation of the Raw Microprocessor An Exposed-Wire-Delay Architecture for ILP and Streams, Proceedings of 31st International Symposium on Computer Architecture (ISCA-31), pp. 2-13, Munich, Germany, June 2004.
- [54] Thies, W., Karczmarek, M., Amarasinghe, S., StreamIt a Language for Streaming Applications, Proceedings of the 11th International Conference on Compiler Construction, pp. 179-196, 2002.
- [55] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P., Brook for GPUs: Stream Computing on Graphics Hardware, ACM Transactions on Graphics, Vol. 23, No. 3, pp. 777-786, August 2004.
- [56] Labonte, F., Mattson, P., Thies, W., Buck, I., Kozyrakis, C., Horowitz, M., The Stream Virtual Machine, Proceedings of 13th International Conference on

Parallel Architectures and Compilation Techniques (PACT'04), pp. 267-277, Antibes Juan-les-Pins, France, September-October 2004.

- [57] Fatahalian, K., Knight, T., Houston, M., Erez, M., Horn, D., Leem, L., Park, J.Y., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P., Sequoia: Programming the Memory Hierarchy, Proceedings of the Conference on Supercomputing (SC'06), p. 4, November 2006.
- [58] Knight, T., Park, J.Y., Ren, M., Houston, M., Erez, M., Fatahalian, K., Aiken, A., Dally, W.J., Hanrahan, P., Compilation for Explicitly Managed Memory Hierarchies, Proceedings of the 12th Symposium on Principles and Practices of Parallel Programming (PPoPP'07), San Jose, CA, March 2007.
- [59] Kapasi, U.J., Rixner, S., Dally, W.J., Khailany, B., Jung Ho Ahn Mattson, P., Owens, J.D., Programmable Stream Processors, IEEE Computer, Vol. 36, Issue 8, pp. 54-62, August 2003.
- [60] Khailany, B., Dally, W.J., Chang, A., Kapasi, U.J., Namkoong, J., Towels, B., VLSI Design and Verification of the Imagine Processor, Proceedings of the 20th International Conference on Computer Design (ICCD'02), p. 289, Freiburg, Germany, September 2002.
- [61] Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D., Introduction to the Cell Multiprocessor, IBM Journal of Research and Development, Vol. 49, No. 4/5, pp. 589-604, July/September 2005.
- [62] Flachs, B., Asano, S., Dhong, S.H., Hofstee, H.P., Gervais, G., Kim, R., Le, T., Liu, P., Leenstra, J., Liberty J., Michael, B., Oh, H., Mueller, S.M., Takahashi, O., Hatakeyama, A., Watanabe, Y., Yano, N., Brokenshire, D.,A., Peyravian,

- M., To, V., Iwata, E., The Microarchitecture of the Synergistic Processor for a Cell Processor, IEEE Journal of Solid-State Circuits, Vol. 41, No. 1, January 2006.
- [63] Adve, S.V., Gharachorloo, K., Shared Memory Consistency Models: A Tutorial, IEEE Computer, Vol. 29, Issue 12, pp 66-76, December 1996.
- [64] Leverich, J., Arakida, H., Solomatnikov, A., Firoozshahian, A., Horowitz, M., Kozyrakis, C., Comparing Memory Systems for Chip Multiprocessors, Proceedings of 34th International Symposium on Computer Architecture (ISCA-34), pp. 358-368, San Diego, CA, June 2007.
- [65] Martin, M.M.K., Hill, M.D., Wood, D.A., Token Coherence: Decoupling Performance and Correctness, Proceedings of 30th International Symposium on Computer Architecture (ISCA-30), pp. 182-193, San Diego, CA, June 2003.
- [66] R. Gonzalez, Configurable and Extensible Processors Change System Design, *Hot Chips 11*, Stanford, CA, August 1999.
- [67] A. Wang, E. Killian, D. Maydan, C. Rowen, Hardware/software instruction set configurability for system-on-chip processors, Proceedings of the 38th Design Automation Conference (DAC-38), pp.184-188, Las Vegas, NV., June 2001.
- [68] D. Jani, G. Ezer, J. Kim, Long Words and Wide Ports: Reinventing the Configurable Processor, *Hot Chips 16*, Stanford, CA, August 2004.
- [69] Culler, D.E., Singh, J.P., Gupta, A., Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufman, 1998.

- [70] Mai, K., Paaske, T., Jayasena, N., Ho, R., Dally, W.J., Horowitz, M., Smart Memories: A Modular Reconfigurable Architecture, Proceedings of 27th International Symposium on Computer Architecture (ISCA-27), pp.161-171, Vancouver, Canada, June 2000.
- [71] Mai, K., Ho, R., Alon, E., Dean, L., Kim, Y., Patil, D., Horowitz, M., Architecture and Circuit Techniques for a Reconfigurable Memory Block, Digest of Technical Papers, IEEE International Solid-State Circuits Conference (ISSCC), pp. 542-543, San Francisco, CA, February 2004.
- [72] McKeown, N., The iSLIP Scheduling Algorithm for Input-Queued Switches, IEEE/ACM Transactions on Networking (TON), Vol. 7, Issue 2, pp.188-201, April 1999.
- [73] Dally, W.J., Towels, B., Principles and Practices of Interconnection Networks, Morgan Kaufman, 2004.
- [74] Gosling, J., Joy, B., Steele, G., Bracha, G., The JavaTM Language Specification, Third edition, Addison-Wesley, 2005.
- [75] OpenMP Official Web Site, <http://openmp.org/wp/>
- [76] Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Parallel Programming in OpenMP, Morgan Kaufman, 2001.
- [77] Martin, M.M.K., Soring, D.J., Ailamaki, A., Alameldeen, A.R., Dickson, R.M., Maur, C.J., Moore, K.E., Plakal, M., Hill, M.D., Wood, D.A.,

Timestamp Snooping: An Approach for Extending SMPs, Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), pp.25-36, Cambridge, MA, October 2000.

- [78] Tensilica official website, Xtensa LX2 processor family product brief, http://www.tensilica.com/pdf/xtensa_LX2_April07.pdf