INVESTIGATING THE SCALABILITY OF TILED CHIP MULTIPROCESSORS USING MULTIPLE NETWORKS

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2009

By
Preethi Sam
School of Computer Science

Contents

List of Tables

List of Figures

Abstract

The era of billion and more transistors on a single silicon chip has already begun and this has changed the direction of future computing towards building chip multiprocessors (CMP) systems. Nevertheless the challenges of maintaining cache coherency as well as providing scalability on CMPs is still in its initial stages of development. This thesis therefore investigates the scalability of cache coherent CMP systems.

Previous studies have shown that single bus based cache coherent CMPs do not scale. Directory based CMPs systems provide better scalability, but have overhead in terms of the space for a full map directory as well as latency in providing for broadcasting of writes to widely shared data.

In this thesis the idea of using two separate (multiple) networks is explored for providing a combination of snoop and directory based protocols on a CMP. The cache coherency traffic is split over two separate interconnects. A limited directory based scheme with low space overhead is used over one network for handling all requests and non-broadcast based cache coherency responses. The second network is specifically used for supporting broadcast based invalidations to widely shared data. The cache coherency protocol is optimized by removing the need to generate acknowledgement messages during writes to widely shared data, as required by directory protocols. A combination of homogeneous and heterogeneous networks is implemented giving rise to two architecturally different CMP systems. The performance of both these CMP architectures is evaluated using multithreaded benchmarks. Results do confirm that the homogeneous networks based scheme is a promising design for small and medium sized CMP systems.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to thank Dr. Ian Watson for giving me an opportunity to pursue a PhD and for all the help and support during the study. I would also like to thank members of the APT group. Special thanks to members of the JAMAICA group for all the motivation and help, especially Matt for all the guidance and inputs with regards to designing the cache coherency protocol and debugging the simulator.

Many Thanks to all my friends in Manchester (as well as in India and US) for making my stay enjoyable inspite of the numerous pressures associated with the PhD. Special thanks to Uncle Partha and Aunt Winifred for being like a family to me in Manchester, Hande for being such a great roomate and Varsha for being a true and lasting friend.

Last but not the least, I thank GOD for giving me loving parents and family members. Without their prayers and constant encouragement this PhD would not be possible.

Chapter 1

Introduction

1.1 Motivation

In the past, parallel machines were relegated to research based and high performance scientific applications. These applications are used to model physical phenomena that are costly, and in some instances impossible to observe without their occurance [CSG99]. Examples include, simulation of weather forecast, interaction between astronomical bodies (N-body simulations), crash analysis in automobiles and studying molecular dynamics as required by chemical and pharmaceutical industries [CSG99]. In contrast, uniprocessor (single processor chip) systems dominated the desktop computing market. With the explosion of the internet, databases and server computing based applications started to encroach the realm of parallel applications. While on the desktop side, gaming, multimedia and mobile computing started fuelling the need for higher performance.

Since its inception around 1970's, and upto the mid 80's [HP07], performance enhancement in uniprocessors was sustained by technological improvements in process generation, resulting in doubling of transistors on a single chip as predicted by Moore's Law (Figure 1.1 [Bha05]) and increase in processor clock frequency. From mid 80's until about 2002, increased on-chip transistor budgets gave rise to several microarchitecture enhancements such as wider pipelines, out of order processing for increased throughput in terms of more number of instructions per cycle, better branch prediction mechanisms and larger on-chip cache memory [HP07]. However, after 2002, the returns from these architectural enhancements and increase in clock frequency started to diminish mainly due to the following factors:

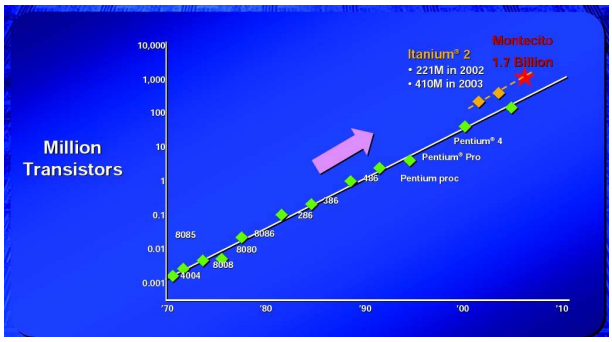

Figure 1.1: Moore's Law [Bha05]

- Power Consumption

  Power consumption within a processor can be classified as static and dynamic power. Static power is equivalent to the power consumed by a circuit when it is in its quiescent state, primarily due to leakage currents (current that flows through transistors when they are in the off state). While dynamic power is the power consumed when transistors switch within a circuit [BKJN99]. The formula for static [HP07] and dynamic power [Bha05] is given by equation 1.1 and equation 1.2 [KM08] respectively:

$$P = I_{static} * V \tag{1.1}$$

$$P = C * V^2 * f \tag{1.2}$$

  Where P = power consumed I_{static} = leakage current C = load capacitance V = voltage of operation f = frequency of operation

  Frequency $\propto$ Voltage. Therefore, P $\propto$ $frequency^3$. It is estimated that almost 40% of the chip's power consumption is due to superscalar uniprocessor core on-chip [Bha05]. Reducing the power consumed using techniques such as voltage scaling or local clock gating [EB00] leads to lower performance in uniprocessor (superscalar processors) systems [Bha05] and does not account for power consumed when wires are driven through idling processor cores [KFJ+03]. Therefore, there is a non-linear increase in power as frequency increases. Figure 1.2 shows how power due to leakage current is catching up with active power.

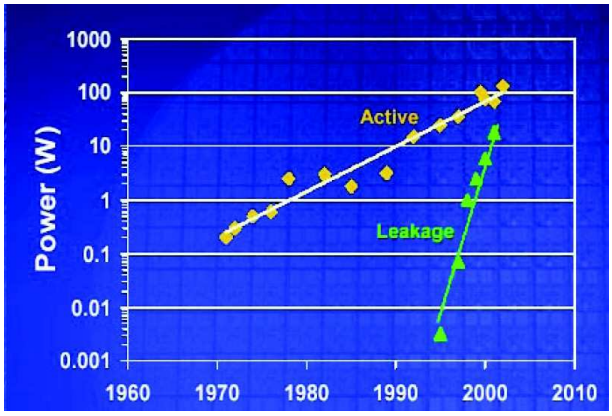

Figure 1.2: Active and Leakage Power increases as process technology improves over the years [Bha05]

- Memory Wall

  In the past twenty years, DRAM[1] sizes have increased by about 40% per year [HP07] and their access latencies have decreased by 76%. However, these enhancements in off-chip memory latencies is unparalleled to the increase in processor clock frequencies (120 times) as compared to 15 times for the clock frequency for the DRAM. Figure 1.3 shows the relative improvement in performance for DRAMs and processors. Therefore, in order to avoid the long wait times to access memory, processor cores normally contain a hierarchy of caches. Apart from using caches, multithreaded and speculative processor cores, and software [GGV90] and hardware prefetching [BC91] is used to reduce the effect of the penalty associated with memory access.

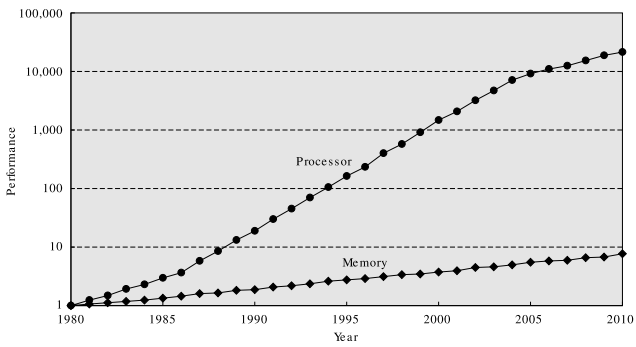

Figure 1.3: Relative Performance of DRAMs w.r.t. Processor [HP03]

[1]Dynamic Random Access Memory

- Wire Delay

  With shrinking feature sizes (minimum size of a transistor or wire in the X or Y dimension on chip) [HP07], wires become thinner and transistor size and CMOS switching times decrease. The resistance of a wire of length l is calculated using the formula as given below [BKJN99][Ye03]:

$$R = \frac{\rho \times l}{A} \tag{1.3}$$

  where ρ is a constant; l = length of the wire and A = cross-section area of the wire. As the wires becomer thinner, A decreases and hence R increases. The delay for propagating a signal through a wire is given by the equation [HP07]:

$$\tau = R \times C \tag{1.4}$$

  We see that the delay is proportional to R and C. In order to reduce R and thereby the delay, according to equation 1.3, l needs to decrease. Therefore, the wire gets shorter in length. If the wire length decreases, the distance travelled by the signal on the wire also decreases, which means that in a single clock cycle only a certain number of gates are traversed. Figure 1.4 shows the percentage of a processor die that is reached with reducing feature sizes.

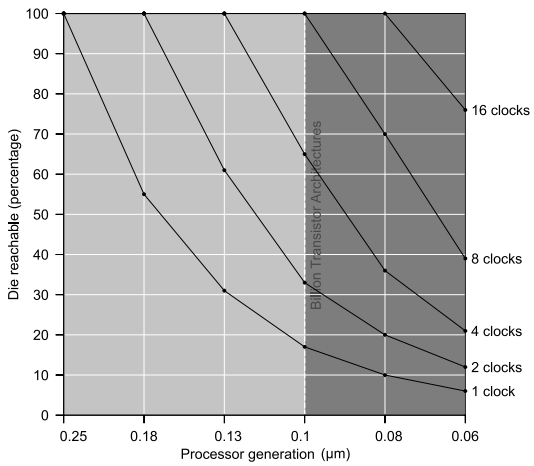

Figure 1.4: Reachability of the signal with shrinking process technologies [Mat97]

- Saturating Instruction-Level Parallelism (ILP)

  The main objective of any newly developed processor system is to enhance system performance. On uniprocessors executing sequential applications, this aim is achieved by issuing independent instructions simultaneously within the processor pipeline and provide for multiple execution units to process these instructions. Such processors are known as out-of-order (OOO) processors. If the *Decode* stage within the processor pipeline, resolves instruction dependencies then it is known as a superscalar processor. If the compiler identifies a window of instructions that could be executed in parallel then such processors are known as Very Large Instruction Word (VLIW). OOO processors became popular around the mid 90's with Intel's P6 microarchitecture [Int98]. However the number of comparisons to be made in order to determine independent instructions increases as the square of the window size [HP07], assuming all instructions were register based. In reality, the storage space required for a window, branch prediction mechanism, limited number of functional units(integer and floating point *Execute* units) and registers cause the actual number of instructions that are issued in parallel to be less than the window size [HP07]. One of the most powerful OOO processor is the Dual Core IBM Power 5 that has an issue width of 8 instructions from a single thread [SKT+05]. However, studies in the past have shown that the average parallelism in terms of number of instructions that can be executed simultaneously is 7 [Wal90].

In addition to the above mentioned issues, design productivity is of great concern to the IC[2] industry. Figure 1.5 shows the gap between productivity and complexity as number of transistors increase.

1.2 Why CMPs?

The main goal of a CMP system is to use multiple simple scalar or moderately complex superscalar cores that work in tandem to improve or better the throughput of a complex superscalar uniprocessor core. It addresses the power consumption problem, by using relatively lower frequency based processor cores in

[2]Integrated Circuit

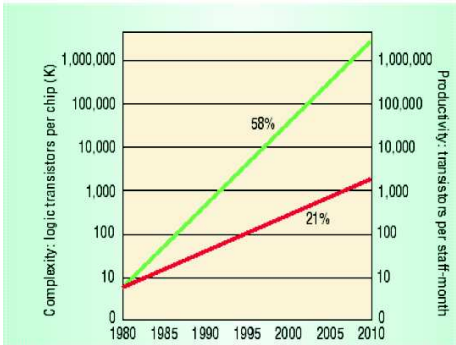

Figure 1.5: Productivity vs. Increasing transistor count [Bha05]

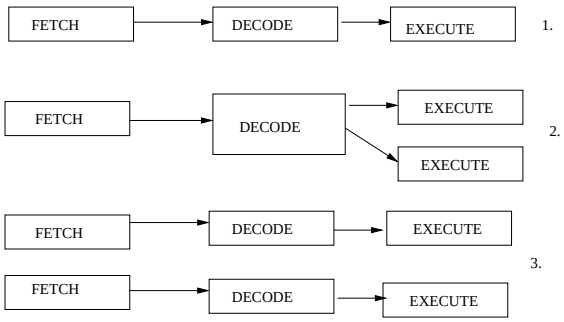

Figure 1.6: Different types of parallelism: 1.Single-Issue 2.ILP 3.TLP [Wri01]

constrast to a single high speed uniprocessor core. It alleviates the chip area reachability problem by partioning its architecture, thereby using smaller length wires to interconnect logic within and between cores and curtailing the wire delay. Given that the cores on a CMP are simple, they do not occupy large area on chip giving more room for incorporating a larger portion of the memory hierarchy on chip. Most CMPs offer L2 and L3 caches on chip thereby hiding memory latency as the bandwidth offered on chip is higher than off-chip. The saturating limits within ILP and the design complexity associated in hardware has led application and system developers to develop different means of extracting parallelism. One such instance of parallelism is Thread Level Parallelism (TLP), as shown in Figure 1.6. CMP systems can use TLP to their benefit simply because there are several processors that are potentially available to execute these threads in parallel.

1.3 Design Challenges in CMPs

The transition from uniprocessor to CMP systems has already been made for both high end server as well as desktop machines, as seen in Sun's Niagara, and Intel and AMD's quad and dual core systems. However, there are some key design challenges for CMPs, mainly, the cache coherency issue and architecture scalability. A major difference between uniprocessor and multiprocessor systems is that uniprocessors do not need to address the issue of cache coherency. However, CMP systems are in essence, architecturally, a smaller version of cache coherent multiprocessor systems. Since CMPs are targeted towards applications that will most likely use the shared memory programming paradigm (a detailed explanation of cache coherency in shared memory systems is given in Chapter 2, Section 2.2), there is a need for a cache coherency protocol to be implemented over such systems. Apart from handling cache coherency, CMP systems should be scalable (in terms of number of processor cores on a single chip) in order to achieve faster execution speeds for parallel applications. It is a known fact that bus based CMP systems do not scale [Wri01][Hor07]. Therefore, alternative switch based interconnect topologies (again, a detailed explanation of on-chip switched networks is given in Chapter 2, Section 2.1), are becoming popular with many CMP systems. One of the most popular switched interconnect is the mesh topology that is being used to integrate multiple processor cores (tiles) on a single chip [DT01][WGHea07]. However, the bigger challenge is to maintain cache coherency on such unordered networks and at the same time achieve application scalabilty.

1.4 Research Aim

The previous sections have listed some of the core problems with increasing chip densities that are addressed by CMP systems. They also highlight the design challenges, namely, the scalability of cache coherent CMPs. Therefore, this thesis explores the scalability of cache coherent tile (interconnection of processor and memory tiles) based CMP systems (tiled CMPs) using multiple networks. It uses an extended version of a formerly developed cycle accurate simulator [Hor07] to model multiple networks and a combination of snoop and directory based cache coherency.

1.5 Contributions

The contributions of the thesis are as follows:

- It uses a novel combination of homogeneous and heterogenous networks to evaluate the scalability of a cache coherent tiled CMP system.
- It adapts the bus based snoop protocol and the limited directory based protocol to perform on multiple networks.
- It optimizes the limited directory based cache coherency protocol for write invalidates to widely shared data by eliminating the need for acknowledgement messages.

1.6 Thesis Layout

The structure of the thesis is as follows:

Chapter 2 provides an overview of the basic components of tiled CMPs, namely on-chip interconnection networks and cache coherency protocol. It also includes a review of related work for Tiled CMPs in the literature.

Chapter 3 presents the JAMAICA bus based CMP. It provides details of the processor architecture, the bus interface, the token ring network and the MOESI cache coherency protocol used. It also gives details of the simulator modelling the architecture.

Chapter 4 provides a detailed description of the heterogenenous tiled architecture - mesh and bus. It includes details of the architecture of the tile and the mesh network. It explains the cache coherency protocol, the design of the dedicated hardware based queuing lock unit and the changes made to the JAMAICA simulator in order to implement this design.

Chapter 5 provides a detailed description of the homogenous tiled architecture - dual mesh. The main architectural differences between the dual mesh and mesh and bus schemes are discussed. The modifications made to the cache coherency protocol, network deadlock problems encountered during simulations and the solutions provided for it, are presented.

Chapter 6 evaluate the performance of the mesh and bus and dual mesh architectures using multithreaded benchmarks. It compares the relative performance of the dual mesh over the mesh and bus and single bus based architectures.

Chapter 7 concludes the thesis based on the performance analysis of the two architectures and suggests future enhancements to the dual mesh scheme in order to improve its performance.

1.7 Publications

- HiPEAC Workshop on Interconnection Network Architectures: On-Chip, Multi-Chip January 2007, Ghent, Belgium: Presented an abstract on the design of a Tiled-Bus based CMP.
- ACACES 2006 2nd International Summer School, Italy, July 2006 Presented a poster A Lock Unit for the JAMAICA CMP
- Accepted paper in the Second Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG'09): A Dual Mesh Tiled CMP

Chapter 2

Background

This chapter provides an insight into network based cache coherent chip multiprocessor (CMP) systems. It explains the basic components for on-chip networks (OCNs) - topology, routing, flow control, switching; followed by a general discussion on cache coherency protocols as well as alternatives to it. It reviews the design of some Tiled CMPs cited in both industry and academia.

2.1 On Chip Networks

With the onset of embedded systems, wherein multiple intellectual property (IP) blocks on chip have to be connected in a manner which reduces the latency and power consumption - factors that are critical for the performance of the real-time system, the design of *OCNs* has generated great interest within the research community as well as industry. Motivation for networks on chip for homogeneous CMP systems stems from the fact that, as process technology shrinks, latency and power become important deciding factors in the performance of systems. With shrinking feature sizes and the associated wire delay problem (shorter wire lengths for signal reachability - (Chapter 1), long interconnects, such as buses and others that rely on global wiring, will be required to clock at a much smaller frequency compared to the processor, in order to maintain signal reachability throughout the chip and ensure the in order property of the network. This means that signal propagation delays on global wires grows logarithmically as opposed to that on local wires [itr05], as shown in Figure 2.1. Also, network topologies that rely on global wiring will dissipate more power than their local counterparts , because of repeaters added to the long wires to maintain signal integrity

[DT01]. OCNs on the other hand have several advantages over global interconnects. Firstly, depending on the topology, they require smaller length wires to interconnect components on chip, thereby reducing the power dissipation. They allow for structuring wiring resources, which simplifies layout on chip, increasing and sharing the bandwidth among components and hence enhancing scalability [DT01]. They can use local clocks, thereby reducing the area, power and clock skew problems associated with global clock trees [OS02]. Given these advantages, OCNs have been favoured over traditional bus based interconnects in many CMP systems.

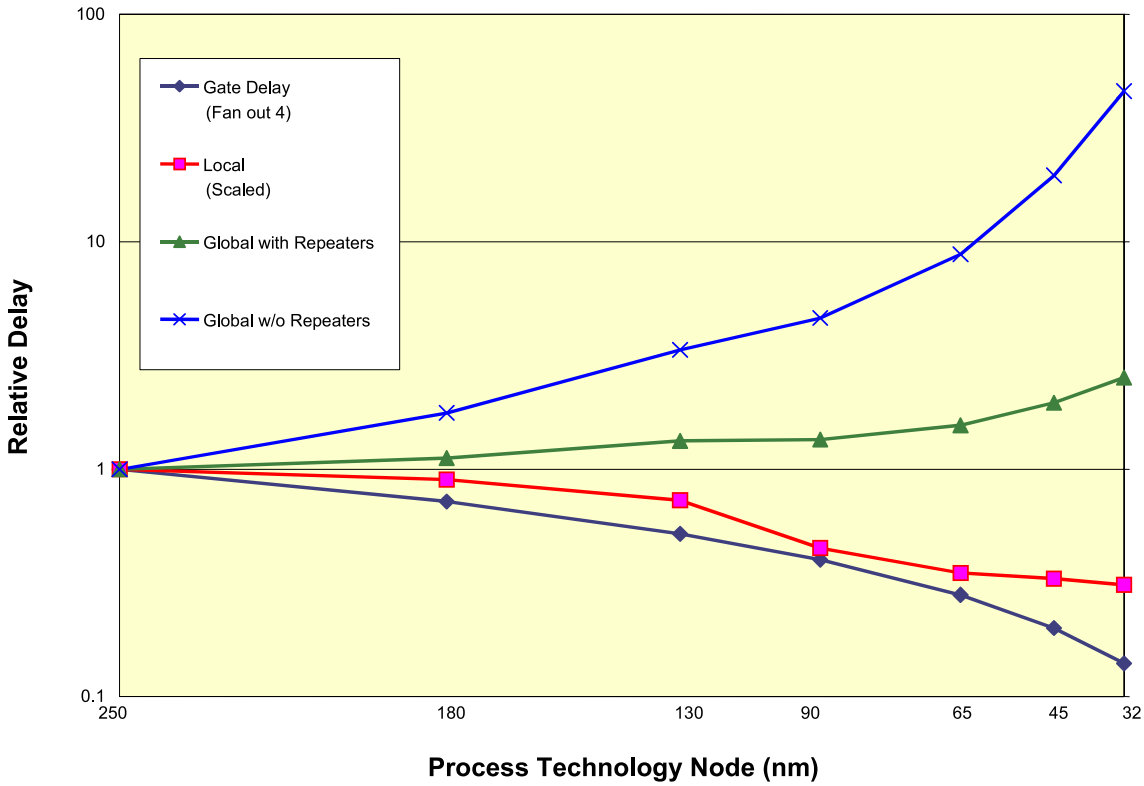

Figure 2.1: Wire Delays Projections On Chip for different metal layers [itr05]

2.1.1 Network Topologies

Network topologies define the various paths that exist between a communicating source and its destination. Multiprocessor and multicomputer systems use a variety of network topologies, such as hierarchical buses, trees, ring, meshes, hypercubes, crossbar [DYL03] [Sta97] [CSG99]. A broad classification of various types of networks is shown in Figure 2.2. All these topologies can be used both

on and off-chip. Off-chip networks are constrained by the pin bandwidth. In this discussion we concentrate on OCNs. OCNs normally divide a message into packets and transmit them as information carriers between communicating nodes. This mode of transportation is known as packet switching, wherein a packet enters a network and traverses through it without knowing the path that it is going to take to reach its destination. Another mode of transportation is circuit switching, that which is used by traditional land line phone networks. In circuit switching a route from source to destination is reserved in advance before sending the packet. Although, this scheme guarantees packet delivery within a certain period of time, it cannot compete with packet switching in terms of adaptability, in an ever changing network traffic scenario. Also, the idea of reserving a network resource (buffers or channel/links) way ahead in time can lead to inefficient utilization of that resource. Some of the commonly used OCN topologies are shown in Figure 2.3.

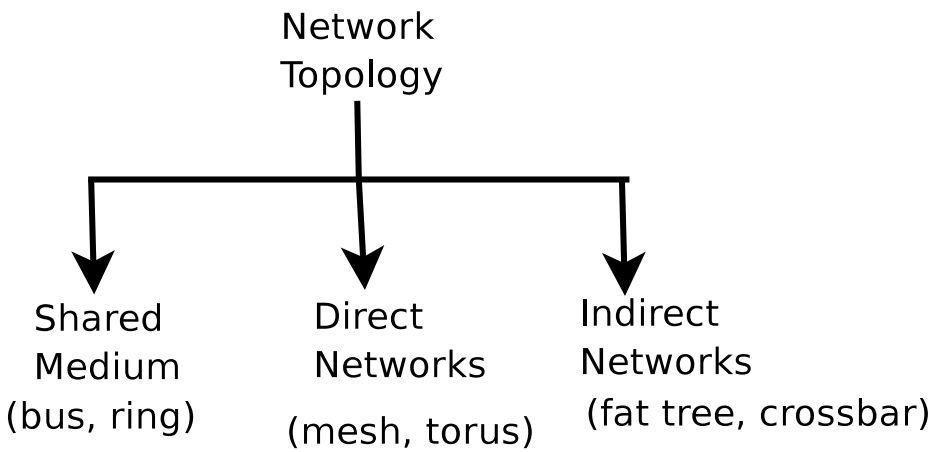

Figure 2.2: Classification of Network Topologies [DYL03]

2.1.1.1 Direct Networks

Direct networks, as their name implies, connect to neighbouring devices directly, while remote devices are accessed using several hops. Each node within a direct network uses a router to interconnect with other neighbouring nodes. The routers per node perform switching and arbitration. They are also known as orthogonal networks, i.e. there is at least one link in each dimension of the network, for every node [DYL03]. This also makes the routing protocol easier to implement.

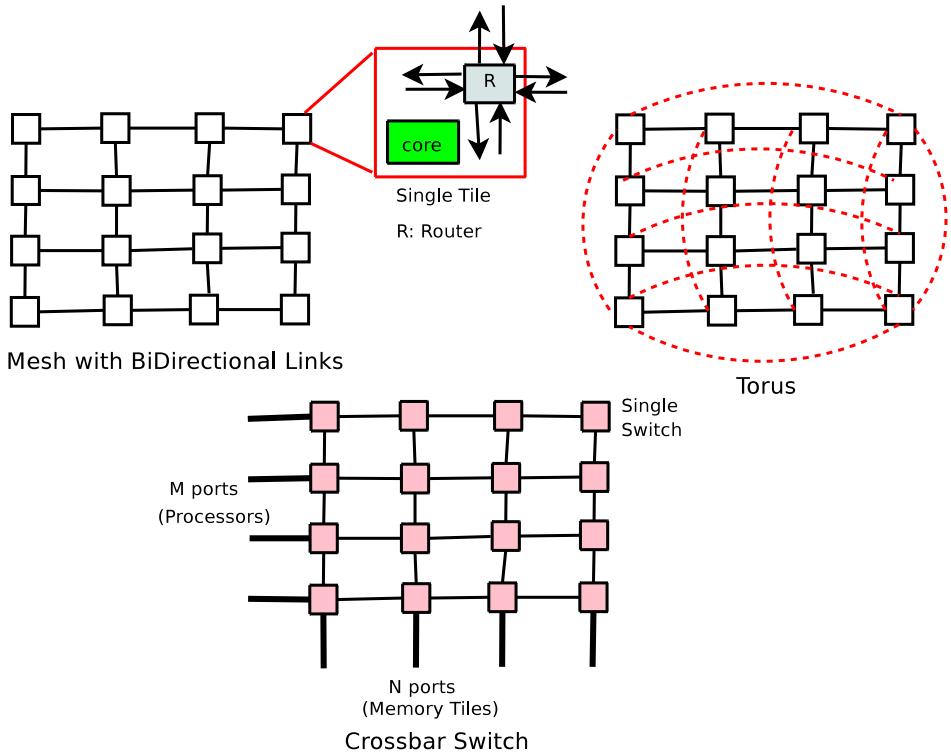

Figure 2.3: Various Types of OCNs [ODH+07] [DYL03]

Direct networks are the preferred topology given the on-chip area constraints and wiring availability [DT01][GK08]. Examples include meshes, tori, hypercubes, etc. Equation 2.1 shows the average latency that a packet encounters when travelling through a direct network from source s to destination d [BD06][KBD07]:

$$L(s,d) = T_h + T_s + T_w \tag{2.1}$$

where, $T_h = Ht_r$, is the header delay given by H (the hop count) and t_r the router traversal delay; $T_s = L/W$ is the serialization delay, with $L =$ the packet length and $W =$ the channel width (the number of wires that connects two adjacent communicating nodes; T_w is the time taken to send the packet over the wires.

2.1.1.2 Indirect Networks

Indirect networks use multiple switches connected in a manner so as to form a centralized switch for interconnecting multiple communicating nodes [DYL03].

The centralized switch is responsible for packet routing and arbitration. Examples include crossbars and multistage interconnection networks (MIN) (such as butterfly, omega, clos, fat tree). The number of switches required for crossbars is N^2, and that for MINs is $(N/k)(log_k N)$, where N is the number of communicating input and output ports and k is the number of input/output ports per switch (switch degree). In order to reduce the number of switches, MINs use bristling, a technique wherein multiple ports are connected to a single switch. This is used in fat tree networks [HP07].

2.1.1.3 Shared Medium Interconnects

Shared medium interconnects, such as arrays, buses and rings are characterized by a single link that is shared by all components. Buses, for example also provide for total order. Although they are simpler to implement, these networks are not scalable and require arbitration in order to access the network [DYL03].

2.1.1.4 Commonly Used OCNs

Mesh networks are by far the most popular [WGHea07] [GKS+07], mainly because of the short channel distances between neighbouring nodes resulting in one cycle hop delay and thereby addressing the wire delay problem for deep submicron technologies. However, as the number of nodes increases, the router radix (connectivity of the router) is increased to reduce the hop latency. Equation 2.2 gives the power dissipated in sending a flit (basic flow control unit, normally the minimum amount of information that can be transmitted over the physical link [DYL03]) per hop [DT01]:

$$P_{hop} = P_{iocontrl} + P_{wire} \tag{2.2}$$

where, $P_{iocontrl}$ is the power dissipated in the input and output controller of the transmitting node and total power dissipated because of this factor is dependent on the number of hops; P_{wire} is the power dissipated as the flit traverses the wire and the total power dissipated because of this factor is dependent on the length of the wire.

In topologies such as torus, the power dissipation increases mainly because of the varying wire lengths to interconnect different nodes [BD06] [DT01]. Therefore, nowadays a more popular topology is the concentrated mesh. Concentrated

meshes increase the radix of a router using the concept of bristling, wherein each router node services four processing nodes. The router itself is then connected to four such routers using the mesh network links and express links as shown in Figure 2.4. Express links connect a router situated on the edge of the mesh network to other alternating edge routers. This layout improves area efficiency, the hop latency (more connectivity compared to a torus) and does not require complex wiring layout (no large varying wire lengths), thereby reducing power dissipation. However, the disadvantage of this scheme is that there is an increase in design complexity of the router resulting in larger router delay. Recent papers have proposed high radix router based concentrated mesh, also known as the flattened butterfly topology, to reduce the routing latency and power dissipation [KBD07]. Each router within a row of the 2-D mesh network connects to all other routers in that row as well as all routers within the same column. This type of network incorporates the high radix feature of indirect networks on a symmetric direct network. Although this scheme uses varying wire lengths similar to the torus network, it reduces the hop count (lower than the concentrated mesh) and therefore reduces the total power dissipated in the input and output controllers, which compensates for the increase in power due to wiring complexity.

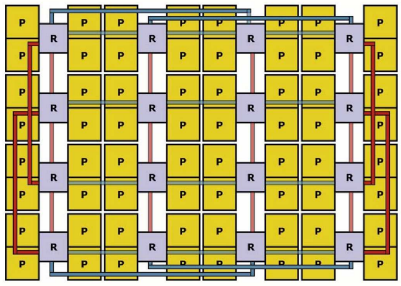

Figure 2.4: Concentrated Mesh [BD06]

2.1.2 Routing Protocols

A routing algorithm determines the path between source and destination and gives an indication of how effectively the bandwidth on this path is utilized [Pin06] [DYL03]. The routing algorithm also determines how the load is distributed within the network [HP07]. The main purpose of a routing algorithm is to guarantee packet delivery to its destination. It does this by restricting certain

paths within the network. Depending on the number of paths restricted routing algorithms are classified as deterministic or adaptive. Subcategories within each class of routing determine whether the route for the packet to be sent is computed in advance and set within the header of the packet (source routing), or if the route is computed on the fly, given the information about the source and destination (distributed routing) [DYL03]. Source routing is simpler to implement, but requires prior knowledge of traffic conditions before the route is computed and is almost obsolete in OCN [GLD06]. Distributed routing is harder to implement requires fixed hardware tables (not scalable) or finite state machines for determining the link on which the packet needs to be sent [DYL03]. The reason distributed routing is preferred over source routing is because it is more adaptive to changing network conditions.

2.1.2.1 Deterministic Routing

In this routing scheme, the path from source to destination is always fixed. The most popular routing algorithm in this category is the Dimension Order Routing Protocol (DOR) [SB77], which is used for direct networks, such as meshes, tori and hypercubes. In this routing protocol, the packet changes its dimension only when the link or channel needed to the send the packet is not on the same dimension [HP07]. Figure 2.6 shows the DOR algorithm and Figure 2.5 shows a mesh network using the DOR algorithm to route packets [DYL03]. DOR is also known as a minimal routing algorithm because it selects the shortest path from source to destination and always ensures that the packet is progressing towards the destination.

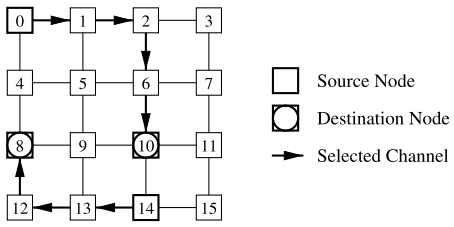

Figure 2.5: Mesh Network Using DOR
[DYL03]

Algorithm: XY Routing for 2-D Meshes
Inputs: Coordinates of current node ($Xcurrent, Ycurrent$)
and destination node ($Xdest, Ydest$)
Output: Selected output *Channel*
Procedure:

```
Xoffset := Xdest − Xcurrent;
Yoffset := Ydest − Ycurrent;
if Xoffset < 0 then
    Channel := X−;
endif
if Xoffset > 0 then
    Channel := X+;
endif
if Xoffset = 0 and Yoffset < 0 then
    Channel := Y−;
endif
if Xoffset = 0 and Yoffset > 0 then
    Channel := Y+;
endif
if Xoffset = 0 and Yoffset = 0 then
    Channel := Internal;
endif
```

Figure 2.6: Dimension Order Routing Algorithm [DYL03]

2.1.2.2 Adaptive Routing

In this routing scheme, multiple paths exist between a source and destination. The path can change depending on the traffic level within the network [DYL03]. This type of routing algorithm provides better load balancing and is also fault tolerant [CSG99]. It is also know as non-minimal routing because the packet may get routed in a direction away from the destination in order to avoid or reduce traffic congestion [DYL03]. The hardware required to make these routing decisions is complex and hence could potentially increase the route calculation time. Depending on the number of channels that are available for routing, fully and partially adaptive routing algorithms exist [DYL03].

2.1.2.3 Topology Agnostic Routing

Routing algorithms that are topology dependent are not fault tolerant because failure of a single link within the network could lead to stalling the system [CSG99]. Therefore, protocols that do not make any assumptions about the underlying network, also known as topology agnostic algorithms, are proposed. One such algorithm is the $Up^* - Down^*$ routing algorithm [SBB+91]. In this routing algorithm, it is necessary for the node that is generating a packet to determine the topology of the network and create a tree based on the location of the source and destination. Further on, it assumes that nodes within a tree are interconnected using bi-directional links. Similar to tree routing, packets move up the tree towards the root and then down towards the destination. Restrictions to the routing protocol requires that a packet cannot re-traverse in a direction(up/down), once taken. Another algorithm in this category, also known as segment based routing [MFD+06], proposes to divide the network into segments and places route turn restrictions within each segment, allowing non-minimal routing within a segment.

2.1.3 Flow Control

Flow control is technique that is used to prevent a sender from sending packets at a rate that is faster than that the receiver can process [HP07]. It helps in controlling the traffic congestion level in the network [DYL03]. There are different flow control schemes, the simplest one involving performing a handshake protocol between the sender and receiver (request and ack scheme). Another scheme, Stop and Go, notifies a sender that the receiver's buffers are nearly full and the sender should stop sending packets. Credit based flow control allows a sender to keep track of the number of available buffers at the receiver end. The sender maintains a count (credit) of the number of buffers at the receiver and decrements a counter every time it sends a packet. The receiver on accepting the packet and freeing the buffer, sends a credit increment message to the sender [Pin06]. A more recent flow control technique known as Adaptive Bubble flow control [PIB+01] relies on ensuring at least two packet buffers free at the receiver before the packet is sent out of the sender.

2.1.4 Switching Technique

Switching techniques allow a packet to get forwarded through the network based on the buffer requirements. There are three main switching techniques used, Store and Forward (S&F), Virtual Cut-Through (VCT) and Wormhole [DYL03]. S&F switching technique buffers the complete packet before sending to the next node in the route direction. VCT allows for packet pipelining once the header packet is received at a node, i.e. the header is forwarded to the next node before the trailing packets arrive from the sending node. If the header packet is blocked due to congestion, then the complete message is buffered, similar to S&F [DYL03]. Wormhole applies switching to flits, in order to save buffer space. The packet is split into flits and they are transmitted through the network similar to that in VCT, except that packet does not buffer at flit blockage [DYL03]. The greater chance of message blockage in wormhole scheme requires the network to be associated with extra buffers per physical channel (virtual channels) and flow control techniques [DS87].

2.1.5 Deadlock Avoidance

For packet progression through a network, network resources (buffers, channels/links) must be available. Non-availability of a link or buffer that is blocked results in a packet waiting for the resource to be freed and hence leads to deadlock of the system. During deadlock, there is no progression of packets through the network. All networks are prone to deadlock because they do not contain infinite buffer resources [HP07] [HGR07]. A simple deadlock scheme is illustrated in Figure 2.7, where in all packets are destined for alternating nodes, with each node unable to send its packet because the successive node buffers are full. The cause of a deadlock is due to the formation of a cycle by the turns contained in a packet's route [DYL03]. This deadlock can be broken by adding extra buffers per physical channel. These buffers are also known as Virtual Channels [DS87]. Virtual channels are one of the most popular ways to avoid deadlock, especially for adaptive routing based networks. However they consume extra space and power on chip [HGR07].

While the routing algorithm is responsible for controlling packet flow through the network, one of the easiest ways to avoid deadlock is through using a deadlock free routing algorithm, e.g. DOR. DOR ensures that path taken by a packet from

source to destination never forms cycles. It does this by allowing the packet to perform only a single change in dimension (direction) when moving from source to destination. DOR is restrictive in the number of paths that a packet can take and therefore the turn model [GN92] was proposed to provide a minimal combination of *turns* that a packet can make without creating cycles in its route. The turn model does not require addition of VCs. The $Up^* - Down^*$ routing algorithm is a instance of the turn model and is a partially adaptive routing algorithm [DYL03].

Another proposal for avoiding deadlock is to use flow control. Credit based flow control is shown to be most effective in avoiding deadlock, by controlling the amount of traffic that enters the network [HGR07]. If deadlock is detected then packets are dropped to ensure forward progress in the network [DYL03]. Apart from deadlocks, networks are also prone to livelock. Livelock occurs when a packet travels through the network towards the destination but gets rerouted in a different direction every time because of blocked buffers [DYL03]. This situation normally occurs in routing protocols that allow non-minimal routing. Livelock occurrence probability is reduced by limiting the number of non-minimal routes taken.

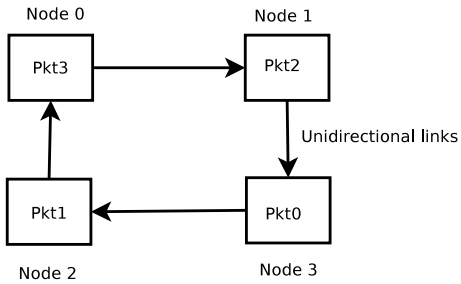

Figure 2.7: Classic Example of Deadlock
[DS87]

2.2 Cache Coherency Protocols

Studies in the past have shown that computation intensive applications, such as scientific workloads, databases and internet applications support inherent parallelism and can perform better if this parallelism feature is exploited by the underlying hardware [HP07] [CSG99]. The quest for higher speed in application executions as well as the growth in microprocessors fuelled by the enhancement

in VLSI technology, gave rise to multiprocessor systems [CSG99]. Multiprocessor systems, utilise multiple microprocessors to speed up the execution time of a single parallel application or can run several applications to produce increased performance. Unlike their uniprocessor counterpart, multiprocessor systems require an enhanced communication based programming model and this gives rise to two types of parallel programming paradigms, namely shared memory and message passing. Message passing models require explicit programmer defined messages to be exchanged among communicating processors [PTM96]. Also data distribution plays a key role in the performance of such systems. Programming such systems is non-trivial as each processor uses a separate address space, unlike uniprocessors that use a single address space. On the other hand, the shared memory programming model is simpler to program as it achieves communication through a shared single address space and allows for easy porting of parallel applications [PTM96]. Parallel programs that run on shared memory multiprocessors always need to ensure that the value of data being read is the last written [1] [HP07]. However, having multiple copies of the same data in different caches in multiprocessor systems poses problems in maintaining this constraint. Cache coherency protocols use hardware or software techniques in order to ensure this condition [HP07]. Of the hardware based cache coherency protocols, the simplest one is the snoop based protocol, wherein all caches snoop a bus or any ordered network to maintain the correct state of the cache line. Snoop based multiprocessor systems are also known as symmetric shared memory multiprocessor systems because the access time to the shared memory is uniform irrespective of the location of the processor. One of the popular cache coherency protocols is the non-inclusive based, MOESI. It is a combination of the Dragon (MOESI with write invalidate feature to Shared cache lines) and CRAC (with the Owned state transfer feature between L1 caches) cache coherency protocols [AB86]. The state diagram for the MOESI protocol is shown in Figure 2.8 and Table 2.1 explains the various cache states.

While cache coherency protocols provides data consistency for a single location by deciding on what value of shared data is read [HP03]. Memory consistency models on the other hand handles the data consistency for multiple locations [Ste05]. It decides when the value of the data can be read [HP03]. In short, memory consistency is a formal specification of how ordering of reads and writes

[1] Sequentially consistent memory model

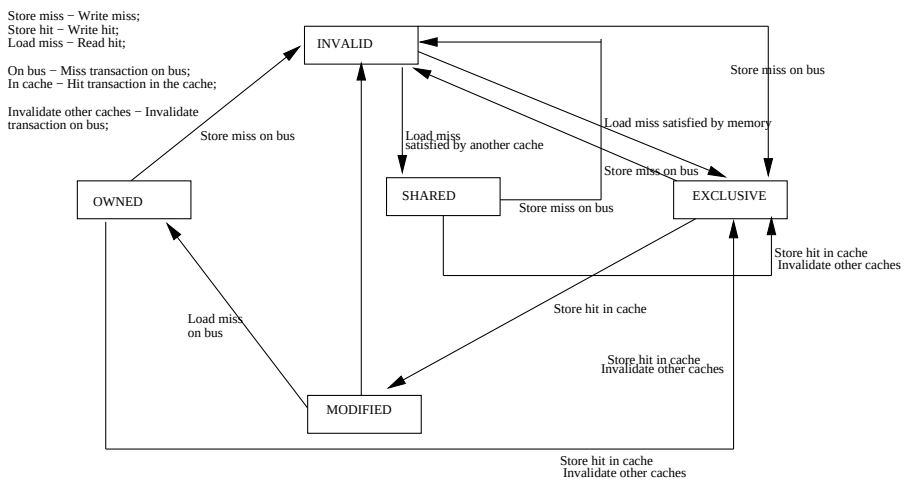

Figure 2.8: The MOESI state diagram [Sun03]

should take place within a program [Dig95]. All memory consistency models require programmers to devise mechanisms to ensure correctness of program execution [Dig95]. Memory consistency models for multiprocessor systems are classified broadly as either sequential or relaxed consistent. Sequentially consistency models require that program order is maintaind within a processor and that writes to a single location are seen in the same order by all processors (write serialization) [Lam79]. Although memory operations from different processors may be interleaved, they should commit in the same order as if the program was being executed sequentially on a single processor, i.e. a memory operation is atomic and completes in the same order that it was issued [Dig95].

Relaxed consistency models optimize on the commit ordering of reads and writes to different addresses within a processor as well as on different processors. Relaxed models are normally classified depending on the read and write ordering they relax, i.e. either read-after-write (RAW) or write-after-write (WAW) or read-after-read (RAR) and write-after-read (WAR) [Dig95]. All relaxed consistency models maintain the write serialization property [Dig95]. However, they require programmers to use special instructions known as fence instructions to overide program order and introduce synchronizing operations for accessing shared memory [Dig95]. Use of relaxed memory models allows for implementation of hardware and software optimizations on the application.

Cache State	State Description
M	Modified-Line is dirty and the latest copy of the data exists in only one L1 cache which is responsible for writeback to main memory.
O	Owned-Line is dirty and shared between multiple L1 caches. L2 is not inclusive. The L1 cache that contains the O cache line is responsible for a writeback to memory.
E	Exclusive-Line is clean and a copy of the data exists in only one L1 and L2 cache.
S	Shared-Line is clean and exists in one or more L1 caches and the L2.
I	Invalid-Cache line contains invalid data in L1.

Table 2.1: Cache Line State Description

2.2.1 Snoop Based Cache Coherency Protocol

In snoop based cache coherency protocols, broadcast is used as a means to maintain the state of a cache line. All caches snoop a bus or any ordered interconnect to determine the current state of the cache line. Bus based snoop cache coherency protocols have been used extensively in the past on most multiprocessor systems but suffer from lack of scalability, because a single interconnect (lower bandwidth) is used for broadcast traffic. Another disadvantage of snoop based system is the interference caused by processor and bus interfaces trying to access or snoop the cache tags at the same time instant, resulting in the reduced performance of the system [ASHH88]. A solution to this problem is to use duplicate tags [ARM06] [ASHH88]. But this increases the power and space overhead within the system. Duplicate tag based snoop systems also have the requirement of ensuring that any updates to duplicate tags and original cache tags is atomic in nature. In order to increase the snoop bandwidth, systems such as Sun's FirePlane servers use separate address and data networks, the former being ordered and the latter is unordered [Cha02]. Alternatively, in order to reduce the broadcast snoop traffic and power dissipation associated with snooping the cache tags, schemes such as course-grained coherence tracking and JETTY filters have been proposed and their working is described below [CSL+06] [BDH+99] [MMFC01]. Figure 2.9 shows a snoop based multiprocessor system.

- Course Grained Coherence [CSL+06]

  Course grained coherence relies on tracking snoop requests for large regions of memory within each processor node. Each snoop request is mapped to a memory region. Two different schemes are used in order to store the region information.

  In the first scheme, cache lines from multiple regions are hashed to the same entry. To implement this, two tables are used, the first one is a hash table into the line count of the cache lines from multiple regions - indicating a line from a region *may be* present in the cache, and the second a non-shared table that tracks regions (using the address) that are not shared. If a processor generates a request it takes the following actions:

  - It first checks its non-shared table to see if the request's region is not shared. If no information is available, then the request is broadcast, else the line is fetched from memory.
  - On a broadcast, other processors check their hash region table and if the line count is > 0 for the memory region corresponding to the broadcast address, then the non-shared table is checked.
  - If an entry is present in the non-shared table it is invalidated. Along with the hash and non-shared tables, the data and instruction caches are also checked and their state is updated on a cache hit. Data and state information is then provided to the requesting processor.
  - The requesting processor, on receiving data and state, will increment its own hash table and update its cache states. It sets an entry in the non-shared table, if the data was provided by memory.

  Although this method is space efficient and does not require evicting entries from the hash table, it does not provide high accuracy in predicting the existence of a line from a region.

  The second scheme, has higher space overhead and maintains a single table (region coherence array - RCA) whose entries provide for a one-to-one mapping between the cache line and the memory region. An

RCA entry also contains the number of lines from a region that are cached by the processor and the state of the region, i.e. if the region is shared or modified. RCA's are checked by both local cache requests and by snoop broadcasts. If an RCA entry matches a broadcast request, it updates its own state to shared as well as takes the necessary cache action. The limitation of this scheme is that on eviction of an RCA entry, the corresponding cache lines have to be invalidated or written back to main memory. The effect of this disadvantage is reduced by using a policy that requires evicting regions with small number of cached lines. The space overhead for an RCA configuration of 512B with same number of lines as the L1 cache is about 5.9% of a 1 MB L2 cache area. The scheme has been shown to reduce broadcasts by 55-97% on some of the benchmarks that were evaluated and an overall decrease in runtime execution by about 8.8% on average.

- JETTY Filters [MMFC01]

  JETTY filters (per processor) are used to snoop requests over the bus in order to determine if it is worthwhile accessing the L1 or L2 cache for the same request. It derives its inspiration from the fact that most snoop requests miss in L1 or L2 caches and hence cause wastage of power when accessing their tags. For efficient working of the JETTY filter it is imperative that the structure of the filter is small enough so as to not waste space as well as energy in searching the filter itself. Also, the filter should never mis-predict, i.e. a request for a cached line should always be allowed to snoop the L1 and L2 tags. Three types of JETTY filters are described. The first one keeps track of cache requests that were snooped and missed within the L1 and L2 of the processor. The second one keeps track of all cached lines and finally the third one is a hybrid version that is a combination of the first two methods. On average, around 41% power reduction was observed from the hybrid version compared to accessing the cache tags during every snoop request. Of all the three types of JETTY filters the hybrid version gave the best performance for power reduction. These filters act as duplicate cache tag arrays and help in reducing the latencies caused when both processor and snoop interfaces try to access the cache tags simultaneously.

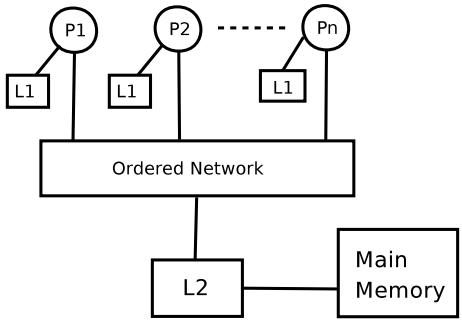

Figure 2.9: Snoop based Symmetric Shared Memory Multiprocessor [HP07]

2.2.2 Directory Based Cache Coherency Protocols

For highly scalable systems, the shared memory is physically distributed across various nodes and a switched network is used to interconnect them. These types of multiprocessors are known as distributed shared memory system and they use directory based cache coherency protocols [HP07]. Directories are separate memory modules that maintain the state of every cached line along with a vector that indicates the list of sharers for that cache line. They can be either centralized or distributed among nodes to keep track of the state of the shared data within the local memory in the node. Schemes with both static and dynamic directory entry placement have been described in the literature [PTM96]. A simple directory protocol is shown in figure 2.10 [HP07]. Nodes within this protocol scheme are classified as either local (where the request originates), home (where the directory for the requested memory word resides) or remote (other nodes that contain the same memory block in their caches). For a read miss, the request from the local node is forwarded to the home node, which in turn either supplies the data if it is in memory or forwards the request to the remote node where the actual data resides. The local node gets the data and if the remote node supplied the data, it updates its own cache and also the state of the directory entry in the home node to indicate the list of sharers for this address. For a write request, the local node sends invalidation messages or update messages to all the sharers and waits for an acknowledgement from each of them thus guaranteeing that the write was seen by all sharers. The list of sharers is fetched from the home node. The local node now has exclusive access to the cache line and the directory in the home node is updated with the new owner information. Ordering is ensured during read and write access by the directory that waits for an acknowledgement message from

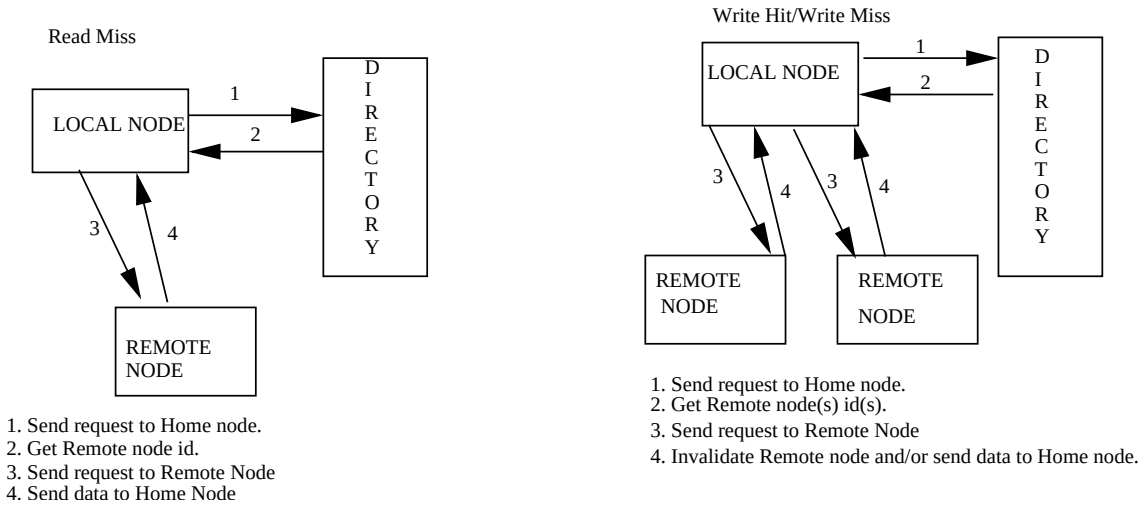

Figure 2.10: Simple Directory Based Scheme [CSG99]

the requestor and does not allow any new request for the same address to proceed until the original request for that address has completed.

2.2.2.1 Directory Design Alternatives

The design of the directory has significant impact on the latency introduced due to indirection during coherence transactions among processor nodes [PTM96]. The full map directory is not very efficient in terms of space complexity ($O(N^2)$) and because cache lines are normally shared by a small number of processors, several alternatives schemes have been proposed to reduce this overhead. Note that all schemes provide reduced performance compared to the full map directory as a trade off over smaller space overhead.

- SCI - Scalable Coherent Interface [PTM96][NS92]

  In this scheme caches are used as distributed directories and cache lines are logically joined together in the form of doubly linked lists. The home node directory maintains a pointer to the head of this list. Every read request is satisfied either by the home node or the head of the list. A requestor always add itself to the head of the list. On write requests, invalidation messages propagate down the list. In order to reduce the write latency on invalidations (O(N)), the list structure is replaced with a tree. This reduces the write latency to $O(\log_2 N)$. Although SCI reduces the space complexity at the directory in the home node, it increases the size of L1 caches.

- 2 bit Directory Scheme [AB84]

  In order to reduce the space overhead associated with each entry in the directory, only 2 bits are used per directory entry to encode the cache states - Absent, Present in one cache, Present in multiple caches and Modified in a cache. No owner information is stored and therefore on a read miss to modified data or write miss to shared or modified data, broadcast is used. This scheme increases the number of broadcasts and also the latency for read and write misses to lines present only in one processor node. This scheme is suitable for applications that have a small sharing set.

- Limited Pointer Schemes [ASHH88]

  These schemes store x pointers per directory entry where, x $\leq N$, and N = number of nodes in the system. Each of these pointers further contains $\log_2 N$ bits. They handle overflow by broadcasting or software intervention to store the additional sharer information in the memory associated with the directory's node. The penalty paid for reducing the space overhead is increased latency in the case of overflow of directory pointers. Limited directory schemes are classified depending on the number of pointers as well as the invalidation scheme used, i.e. broadcast or non-broadcast. These schemes can be generalized as Dir_iX, where i is the number of indices per directory to store the cache line sharers and X depends on whether broadcast (B) or non-broadcast (NB) is being used. They range from storing 0 pointers (broadcast scheme) to x pointers with broadcast as well as non-broadcast facility. A scheme with x pointers and non-broadcast implies that the number of sharers cannot exceed x. The directory size grows as $O(N \log_2 N)$ [SH91].

- Compressed Directory Entries [ASHH88] [MH94]

  Here the directory entries use compressed sharing codes to reduce the width of the sharers. In a coarse vector based directory, each entry uses x bits, wherein each one of the x bits represents a k processor group [GWM90]. This scheme allows for multicast based invalidations, instead of broadcasts. Alternatively, codes such as, tristate [ASHH88] or gray-tristate [MH94] are used to reduce the number of bits per

directory entry, by using $2 * (\log_2 N)$ bits, where N is the total number of nodes. If each 2 bit pair is either coded as both zeros or ones then they represent a single sharer. If any one 2 bit pair is coded with both zero and one, then the directory entry represents a superset of cache sharers for that particular pair of bits. If all 2 bit pairs are coded with both zeros and ones, i.e. 0 and 1 per 2 bit, then it means that all caches are sharers. This scheme could result in several unnecessary invalidations because of the superset case and is proved to be marginally better than the limited pointer based broadcast scheme [GWM90].

- Segment Based Directories [CP99]

  Segment directories use both full-map as well as a limited pointer scheme to reduce both the space and number of overflow instances from the limited pointer scheme. Here, each directory entry termed as SDK_k, consists of a segment of k bits, and a segment pointer of $\log_2(N/K)$ bits to determine the position of the k bit segment within a N bit full map directory entry. In effect, the k-bit segment acts as a full map directory for k processors within a system of N nodes, instead of acting as a pointer to a single node as in the limited directory scheme. Thus the segment directory can encode many more processors using a few more bits, compared to the limited directory scheme. Results have shown that this scheme reduces the directory overflow by up to 85% on some benchmarks.

- Interconnect Based Directories [RL97]

  In this scheme, the topology of the mesh network as well as the property of the wormhole routing algorithm allows all nodes within a path from source to destination to inspect a packet. The snooping of packet reduces the number of invalidation messages that need to be generated during broadcasts. Broadcast messages are generated only when the limited pointer based directory entry overflows. The limited directory entry pointers are then utilized to generate snoop paths over which the broadcast packets are sent out. The space overhead is $O(N^{3/2} * \log_2 N)$, which is greater than the limited pointer scheme, but lesser than a full-map directory.

- Two Level Directory Scheme [AGGD05]

  Here two levels of directories - a small uncompressed directory (full-map) and a larger compressed directory that covers all lines that are cached, are maintained. The larger directory entries contain compressed sharing codes (some with $O(\log_2 \log_2 N)$ as space overhead) based on the fact that nodes are arranged as a logical binary tree. The entries in the smaller uncompressed directory are cache lines that exhibit high temporal locality. This scheme provides lower space overhead compared to a full-map, course vector and tristate based directory schemes. Its provides better runtime performance than using Dir_0B and Dir_1B based limited directory used as the compression scheme within the larger directory, for most of the evaluated benchmarks. However, it does not outperform other compression schemes in terms of runtime execution, such as gray-tristate [MH94] and course vector [GWM90].

All the limited directory schemes discussed here reduce the area overhead in comparison with that of a full directory. They range from storing no pointers (2bit directory - smallest area overhead) to a hybrid version of using limited and full directory. In order to reduce the number of broadcasts and hence store more sharer information, limited directory schemes with compression, utilising the network topology (more area overhead) and dividing the network into segments, is discussed. It is seen that smaller the directory entry gets, the larger the write invalidation latency overhead (broadcast) it suffers from. This in turn affects the run-time performance of the application. In this thesis, we have used a Dir_1B limited directory scheme, however the broadcast invalidation overhead is reduced by using a separate ordered network and thereby avoiding the need to generate acknowledgement messages.

2.3 Alternative to Snoop and Directory Protocols

2.3.1 Token Coherence

While snoop based and directory cache coherency protocols dominate shared memory systems, another protocol known as *Token coherence* has been proposed

to work on unordered interconnects, without incurring the space overhead of directory protocols or the indirection latency that occurs due to sending all requests to the directory [MHW03]. The coherency scheme uses an optimized performance protocol in the absence of races, such as using prediction of cache line owners. The protocol assumes a correctness substrate that enforces the following rules:

- Each memory block is associated with N tokens, where N is equal to the number of processors in the system. One of the N tokens is an owner token.
- When performing a write to shared data, the writing processor requests all the tokens associated with that shared memory location from all the sharer processors. If a processor has just one token and if it supplies that token on a token request, it invalidates the line in its cache. Only after receiving all the tokens associated with a shared memory location can the processor write to the location or cache line. The idea behind getting all tokens is to ensure that no other processor is attempting an operation (read or write) on the same cache line.
- On a read miss, a processor needs just one token and gets it by broadcasting a request to all the processors for a token and valid data. A processor with the owner token should supply the data along with a token.

Write to a cache line not present in the cache, i.e a write miss, works in a similar way to reading the data in a Shared(S) mode and then writing to it. In this system, a cache line need not store the cache states, however a cache line with $0 \leq t \leq N$ tokens, represents the S state. If a cache line contains all the N tokens, it is in the Modified (M) or Exclusive (E) state. A processor with no tokens is in the Invalid(I) state. When a processor detects starvation of its request, it issues a persistent request that is handled with priority by all other processors.

This protocol uses broadcast to issue transient and persistent requests and this increases the traffic within the network. Simulation results have also shown that token coherence does not perform better than a snoop based protocol on ordered networks, because of the overhead of gathering tokens to perform a transaction, but can outperform directory based protocols. It also adds overhead to L1 caches by storing token count. Recent papers have utilised token coherence for designing

fault tolerant cache coherency protocols as well as using priority instead of persistent requests in the case of starvation [CRD07][FPGAD07]. Token coherence has also been used as an inter-chip cache coherency protocol [MBH+05].

2.3.2 Multicast Snooping

This scheme uses a combination of snoop and directory protocols [BDH+99]. By multicasting (prediction based) it allows a selected group of processors to snoop a request. For this purpose, it requires a multicast address network as well as a multicast group predictor. It switches to broadcast, if no multicast prediction is made or the multicast mask is wrong even after a few retries. The multicast address network has to be ordered and hence eliminates the need for acknowledgement (ack) messages during invalidation to shared data. All requests that are multicast are also sent to a directory to ensure correct prediction of the multicast mask. Any request, read or write that does not contain the previous owner in its multicast mask receives a negative acknowledgement (nack) from the directory. Lines within the cache can assume shared, owned (dirty shared) and modified state. A distributed directory scheme is assumed, with each processor associated with some physical portion of the main memory. For a write, if the multicast mask contains the previous owner as well as the list of all sharers then the write is successful and the line is fetched in modified state. If the mask includes the previous owner but not all the sharers (or has a partial list of sharers), the line is fetched in the owned state with newer mask information from the directory. Also, all requests that arrive at a processor for a pending block, i.e. the processor has itself requested for the same block, will be negatively acknowledged (nacked). The advantage of this scheme is that it reduces the number of broadcasts (if prediction is correct or partially correct - achieved about 73 - 95% accuracy) on the address network by performing multicasts to 2–6 destinations. and does not incur the indirection delay (directory forwarding the request) that is associated with directory protocols especially when a request is made to modified lines and the prediction mask is correct. This scheme requires a logically ordered address network as well as a directory that maintains the complete list of all sharers. Requests have to wait for an ack or nack from the directory before they can retry or accept the data. Therefore the overall system performance might not surpass a directory protocol scheme.

2.3.3 Bandwidth Adaptive Snooping

This CMP system also uses a combination of snoop and directory protocols [MSHW02]. It is primarily motivated by the idea that if a large bandwidth is available, then snooping protocols perform better, while in constrained bandwidth situations, directory protocols are preferable. It uses a saturating counter to determine if the link utilization is above or below a particular threshold. A policy counter is used to average the link utilization and determine the fraction of requests that need to be broadcast or sent to the directory controller. The cache coherence protocol used in this scheme is MOSI. The snoop protocol assumes an ordered address network and an unordered data network. Both these networks are implemented as separate virtual channels. A multicast snoop based full-directory protocol [BDH$^+$99] is implemented on the two virtual networks. All requests that are generated are sent from the directory controller back to the requestor. This is to ensure that the requestor can determine where its request is positioned in the total order. The ordered address network prevents the need for explicit acknowledgement messages. Results indicate that for small bandwidth networks, the directory protocol performs better that snooping or the bandwidth adaptive protocol. As the bandwidth increases the bandwidth adaptive protocol outperforms both snoop and directories. For large bandwidth networks, the bandwidth adaptive protocol imitates the snoop protocol behaviour and hence outperforms directories. While this approach seems promising for varying workload characteristics, it is a complicated protocol to design and verify, given the increased number of cache state transition in order to maintain cache coherency and the constant monitoring of the utilization of the network. The approach taken in this thesis is also similar to that described in this system, with the distinguishing features such as, no attempt to determine network utilization (assuming that the bandwidth available on the on-chip network is large), use of area conserving limited directories and no requirement for an ordered address network.

2.4 Tile Based CMPs

Nowadays, the abundant on-chip wire and transistor resources [DT01] have resulted in CMP systems that implement processing logic in a tile fashion and interconnect them using some symmetric network for simplification of the layout on chip. Tile based CMPs, as they are popularly known, can be broadly classified

based on the programming paradigm adopted within the system - control flow and data flow model. In control flow based systems instructions that are executed in some order control the flow of data. In dataflow systems, the availability of data drives the execution of the program. Most tile based CMPs use a distributed directory protocol for cache coherence, which means that each tile is designated as a home node for a certain range of physical addresses. In traditional multiprocessor systems the main memory is physically distributed across multiple nodes and has a directory associated with it. However, due to space constraints for on-chip tiled CMPs, the main memory is off-chip and the directory is implemented as either separate directory caches [BKT07] within each processing node (for private L2 caches at the node), or is a part of the tag array within a shared L2 cache node [BGC+07]. This section reviews some of the popular cache coherent control flow and dataflow driven tile based CMP architectures, studied in academia as well as developed by industry.

2.4.1 Tilera

Tilera (Figure 2.11) is a 64 tile architecture that contains 5 mesh networks (iMesh), utilizing the abundant on-chip wires for enhanced communication in embedded applications [WGHea07]. It is an extension of the RAW tiled architecture [TKMea02]. It provides shared memory communication as well as dataflow based computation using an API[2], *iLib*, which allows for customized data placement and routing on the underlying network without the intervention of the operating system. Each tile contains a 3-way VLIW[3] pipelined processor and with private L1 and L2 caches, a DMA[4] engine and support for interrupts, protection and virtual memory. Every tile is capable of booting an OS and running C or C++ programs, using an in-house compiler [Til]. Each network within the system has a channel width of 32 bits. Out of the five mesh networks, four are dynamic networks - User Dynamic Network (UDN), Memory Dynamic Network (MDN), I/O dynamic network (IDN) and Tile Dynamic Network (TDN); and one static network (STN). Communication over dynamic networks is through packets that use a wormhole based dimension order routing protocol to provide ordering between any two communicating nodes. The functionality of each dynamic

[2]Application Programming Interface
[3]Very Large Instruction Word
[4]Direct Memory Access

network is specified below:

- UDN

  This network is configured using the iLib API and performs low latency scalar operations between executing threads on different tiles. Flow control is ensured using software generated acknowledgment messages. Nodes communicate with each other using registers that are network mapped. The iLib API allows for both channel and message passing based communication. Channel based communication can be further classified as raw or buffered, depending on the buffer requirements at the receiver. In order to support these different communications, each tile is provided with a fast hardware demultiplexing logic to route packets into several hardware queues that are configurable to accept certain types of messages. The UDN network uses 5 hardware demux queues.

- IDN

  This network is used for communication between tiles and off-chip IO devices. Flow control is implemented using pre–allocated buffers before sending data on the network. The IDN contains 3 hardware demux queues.

- MDN

  Used when a shared memory communication mode is implemented within the application. It is used for fetching data into tile caches from off-chip DRAM[5]. The cache coherency protocol allows for caching of read-only data. If a memory address is marked by software as local, the data is fetched from the tile's local cache or DRAM. A Write-through policy is used for writes. Locks and synchronization are implemented in software. Applications that access the same data using different networks require a fence instruction inserted in software to synchronize the network traffic. The DRAM controller contains a pre-allocated buffer for each node, thereby ensuring that it never congests or deadlocks the MDN.

[5]Dynamic Random Access Memory

- TDN

  Requests for memory addresses that are marked as remote are sent on the TDN network. Remote data is not cached in any other tile, except the home node tile. The forward progress of the packets sent on it depend on the messages sent over the MDN, which is guaranteed not to deadlock. Therefore, it only implements flow control between adjacent communicating nodes and not end to end flow control.

Static networks on the contrary, do not use packets, but allow users to configure them so that they behave as a circuit switched network and send and receive data streams on the set-up routes. A memory protection scheme known as Multicore Hardwall, is used to block links on UDN, IDN and STN networks that are user configurable and hence support multiple networks or different execution environments on a single chip.

Tilera requires extensive programmer intervention - the programmer should be aware of the underlying architecture to achieve significant speed ups for parallel applications. This constraint is in contrast with the simple programming model that other shared memory systems require. Speed ups are achieved up to 96% improvement in execution time (for 64 tiles) compared to a shared memory communication based version of certain micro benchmarks, such as dot product and 2D frequency transform.

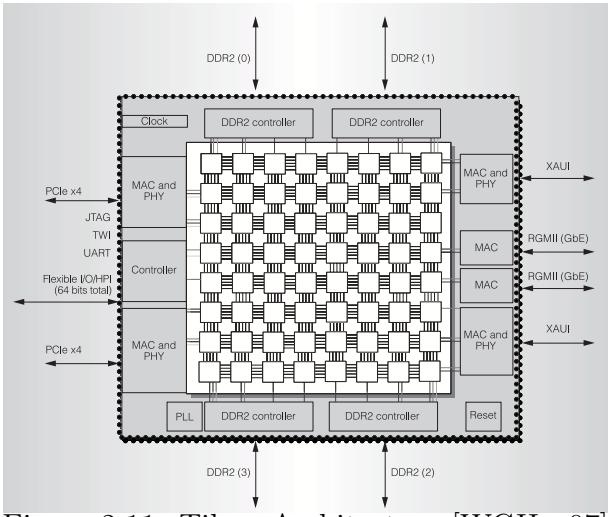

Figure 2.11: Tilera Architecture [WGHea07]

2.4.2 TRIPS

The Tera-Ops, Reliable, Intelligently adaptive Processing System (TRIPS) architecture consists of two processor cores and two networks - On-Chip Network for memory (OCN) and OPerand Network (OPN) [GKS+07]. The OCN is a 4X10 wormhole routed, credit based flow control mesh network that connects processors to 16 L2 cache tiles as well as IO (SDRAM, DMA, chip-to-chip and external bus controller) and network interface tiles. The channel width of each link within the OCN is 138 bits. The network interface tiles contain a network address translation table that maps memory addresses to destination network addresses. This gives flexibility in placing data in any L2 memory tile. Each tile in the OCN contains a router and employs virtual channels in order to avoid deadlock. The banked L2 tiles can be configured as either shared static Non-Uniform Cache Access (NUCA) [GCM+06] L2, or private NUCA L2 for each processor or used as a scratch pad memory. Each processor tile contains instruction and data cache tiles, register tiles, global tiles and execution tiles, all interconnected using the OPN. Similar to Tilera, it supports Instruction Level Parallelism (ILP) by using multiple execution units to support the individual instructions within a block, Thread Level Parallelism (TLP) by supporting execution of multiple hyperblocks and Data Level Parallelism (DLP) using the OPN. The compiler creates basic blocks of code that have no loops, have a single entry point and possibly multiple exit points. These blocks of code are then combined to produce an hyper block [MLC+92], which is then scheduled on the execution units. The instructions communicate with other instructions using the OPN rather than a register file. The instruction format contains the target instruction to whom the data needs to be passed. The global tile acts like a control unit that monitors the fetching, execution and committing of instructions. In total the number of tiles on the chip equals 109. Results show that the architecture performs within 28% compared to a system that has no network contention. It uses a data flow based computation model in contrast to a control flow based system and hence requires extensive compiler support.

2.4.3 OS Based Coherence

This scheme uses OS intervention instead of a hardware based cache coherence protocol for shared memory applications on tiled CMPs [FC08]. It proposes a

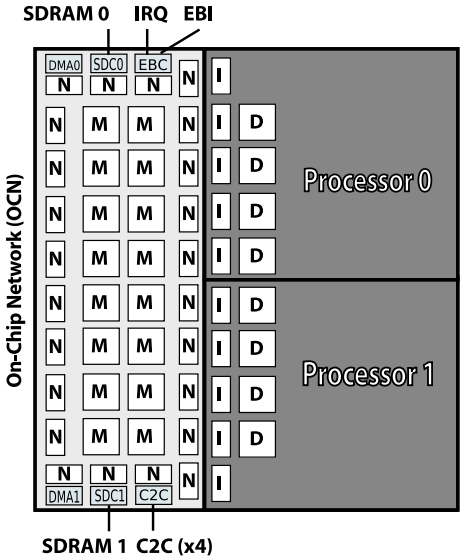

Figure 2.12: TRIPS Tile Architecture [GCM+06]

single logical cache and requires the OS to map memory pages to physical tiles. The default mapping is based on a first touch policy. Each tile on the CMP uses a hardware based structure similar to the TLB[6], also known as MAP, that caches the virtual address to physical tile mapping. The virtual to physical address translation is performed at the home tile. To allow for pages to migrate between tiles (owner migration), the MAPs are invalidated in all tiles. Before allowing migration of pages, it is necessary to ensure that all modified data is written back to main memory. No caching of shared data (shared cache line) is performed, however sharing of a page is permissable. On the first write within a page, the OS marks the page as Modified and tracks the writer tile. It ensures that any subsequent first time reads to the same page contain a MAP entry that points to the owner tile. All subsequent writes to the same page after the first write will require writing to the cache line in the owner tile. Tiles that had performed a read previous to the Ist write continue to use the local mapping and continue reading from the Ist touch tile. However, before a lock access to shared data, all MAP entries for shared pages are invalidated, a cache flush is performed and the owner tile writes the modified data to main memory. Results do show that this scheme performs within 16% on average compared to a full map directory scheme.

[6]Translation Look Aside Buffer

2.4.4 Priority Based Cache Coherent NoC

This tiled CMP architecture relies on multiple optimizations to the network router architecture, packet switching technique to reduce the delays (queuing and network traversal) for request and responses messages [BGC+07][BCGK04]. This architecture presents a NUCA based cache coherent system that uses 8 processors and a 64 bank, 16MB shared L2 cache. L2 cache banks are connected using a mesh network that uses credit based flow control for a wormhole based static routing protocol to preserve ordering of messages. Processor tiles are connected to the edges of the mesh network. The cache coherency protocol used is the MESI based directory protocol. It optimizes on the delays encountered in transmitting cache coherence messages, by giving priority to short request or control (ack) messages. It does this by employing several service lanes per physical link for each input/output port and a multiple-flit based buffer for each service lane. When different types of messages are assigned different priorities, there is a loss of ordering of request and response messages between any two communicating nodes. Therefore, additional state preservation serialization measures are employed within the processor node, to check for violations before performing any cache state transition. An example being, receiving an invalidation for a cache line that is being requested, before the response arrives. In this case, the response on arrival should be invalidated. Other optimizations to the NoC include reducing the delays encountered for invalidations sent out by the directory at the L2 during writes to shared data. Instead of sending out unicast messages during invalidations, the invalidation packet is replicated in the routers in the X direction of the mesh network and each of these routers in turn forward the packet on the Y direction. In order to implement the new broadcast scheme the wormhole based router is extended to ensure that the output port schedules a packet for transmission only when all the following nodes that receive the packet contain enough space for the complete packet, i.e. store and forward based packet flow control is implemented at no buffer space increment for short control messages. While ack messages traverse the same path as broadcast messages, a single node is designated as the gatherer for all the ack messages generated and this accumulated ack message is passed to the invalidator node. This further reduces the request-response delay. An alternative to the store and forward broadcast scheme, a virtual ring based invalidation scheme, that connects all processors in a ring, is also proposed. Simulation results do show delay reductions up to 33%

in read and write request-response transactions and speed up in execution time for up to 9.5% on some benchmarks compared to a run on a NoC without the above mentioned optimizations.

2.4.5 DiCo: Efficient Cache Coherency for Tiled CMPs

The Direct Coherence or 'DiCo' CMP proposes a novel directory scheme that aims at reducing the indirection latency incurred when sending all requests to a fixed directory controller, consequently decreasing the network traffic [RAG08]. Instead of having the directory at the L2 storing all the sharer information, this task is given to a separate L1 coherence cache per processor core. It assumes a unified shared L2, that is physically distributed among multiple cores. On a miss in the L1 cache, the request is sent to the L1 coherence cache. If the coherence cache produces a miss, i.e. no owner information is stored, the request is sent to the L2 cache (depending on the miss address it may or may not be on the same tile). The L2 cache checks its L2 coherence cache to see if there is a hit or miss. On a miss, the line is fetched from main memory and the L2 coherence cache updates itself to indicate the requestor as the owner cache. The data is not stored within the L2 cache. If a hit occurs, the L2 forwards the request to the owner cache. The owner L1 cache supplies the data and updates its L1 coherence cache to reflect the requestor as the new sharer. The requestor also updates its L1 coherence cache to point to the owner L1 so that all future requests could be sent to the owner L1. On a write, the owner L1 sends invalidation to all sharers and requests the new owner to collect all the acks. The owner L1 also sends the requestor processor id as the new owner to the L2. The L2 updates its coherence cache and sends an ack to the new owner. Until the new owner receives all the acks, it does not commit its request, thereby maintaining total ordering. When an owner L1 cache evicts a line, the L2 coherence cache is updated with sharer information and the L2 becomes the new owner. If the L2 coherence cache evicts an entry, it sends an invalidation or writeback message to the owner L1. In order to further reduce the indirection to L2 on a miss within the L1 cache and L1 coherence cache, two *hints* schemes are employed. The first policy (DiCo–base), on an invalidation, the previous L1 owner stores the id of the requestor as the potential owner of the cache line within the L1 coherence cache. The second policy (DiCo–hints), when a change of owner occurs, during write to a cache line, the new owner gets a list of all sharers and the previous sharers invalidate their L1

caches, but update their coherence cache to indicate the new owner. This policy increases the network traffic compared to a normal directory protocol. A request that mispredicts or relies on indirection for the correct owner might experience starvation if the owner changes rapidly. In order to avoid this situation, a counter is maintained for each request. If the counter expires, the L2 delays the ack that it sends to the new owner until the starved request is satisfied. Simulation results do show up to an average 8% reduction in network traffic when comparing the DiCo–base protocol to a baseline directory protocol. However, the average execution time is almost the same for baseline directory and DiCo–base protocol for all benchmarks that were evaluated. The DiCo–hints protocols provides up to 8% average speed up in execution time compared to the baseline directory protocol. The following increase in network traffic is comparable to a baseline directory protocol. The cache coherency protocol is fairly complex. It requires correct prediction of the owner cache in order to improve performance, however the higher accuracy prediction scheme increases network traffic.

2.4.6 Virtual Hierarchies

In this tiled architecture a homogeneous tiled CMP is used in a heterogeneous manner. It contains 64 tiles interconnected using a mesh network [MH08]. It groups communicating tiles operating under a virtual machine (VM) and provides global shared memory for inter group communication. Each tile contains its own L1 instruction and data cache as well as a private L2 cache. A virtual cache hierarchy is formed by tiles that are grouped to run a workload under a single VM. This involves, caches searching for data in the group before trying to fetch data from the main memory. The advantage of grouping tiles under a VM implies that they can be reconfigured or reassigned to different VMs during runtime. For inter group communication, two cache coherency protocols are used; one a logical directory at main memory that consists of two levels (VH_A); the other a directory space conserving approach by using a broadcast based token coherence protocol (VH_B). In VH_A a full map directory containing the tile id of each home node within a VM is maintained at the main memory and a smaller directory is maintained at the home node within the VM. VH_B uses a single bit instead of a full map directory at the main memory. If the bit is set, it means that the line is shared by all tiles. The directory also stores a count of tokens for each cache address, thereby allowing for tiles not caching a line to not respond on a

broadcast request. Simulation results show that VH_A and VH_B protocols out perform the following directory schemes - a baseline directory that contains a single centralized directory at the main memory; a distributed directory scheme; and a duplicate tag based directory scheme that stores the directory information at the centre of the chip, by 22 - 42%.

2.4.7 Network Based Coherence

The features within the network itself, i.e. either the topology, routing algorithm or on chip wiring are used in order to reduce the effects of cache coherency [EPS06][CMR+06]. It implements a directory within each router in the network which allows for in transit optimizations of cache coherency messages. The node that gets a cache line from off–chip memory (via the home node) is designated as the root node. When the reply from off–chip memory traverses the network, it creates virtual links that point towards the root node. Each router node on the way to the root node caches the memory address as well as sets certain bits (NEWS) to designate its connectivity with other nodes in the path. 2 bits are used to indicate if the router node links to the root node, and if so through which link. A valid bit indicates that the node that the router is attached to contains data. A busy bit is applicable to the home node only and if set means that the virtual tree path is being changed. A request bit suggests that the node has sent a request for the same address and is yet to receive a response. The root node always points to the home node. It uses a complex protocol to maintain sequential consistency and ensure that the virtual tree is up to date. The main aim is to reduce the read and write latency due to the indirection overhead of directories.

Another work advocates the use of different types of on–chip wires to transmit coherency messages depending on whether they are critical or non–critical. The wires on this chip are characterized based on their latency, bandwidth and power consumption. Coherency messages are mapped on to these wires based on their latency and bandwidth requirements.

2.4.8 Proximity Aware Directory

Explores the idea of having private L2 caches and a distributed directory scheme implemented using a separate directory controller and directory cache per tile

[BKT07]. The actual directory and main memory is off-chip. The directory cache is used to store the state of some of the directory entries associated with the range of memory assigned to that tile. This architecture is compared to a traditional directory protocol scheme wherein the memory and its associated directory is distributed across multiple chips. While in traditional multiprocessors, node to node communication is the dominant factor, in CMPs it is off-chip memory access. In a traditional directory based protocol, the directory controller at the home node would have to fetch requested data from the physical memory attached to it. However, in a CMP, the memory is off-chip and access to it is costly. It does not require the home L2 tile to have the data in its cache, instead if the data is present in another tile, nearer the requestor, the request is forwarded to the remote tile. This protocol implements a full-map directory cache. Again, as before, the aim is to reduce the read and write latencies of coherence messages. Three different policies are used in choosing the nearest sharer. The Ist one depends on the manhattan distance between the remote tile and the requestor, the second one depends on the sum of distance between the home tile and remote tile and remote tile and requestor. The final policy randomly selects the remote tile. The average reduction in latency for serving a request that misses in the L2 of the requestor tile is about 26% compared to a baseline directory protocol. Average speed up observed was about 16%.

2.4.9 Alternative Home Node in Directory Based Tiled CMPs

This scheme relies on load-balancing the number of directory entries, by using two home nodes per cache block, thereby reducing the number of home-node directory misses [HSXP08]. Directory overflow also is reduced due to the addition of another directory to store the cache blocked information. Similar to Proximity Aware Directory Coherence protocol, nodes are associated with their own private L2 caches and a distributed (banked) directory. The main idea is to associate each cache block with either one of the two home nodes, so that the distribution of directory entries is fairly even. The first home node is determined using a direct-mapped scheme (using the least significant $\log_2 N$ bits of the cache line address, where N is the total number of nodes). The second home node is determined using a random function or flipping the most significant bit of the direct mapped

scheme. On a cache miss, the request is sent to both home nodes, and if both do not contain the directory information, the primary or first home node fetches the block from main memory. This block can now be stored in either the primary home node or secondary home node, depending on space availability. If the block is present in the secondary node, proper directory based cache coherency action is initiated and the primary node is informed about the existence of the block. Results show reduced number of invalidation messages generated due to insufficient space in the directory. Also, the number of L2 misses per instruction reduces up to 50% because of the increased probability of finding the block on-chip rather than going to main memory. However, this scheme would certainly increase the traffic in the network and this might cause reduced performance from the system.

2.4.10 Tera Scale

Intel's Tera Scale [ACJea07] architecture uses 80 tiles interconnected using a mesh network. Tiles can be general purpose processor cores with their own private L1 and L2 caches. It also has the provision for a group of processors to share a L2 cache. Tiles can also be a special-purpose computing logic, such as graphic co-processors, network accelerators or security engines. The use of both private and shared L2 caches per tile is motivated by the mixture of workloads that can be run on this architecture. Private L2 caches can be used efficiently for non-sharing multiprogrammed workloads as well as highly parallel applications. Shared L2 caches among multiple tiles can be used for moderately parallel workloads with an optimization to allow placement of shared cache lines nearer to cores that frequently need to access them [CS06]. A MESI based directory protocol is used. On-chip memory consists of 3D stacked SRAM and efforts are in progress to incorporate on-chip stacked DRAM.

2.5 Summary

This chapter provides an overview of the two main components within a cache coherent tiled based CMP, the OCN and the cache coherency protocol. It starts with the different OCN topologies used, concentrating mainly on mesh based networks that are most popular with tiled CMPs. This is mainly due to the ability of the mesh network to overcome the wire delay problem that requires shorter

wires with decreasing feature sizes. The performance of mesh based networks is dependent on the routing, switching, flow control and deadlock avoidance techniques, that is discussed further on. Cache coherency is introduced with a review of some of the multiprocessor and CMP systems that implement either snoop or directory protocols. More emphasis is laid on limited directory protocols, which is implemented in this thesis. It concludes with a review of several current tiled CMP systems that are proposed in academia as well as implemented and in use by industry. This aim of this chapter is to familirize the reader with various concepts and terminologies that are neccessary for the design of cache coherent tiled CMPs, which forms the crux of this thesis.

The next chapter introduces the JAMAICA processor, system archictecture and the simulator platform that is used as a base for the design of the tiled CMP.

Chapter 3

Jamaica

The JAva MAchine and Integrated Chip Architecture (JAMAICA) chip multiprocessor (CMP) is a symmetric shared memory multiprocessor system on a single chip [Wri01]. Figure 3.1 shows the architecture of the JAMAICA CMP. It consists of several block multithreaded processor cores [URS02], each with its own private level1 (L1) instruction and data cache. The cores are connected to a shared level2 (L2) cache using a bus interconnect. The L2 cache connects to an on-chip memory controller to access the off-chip main memory. The cache coherency protocol used is MOESI [AB86]. Cores are also connected to a token ring network using a Thread Interface Unit (TIU) for light weight thread distribution [Wri01]. This chapter discusses the JAMAICA architecture, mainly concentrating on the processor design, memory hierarchy, interconnect design(bus and token ring) and the simulation details.

3.1 Processor Architecture

The JAMAICA processor architecture contains a simple in-order, 5-stage RISC pipeline core, along with private L1 data and instruction caches. The processor core supports block multithreading thereby allowing thread level parallelism (TLP) to be implemented [Wri01]. Support for up to four hardware contexts is provided to allow for context switching which can be used to compensate for the high latency stalls during cache misses. A hardware context can be viewed as a virtual processor, allowing for interleaved execution of multiple threads, thereby improving performance [URS02]. Context switching also occurs on a quantum timer policy that allows a new context to be run every 1000 cycles. This ensures

that events such as spin locks, that hold processor resources, do not adversely affect the performance. Every time a thread misses in the L1 cache a request is created. The request consists of the type of L1 miss (read or write) and the missed cache line address. Each L1 cache within the processor core is provided with its own outstanding request table with a maximum of four entries (one entry per context). The request table is used to store a request generated by a thread during a cache miss. Along with the request table, a writeback buffer is provided to store dirty and evicted cache lines that are no longer required by the processor, before being written back to the L2 cache or main memory. The writeback buffer also serves as a victim cache, allowing for lines to be pulled back into the L1 on a cache miss. A Least Recently Used (LRU) cache line eviction policy is used within the L1. The pipeline uses a 2 bit up/down saturating branch prediction table and a hardware based register window scheme on a single register file that is shared by all contexts. Previous studies have shown that a windowed register file is more efficient as opposed to a stack based approach in order to support the frequent method calling feature that exists in object oriented languages [QDT88]. Figure 3.2, shows the architectural details of the JAMAICA processor core with the 5 stage pipeline, branch prediction unit (BPU), L1 I and D caches with their respective bus interfaces.

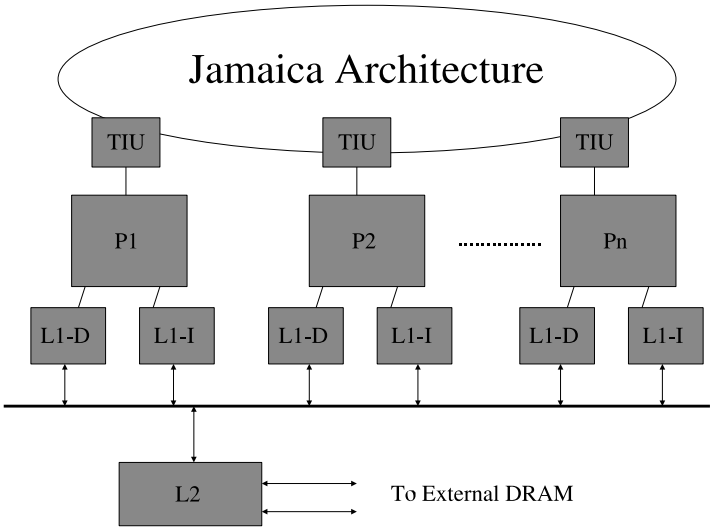

Figure 3.1: Jamaica Architecture[Wri01]

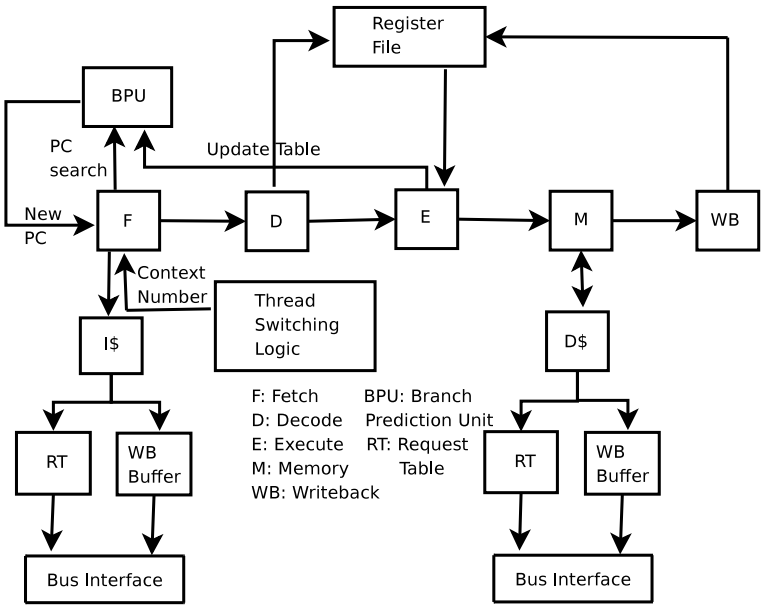

Figure 3.2: Jamaica Processor Core[Hor07]

3.2 Token Ring

[Wri01] JAMAICA uses a seperate token ring network that is used by processors to detect idle contexts on processor cores (contexts that are not executing any thread and are not stalled [Wri01]) and perform load balancing by sending threads to such processors. The JAMAICA instruction set architecture (ISA) provides two special instructions, namely *THJ* and *TRQ*, to send threads to idling contexts and to detect idle contexts, respectively. When a thread on a context exits from the last window in its execution stack, the context becomes idle and generates a token that is sent out on the token ring network as shown in Figure 3.3. If a thread on another processor's context needs to create a new thread in order to distribute the work load, it executes the TRQ instruction. The TRQ instruction stalls the pipeline for an average of N/2 cycles(where N is the number of processor cores) in order to detect the presence of an idle token on the ring network. If a token is found within N cycles, the current thread executes the THJ instruction that sends the newly created thread's setup information, such as the program counter and stack pointer of the parent thread's software stack to the idle context, over the bus [Wri01]. If no token is found, the newly created thread is executed locally. This light weight thread distribution mechanism is a novel feature of the JAMAICA architecture and provides better performance over a shared memory queue based

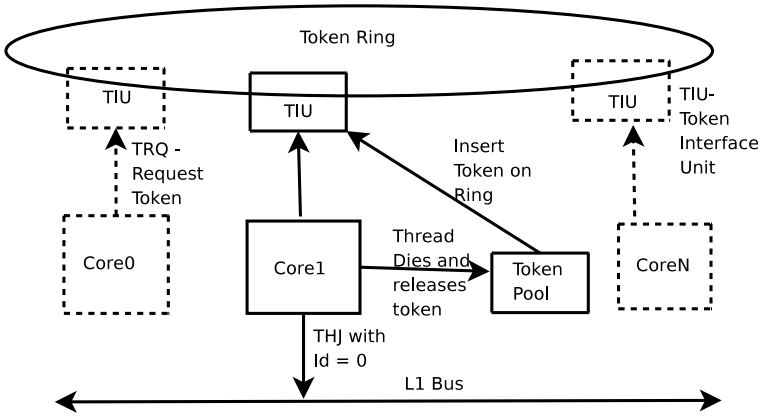

Figure 3.3: Processor interface to the Token Ring Network[Hor07]

thread distribution mechanism [DWKEM05].

3.3 Locking Mechanism

As in Alpha, MIPS and PowerPC processors , JAMAICA uses a pair of non-blocking atomic instructions, LDL_L (load linked) and STL_C(store conditional), to provide locking and synchronization primitives. The JAMAICA processor core provides a local lock table, with the number of entries equal to the number of contexts on the core. Each entry within the local lock table, contains the lock address along with a flag to indicate that the lock word has been loaded [Wri01]. A thread that executes the LDL_L instruction records the lock address within the local lock table and sets the flag associated with the lock entry. The same thread then tries to acquire a lock, by executing the STL_C instruction, which commits provided no other intervening write has invalidated the flag or the lock word cache line. The success or failure of the STL_C determines if the lock is acquired or not. If the STL_C succeeds at the local lock table and acquires the bus, it broadcasts the write, invalidating any other L1 cache lines and flag bits (if set) within other processors local lock tables. The code for a simple lock using LDL_L/STL_C instructions is as follows [Wri01]:

```
lock: LDL_L, reg,locklocn; /*set a flag*/
      BNZ L1;
      ADD reg,1;
      STL_C reg,locklocn;
      /*check the value of the flag, if flag=1,
```

```
      return 1; set flag to zero; broadcast STL
      to lock variable, else return 0(STL_C fails)*/
      BNE lock;
      //enter CS;
      .....
      STL 0,locklocn; //release lock;

L1:   WAIT;
      BNZ lock;
```

JAMAICA uses the WAIT instruction to avoid spin locking when a thread does not succeed on a LDL_L. The WAIT instruction stalls the context on which the locking thread is executing, until another thread writes to the same cache line [Wri01]. The broadcast write causes the *waiting* context to wake up the stalled thread and retry the LDL_L instruction. However, use of the WAIT instruction can cause deadlocks in scenarios wherein the number of threads exceeds the total number of contexts. In this case if all threads are in WAIT state, then the non-waiting thread may never get a chance to run.

3.4 Interrupt Mechanism

JAMAICA provides both hardware and software interrupts for handling hardware exception and providing support to the JaVM for implementing thread scheduling, respectively [Hor07][Wri01]. An explicit vectored software interrupt instruction (SIRQ) is used by the JaVM thread to wake up all other contexts during the boot process, by broadcasting an interrupt message (SIRQ) on the bus. The SIRQ broadcast message contains the identifier(id) of the thread that needs to be woken up as well as a 32 bit interrupt mask. The interrupt mask serves as a program counter for the woken up thread in order to access the interrupt code located in main memory. During the boot process, this interrupt code is used to set up the software stack for the woken up context. A hardware trap interrupt is used for handling invalid memory accesses from the processor core [Hor07].

3.5 Bus Transactions

JAMAICA implements a snoop based cache coherency protocol over a bus interconnect. All L1 caches and the L2 cache interface the bus using a master and slave bus interface component. The master handler is responsible for sending requests over the bus, while the slave handler snoops the bus-either to satisfy a request or service another request. All requests for either satisfying a cache miss or upgrading the state of a cache line are broadcast on the bus. A bus arbiter is provided and uses the LRU bus requestor policy for choosing a single requestor from the multiple masters arbitrating for the bus. Figure 3.4 shows the control and data components of the bus along with the timing diagram for a sample request. Each transaction on the bus takes eight bus cycles to complete. Cycle by cycle operation of the bus is explained below:

- Bus cycle1-Bus Request

  Bus Master sets a bus request - MReq.

- Bus cycle2-Bus Grant

  Bus arbiter issues a grant - MGnt.

- Bus cycle3 - Information

  The granted master sets the following lines:

  - Cache Address - for which the request had been generated,
  - Id - unique number associated with each context and assigned during the boot up process.
  - Type
    1. RD_SH - generated due to a read miss in a cache.
    2. RD_EX - generated due to a write miss in a cache.
    3. UPGRADE - generated due to a write hit in a cache.
    4. WB - writeback to L2 or main memory.
    5. MEM_SH, MEM_EX, MEM_UP - all memory originated transactions for requests not satisfied by other caches.
    6. THJ - generated when a processor needs to ship a task to another processor for load distribution purpose.

- Bus cycle4 - Wait(check)

  All other processors check their caches, request tables and writeback buffers to see if they can satisfy the request or if they have the same pending request in their request tables.

- Bus cycle5 - Status

  If the request was found in the cache or writeback buffer, the FOUND (F) line is set. In addition the state of the found data - SHARED (S), EXCLUSIVE (E) and OWNED (O) is also set on the S,E,O lines respectively. These lines are wired-OR and only one processor can set them at a time. On a WB of an O line, if a slave has the line in its cache in the S state or if it has a pending miss for the same address in its request table, then a slave request (SReq) is sent to the bus arbiter.

- Bus cycle6 - Memory check

  If the FOUND line is not set, the L2/main memory responds back with a memory accept(MAccept) signal.

- Bus cycle7 - Data1

  The 1st 16 bytes of the cache line is put on the data lines. If SReq had been asserted previously, then a SGnt for a slave implies that that a line in a cache in S state can be upgraded to O state. If the SReq was for a pending miss, then data can be accepted into the L1 in the O state. All other slaves that asserted their SReq lines, but did not get an SGnt, can accept the data into their L1 on a pending miss in the S state.

- Bus cycle8 - Data2

  The 2nd 16B of the cache line is put on the data lines.

The requestor *commits the request*, by writing the data value into the registers, cache and setting the cache line state.

3.6 Simulation Details

The JAMAICA architecture is simulated using an in-house Java based cycle accurate simulator-JAMSIM [Hor07]. The simulator itself runs on a Java Virtual

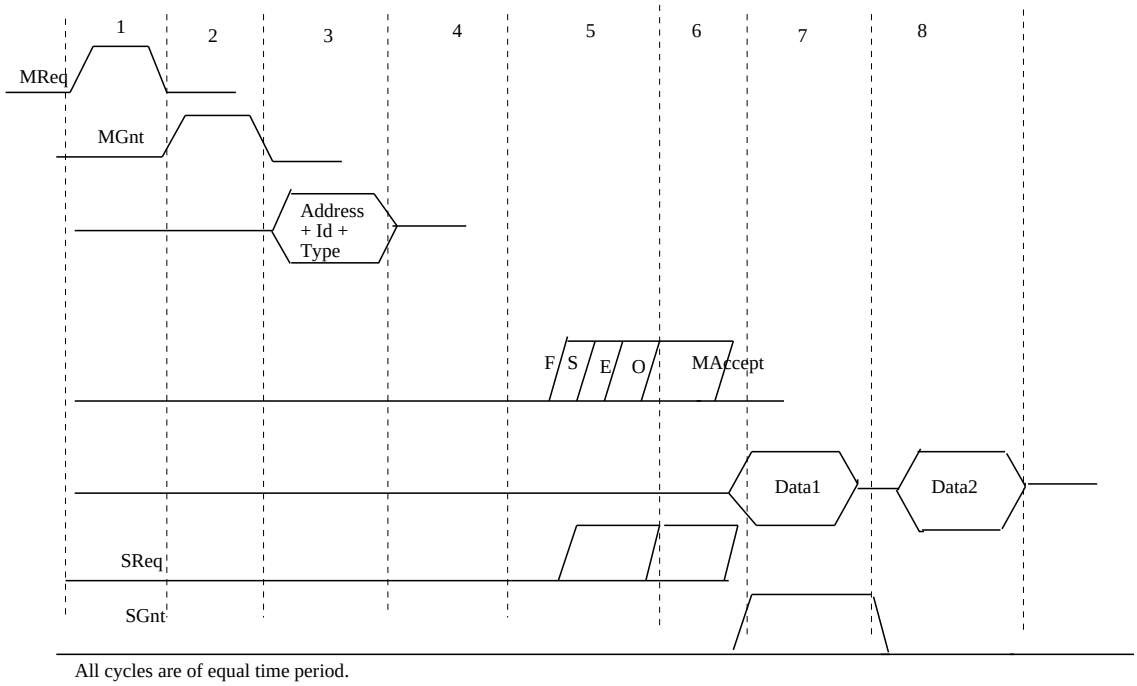

Figure 3.4: Jamaica Bus Transaction[Wri01]

Machine (JVM) [Inc] that executes on a host machine. It is capable of running applications that are compiled for the JAMAICA instruction set architecture (ISA). The simulator models hardware components as software objects, allowing for instantiation of multiple hardware components. It models pipeline stalls, bus interconnect transfers, bus arbitration contention, L1 cache access contention, on-chip memory controller queuing delays to access off-chip main memory and token circulation on the token ring [Hor07]. L1 cache access delay is modelled as 1 processor cycle, while main memory access delay is assumed to be 200 processor cycles without contention from other requests for the same bank of memory. Command line parameters are provided by the simulator to allow the user to control the number of processors as well as the number of contexts per processor. Two types of debugger - TTY and GUI allow for cycle by cycle viewing of the internal state of each component within the architecture. The use of object oriented principles in designing the simulator allows for easy modification of existing components, such as changing the cache sizes of L1 and L2, as well as allowing for addition of newer components, such as mesh or crossbar interconnects or increasing the memory hierarchy [Hor07].

Both functional and cycle accurate models are simulated using Java and a

JAMAICA ported version of the Jikes RVM (JaVM) [Aea00] [DWKEM05][Din06] provides a Java run-time environment. The JaVM allows execution of Java byte codes that are compiled to a JAMAICA executable, using the simulator. The simulator does not support any IO devices and so any calls to them within the application code result in a JAMAICA *subroutine call* instruction generated by the JaVM, jumping to a negative address in memory. These instructions are detected in the simulator and are handled as calls to the underlying OS running on the host machine [Hor07].

3.7 Summary

This chapter highlighted the architectural design of the JAMAICA CMP with a brief overview of the processor architecture, the memory system design, the cache coherency protocol, the task distribution token ring network and the simulator platform used to implement this architecture. It serves as a precursor for the following chapters that involve modification of the existing JAMAICA architecture to implement a tiled CMP.

Chapter 4

A Tiled Bus CMP

4.1 Motivation

It is widely believed bus based cache coherent CMPs do not scale to large number of the processors as seen in switched network based CMPs [GKS+07] [WGHea07]. This is mainly because of the limited bus bandwidth [KZT05] [VAG05]. Bus bandwidth is constrained by the speed at which the bus is clocked [Wri01]. Figure 4.1 shows the theoretical peak bandwidth utilisation of the Jamaica bus [Hor07]. The Level1 (L1) instruction and data caches assume a hit rate of 99% and 98% respectively and the L2 has a perfect hit rate. The running application is a RISC code that contains an instruction mix of 22% loads and 12% stores [HP07]. As shown in Figure 4.1, it would be ideal to clock the bus at processsor clock speeds [Hor07]. However, the wire delay problem in current submicron technologies limits the clock speed of the bus, resulting in bus saturation at low processor counts. An alternative to bus interconnects are high bandwidth switched networks. Although switch based CMP systems are highly scalable compared to their bus based counterparts, they suffer from indirection latency, invalidation latency for writes to widely shared data (cache line shared by many processors) and induce storage overheads for widely shared data in order to maintain cache coherency using directory protocols [RL97] [ZHWH07]. An ideal cache coherent CMP system would perform with the (low) latency of a snoop based cache coherent protocol and provide the scalability of a directory based protocol [BDH+99]. In a step towards realizing this ideal cache coherency protocol that utilizes the benefits of both snooping and directories, a dual network CMP architecture - Mesh and Bus, is investigated. The cache coherency

protocol uses a broadcast mechanism for writes to widely shared data, while a directory that records the current cache line owner is used for reads and writes to private and shared data. The bus network is used for the broadcast mechanism and the limited directory protocol is implemented over the mesh network. As with sequential consistency, the bus provides global ordering (write serialization) for writes and each processor core is in-ordered and follows program order. In this chapter we detail the architecture and cache coherency of this dual network based CMP system. The main aim is to investigate the scalability of this system, without incurring the low bandwidth problem of bus based CMPs and space and invalidation latency overhead of directory based cache coherency protocols.

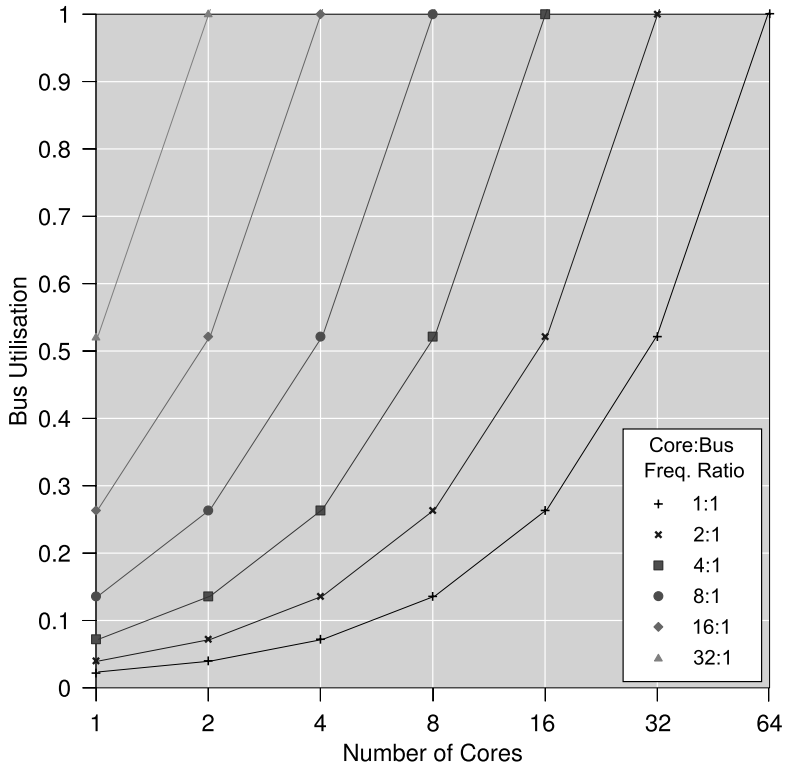

Figure 4.1: Bus Clock Speed vs. Scalability of Processors [Hor07]

4.2 System Architecture

As seen from Figure 4.2, the architecture consists of processors and memory tiles that are interconnected using a mesh network and a snoop bus. All tiles connect to an arbiter that is provided for each network. The arbiter is used to ensure that only non-conflicting requests enter the network. It should be noted that

the placement of processor and memory tiles can be interleaved in any order and need not adhere to the specific configuration as shown within Figure 4.2. Architectural details of each component within the system and the description of the cache coherency protocol forms the subsequent sections of this chapter.

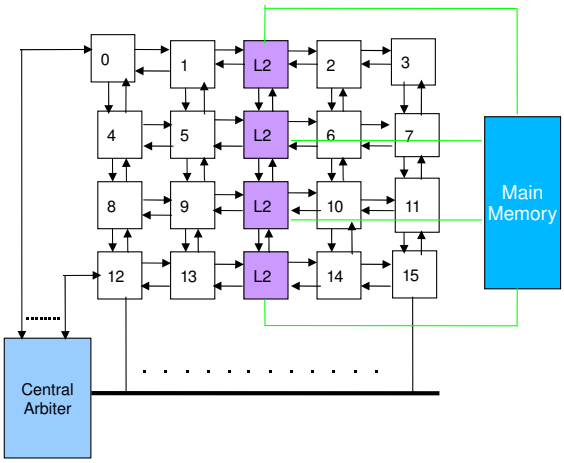

Figure 4.2: Tiled-Bus Based CMP

4.3 Processor Tile

Every processor tile consists of a JAMAICA multithreaded core containing L1 instruction and data caches, as shown in Figure 4.3. A request table and writeback buffer are provided for each cache. The tile uses a bus interface that contains a master and slave handler to send and receive data respectively, on the bus. The bus master handler is used for sending SIRQ and THJ transactions, while the slave snoops THJ, SIRQ and all invalidations sent on the bus.

The tile interfaces to the mesh network using four Input and Output (IO) ports - North, East, West and South (NEWS) [HP07]. Each port consists of a master and slave handler, referred to as port master and port slave, respectively. Port slaves receive and process packets while masters transmit packets. The processor core interfaces with the IO ports using a local port. All port masters and slaves are interconnected using a crossbar switch as shown in Figure 4.4. The routing unit uses a dimension order routing protocol or XY routing for determining the next tile that receives the packet enroute to its destination [DYL03]. The packet switching scheme at each port is store and forward, i.e. the complete packet is buffered before it is sent out of the port master [DYL03]. Each port slave is

equipped with its own finite state machine (FSM) based control logic to *fetch*, *decode* and *forward* the packet. The FSM at the port master is used for buffering and transmitting the packet. In order to buffer the packet, each port master and port slave contain a single separate packet buffer. The buffer width at each of the masters and slaves is equal to one cache line width (256 bits) plus the header packet size (64 bits). The message format used is shown in Figure 4.5. The header packet consists of the following five fields:

- *Source identifier(id)* of the tile that generates the packet.
- *Destination id* of the tile that is to receive the packet.
- *Type* of packet - if the packet is a read or write request, or if it is a response packet.
- *Control* indicates the state of the cache line that is being requested/sent.
- *Address* of the cache line that is being requested.

The mesh network uses 32 bits for control (first four fields of the header packet) and a 128 bit wide data path for transmitting request and data packets across adjacent tiles. The port controller component oversees the scheduling of port slaves for receiving a packet and processing it. It ensures that only one port slave is accessing the caches, request table and writeback buffer at any instant in time, thereby causing the contention experienced at slave ports during packet processing. Also, if multiple slaves access the same master, only one slave is granted access, again simulating intra tile contention between slaves. A bus slave gets higher priority over port slaves for accessing the caches, request table and writeback buffers. The flow control technique between adjacent tiles is a request-grant scheme [DYL03], with the master sending a request to the slave, and the slave responding with a grant if it is ready to process the packet.

4.4 Router Architecture

The router within a tile contains the port master and slaves, the routing logic unit, the switch allocator unit and the crossbar switch as shown in Figure 4.4. The functionality of port master and slaves has been explained in Section 4.3. The routing logic uses the DOR routing protocol and is accessed by the port slave

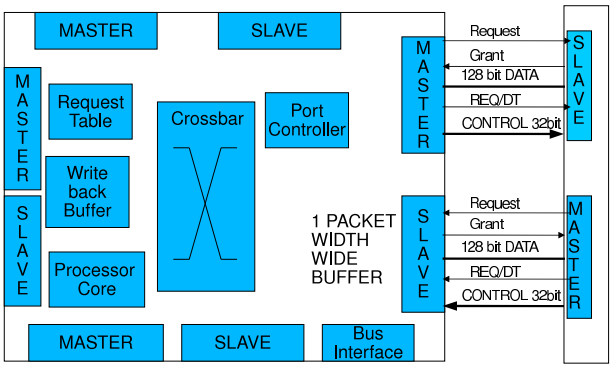

Figure 4.3: Structure of Processor Tile

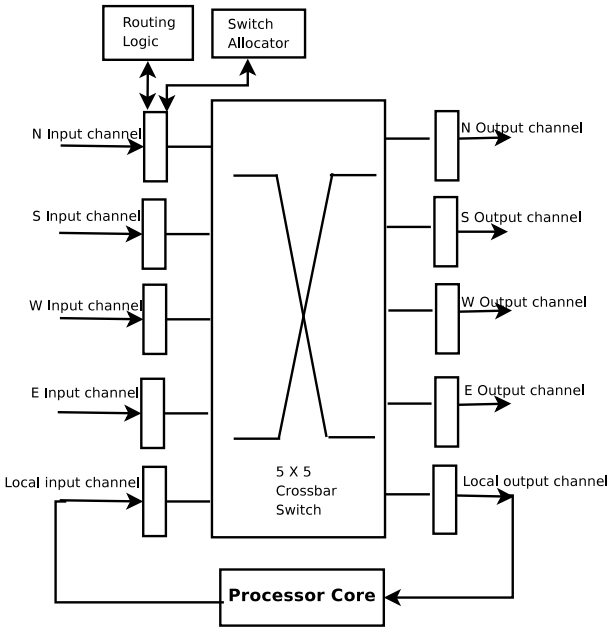

Figure 4.4: Crossbar Logic within Processor Tile

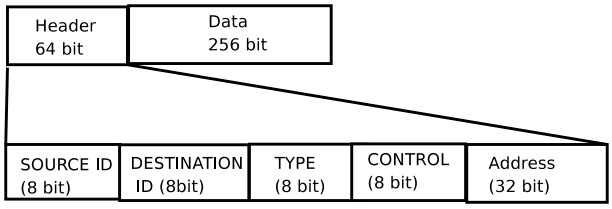

Figure 4.5: Message Format

during the *decode* phase to determine the port master that the packet needs to be sent to. The port master that a packet needs be sent out on is selected depending on the location of the destination id of the node. The switch allocation unit (part of the port controller) is accessed by the port slave during the *forwarding* phase

of the FSM in order gain access to the port master. During this phase if the port slave wins the arbitration to the port master, it sends the packet through the crossbar switch to the port master.

4.5 L2 Tile

The memory hierarchy in this system consists of on-chip L1 and L2 caches with off-chip main memory modules. The L2 is a Non-Uniform Cache Access architecture (NUCA) [KBK02] based memory that is split into banks, with each bank represented as a separate tile. Each L2 tile uses a memory controller to interface with the off-chip DRAM or main memory. All read and write requests generated with the processor tile are sent to the appropriate L2 tile, based on equation 4.1. The L2 tile contains the cache data and tag array, cache controller, port controller, request table and bus interface. The port structure is similar to the processor tile, except that there are eight masters and slaves per port. The port controller ensures that at a given port, only a single port master, out of the eight masters, is engaged in packet transmission and only a single port slave at a given port is receiving a packet, at a given instant of time. At the same time other ports could be decoding or processing a packet. Only one port slave can access the L2 cache data and tag array, and request table at any instant in time. The request table is provided to track any pending requests (writebacks issued by L2), or any invalidations that need to be sent out on the bus. The bus interface is responsible for broadcasting any invalidation requests from the request table onto the bus. The L2 tile is solely responsible for the invalidation broadcasts and hence does not need a slave handler within the bus interface, for snooping the bus. Figure 4.6 shows the structure of the L2 tile and Figure 4.7 shows the L2 cache line. The L2 cache stores the following information for each cache line along with the data:

1. The *owner of each cache line*, the processor id that has the cache line in Modified state within its L1 or if it accessed it first.

2. A flag (F) to indicate if a line is shared by multiple caches or if it is present in a single cache. It uses this flag to decide if a broadcast needs to occur on a write to this line.

3. A lock bit (L) to indicate if a lock access has been performed on that line.

4. The state of the cache line, Exclusive (E), Shared (S), Modified (M) or Pending (P).

The L2 tile to which a request needs to be sent is determined using the following equation:

$$bankid = (address >>> 5)\&(2^{\log_2(numberofL2tiles)} - 1) \tag{4.1}$$

Where, & represents bit-wise AND and >>> represents bit wise right shift operation.

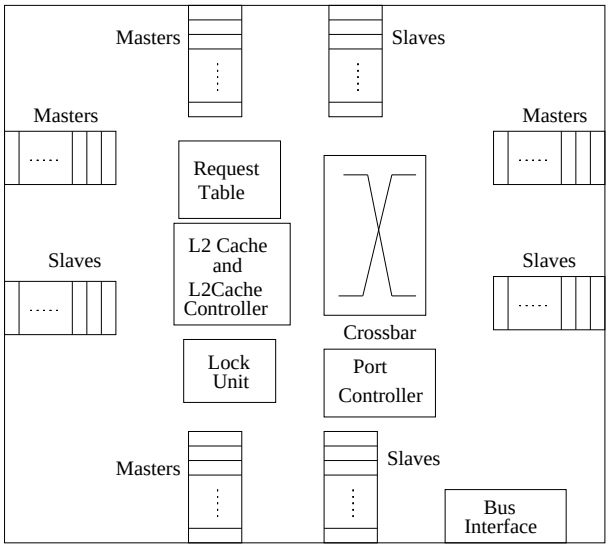

Figure 4.6: L2 Tile Structure

Tag	V	E	O	F	L	Owner ID	Cache Data

Figure 4.7: L2 Cache Line Structure

The L2 uses a different line eviction policy from its cache compared to the Least Recently Used (LRU) cache eviction policy in the L1. For evicting a line, the L2 checks the state of the line and the lock bit status. Lines with lock bit set are not evicted. Among the different cache states, lines with states such as E (first choice) or S are preferred for eviction compared to M lines. The LRU policy is used when there are multiple lines with E or S states. In the outer case that all lines within a set are locked and it is imperative to evict a locked line,

then certain coherence messages need to be generated and will be explained in Section 4.8.4.

4.6 Central Arbiter

The central arbiter component consists of two arbiters - one for the mesh network, the other for the bus interconnect, as shown in Figure 4.8. All processor and L2 tiles access the arbiter for access to the mesh or bus. Only requests need access via the arbiter; response messages do not need arbitration. On every arbitration cycle only one mesh and bus request is granted. Processors and L2 tiles that get access to the mesh need to inform the mesh arbiter on completion of a request. This also guarantees that no two requests for the same address can exist at the same time within the mesh network, providing for serialization of requests to the same address. An exception to this rule is when a processor does a write back of data to the L2. In this case, the processor releases the arbiter as soon as the writeback message leaves its master port. There is no acknowledge message from the L2 to indicate that the writeback has reached its destination.

Bus arbitration relies on request completion within a fixed number of cycles; 8 in the case of the JAMAICA bus [Wri01]. A separate snoop signal is provided for each component on the JAMAICA bus. This snoop signal prevents these components from snooping the bus on every bus cycle. The reason for providing this snoop signal was mainly due to the fact that the number of transactions on the bus is reduced significantly as compared to that of a bus-only CMP. Therefore, disallowing snooping the bus when not in use would reduce power consumption. All bus components check this signal every bus cycle to determine if a bus snoop is required in the following cycle. Figure 4.9 shows the 8 cycle JAMAICA bus.

4.7 Cache Coherency Protocol

The cache coherency uses a combination of both snoop and directory based protocols. The main idea is to use a directory based protocol without the overhead of storage space for tracking the list of sharers and reducing the latency during broadcasts to widely shared data. For writes to widely shared data, the snoop based protocol is used and the directory protocol for all other reads and writes. The cache coherency protocol aims to decrease the write latency to widely shared

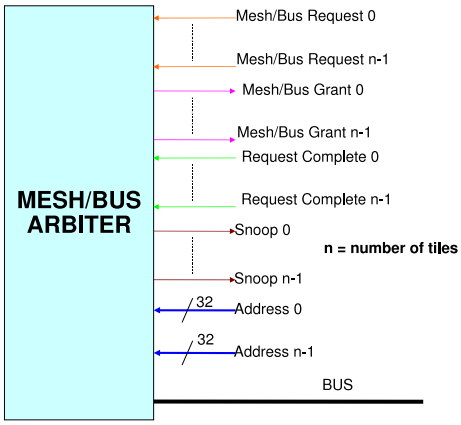

Figure 4.8: Central Arbiter

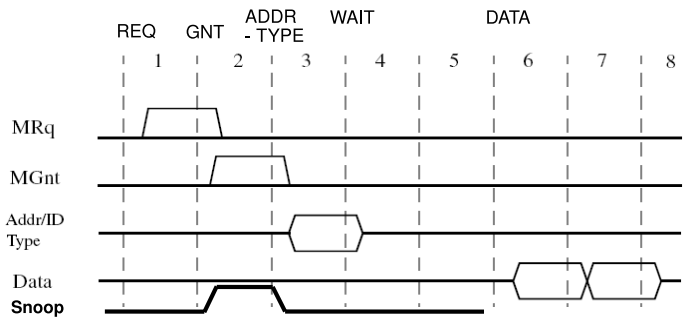

Figure 4.9: Timing Diagram of the JAMAICA bus

data and at the same time reduce the space overhead ($M * \ln N$ bits, where M is the total number of L2 cache lines and N is the number of processors) associated with full map directories protocols. The cache states used in the L1 caches are Modified-M, Owned-O, Shared-S and Invalid-I, while those used at the L2 are Modified, Exclusive-E, Shared, Invalid and Pending-P. The S state in L2 can be further categorized as either Shared Single Owner S(SO) or Shared Multiple Owner S(MO). The S(SO) state implies that the line exists in the S state within a single L1 cache. On the contrary, the S(MO) state implies that the line is shared by multiple L1 caches. Table 4.1 describes the various cache states in both L1 and L2. The cache coherency protocol is best explained with the cache state transition diagrams shown in Figures 4.10 & 4.11, for both L1 and L2 caches respectively.

The various requests that are generated by the L1/L2 caches can be classified in general as Read Misses, Write Misses, Writebacks and Evictions. Tables 4.2,

Cache State	State Description
M	Modified-Line is dirty and the latest copy of the data exists in the L1cache.
O	Owned-Line is dirty and shared between I and D L1 cache on the same processor core. This is because of the self-modifying code feature within the JIKES RVM
E	Exclusive-Line is clean and a single copy of the data exists in the L2 cache.
S	Shared-Line is clean and exists in one or more L1 caches.
I	Invalid-Cache line contains invalid data in both L1 and L2 cache.
P	Pending-Cache line is in a flux state at the L2 waiting for an acknowledgement or data from the L1

Table 4.1: Cache Line State Description

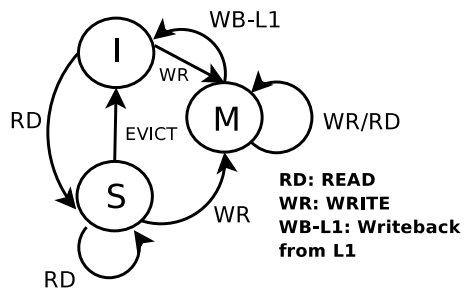

Figure 4.10: L1 Cache State Transitions

4.3 and 4.4 shows the different request, response type fields and control fields used within the packet header, respectively. Different requests and responses are identified by a combination of *type* and *control* fields from the packet header. Henceforth, all requests and responses will be designated as a tuple, with the first entry in the tuple designating the *type* field from the packet header and the second entry representing the *control* field from the packet header. Each of these requests and the corresponding responses they generate are described in detail in the following paragraphs. Note that for any request to be sent on either the bus or mesh, the requestor (either L1 or L2) needs access from the central arbiter.

Requests	**Description**
RD_SH	Read miss from the L1
RD_EX	Write miss from the L1
UP	Write hit to a S line in L1
WB	eviction of M lines from L1
LOCKREQ	L1 trying to access the lock at L2
LOCKFREE	L1 frees the lock at L2
RD_SH_L2FWD	Generated on a read request to M line in the L2
RD_EX_L2FWD	Generated on a write request to M line in the L2
LOCKGNT_L2FWD	Generated on LOCKREQ message to a M line in the L2
WB_PENDING	On evicting a M line from L2
INVALIDATE	On evicting a S(SO) line from L2
B_INV	On evicting a S(MO) line from L2

Table 4.2: Type field: L1 and L2 Requests

Responses	**Description**
RD_SH_RESP	Generated by the L2 on a RD_SH, or generated by a L1 on a RD_SH_L2FWD to a M line in cache
RD_EX_RESP	Generated by the L2 on a RD_EX or generated by L1 on a RD_EX_L2FWD to a M line in cache
UP_RESP	Generated by the L2 on a UP
WB	Generated by the L1 on a WB_PENDING from L2 or on evicting a line
LOCKGNT	Generated by L2 on a successful LOCKREQ or generated by L1 on a LOCKGNT_L2FWD to a M line in cache
LOCKQUEUE	Generated by L2 on a unsuccessful LOCKREQ
OWNERDATAUP	Generated by L1 on a RD_SH_L2FWD to a M line in cache
OWNERUP	Generated by L1 on a RD_SH_L2FWD to a M line that is being evicted from the cache or is present within the other cache of the requestor, i.e. request from data cache, line present in instruction cache
LOCKGNT_L2FWD	Generated on LOCKREQ message to a M line in the L2
WB	Generated by L1 on receving a WB_PENDING from L2

Table 4.3: Type field: L1 and L2 Responses

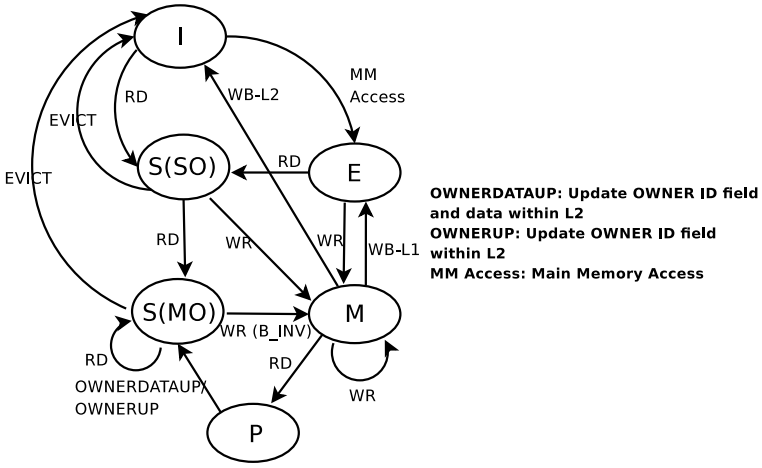

Figure 4.11: L2 Cache State Transitions

Control Field	Description
FWD	State of the cache line in the response message is unknown
ACK_M	State of the cache line in the response message is M
ACK_S	State of the cache line in the response message is S
ACK_M_INVPENDING	State of the cache line in response message is M, wait for B_INV to complete

Table 4.4: Control fields within the packet header

- **Read Miss**

A read miss is generated when the cache line is in the I state or not present within the L1. The read miss is sent out as a (RD_SH,FWD) packet to the L2, FWD within the control field indicates that the state of the cache line is unknown. If the line is in E or S(SO) or S(MO) state at the L2, a read response packet (RD_SH_RESP, ACK_S) is generated by the L2 and sent back to the requestor. The ACK_S control field indicates that the requestor should set the state of the line in the L1 to the S state. The L2 takes the following actions depending on the state of the line in its cache:

1. If the line was in S(SO) state before the read, the L2 resets the single owner flag within the cache line to indicate that the line is shared between multiple caches, i.e. the L2 is now in S(MO) state.

2. If the line was in E state, the L2 sets the single owner flag and records the requestor's processor id.

In both cases, the L2 transits to the S state. On receipt of the (RD_SH_RESP, ACK_S), the L1 transits to the S state.

3. If the line is in the M state at the L2, the request gets forwarded to the to owner L1 (RD_SH_L2FWD, FWD) and the L2 transits to P state. Owner L1 responds with data to both L2 (OWNERDATAUP, ACK_S) and requestor (RD_SH_RESP, ACK_S), and changes its L1 cache state to S. On receipt of the OWNERDATAUP packet, the L2 updates its state to S(MO) and resets the single owner flag. Requestor L1, on receipt of RD_SH_RESP transits to S state.

   There are instances when the owner L1 does a write back (WB) of the M line from its cache and at the same time the RD_SH_L2FWD packet is in flight in the network. In this case, the owner L1 will redirect the packet as a RD_SH packet back to the L2. As long as static routing (DOR) is employed over the mesh network, it is guaranteed that the redirected RD_SH packet will always arrive after the WB at the L2. It should be noted that while the L2 is in P state, it does not process any packet for the same address. This is done so as to guarantee completion of service of a request once it has been accepted at the L2.

   There is also the possibility that the RD_SH packet arrives from the L1 instruction cache of the requestor, while the data is present in M state within the L1 data cache of the requestor. This happens only when JIKES is generating new code during run-time, resulting in code being written to the data cache. In this case, the owner L1 will generate a non-state change packet (OWNERUP, ACK_M)(owner is the requestor) to the L2 and changes the L1 instruction cache to O state. Alternatively, the owner L1 might be in the process of evicting the line from its L1 and the request from L2 arrives. In this case, the owner L1, generates a (OWNERUP, ACK_M) packet to L2 and a (RD_SH_RESP, ACK_M) packet to the requesting L1. L2 on receipt of OWNERUP packet will switch back to the M state from the P state and update the owner id field to reflect the requestor's id. Requestor L1 on receiving the RD_SH_RESP packet transits to M state within its L1.

4. If the line is in I state, L2 fetches the line from main memory in E state and processes the request as a read miss to an E state line.

- **Write Miss**

A write to the L1 cache line results in either a write miss (RD_EX, FWD) to an I state or non existing cache line, or write hit to a S line (UP, FWD) request. Writes to M lines do not generate a request. The L2 takes the following actions on receiving a RD_EX/UP request:

1. On an UP request, the L2 will either respond with a upgrade grant message (UP_RESP, ACK_M) on a S(SO) line, or broadcasts an invalidate message on the bus (B_INV) if the line is S(MO) state. L2 changes state to M state, records the requestor (writer) processor id as the owner of the cache line and sets the single owner flag. Requestor L1 on receiving the UP_RESP or B_INV will transit to the M state.

2. On a RD_EX request, the L2 takes the following actions:

   (a) If the line in L2 is in E state - it grants the line in M state (RD_EX_RESP, ACK_M) to the requesting L1, sets the owner id field to the requestor's processor id and switches to M state.

   (b) If the line in L2 is S(MO) state - it sends a (RD_EX_RESP, ACK_M_INVPENDING) message to the requesting L1, a B_INV message on the bus, sets the owner id field to the requestor‘s processor id and switches to M state. The L1 requestor on receiving the RD_EX_RESP packet and B_INV will transit to M state.

   (c) If the line in L2 is in M or S(SO) state, it forwards the request to the owner processor (RD_EX_L2FWD, FWD), sets the owner id field to the requestor's processor id and switches to M state. The owner processor will send a (RD_EX_RESP, ACK_M) packet to the requestor and invalidate itself. If the line has already been written back (WB) from the owner L1, then the owner processor will redirect the RD_EX write request back to L2. This redirected RD_EX request is similar to a write to an E line in the L2. The requestor L1 on receiving the RD_EX_RESP message will switch to the M state.

(d) If the line in L2 is in I state - it fetches the line from main memory in E state and processes the RD_EX as a write to an E state line.

- **Writeback**

Space constraints within the L1 and L2 cache often require cache lines to be evicted. The L1 cache uses the Least Recently Used (LRU) cache eviction policy. The processor core is provided with an additional writeback buffer for each cache to store these cast out lines which acts as a victim cache. If the writeback buffer is full, the lines in M state need to be written back to L2. In the case of L2 caches, an eviction of a M line requires notifying the owner of the cache line for a writeback to main memory. The L2 uses a cache line eviction policy based on the LRU count as well as the state of the cache line. Before evicting a line, the L2 checks the state of the line and the lock bit status. Lines with lock bit set are not evicted, this is also true for lines within the L1. Among the different L2 cache states, lines with states such as E (first choice) or S are preferred for eviction compared to M lines. The LRU count is used to determine the final candidate in the case when multiple cache lines based on the cache state policy are chosen for eviction. The following messages are generated when M lines are evicted from the processor writeback buffer or L2 cache:

1. On evicting M lines from the L1, a writeback (WB) message is sent to L2. L2 sets the cache line state to E, clears the owner field and resets the single owner flag.

2. On evicting M lines from L2, a write back request (WB_PENDING, FWD) message is sent by the L2 to the owner L1 and transits to the P state. The owner L1 responds with a WB message. L2 writes the line to main memory on receiving such WB messages and invalidates the line from the cache. It is possible for a WB_PENDING as well as a WB message to be in flight within the network at the same time. In such a case, because of the static routing protocol used, the WB message is guaranteed to reach the L2 before there is any following response from the L1 and the line is written back to MM. The owner L1 sends a (WB_PENDING_FAIL,FWD) message to the L2, to indicate that the line has already been written. Deterministic packet routing guarantees that the WB_PENDING_FAIL reaches the L2 later than

the WB packet from the L1. The L2 discards the WB_PENDING_FAIL message after checking that the request table no longer contains a WB message for the same packet. Although, there is no actual need for the L1 to generate a WB_PENDING_FAIL message, it serves as a debug feature within the simulator to check if the routing protocol ensures in order packet delivery.

- **Eviction**

Evicting a S(SO) line from L2 results in either an invalidation message (INVALIDATE) or a B_INV message on the bus for S(MO) lines. Shared lines from L1 are discarded on eviction and do not generate any network traffic.

4.8 Lock Unit Description

All shared memory multiprocessor systems require mechanisms to protect multiple threads from accessing and modifying shared data at the same time. Synchronization is a technique used in multiprocessor systems to allow for exclusive access to shared data. Synchronization is either used to implement mutual exclusion, i.e. allowing only one thread to gain access to the shared data at any point in time; or provide for conditional execution, i.e. threads can continue executing only after a certain condition is satisfied [GVA+08]. Locks, semaphores and mutexes are used to implement mutual exclusion, and conditional synchronization is implemented using flags or barriers. Mutual exclusion can be further classified as course grained or fine grained. Course grained synchronization typically requires locking a complete shared data structure (large region of memory) that is accessible by several threads. Fine grained synchronization techniques concentrate on locking smaller regions of memory, typically a memory word or a cache line. The traditional lock based scheme, also known as blocking synchronization, is prone to deadlocks and therefore, non-blocking or lock-free synchronization algorithms that do not require accessing a lock to update a shared memory location are used in certain synchronization scenarios.

4.8.1 Synchronization Primitives

In multithreaded applications, the user defines the synchronization algorithm in software (using in-built library functions) and the compiler implements the

algorithm using specialized hardware instructions or a sequence of hardware instructions [CSG99][HP07], known as synchronization primitives, that perform hardware atomic read-modify-write (RMW) operations on a memory location, i.e. the lock word. There are several primitives, such as exchange, test and set, fetch and increment, compare and swap [HP07]. They all differ in the manner in which they access and update the lock. Atomically performing a RMW on a memory location can be a challenge to implement in hardware, especially in bus interconnect based multiprocessor systems, as it requires locking the bus until the lock operation completes [CSG99]. Synchronization primitives, such as compare and swap, require the address of a memory location and two registers within a single instruction, making it hard to implement in RISC[1] based architectures, that use at most a single register and one memory location within an load or store instruction [CSG99]. Therefore, modern systems, such as Alpha, IBM PowerPC and JAMAICA, use a pair of non-atomic instructions, Load Linked (LDL_L) and Store Conditional (STL_C), in order to implement the RMW operations [Dig92][CSG99][Wri01]. During a LDL_L operation, the lock value is read and if it is determined to be free (normally if the lock memory word represents a mutex, value 0 represents a free lock and 1 vice versa), the STL_C instruction is attempted. The thread that succeeds in doing the STL_C is the eventual owner of the lock. In a cache coherent system, the STL_C succeeds only if the lock cache line is valid in the cache and the thread that is executing the instruction accesses the bus or directory (in the case of a switched network) and sends out an invalidation to all other threads. The advantage of this approach is that it does not rely on locking the bus during a synchronization operation and can use a RISC based instruction set. During the lock access phase, it requires loading the cache line in order to test the lock value and does not perform any invalidations on unsuccessful access of the lock, such as in compare and swap. However, if the lock is heavily contended, threads that cannot gain access to the lock normally spin on the lock variable (especially if the critical section is small [CSG99]). This consumes processor cycles, and is wasteful in the case of multithreaded cores that have the provision for another thread to run and perform useful work.

[1]Reduced Instruction Set Computer

4.8.2 Hardware Queue Based Locking in Tiled Bus CMP

In order to improve on the original bus based locking protocol a queue based locking scheme [MPS06] was implemented in the tiled and bus CMP. In this locking scheme, a queue based hardware synchronization unit (lock unit) is used to handle the *grant* and *release* of locks, *in hardware*. The lock unit receives multiple requests (LDL_L) for a lock (memory word), but grants access to only one thread and queues all other threads by storing the processor id associated with requestor thread. The thread that gets the lock access eventually executes the STL_C and releases the lock. The subsequent queued thread then gets access to the lock. When a thread is queued, the context is stalled on the processor, similar to a cache miss, allowing the JAMAICA processor core to run another thread. The process of granting the lock in hardware essentially controls the lock acquire or release in software. This is so because the thread that gets the grant in hardware is permitted to perform the STL_C operation, which could signal either a lock acquire or release operation in software depending on the synchronization algorithm that is being implemented. The semantics of the hardware based lock unit is similar to compare and swap and is implemented using the LDL_L and STL_C instructions in order to maintain compatibility with the JAMAICA instruction set.

Queue based locks have proved to be efficient in software and hardware implementations [And90][GW88]. In the current architecture, each L2 bank is associated with a dedicated lock unit. As shown in Figure 4.12, the minimum number of entries within the lock unit is equal to number of processors multiplied by number of contexts per processor. Each entry has separate fields to store the address of the lock (cache line address) and a pointer to a queue that is associated with the lock entry. The queue contains a list of current and pending owners for the lock. The current and pending owners for a lock are represented by the ids of the processors on which the threads are executing. The lock unit entry is updated on a STL_C or normal STL to the same cache line as the lock word.

4.8.3 Modification to Jikes RVM

The locking code generated by the original JaVM does not guarantee to perform a STL_C after a LDL_L, as seen in Section 3.3. However, in this locking scheme, it is imperative for a thread that succeeds at the lock unit at the LDL_L stage to

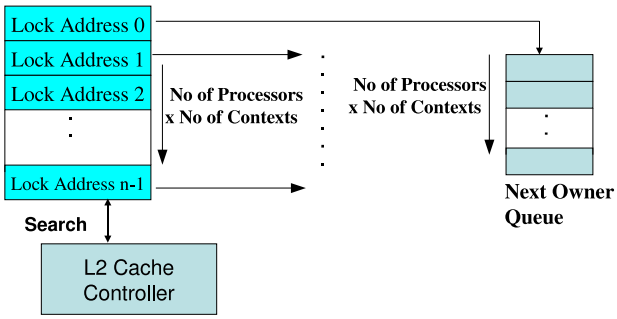

Figure 4.12: Lock Unit

perform a STL_C in order not to deadlock. As shown in Figure 4.13, the original locking code in JaVM (a) was modified (b), such that a thread that succeeds at the LDL_L stage performs the STL_C and writes either the same value that it read during the LDL_L stage if the compare operation fails, or writes the new value, if the compare operation succeeds. Therefore, the queue based locking scheme supports compare and swap using LDL_L and STL_C instructions. Although the lock unit provides hardware atomic operations to access locks, it is up to the software to ensure that the appropriate value is written to the lock word in order to guarantee the property of mutual exclusion when accessing the critical section code. In the modified Jikes implementation, if the LDL_L/STL_C pair is used to implement a lock operation, as in mutexes or semaphores, and a normal STL is used for the unlock operation, then the queue based compare and swap scheme performs worse than the original JAMAICA LDL_L/STL_C scheme. This is because when the next in line waiting thread (assuming a heavily contended lock) gets a lock access from the lock unit, it reads the cache line to determine if the lock has been freed by the previous lock holder, if not then the new lock owner will perform a STL_C in order to release the hardware lock unit and this results in unnecessary invalidations on the bus. However, in the current architecture, the two networks provide more bandwidth compared to a bus only JAMAICA system and therefore the unnecessary invalidation messages that could possibly be generated in the case of blocking synchronization scenario would not hinder other requests that would be sent over the mesh network. If the LDL_L/STL_C pair is used to implement non-blocking synchronization, then the STL_C would signal the release of the lock in hardware and software and the subsequent queued thread will access a freed lock, thereby providing serialization to lock access and better utilization of processor cycles by not performing spinning.

```
lock: LDL_L reg, locklocn; /* set a flag */
BNZ L1:
ADD reg, 1;
STL_C reg, locklocn;

/*Check the value of the flag, if flag = 1
return 1; set flag to zero; broadcast STL
to lock variable; else return 0 (STL_C fails) */

BNE lock; //spin on lock
//enter CS;

..............

STL 0, locklocn; //release lock;

L1: WAIT;
BNZ lock;
```

a) Original Locking Code

```
lock: LDL_L reg, locklocn;
CMPNE reg1, reg, newlockvalue;

/*compare reg and newlockvalue (data value);
reg1 = 0, if reg == newlockvalue (compare fails);
reg1 = 1, if reg != newlockvalue (compare succeeds) */
BZ  L1:  /*if reg1 == 0 branch to L1 */

STL_C newlockvalue, locklocn; /*STL_C succeeds */
//enter CS

.......................

L1: STL_C reg, locklocn; /*STL_C fails */
BR lock; //spin on lock;
```

b) Modified Locking Code

Figure 4.13: Original and Modified Locking Code in JaVM

4.8.4 Operation using the lock unit

The queue based lock unit, requires explicit lock access and release messages to be sent to it for the locking protocol to be implemented over the mesh network. This section details the network messages that are generated when a thread executes the LDL_L and STL_C instructions and the actions that the lock unit takes on receiving the messages from the lock requestor thread.

1. As shown in Figure 4.14, the LDL_L instruction causes a thread executing on Processor1 (P1) to generate a (LOCKREQ, ACK_S/ACK_M/FWD) message to the lock unit.

2. The lock unit checks if the lock is free within the lock table.

3. The lock unit responds with either a lock grant (LOCKGNT, ACK_S/ACK_M/) or lock non-grant (LOCKQUEUE,FWD) message depending whether the lock queue (next owner queue) associated with the lock address is empty or non-empty, respectively. The LOCKGNT message implies that the lock is not being held by any other thread and the requestor thread has access to the lock. The LOCKQUEUE message (P0 is holding the lock) implies that the thread should stall until it gets an explicit LOCKGNT message from the lock unit.

4. On a LOCKGNT, the lock unit stores the id of the processor that

sent the LOCKREQ message, as the current lock owner. On a LOCKQUEUE the lock unit stores the processor id in the lock queue associated with the lock address.

5. The LOCKREQ message is generated on a LDL_L instruction irrespective of whether line is in L1 cache or not. On receiving a LOCKGNT at the processor, the thread sets an entry within the lock table local to the processor, to indicate that a LOCKREQ operation has completed and the state of the L1 cache line is either S or M. On a LOCKQUEUE message, the context on which the thread is executing stalls. Because the LOCKREQ is similar to cache miss, another thread can be scheduled to start executing on the same processor core as soon as the LOCKREQ is generated, if multithreading is supported.

6. On a STL_C, the thread (assuming P0 is the lock owner and P1 received a LOCKQUEUE) first checks if a corresponding entry exists within the local lock table and the cache. Entries from the local lock table and the cache are invalidated on a write miss operation (UP or RD_EX) to the same cache line as the lock word. The lock unit in L2 on seeing the UP or RD_EX will clear the id of the present lock owner from its lock queue. If any one of the conditions fail, the STL_C operation is aborted and the LDL_L instruction is retried. On success of both conditions, i.e. entry present within the local lock table and line present in the cache (either in S or M state), a lock release (LOCKFREE,FWD) message is sent to the lock unit and the L1 cache transits to M state. The lock unit removes the current lock owner from the lock queue associated with the lock address, and generates a new LOCKGNT message to the next waiting processor. If the queue is empty no message is generated by the lock unit and the L bit in the cache line at the L2 is reset.

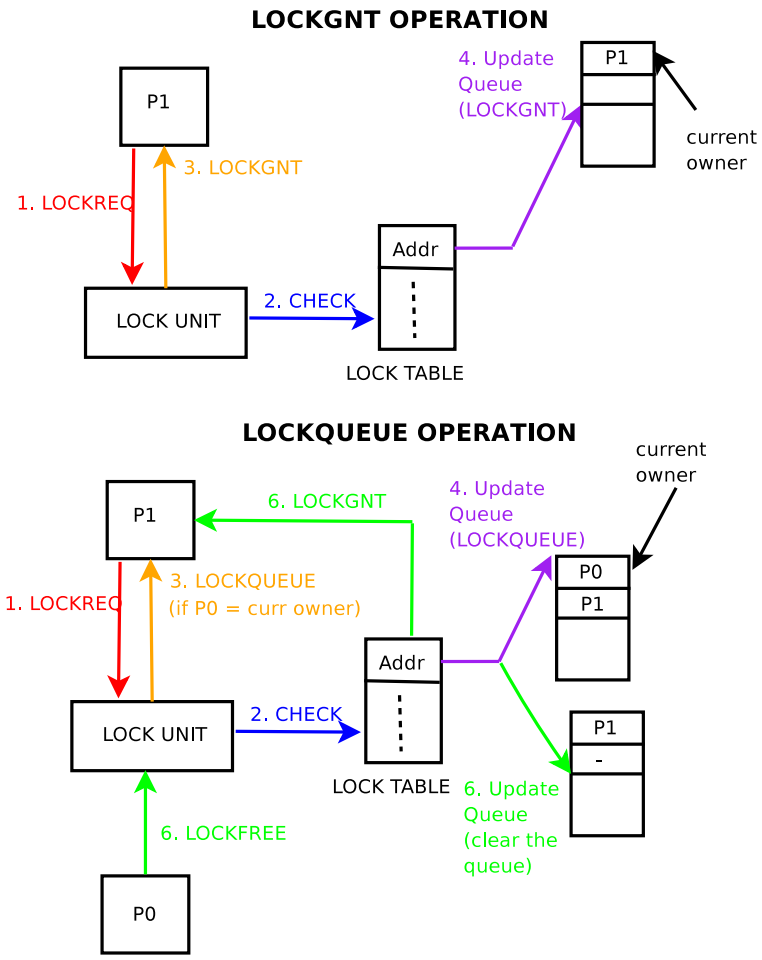

Figure 4.14: Queuing Lock Protocol

- *Advantages*

  1. Provides for serialised access to locks and reduces the traffic over the network for non-blocking synchronization cases.

  2. Serialization of locks prevents lock spinning thereby saving processor cycles and allows a multithreaded core to run another task while waiting for a lock.

- *Disadvantages*

  1. Pausing a hardware context on a LOCKQUEUE message for a long period of time may cause the OS to schedule a stalled lock queued software thread to wake up on some other hardware context. This means that on a LOCKFREE, the lock unit will send the LOCKGNT message

to the processor id that the lock queued thread was previously scheduled on, resulting in the LOCKGNT message not reaching the correct new owner. A solution to this problem would require the processor that receives the LOCKGNT message to resend a new message back to the lock unit indicating that it was a stale or incorrect LOCKGNT message. Also, the lockqueued thread that is rescheduled on another processor should retry the LDL_L instruction, i.e regenerate a new LOCKREQ message. In the current JaVM, this thread rescheduling does not happen on stalled threads and hence this problem does not arise in this system.

2. During blocking synchronization, unnecessary invalidations are generated by the locking protocol for heavily contented locks. This increases the overall traffic in the network because of multiple reads and writes to the locked cache line.

3. Delays involved in accessing and releasing the lock is greater than a snoop based LDL_L/STL_C scheme.

4. Extra storage space and hardware complexity associated with the lock unit

5. Requires that locked lines within L2 are not evicted until the L bit is reset. However, in case that all lines within a single set, within a set associative L2 are locked, i.e. the L bit is set, then one of the locked lines has to be evicted. The following solution is proposed in such a scenario: The L2 generates a LOCKCANCEL message that, clears the lock table entry and resets the L bit in its cache. The LOCKCANCEL message is broadcast over the bus. All processors that are lockqueued, will invalidate their caches if in S state in their L1, and regenerate a LOCKREQ message. If the line is in M state within the L1, the line is written back(WB) to L2.

4.8.5 Problem with stale LOCKGNT

The traversal of messages on different networks can cause race conditions in handling the acquire of a lock after it has been released. One such case is the stale STL_C problem, that arises when a thread receives a LOCKGNT message

and commits in the S state, when in reality the lock unit has granted the lock to next in line waiting thread. This is illustrated in figure 4.15. The following operations take place when a processor sends a LOCKFREE message to the lock unit and the line is in S(MO) state at the L2:

1. Consider the case, wherein processor1 (P1) gets access to the central arbiter and sends a LOCKFREE to the lock unit at the L2, for a line that is S in P1's L1 and S(MO) state at the L2. P1 transits to M state after sending the LOCKFREE.

2. The lock unit checks for a pending lock requestor (P2) and generates a (LOCKGNT_L2FWD, FWD) to P1. The LOCKGNT_L2FWD implies that the new lock owner P2 requires the M line from P1. The L2 sets the Pending(P) bit within the cache line and transitions to the P state, waiting for either an OWNERDATAUP or OWNERUP from P1. The L2 also sends a B_INV on the bus to indicate P1's write to a S(MO) line.

3. P1 releases the central arbiter on seeing the B_INV message on the bus. On receiving the LOCKGNT_L2FWD, P1 sends a LOCKGNT packet to P2, an OWNERDATAUP packet to the L2 and transits to S state. It is possible for the OWNERDATA_UP packet to reach the L2 before the LOCKGNT reaches the new lock owner. In this case, the L2 goes in S(MO) state and clears the Pending bit within the cache line.

4. If P1 now sends an UP request to the L2. It gets access to the network as it had already released the central arbiter in step 3. On seeing an UP to the locked line, the L2 will perform the same action as described in step 2 . This results in a new LOCKGNT_L2FWD and an old LOCKGNT message in flight within the network at the same time and causes a cache coherency problem (stale LOCKGNT problem). If the LOCKGNT at P2 commits after the B_INV because of the UP from P1, it sets the cache line in L1 in P2 to S state, while at P1 the state of the cache line is in M state, causing inconsistent cache states.

5. To overcome this problem, the L2 always sends the processor id of the new lock owner (in this case, P2) as well as the id of the processor that

did the WR/LOCKFREE (P1), during the B_INV in step 2. This is done so that P2 now knows that it is expecting a LOCKGNT from P1.

6. P2 on seeing another processor id during the B_INV in step 4, will realise that its incoming LOCKGNT packet is stale and will discard it when the packet arrives at one of its port slaves.

Note: This solution assumes that the B_INV in step 2 reaches all the processors before the LOCKGNT reaches the new lock owner. This condition always holds true because P1 does not release the central arbiter nor services the LOCKGNT_L2FWD packet until it sees the B_INV on the bus.

On a RD_EX/UP/LOCKFREE to M or S(SO) lines, L2 does not generate a B_INV. LOCKFREE to S(SO) lines in L2 results in an UP_RESP and the routing protocol guarantees that the LOCKGNT_L2FWD will piggyback the UP_RESP packet. However, if steps 4 and 5 occur, then P2 sees a different processor id in step 6 as opposed to its own id. This indicates that the LOCKGNT packet it receives is stale and should be discarded.

Suppose, P2 received the LOCKGNT packet before the B_INV (due to the UP from P1) from step 6 reaches it, then the LOCKGNT packet is accepted. However, when it attempts the STL_C it would fail because the cache line would have been invalidated in its L1 by this time due to the B_INV in step 6.

It is possible for P2 to accept the LOCKGNT and attempt a LOCKFREE before it sees the B_INV in step 6. In this case, the central arbiter does not grant access to P2's LOCKFREE because P1 will not release the central arbiter until it sees the B_INV generated by L2 in step 6. The processor trying to do the LOCKFREE (P2) will eventually detect that its LOCKFREE has failed and will retry the LOCKREQ instead.

4.9 Simulation Details

A cycle accurate version of JAMSIM [Hor07] was modified to include a mesh network along with the existing bus. Changes to the simulator involved the following:

- Creating ports and their associated logic for processor and L2 tiles, router logic in each tile.
- Incorporating the new cache coherency protocol.

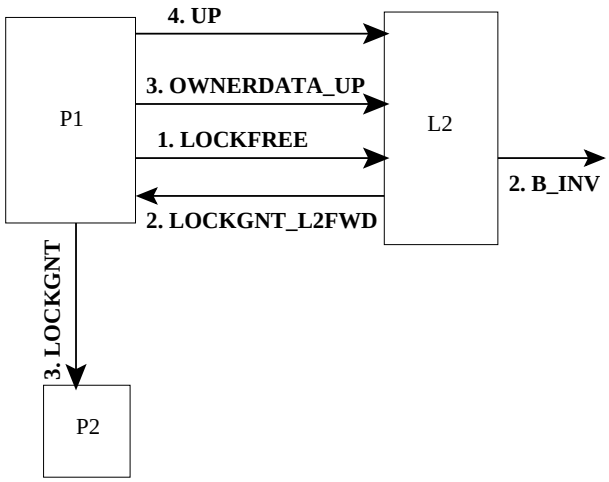

Figure 4.15: Stale STL_C case

- Incorporating a locking unit at the L2.
- Modification to the existing L2 cache controller in the bus based Jamsim, to allow for interfacing between the master and slave ports and the locking unit.

The processor components, such as pipeline, cache structure, request and writeback buffers were retained with the same functionality as that in Jamsim. All ports within a tile are clocked every cycle. State machine based port slaves and masters simulate the delays in accessing, processing and forwarding a packet at all tiles. Intra port delays are modelled by blocking one of the many simultaneous requests that arrive for a single master port. Inter port delays are modelled by having separate request and ack control signals between adjacent tiles. All port slaves within a tile contend for a lock in order to access the cache (L1 or L2) or request table or writeback buffer at any point in time. This lock implements the functionality of a single ported cache, request table and writeback buffer. Packets are sent on the network to model or simulate congestion in the network.

The following delay assumptions are made:

1. Request packets are assumed to take one clock cycle, while data packets take 2 clock cycles per hop transmission. Transmission of data packets on the mesh network is simulated by padding the data part of the packet with 0's.
2. One clock cycle for router within the processor core to compute the output port for the received packet. One clock cycle to transfer a

packet from slave input port to master output port within a tile, across the crossbar, i.e routing latency is 1 clock cycle.

4.10 Summary

A heterogeneous (mesh and bus) network tiled based CMP architecture, is discussed, highlighting on the design of a tile (processor or memory) and its interface to the OCN, issues associated with handling cache coherency protocol on two networks and finally a description of the simulation infrastructure. A dedicated hardware based lock unit is introduced within this architecture to handle acquire and release of locks without the use of a snoop based bus, as required in the bus based JAMAICA CMP. The design continues to use a central arbiter similar to that within the bus based JAMAICA CMP, to restrict requests for the same address from entering the network, thereby simplying the cache coherency protocol (removal of acknowledment messages) in comparison to directory protocol.

It is noted that the use of a central arbiter within the design restricts the bandwidth usage of the mesh network. Although the bus network is used far more sparingly in comparison to a bus-only CMP, on-chip wire delay issues continue to be an impeding factor, with the length of the bus increasing with more processors. Given the difficulties in maintaining a dedicated lock unit and the inefficiencies introduced by the central arbiter and the bus network, the next chapter looks at dual mesh tiled architecture. This new architecture does not use a central arbiter or a dedicated lock unit, and substitutes the bus network described herein with another high bandwidth based mesh network.

4.11 Summary of Request and Response Messages

Request from L1	**Response from L2**
RD_SH (RD miss)	Line in S (SO) or S(MO)/E at L2 - RD_SH_RESP (with data)
	Line in M at L2 - RD_SH_L2FWD (request forward)
RD_EX (WR miss)	Line in E at L2 - RD_EX_RESP (with data)
	Line in M or S(SO) at L2 - RD_EX_L2FWD (request forward)
	Line in S(MO) at L2 - RD_EX_RESP and B_INV (broadcast invalidate on bus)
UP (WR hit)	Line in S(SO) at L2 - UP_RESP (acknowledgement from L2)
	Line in S(MO) at L2 - B_INV
WB - eviction of M lines from L1	None
LOCKREQ	Lock available and line in S(SO) or S(MO) at L2 - LOCKGNT(with data optional as the line may be in S(SO) or S(MO) at L2)
	Lock not available - LOCKQUEUE
	Lock available and line in M state at L2 - LOCKGNT_L2FWD (request forward)
LOCKFREE	Line in S(SO) at L2 - UP_RESP and LOCKGNT_L2FWD, if lock pending
	Line in S(MO) at L2 - B_INV (new lock owner information appended) and LOCKGNT_L2FWD, if lock pending)
	no response on M line

Table 4.5: L1 Request vs. L2 Response

Request from L2	**Response from L1**
WB_PENDING - M line evicted from L2	WB
INVALIDATE - S(SO) line evicted from L2	None
B_INV - S(MO) line evicted from L2	None
RD_SH_L2FWD or LOCK-GNT_L2FWD	OWNERDATA_UP (with data) or OWNER_UP (without data in case the request originates from the same owner in case of sharing between I and D cache) to L2. RD_SH_RESP to original requestor
RD_EX_L2FWD	RD_EX_RESP to original requestor, no response to L2

Table 4.6: L2 Request vs. L1 Response

Chapter 5

A Dual Mesh CMP

All scalable cache coherent multiprocessor systems use directory protocols without a central arbiter over high bandwidth networks. While the Tiled Bus based CMP has more network bandwidth in comparison with a snoop only bus CMP, the central arbiter becomes a bottleneck in accessing the large pool of bandwidth. The need for an efficient use of the network bandwidth as well as non-reliance over a snoop bus, motivated the design of a dual mesh based CMP architecture without a central arbiter. This chapter provides a detailed insight into the architecture, cache coherency protocol and deadlock avoidance mechanism implemented in the dual mesh tile based CMP system.

5.1 System Architecture

In this chapter we describe a CMP architecture which uses two mesh networks instead of a bus and a mesh as described in Chapter 4. One of the mesh networks serves as an unordered invalidation interconnect. Invalidation on this network is supported using a broadcast tree with the memory tile designated as the root node. A mesh based invalidation network provides higher bandwidth and supports more processors as compared to a single bus based network [DYL03]. Also, it does not suffer from the wire delay problems that are associated with a single bus [KZT05]. The other mesh network, processor and memory tiles retain the same functionality as described in Chapter 4. A distinguishing feature of this architecture compared to the mesh and bus based system is that it does not use a central arbiter. This feature eliminates the arbitration delays and better utilizes the mesh network bandwidth. This architecture does not use a lock unit as

explained in chapter 4. Instead, the lock unit is replaced by a simple lock table that is used to detect stale LOCKFREE messages and reduces the hardware complexity that is associated with the lock unit described in Section 4.8. A more detailed description of this locking protocol is described in Section 5.4. The main aim of simulating this architecture was to compare its performance with that of the mesh and bus scheme. Figure 5.1 shows the architecture of this CMP for a 16 processor configuration.

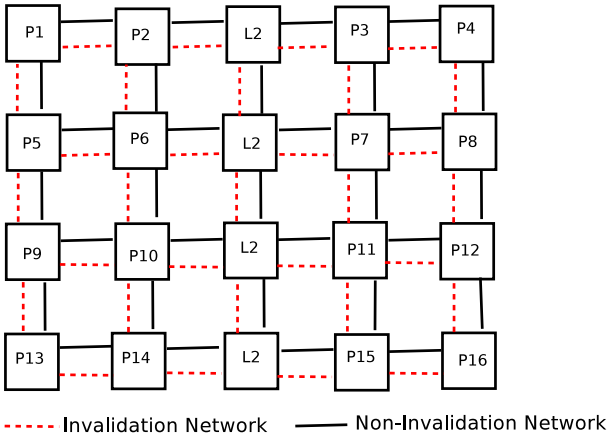

Figure 5.1: Dual Mesh CMP

5.2 Deadlock Avoidance

While removal of the central arbiter unit allows better utilization of the mesh network bandwidth, it also generates more traffic that might eventually lead to deadlock because of buffer space constraint at the L2 and processor tiles [HGR07][HP07]. One option to avoid deadlocking is to provide separate channels or paths for requests and responses. In this system, responses that cause the L2 to transit from the PENDING(P) state to any other state are considered critical. This is because the L2 waits on such responses, stalling requests for the same address until the response arrives. Apart from these critical messages, all responses generated by L2 are also considered critical. This is because requests that piggyback response messages also cause the L2 to transit to P state. Therefore, all critical messages are sent on a separate channel, the sink channel, that is provided for each port handler, i.e. every port master and slave, on a tile. The sink channel is a non-blocking buffer that accepts critical responses

and guarantees consumption of these messages. Both sink and non-sink channels share the physical link that is attached to the port handler. Figure 5.2 shows a tile having both sink and non-sink channels. Irrespective of whether the tile is a processor or L2, there is only one sink buffer per port handler. L2 tiles ensure strict First-in-First-Out (FIFO) ordering on output sink channels, i.e. messages are sent out in the same order that they are generated. The master sink channel has a different buffer structure compared to the master non-sink channel. It contains two header buffers and a single data buffer. This is because, there are instances within the cache coherency protocol that may cause a slave sink channel to generate two response packets from a single request, that need to be sent out of the same master sink channel. If the master sink channel has space for only a single response message, then the second response message that is generated by the slave sink channel will block until the master sink channel releases the first response message. This scenario of blocking the slave sink channel can potentially cause deadlock and will be explained in more detail in Section 5.3.5.

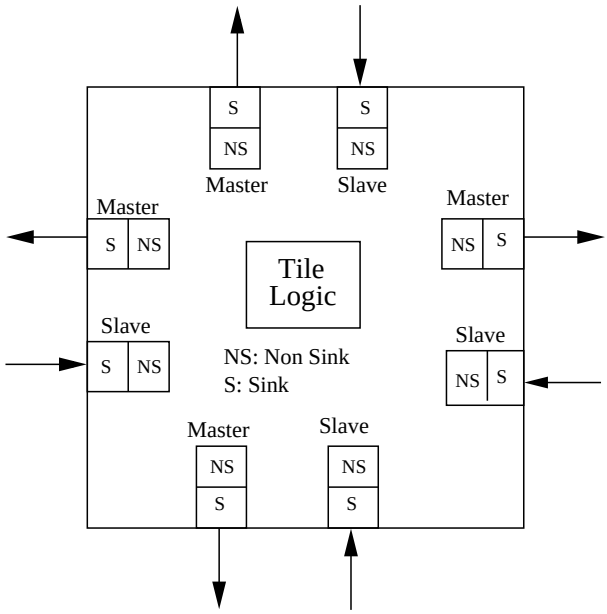

Figure 5.2: Tile with Sink channels

5.3 Cache Coherency Protocol

As described in Chapter 4, the cache coherency protocol uses directories at the L2 for read and write requests to non-widely shared data, while the snoop protocol is

used solely for broadcast purposes. In the mesh and bus based CMP, a requestor signals the completion of its request by releasing the central arbiter. In this CMP design, the lack of a central arbiter means that acknowledgement messages are required to ensure completion of reads and writes. This is an overhead to the existing cache coherency described in Chapter 4. Almost all read and write requests have retained the same naming convention, with request and response messages being designated as a combination of the *type* and *control* field from the packet header, as described in Chapter 4. All requests are sent on the non-sink channels, while most responses are sent on the sink channels. All messages that are sent on the sink channel are explicitly stated, otherwise it is assummed that they are sent on the non-sink channel. Cache lines retain the same states in both L1 and L2 as described in Chapter 4. Subsequent sections describe the actions taken to ensure cache coherency for read and write requests.

5.3.1 Read Miss

On a Read Miss, the processor (requestor) sets a READ request within the request table and sends a RD_SH message to the L2. The L2 takes the following actions depending on the state of the line in its cache:

- **Line in E or S state**

The L2 sends (RD_SH_RESP, ACK_S) packet on the sink channel, indicating that the READ request at the processor should commit the request in S state in the L1 cache.

- **Line in M state**

The L2 forwards the request (RD_SH_L2FWD, FWD) on the sink channel to the processor that holds the line in M state within its L1 (owner cache) and transits to the P state, waiting for an acknowledgement from the owner and the requestor. The owner cache responds with a (OWNERDATAUP,ACK_S) or (OWNERUP, ACK_M) packet, sent on the sink channel to the L2, and a (RD_SH_RESP_RAW, ACK_S) packet sent on the sink channel, to the requestor L1. The RD_SH_RESP_RAW packet indicates that the read response packet is the result of an indirection operation, i.e. the read is a part of a 'Read-After-Write'(RAW) operation and had to be forwarded to the owner processor in order

to fetch the read data. The primary difference between a RD_SH_RESP_RAW packet and the RD_SH_RESP is that the requestor processor needs to generate a acknowledgement packet (ACK - sent on the sink channel) back to the L2 in order to indicate that the read operation has completed. The requestor commits the read request in the S state within its L1 cache on receiving the RD_SH_RESP_RAW packet. The L2 on receiving both the OWNERDATAUP and the ACK packets, transits to the S(MO) state. The need for an ACK message on a READ request to a M line in L2 is illustrated in figure 5.3. There is no certainty on whether the read from P1 is a part of a RAW or a 'Write-After-Read'(WAR) operation. If the read was part of the RAW, as indicated by the *type - RH_SH_RESP_RAW* field within the packet, then the cache line in P1 is not invalidated. If the read was part of a WAR, as indicated by the *type - RD_SH_RESP* field, then the cache line in P1 will be invalidated. The ACK sent by P1 after it receives the RD_SH_RESP_RAW packet serves as a serializability measure ensuring that the RD_SH_RESP_RAW packet always commits in the S state within the cache. If no ACK packet was generated by P1, then P1 has no means of guaranteeing that there was no intervening write before it received the RD_SH_RESP_RAW packet and hence cannot ascertain that it can commit the cache line in S state. Therefore the RD_SH_RESP_RAW along with the ACK packet solves the problem of unnecessary invalidations on reads to modified data.

Note, that an OWNERUP packet indicates that the requestor and the owner processor are identical and that the request was generated by either the D or I cache and the data resides in either the I or D cache. On receipt of an OWNERUP packet the L2 transits to the M state and changes the owner id field within the cache line to reflect the new owner processor, in this case, the same id as the original owner processor.

On some instances, the owner processor might evict the data from its write-back buffer in order to respond to the RD_SH_L2FWD. In this case, the owner processor generates an (OWNERUP, ACK_M) packet with the control field set to ACK_M indicating that the L2 should set the cache line state as M and (RD_SH_RESP_RAW, ACK_M) packet with its control field set to ACK_M indicating that the requestor L1 should also set the cache line state as M. The requestor on receiving the RD_SH_RESP_RAW packet commits the read request in M state within the L1. The L2 on receipt of the OWNERUP packet sets the cache line state to M, and updates the processor id field with that of the requestor

id. There is no need for the requestor to generate an ACK because, even if the OWNERUP packet reaches the L2 much before the RD_SH_RESP_RAW packet reaches the destination, and the L2 generates another forwarding request to the present owner of the line, the dimension order routing protocol and the in order sink channels guarantee that the RD_SH_RESP_RAW will never be blocked.

If the owner processor has already written back the line before the RD_SH_L2FWD packet arrived, then the owner discards the RD_SH_L2FWD. L2 on receiving the WB packet will generate a RD_SH_RESP back to the requestor.

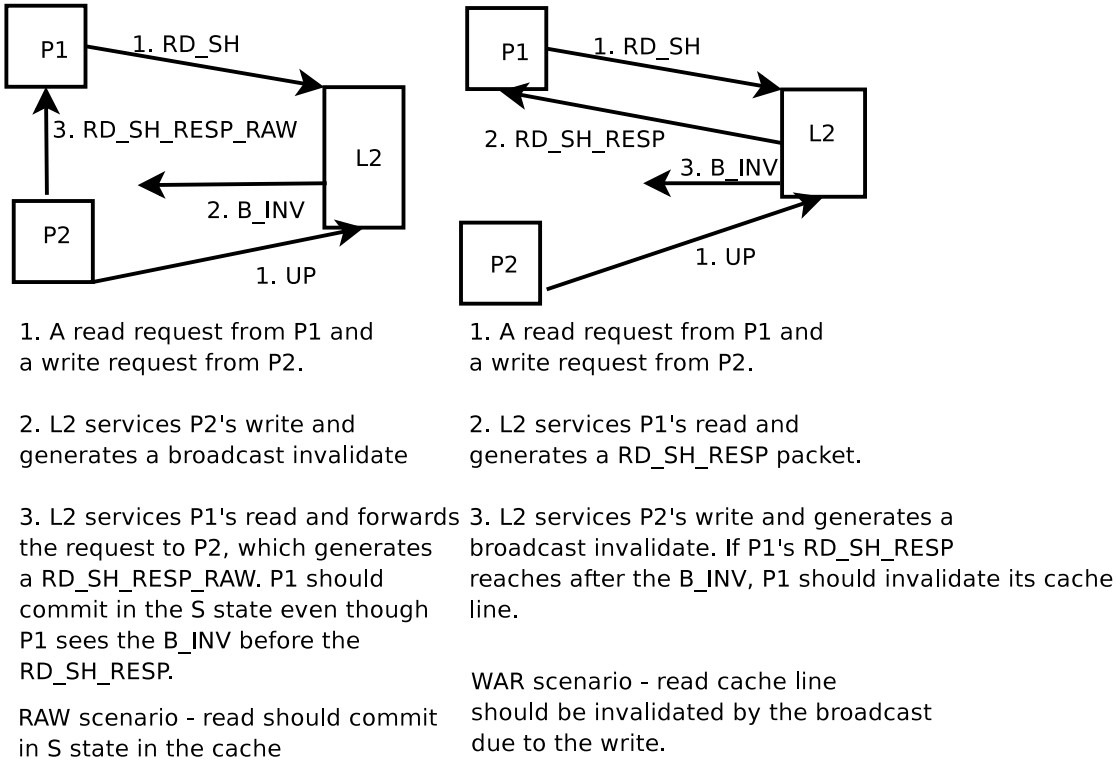

Figure 5.3: RAW and WAR operation

5.3.2 Write Miss

On a write miss in the L1 cache, the requestor creates a write request within the request table and sends out either an UP or RD_EX packet to the L2. The L2 takes the following actions depending on the state of the line in its cache:

- **Line in E state**

The L2 generates a (RD_EX_RESP, ACK_M_NORESP) packet to the requestor, on the sink channel, indicating that response should commit in M state and no ACK packet is required. The owner id field within the L2 cache line is set to the processor id of the requestor and the L2 transits to M state. Note, an UP for an E line in cache is treated as a RD_EX request by the L2. The requestor on receiving the RD_EX_RESP checks the control field and commits the write request in the M state within its L1 cache.

- **Line in S(SO) state**

On a RD_EX, the L2 generates a (RD_EX_L2FWD,FWD) packet that is sent on the sink channel to the owner processor, and transits to the P state. The owner processor on receiving the RD_EX_L2FWD packet takes the following actions:

  - If the line is in the L1 or WB buffer, the owner processor sends a (RD_EX_RESP, ACK_M) packet with the control field set to ACK_M to the requestor and invalidates its L1 or clears the WB buffer. ACK_M indicates that the requestor should generate an acknowledgement to the L2. The requestor processor on receiving the RD_EX_RESP packet generates an ACK packet to the L2, signalling the completion of the write. On receipt of the ACK packet, the L2 changes the owner id field within the cache line to reflect the processor id of the reqeustor and transits to M state.

  - If the line has been evicted because of space constraints within the L1, the owner processor resends a (RD_EX_L2FWD_FAIL, FWD) packet back to L2, on the sink channel. On receipt of this message the L2 generates a (RD_EX_RESP,ACK_M_NORESP) packet for the requestor, changes the owner id field and transits to M state (Figure 5.4).

On an UP, the L2 generates an (UP_RESP, ACK_M_NORESP) packet to the requestor on the sink channel and transits to M state. The requestor on receiving the UP_RESP packet commits the request in M state in the L1.

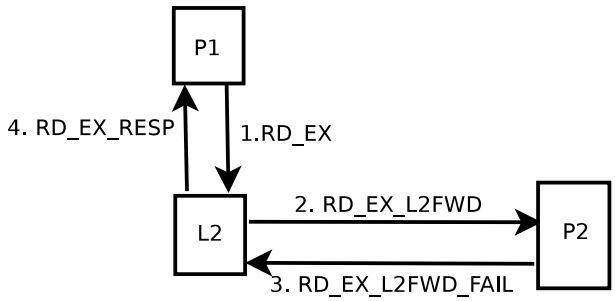

Figure 5.4: RD_EX to S(SO) line evicted from L1

- **Line in S(MO) state**

On a RD_EX, the L2 broadcasts an invalidation packet (B_INV) on the invalidation network and generates a (RD_EX_RESP,ACK_M_INVPENDING) packet for the requestor, changes the owner id field and transits to M state. On an UP, L2 broadcasts an invalidation packet on the invalidation network, changes the owner id field and transits to M state. The ACK_M_INVPENDING control field indicates that the L1 requestor should not commit the request in the M state until it receives the B_INV.

An invalidation message contains the id of the processor responsible for the invalidation, the type of request that generates the invalidation, i.e., RD_EX, UP or LOCKFREE, the thread id-if the invalidation is due to a SIRQ request and the address of the cache line that needs to be invalidated. All processors on receiving an invalidation message at the invalidation port take the following actions:

1. If there is a request for the same line address within the request table, the processor checks the type of request and takes the following actions:

   (a) If the request is a RD_SH, then an invalidation flag is set within the request table entry indicating that the line should be invalidated within the L1 on the arrival of the response for the RD_SH request. Depending on the type of response for the RD_SH request the following actions take place:

   - If the response for the RD_SH is a RD_SH_RESP_RAW packet, the flag is discarded and the line is committed in the state as dictated by the control field of the packet.
   - If the response is a RD_SH_RESP, then the cache line is invalidated, but the request is committed with the read value

written to the registers. The WAR case in Figure 5.3 shows why there is a need to invalidate the cache line on receipt of a RD_SH_RESP packet when the flag is set within the request table entry. However, it is also possible that the response for the read might have been a part of a RAW, e.g. W-R-R case, in which case the L2 will generate a RD_SH_RESP response packet for the second READ request. But, this RD_SH_RESP causes the cache line to be invalidated because of the invalidate flag that was set by the second READ on seeing the original write. Therefore, in the case wherein the RD_SH_RESP packet finds the invalidate flag set, unnecessary invalidations *might* occur. Note, that a full directory protocol that supports ACK messages to be generated during invalidations prevents this scenario, because the processor generating the read never receives an invalidation message.

(b) If the request is a RD_EX and the id on the invalidation network matches the processor id, a flag is set within the request table entry to indicate that the invalidation for the RD_EX request is complete. The processor on receiving a RD_EX_RESP packet with the control field set to ACK_M_INVPENDING, checks this flag and then commits the request in M state.

(c) If the request is an UP and the id on the invalidation network matches the processor id, the UP request commits in M state. If there is no match between the ids, the line is invalidated from the L1 cache, the UP request is changed to RD_EX and a the flag is set within the request table entry to indicate this change.

2. If there is no pending request present for the invalidation address, the L1 and WB buffers are checked and the cache line is invalidated or cleared, if present.

- **Line in M state**

On a RD_EX, the L2 forwards the request as a RD_EX_L2FWD packet to the owner processor and transits to P state. The owner processor on receiving the RD_EX_L2FWD packet, sends a RD_EX_RESP packet with the control field set

to ACK_M to the requestor and invalidates the line from the L1 cache or write-back buffer. The requestor on receiving the RD_EX_RESP packet sends an ACK message to the L2 indicating that write operation has completed and commits the request in M state. The L2 on receiving the ACK packet, updates the owner id field with the processor id associated with the requestor and transits to M state (Figure 5.5).

If the line is not in the cache or writeback buffer, the owner processor assumes that a writeback operation has taken place and discards the RD_EX packet. The L2 on receiving the WB packet will service the RD_EX request as a write to an E line in L2.

On an UP, the L2 assumes that the UP is a RD_EX and services it as described above.

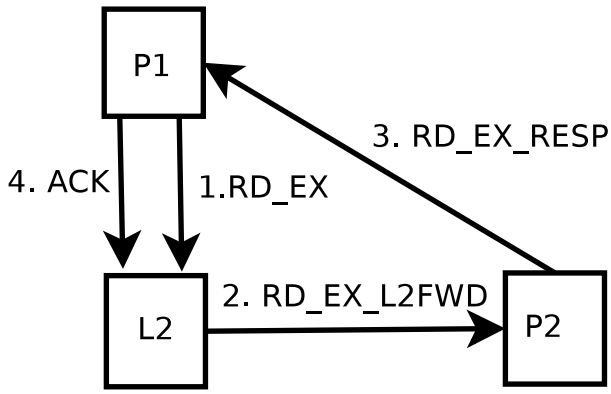

Figure 5.5: RD_EX to M or S(SO) line in L2

5.3.3 Writebacks and Evictions

Modified lines that are evicted by the L1 caches are written back to the L2 in the form of a writeback (WB) packet that is sent on the sink channel. The L2 on receiving the WB packet commits in the E state, if there is space in the L2 cache, or writes the line back to MM.

The L2 sends a WB_PENDING packet on the sink channel to the owner processor on evicting a M line from the cache, forcing a WB response packet from the processor.

5.3.4 Stale UP Problem

The limited directory scheme and the subsequent broadcast that is required on directory pointer index overflow causes problems in detecting UP requests that are no longer valid when they arrive at the directory. Consider the following scenario as shown in Figure 5.6. Processors, P1 and P2 send UP messages and P3 sends a RD_SH message to the L2. Suppose the L2 services P2's UP before P1, it sends a B_INV message and commits in M state. P1 on seeing the B_INV, sets a flag within its request table indicating that the request has changed from an UP to a RD_EX. L2 then services P3's request and transists to S(MO) state. L2 now services P1's UP request, and sends a B_INV packet on the invalidation network with the type field in the packet set to UP, indicating that the B_INV was in response to an UP request. This is a stale UP case and the L2 never detected P1's request change from an UP to a RD_EX.

In such a situation, P1 detects that the L2 has serviced a stale UP packet and sends a RD_EX packet to the L2 with the control field set to UP_INV, on the sink channel. On receipt of this RD_EX packet, L2 transits to E state, clears the owner id field and sends a RD_EX_FAIL packet on the sink channel back to P1 acknowledging that it has detected the stale UP. P1 now sends a RD_EX packet to the L2, instead of an UP.

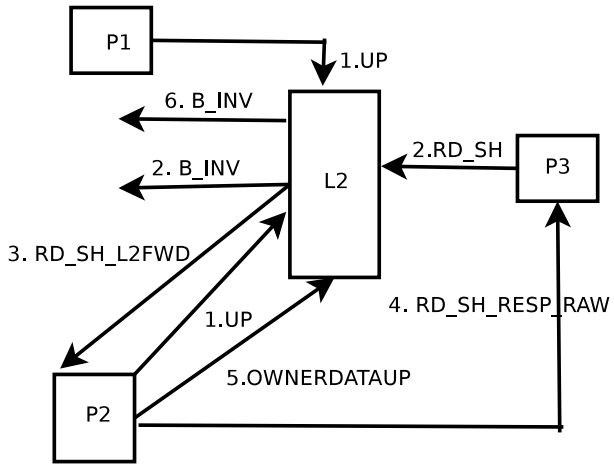

Figure 5.6: Stale UP

5.3.5 Sink Deadlock

As mentioned in Section 5.2, deadlock can also occur on the sink channel in the following scenario as shown in Figure 5.7(a):

1. P1 receives a RD_SH_L2FWD from L2, say for address A. P1 sends an RD_EX packet with control field set to UP_INV packet to L2, say for address B.

2. P1 generates a RD_SH_RESP_RAW for A and inserts the packet into the sink master. The OWNERDATAUP packet is still pending in P1's sink slave (green dashed line). L2 in the mean while, generates a RD_EX_L2FWD for address C for P1 and inserts the packet into its sink master.

3. L2 sink slave generates a RD_EX_FAIL packet (blue dashed line) in response to the UP_INV packet from P1 and requires the same sink master as that of the newly generated RD_EX_L2FWD packet.

4. P1 cannot insert the OWNERDATAUP packet into the sink master, because the sink master is blocked with the RD_SH_RESP_RAW packet. P1's sink master cannot send the RD_SH_RESP_RAW packet out because the L2's sink slave is blocked with the RD_EX_FAIL packet. L2 sink slave cannot send the RD_EX_FAIL packet out because the L2's sink master is blocked with the RD_EX_L2FWD packet. L2 sink master cannot send the RD_EX_L2FWD packet out because P1's sink slave is blocked with the OWNERDATAUP packet. This leads to a formation of a cycle and causes deadlock.

In order to break this cycle, all sink masters are provided with two header buffers and a single data buffer, as shown in Figure 5.7(b). When P1 generates the OWNERDATAUP packet, it inserts it into the second header buffer of the sink master. The data buffer contents remain unchanged because both RD_SH_RESP_RAW and OWNERDATAUP use the same data. P1 then serves the RD_EX_L2FWD packet and L2's sink master accepts the RD_EX_FAIL packet.

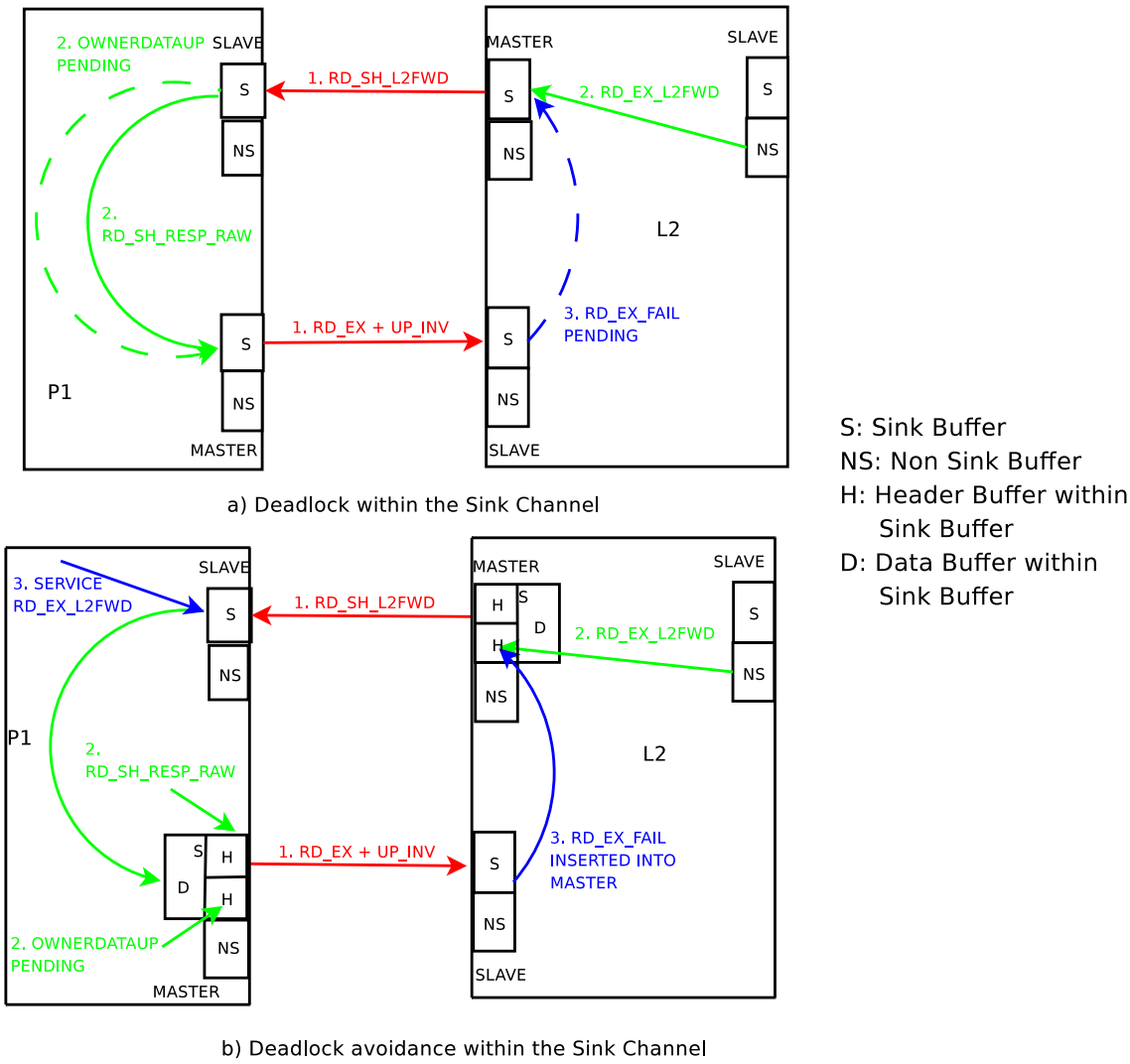

Figure 5.7: Deadlock Avoidance within the sink channels

5.4 Locking Protocol

In Chapter 4, locks were accessed and released using a dedicated hardware locking unit. However, the disadvantages of the scheme mentioned in section 4.8.4, motivated the need for a locking scheme that allows for accessing and releasing locks on a non-ordered network. The new locking scheme continues to use a lock table at the L2, to track *potential would be* lock owners and detects any stale lock access requests. Unlike the previous locking scheme, there is no queue maintained to track the list of lock requestors (processor ids) and therefore the hardware complexity associated with each lock entry is reduced. In this scheme, lock requestors need to retry for the lock on receipt of a failed lock response.

The lock table structure is shown in figure 5.8. Each entry consists of a lock address and a bit vector whose length is equal to the number of processors. Each cache line within the L2 contains two bits to detect lock operations, an L bit to indicate that the line is locked and a W bit to indicate a write operation has been performed on a locked cache line. The lock table along with the L and W bits on the L2 cache line are used to determine the success or failure of a lock operation at the L2. As with the L2, the L1 also contains a L bit on each cache line indicating that the line is locked. The L1 also contains a local lock table to keep track of lines that are potentially marked for lock ownership. The working of the locking protocol is as follows:

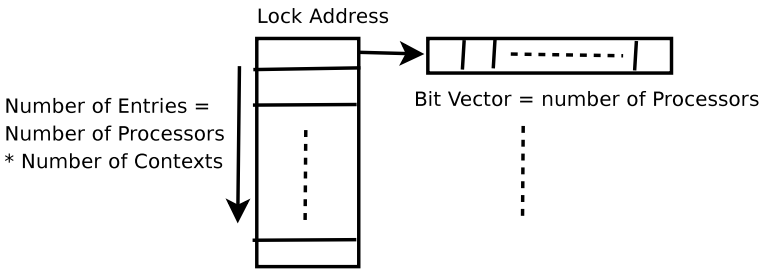

Figure 5.8: Lock Unit at L2

- On a LDL_L instruction and a L1 cache miss, the processor generates a normal read request to the L2. If the read request is to a cache line that has either the L or W bit set, the L2 checks the lock table and takes the following action:

  1. If the lock entry is present in the lock table, a flag is set within the bit vector to indicate the processor id that requested the lock.
  2. If there is no lock entry present in the lock table, a new entry is created and the bit vector associated with that entry is set to indicate the processor id that requested the lock.

- The L2 responds to the read (LDL_L) request as it would do for any normal read. On receipt of the read response from the L2 or a cache hit, the processor sets an entry within its local lock table and attempts to perform a STL_C.

  Note, a read to a line in L2 where both L and W bits are not set is treated as a normal read, even though this read might have been generated due a LDL_L instruction.

- On a STL_C instruction, the processor checks it's L1 cache if it in the S state and a correponding enry exists within the local lock table. If both conditions are satisfied, it sends a LOCKFREE message to the L2. No LOCKFREE message is generated on a M line in L1. STL_C on a line invalidated in cache implies that the lock operation has failed and that the processor should retry the LDL_L instruction in order to access the lock.

- On receipt of a LOCKFREE message, the L2 checks the lock table and takes the following actions:

  1. If a lock entry is present and the processor id in the LOCKFREE message has a corresponding flag set in the bit vector associated with the lock entry or if there is a match with the owner id field on the L2 cache line, then the LOCKFREE is assummed to be a success. The L2 clears the lock table entry and sets the L and W bits on the cache line; transits to M state; and sets the owner id field to the processor id that generated the LOCKFREE message. The LOCKFREE processor on receiving either a RD_EX_RESP or UP_RESP or seeing a B_INV on the invalidation network, will clear its local lock table and commit the STL_C instruction.
  2. If a lock entry is present and the processor id does not match with the flag set in the bit vector, the LOCKFREE is assumed to be a failure and a LOCKFREE_FAIL message is sent to the processor id that generated the LOCKFREE message. On receipt of a LOCKFREE_FAIL message, the receiving processor clears the lock table and invalidates the cache line, if present in the L1, so that the STL_C instruction on retry will fail within in the L1.
  3. If no lock entry is present and the W or L bit is set the the LOCKFREE is assumed to be a failure.
  4. If a lock entry from the L2 lock table is evicted because of space constraints, then the L and W bits on the L2 cache line is reset. Now, if a LOCKFREE arrives for the same cache line and the state of the cache line at L2 is S(MO), then L2 will accept the LOCKFREE. On receiving the B_INV in response to the LOCKFREE, the processor will check the state of the cache line in its

L1. If the line is in S state within the L1, the LOCKFREE succeeds, otherwise it is considered a failure. On detecting a failure, the processor will send a LOCKFREE_FAIL message back to the L2. On receipt of the LOCKFREE_FAIL message, the will reset the lock table entry and clear the L and W bit flags on its cache line.

Figure 5.9 shows the various combinations of lock entry, W and L bits, required for the success of a LOCKFREE or a LOCKFREE_FAIL. 1 indicates LOCKFREE succeeds, 0 indicates LOCKFREE_FAIL is generated. It also shows the next state transitions associated with other types of requests, such as reads, writes and writebacks. Certain states are invalid, these states are indicated by having no next state (NS) entry.

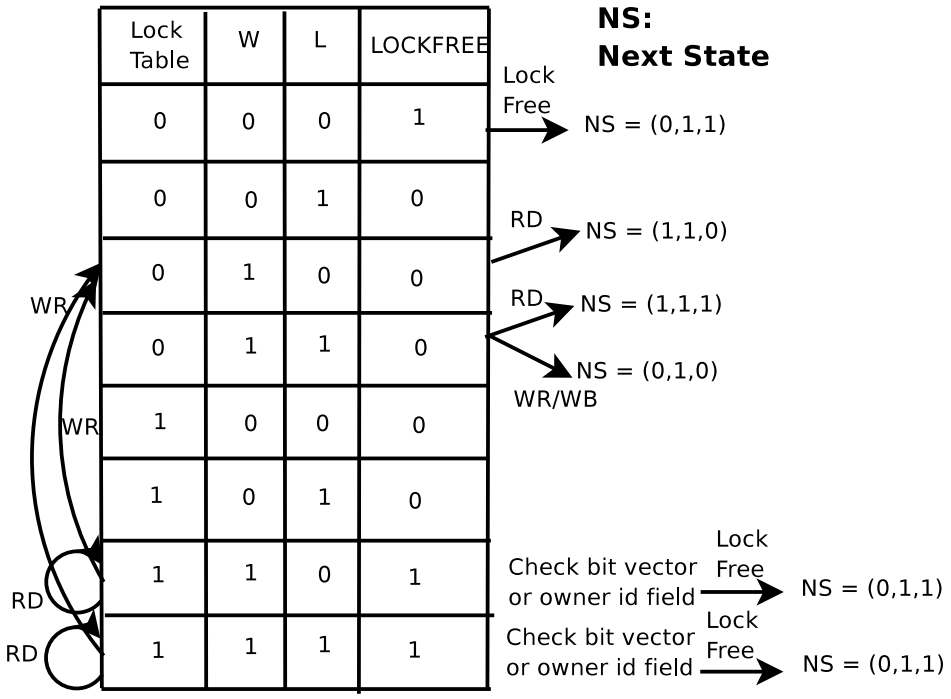

Figure 5.9: A state table describing the combinations for LOCKFREE to succeed

The reason for having an additional bit, i.e the W bit, along with the L bit at the L2, is to detect stale LOCKFREE messages. This is illustrated by the following example as shown in Figure 5.10.

1. Assume Processor1 (P1) and P2 both have the line in S state in L1. Both attempt the LOCKFREE at the same time. Because P1 is closer to the L2 tile, its LOCKFREE message reaches first and succeeds.

2. L2 sends a B_INV message because the line was in S(MO) state and sets both W and L bits, sets the the owner id field to P1 and transits to M state. P1's L1 has the line in M state on seeing the B_INV.
3. If P1 does a writeback, L2 will transit from M to E state and clears the L bit, but not the W bit.
4. P3 and P4 generate RD_SH messages for the same line
5. The L2 sets an entry in the lock table because the W bit was set and responds with a RD_SH_RESP messages to both P3 and P4 and transits to S(MO) state.
6. If, now say, P2's LOCKFREE finally arrives at the L2, it is detected as a stale LOCKFREE because, the lock table entry does not contain a flag set that is associated with P2 nor does the owner id field on the L2 cache line match with P2.

One possible case wherein a stale LOCKFREE can't be detected is, say if L2 requires to writeback a M line with L or W bit and generates a WB_PENDING request to P1. On receiving the WB packet from P1, L2 puts the line(data) into the write queue associated with the memory controller, invalidates the cache line and clears both L and W bits. If now, one or more processors request the same line, the L2 checks the write queue and brings the data into the cache in the E state. Because of the multiple read requests for the same line, L2 eventually transits from E to S(MO) state. Now, if a stale LOCKFREE arrives from P2, L2 assumes that the LOCKFREE has succeeded. A possible solution for this case is for P2 to detects that its LOCKFREE failed when it sees the B_INV due to P1's LOCKFREE. If P2 receives any other response from the L2 apart from LOCKFREE_FAIL, it should generate a message back to the L2 indicating that the it is a stale LOCKFREE and the L2 should transit back to E state.

5.5 Simulation Details

The simulator model used for the Mesh and Bus architecture was extended to include the following new features:

- Buffering and packet processing within the sink channels

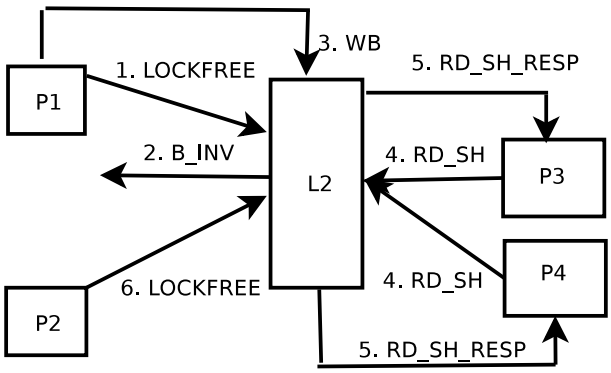

Figure 5.10: Stale LockFree Detection at L2

- Contention for intra tile sink and non-sink masters from both sink and non-sink slaves, by providing a round-robin scheme that schedules only one slave to access the master, while the other slaves are blocked.
- Port controller logic that multiplexes between sink and non-sink slaves and masters for accessing the physical channel attached to a port.

Apart from these changes all delays and assumptions that were made in the mesh and bus based simulator were retained.

5.6 Summary

This chapter overviews the architecture of the dual mesh tiled CMP, concentrating on the interconnect topology and the cache coherence protocol used. It highlights the cache coherency problems that are associated without the use of a central arbiter (generation of ack messages for RAW and WAR scenarios and detecting stale UPs), network deadlocking scenarios that were encoutered when simulating this architecture and the corresponding measures taken to overcome these problems. Instead of using a dedicated queue lock unit, a simple lock table along with an additional bit added to the L2 tag was used to handle acquire and release of locks. This locking scheme avoids some of the disadvantages and is simpler in design and complexity in comparison to the one described in Chapter 4. This architecture better utilises the bandwidth available on the mesh network and mitigates the wire delay issue.

The next chapter provides a quantitative evaluation of the dual mesh, and mesh and bus architectures w.r.t a single bus based JAMAICA CMP.

5.7 Summary of Request and Response Messages

Tables 5.1 and 5.2 summarize the request and response messages that are generated by the dual mesh cache coherency protocol and the channels (sink or non-sink) that they are sent on.

L1 Request	**L2 Response**
RD_SH (non-sink)	Line in S (SO) or S(MO)/E at L2 - RD_SH_RESP (with data on sink)
	Line in M at L2 - RD_SH_L2FWD (request forward on sink)
RD_EX (non-sink)	Line in E at L2 - RD_EX_RESP (with data on sink)
	Line in M or S(SO) at L2 - RD_EX_L2FWD (request forward on sink)
	Line in S(MO) at L2 - RD_EX_RESP and B_INV (broadcast invalidate on mesh)
UP (non-sink)	Line in S(SO) at L2 - UP_RESP (acknowledgement from L2 on sink)
	Line in S(MO) at L2 - B_INV
WB (sink)	None
LOCKFREE (non-sink)	Line in S(SO) at L2 - UP_RESP
	Line in S(MO) at L2 - B_INV
	no response on M line - Error
RD_EX with UP_INV (sink)	RD_EX_FAIL (sink)

Table 5.1: L1 Request vs. L2 Response for Dual Mesh

Request from L2	**Response from L1**
WB_PENDING (sink)	WB (sink)
INVALIDATE (sink)	None
B_INV (invalidate mesh)	None
RD_SH_L2FWD (sink)	OWNERDATA_UP (with data on sink) or OWNER_UP (without data on sink). RD_SH_RESP_RAW to requestor (sink). Requestor responds with ACK (sink).
RD_EX_L2FWD (sink)	RD_EX_RESP (sink) to original requestor. Requestor responds with ACK (sink)

Table 5.2: L2 Request vs. L1 Response for Dual Mesh

Chapter 6

Results

This chapter presents the performance of the dual mesh and mesh and bus architectures, using kernel based multithreaded benchmarks from the Java Grande Benchmark suite. It studies the absolute scalability and the relative performance improvement by comparing the execution times of the benchmarks running on both the architectures.

6.1 Benchmarks

This section describes the evaluation of the system using some of the kernel based Java Grande benchmarks [BSW+99][SBO01]. Kernel benchmarks are used to determine the performance related properties of a real application [CSG99]. The description of the five multithreaded kernel benchmarks used are as follows:

- LUFact

  LUFact is used to solve a linear system of equations that are represented in an 500X500 matrix format. It relies on dividing the matrix into two triangular matrices, upper and lower, and in turn factorizing each of them into sub-matrices, until the matrix is no longer factorisable and the computation is performed sequentially. Only the factorization part is done in parallel.

- Series

  The Series benchmark computes 1000 Fourier coefficients of the function given by the equation $f(x) = (x+1)^x$ over the interval $[0, 2]$. The

iterations of a loop that calculates the Fourier coefficients are independent and are distributed as threads over the available processors.

- SOR

  The SOR benchmark, Successive Over Relaxation algorithm, is used in grid computing to calculate the solution to a partial differential equation [Fen01] and performs 200 iterations over a 500X500 grid. For parallelization purposes, the grid is arranged as alternating red and black squares. Three loops are used, the first one performs the iterations, the second and third inner loops iterate over the red and black squares respectively, updating the value of a red square using the values from four adjacent black squares and vice versa. Each of the inner loops is partitioned into threads to ensure that only the red or black elements are being updated at any point in time [Hor07]. All threads synchronize before the next update can start and ensure that during an update the previous value of the adjacent square is being read.

- Crypt

  The crypt benchmark performs encryption and decryption using the IDEA (International Data Encryption Algorithm) over a 50000 byte array. The algorithm contains two loops whose iterations are independent and can be divided among threads.

- Sparse

  The Sparse benchmark performs sparse matrix multiply using a 10000X10000 unstructured sparse matrix grid that is stored in compressed row format.

6.2 Simulator Configuration

In order to evaluate the performance of the mesh and bus and dual mesh architectures each of the five benchmarks were run on cycle accurate simulators that model four architectures namely, mesh and bus; dual mesh,; single bus and perfect memory; with a varying number of processors. The mesh and bus and dual mesh architectures were simulated using a modified version of the JAMSIM

simulator as mentioned in Chapters 4 and 5. The single bus and perfect memory architectures were simulated using the JAMSIM simulator as described in Chapter 3 [Hor07]. The benchmarks were statically compiled into an optimized version of the Jikes RVM boot image in order not to simulate the dynamic class loading feature which is unnecessary for performance measurements.

Firstly, it was necessary to decide if the processors needed to be configured in single or dual context mode. Figure 6.1 shows the decrease in execution time for all benchmarks run on a 16 processor set up for both dual-mesh and mesh-bus configurations of the simulator, respectively. Apart from SOR, that has better performance with single context, all other benchmarks perform better on using dual context as compared to a single context JAMAICA core. Therefore, the simulations were performed using dual context JAMAICA processor cores.

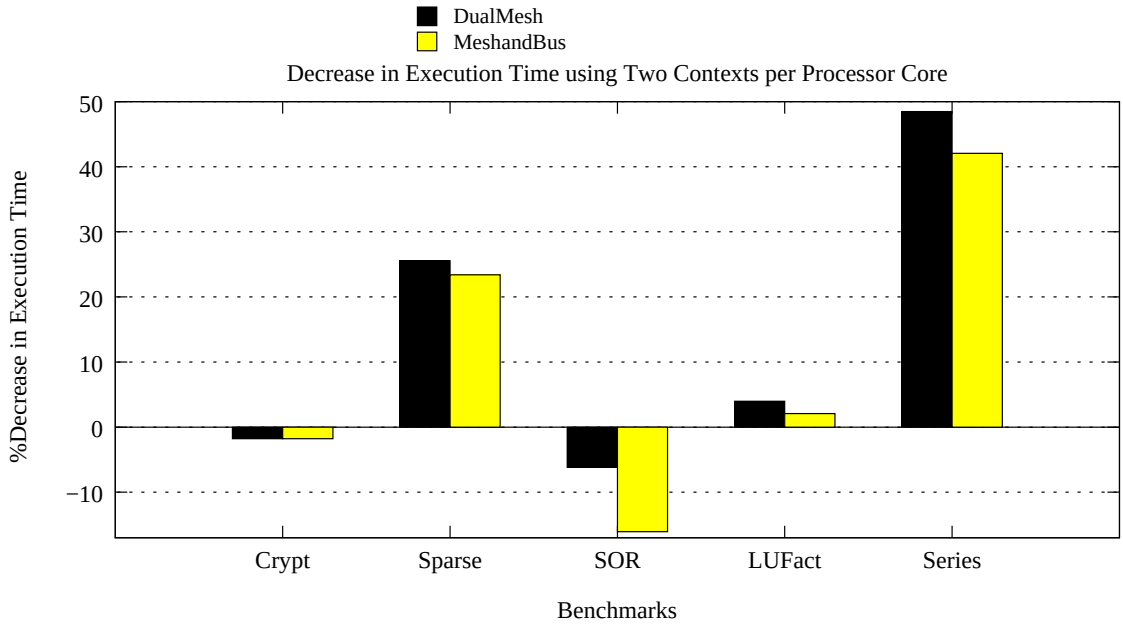

Figure 6.1: Performance Improvement using Dual Context Processor Cores vs. Single Context

Table 6.1 shows the various processor and L2 tile combinations.

Table 6.2 shows the access times (in processor clock cycles) for different cache sizes and the main memory delay assumed [MB07].

Simulation configurations for the four architectures that were modelled are described as follows: For all simulation configurations, dual context JAMAICA cores were assumed.

Table 6.1: Processor and L2 Combinations

Number of Processors	Number of L2 Tiles
1	2
2, 4 , 8	2
16	4
32	8
64	16

Table 6.2: Multi Threaded Processor Configuration

Component	Access Time
L1 Data Cache	64KB, 1 cycle
L1 Instruction Cache	64KB, 1 cycle
1 L2 tile	4MB, 26 cycles
2 L2 tiles	2MB, 17 cycles
4/8/16 L2 tiles	1MB, 9 cycles
Main Memory	200 Cycles

- Mesh and Bus

  The architecture is simulated for 1, 2, 4, 8 and 16 processors. Therefore, a 16 processor configuration would run with 32 threads. The bus is clocked at half the frequency as that of the processor core. The architecture is not tested for more than 16 processors, primarily because studies [KZT05] in the past have shown that practically, a chip with 16 processors on a single bus has considerable area and power overhead, and requires the single bus to be split into two separate buses. Also, with increased processor count a single bus suffers from network saturation (up to 60% on average and 90% peak) and hence reduced speedup. A detailed analysis of this network saturation and speedup is presented by Horsnell [Hor07].

- Dual Mesh

  The architecture is simulated for 1, 2, 4, 8, 16, 32 and 64 processors. For 64 processors, only 3 out of the 5 benchmarks (crypt, series and sparse) were tested, mainly due to the large simulation time overhead, as well as dimninishing scalability from the remaining benchmarks as shown in Figures 6.14 and 6.15.

- Single Bus

  The simulator used for this architecture can simulate up to 128 processors with multiple cache hierarchies [Hor07]. However, for the same reasons as cited in Mesh and Bus this architecture is simulated up to 16 processors only. The configuration tested assumes a single cluster based L1 bus that interconnects all processors to the L2. The bus is run at 1/4 the processor frequency, ensuring that the time taken to access the L2 (2 cycles for a 1:1 bus to processor core frequency ratio) is 16 cycles.

- Perfect Memory

  In order to determine the scalability of the benchmark, a perfect memory (perf mem) simulator [Hor07] is used. The perfect memory simulator assumes no memory hierarchy, i.e. no caches and cache coherency, and hence all requests are satisfied instantaneously without experiencing any contention . Simulator configurations of 1, 2, 4, 8, 16, 32 and 64 processors were used for all benchmarks.

6.3 Speedup

This section describes the speedup obtained by running the multithreaded benchmarks over different architectures with varying processor configurations. All results reported are with respect to using two contexts per JAMAICA core and from the parallel execution phase of the benchmarks.

The speedup of a benchmark is calculated using the formula:

$$SpeedUp = \frac{ExecutionTime(1Processor)}{ExecutionTime(Nprocessors)} \tag{6.1}$$

Figures 6.2 and 6.3 shows the relative speedup obtained by comparing the actual execution time of the benchmark running on different architectures for a 16 processor configuration. It must be noted that the absolute speedup is measured w.r.t the performance of the benchmark using a single processor for the same architecture. It is the relative speedup that actually measures the effectiveness of the architecture. For all benchmarks, the perfect memory case outperforms the dual mesh scheme (Figure 6.3), because there is no contention for memory accesses.

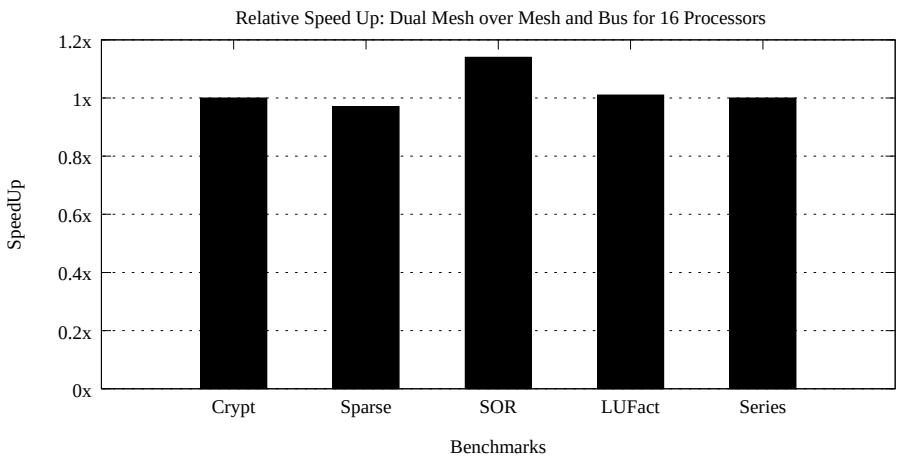

Figure 6.2: Relative Speedup Obtained by Using Dual Mesh over Mesh and Bus for 16 Processors

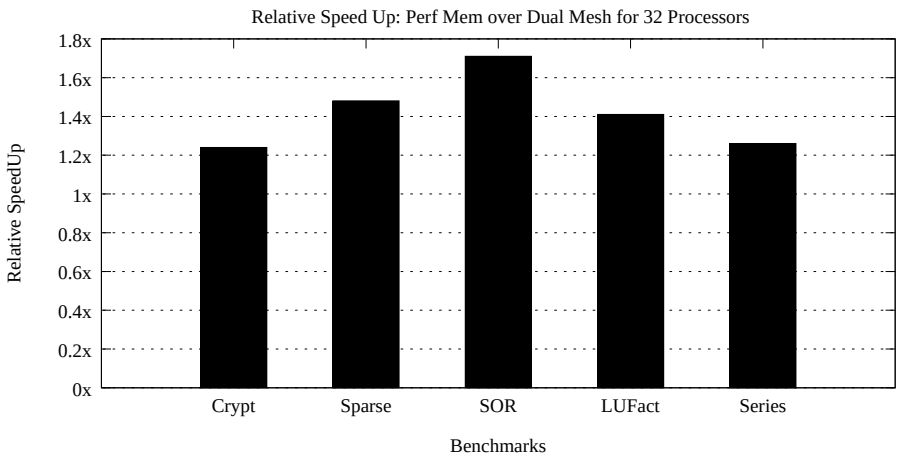

Figure 6.3: Relative Speedup Obtained by Using Perfect Memory over Dual Mesh for 32 Processors

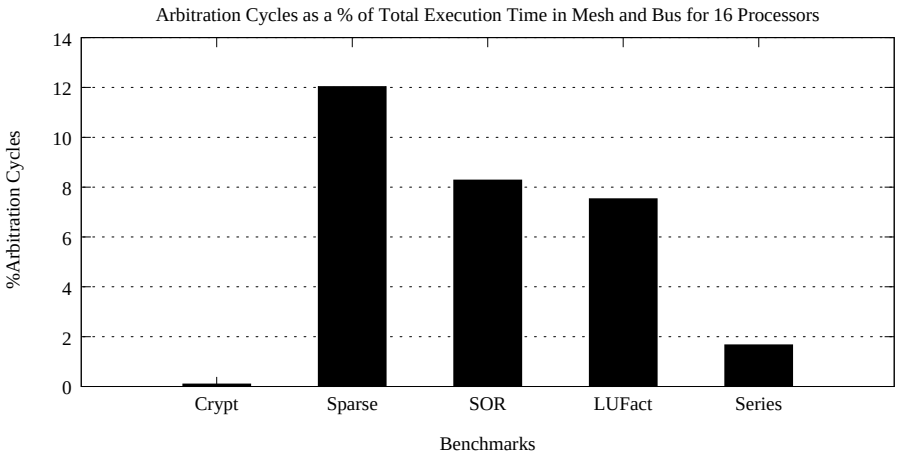

Figure 6.4: Arbitration Cycles as % of the Total Execution Time: Mesh and Bus

Figure 6.4 shows the time spent in arbitration as a % of the total execution time in the mesh and bus architecture. Figure 6.5 shows the effect the arbiter in the mesh and bus architecture has on the average read and write delays in comparison to the dual mesh scheme that does not implement an arbiter. Figure 6.6 shows the increase in total memory loads (MEML), memory stores (MEMS) and total number of instructions executed (TOTINSTRS) in the mesh and bus architecture over the dual mesh scheme. The reason for this increase being, the mesh and bus use a Jikes build with the new locking code as mentioned in Chapter 4, Section 4.8. Whereas, the dual mesh, single bus and perfect memory schemes use the Jikes build without the locking code changes, as mentioned in Chapter 3, Section 3.6.

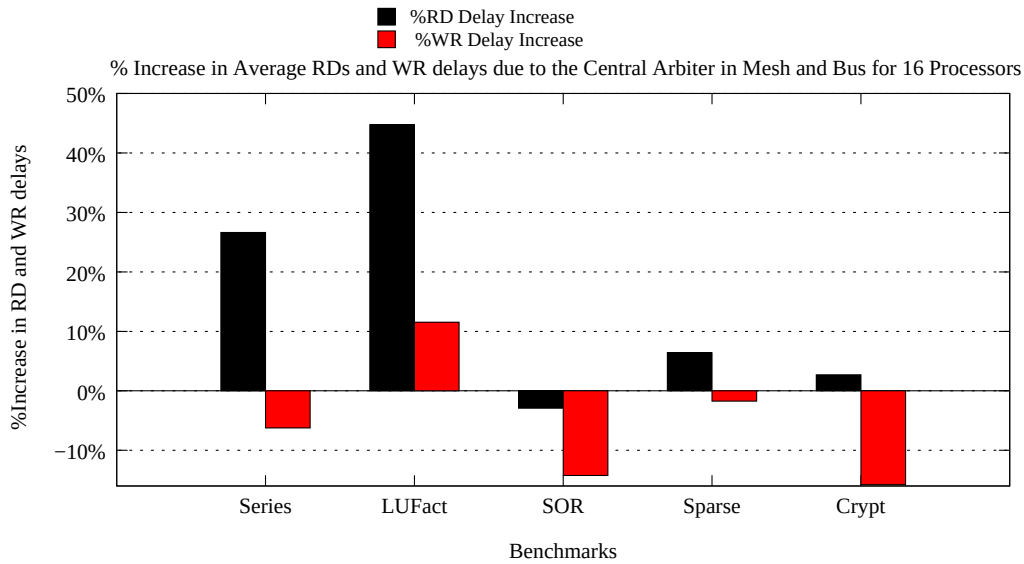

Figure 6.5: %Increase in Average Read and Write Delays in Mesh and Bus over Dual Mesh

Figure 6.7 shows the effect of stale UP, and unnecessary invalidations of cache lines on receiving a read response packet (both mentioned in Chapter 5 and in Sections 5.3.4 and 5.3.2, respectively) that might occur in the dual mesh scheme because of not having an ACK message during writes to widely shared data. This leads to increased indirection messages in the dual mesh scheme compared to the mesh and bus, as shown in Figure 6.8. Figure 6.9 shows the average number of ack messages that are generated per broadcast invalidation on a 16 processor dual mesh configuration. Benchmarks such as, LUFact and Sparse that show more than one processor on an average sharing characteristics should potentially

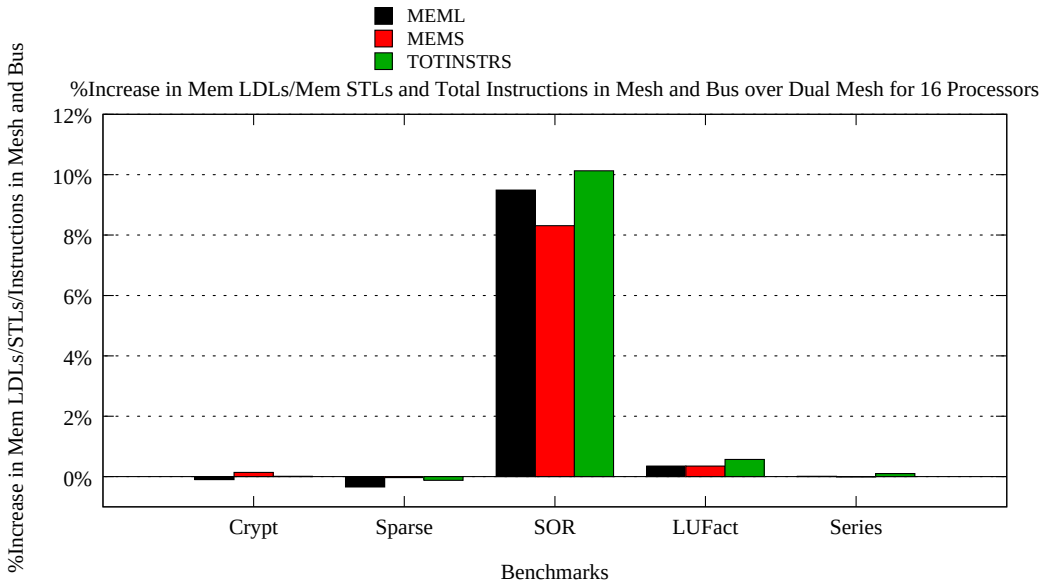

Figure 6.6: Increase in instruction count in Mesh and Bus over Dual Mesh

benefit in reduced write completion latency by avoiding the ack messages.

The absolute and relative speedup for each of the benchmarks is discussed below:

- Crypt

  The perfect memory simulator (Figure 6.10) shows that Crypt has near linear speedup up to 64 processors. This holds true for the other architectures, including the dual mesh scheme, but only up to 16 processors. However, starting from 32 processors, the rate of speedup reduces in the dual mesh scheme and drops down further for 64 processors.

  Figure 6.12 shows the worst case delays (assuming no contention) for 16, 32 and 64 processor mesh network. These delays assume two cycles per node processing time, 1 cycle hop delay, for request (when a packet travels between adjacent nodes) and 2 cycle hop delay for a data message. Figure 6.11 shows the average read and write latencies (average delay to satisfy a read miss and a write miss, respectively) for the crypt benchmark for 16 to 64 processors, with the 64 processor configuration run for different data sets (50000 and 500000) on the dual mesh scheme. It can be seen that as the number of processors increases the average read and write latencies also increase. Increasing the data set

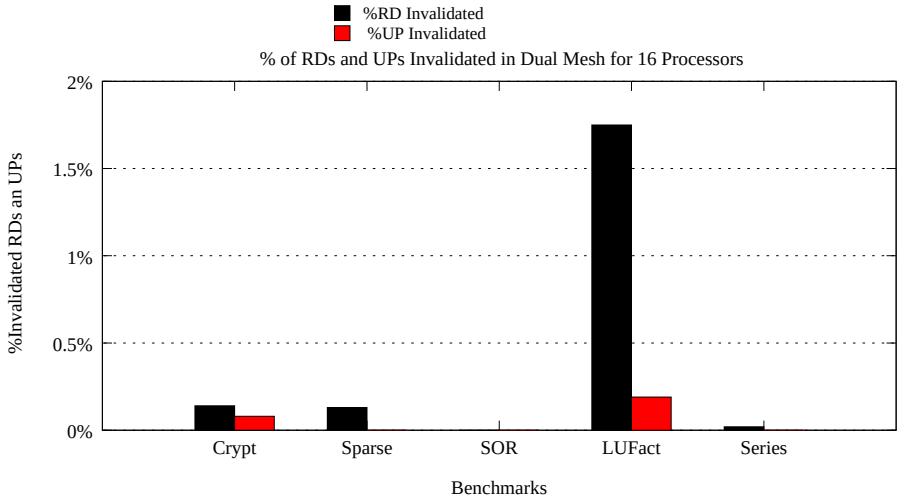

Figure 6.7: % of Invalidated RD and UPs in Dual Mesh

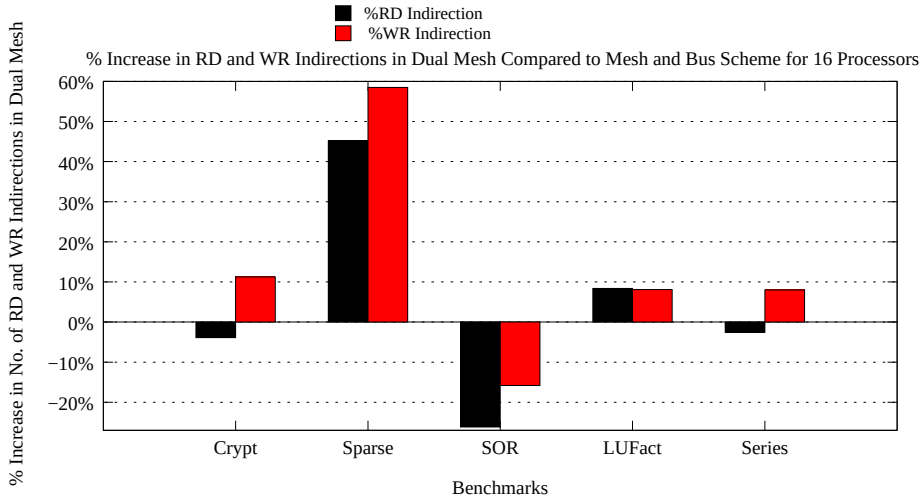

Figure 6.8: Increase in Read and Write Indirections in Dual Mesh over Mesh and Bus

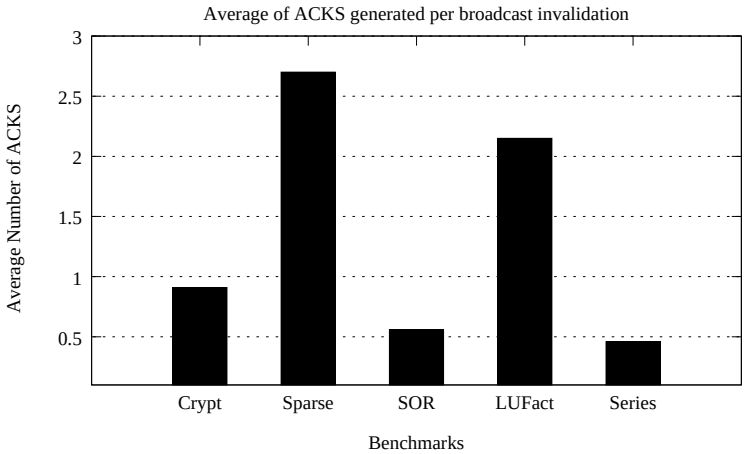

Figure 6.9: Average Number of ACKS that are generated per Broadcast Invalidation on a 16 Processor Dual Mesh Configuration

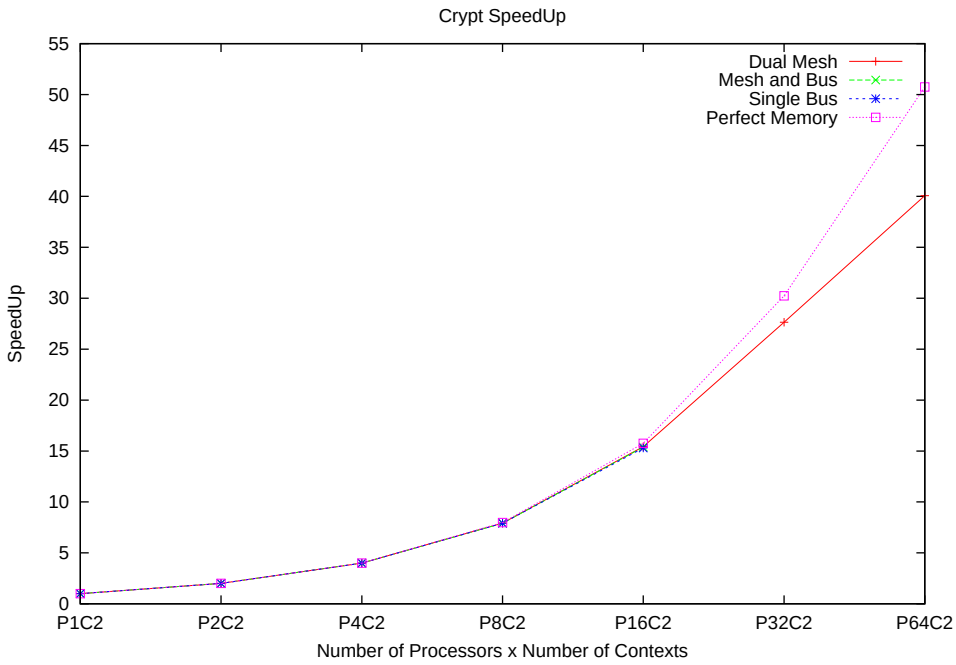

Figure 6.10: Speedup Crypt

size (thereby increasing the number of requests to L2) has no effect on the read and write latencies. This proves that the read and write latencies are dependent on the network architecture rather than the contention induced by the benchmark. However increasing the data set size (from 50000 to 500000) does increase the speedup from 40 to 60. This is because of the increase in computation to communication ratio, as shown in Figure 6.13. This increase in computation to communication ratio and hence the speedup is explained by Gustafson's law [Gus88] which states that the parallel component within an application scales with the data set size. Bigger data sets require more processors and therefore the speedup of an application is proportional to the number of processors. Equation 6.2 gives the speedup according to Gustafson's Law:

$$S = N + (N - 1)F_s \tag{6.2}$$

Where, S = Speedup; N = Number of Processors; F_s = fraction of time spent in the non-parallel section of the code.

From the relative speedup graph (Figure 6.2), there is nothing to be gained from running this benchmark on either the dual mesh or mesh

and bus (1.0x speedup. The dual mesh scheme does not benefit from any savings from the arbitration cycles lost in the mesh and bus scheme (Figure 6.4). Although the mesh and bus scheme suffers from higher average read latencies compared to the dual mesh scheme (Figure 6.5). In the case of crypt, this is offset by the increased indirection message generated in the dual mesh scheme (Figure 6.8). Both increase in the number of indirection messages and increase in the average read and write latencies leads to higher execution time. For 64 processors, the dual mesh scheme performs within 33% of the ideal speed.

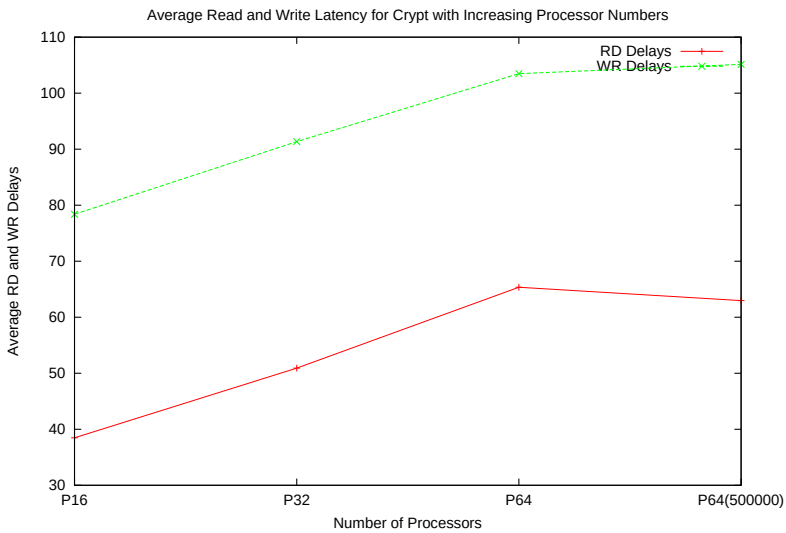

Figure 6.11: Read and Write Latencies with Varying Processor and DataSet Configurations for Crypt: Dual Mesh

- LUFact

  The perfect memory speedup graph for LUFact(Figure 6.14) shows that inherently the benchmark does not scale linearly after 16 processors. The relative speedup graph (Figure 6.2) up to 16 processors shows that the dual mesh scheme performs almost comparably to the mesh and bus (1.01x speedup, Figure 6.2). Although the dual mesh scheme gains (7.8% of the execution time in mesh and bus) by not having a central arbiter (Figure 6.4), there is an increase in the number of read and write indirection messages in the dual mesh scheme (Figure 6.8) thereby increasing the overall execution time. For 32 processors, the dual mesh scheme performs within 20% of the theoretical ideal

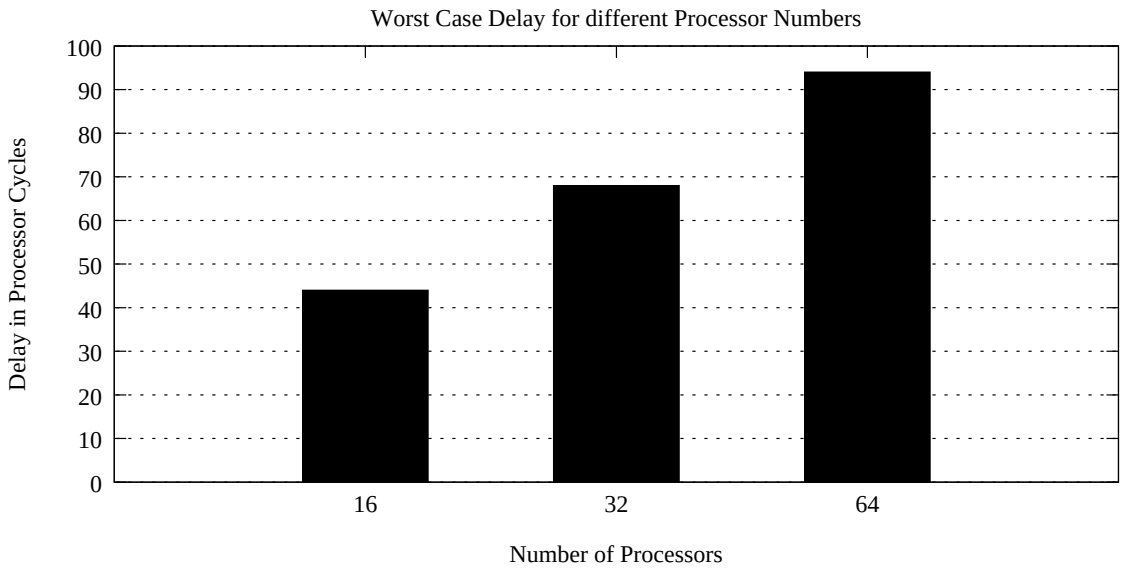

Figure 6.12: Maximum Delay on the Mesh Network with Varying Number of Processor Nodes without Contention

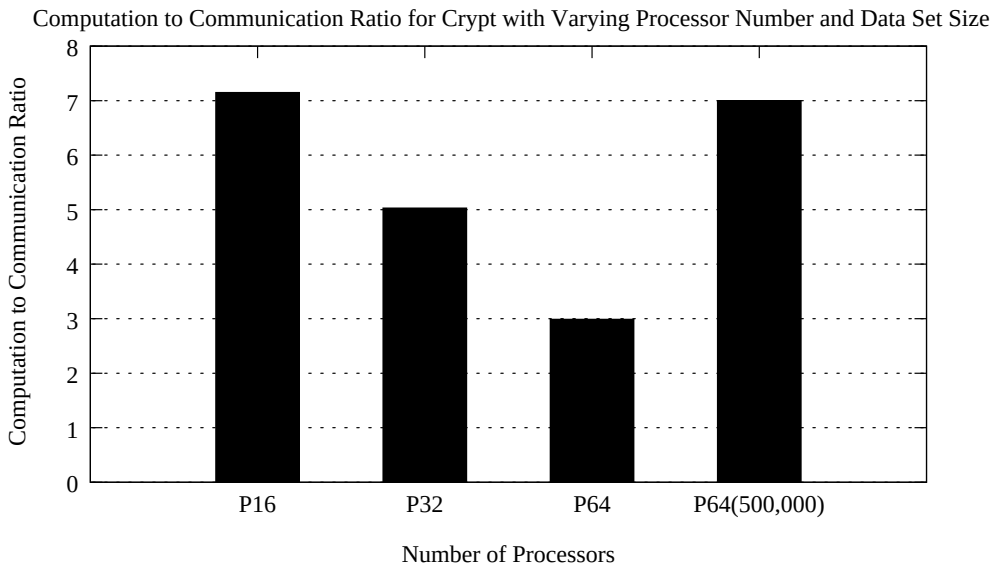

Figure 6.13: Computation to Communication Ratio For Crypt Varying Number of Processor Nodes and DataSet Size

speedup of 32. This benchmark was not exercised on the 64 processor configuration for dual mesh, primarily because of the low inherent scalability of the benchmark and the large simulation time overhead.

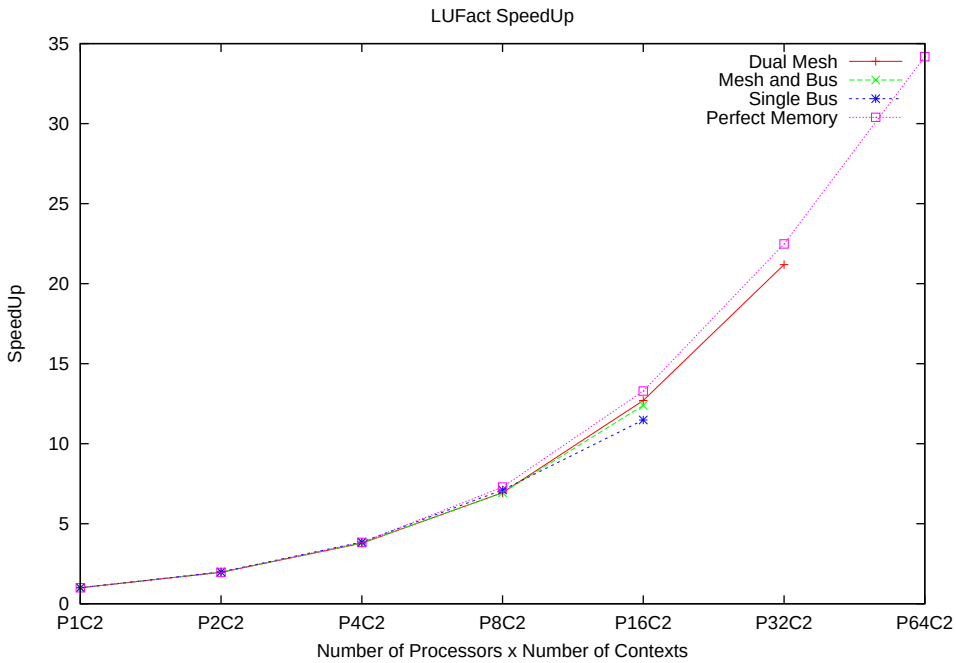

Figure 6.14: Speedup LUFact

- SOR

  The perfect memory speedup graph for SOR (Figure 6.15) indicates that the benchmark drops in performance after 16 processors. In terms of relative speedup, the dual mesh scheme performs better than the mesh and bus architecture (1.14x speedup) as shown in Figure 6.2. SOR has lower read and write latencies on the mesh and bus in comparison to the dual mesh scheme (Figure 6.5). But the decrease in the number of indirection messages in the dual mesh scheme (Figure 6.8), due to reduced number of instructions being executed Figure 6.6) along with the savings in arbitration cycles (Figure 6.4) overcompensate for the higher read and write latencies. This overcompensation leads to reduced execution time for the SOR benchmark on the dual mesh architecture in comparison to the mesh and bus scheme.

  Figure 6.16 shows the performance of the SOR benchmark run on the perfect memory architecture with simulator configurations of 1,

32 and 64 processors with a 1000X1000 grid. In comparison to the results obtained on a 500X500 grid, the speedup improves by 3x for 32 processors and 4x for 64 processors. However, the sharp drop in speedup after 32 processors, indicates that the benchmark does not scale linearly.

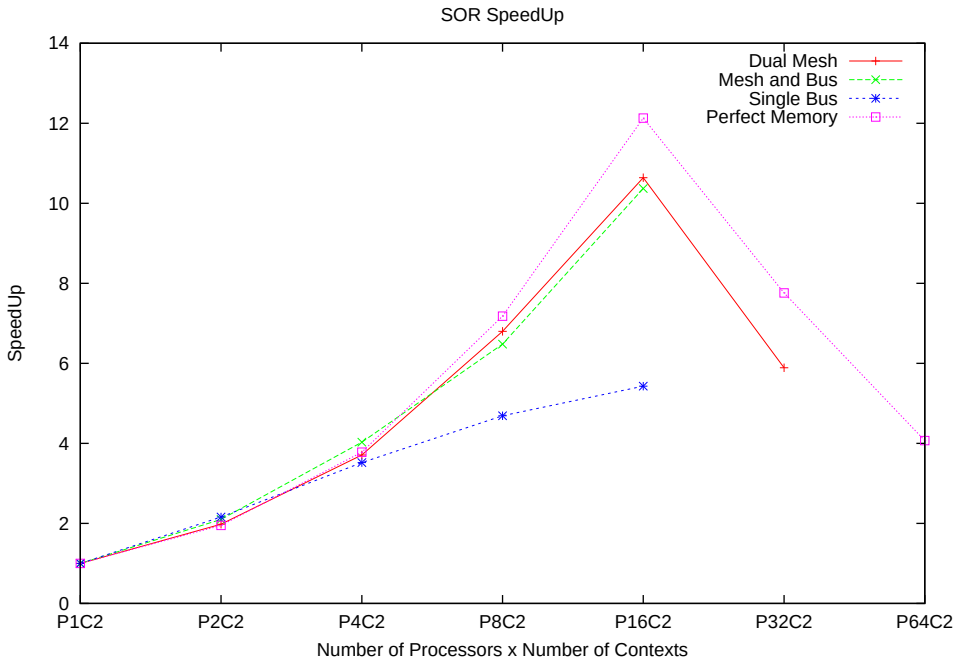

Figure 6.15: Speedup SOR

- Series

  The benchmark shows linear scalability, implying that it is inherently scalable as shown by the perfect memory graph in Figure 6.17. However, in terms of relative speedup (Figure 6.2), this benchmark performs similarly on both dual mesh and mesh and bus architectures (1.0x speedup) for 16 processors. The benchmark is dominated by reads and writes to main memory and has very little sharing among threads (Figure 6.18). Therefore, its performance is dominated by main memory delay (200 cycles, the same for both dual mesh and mesh and bus).

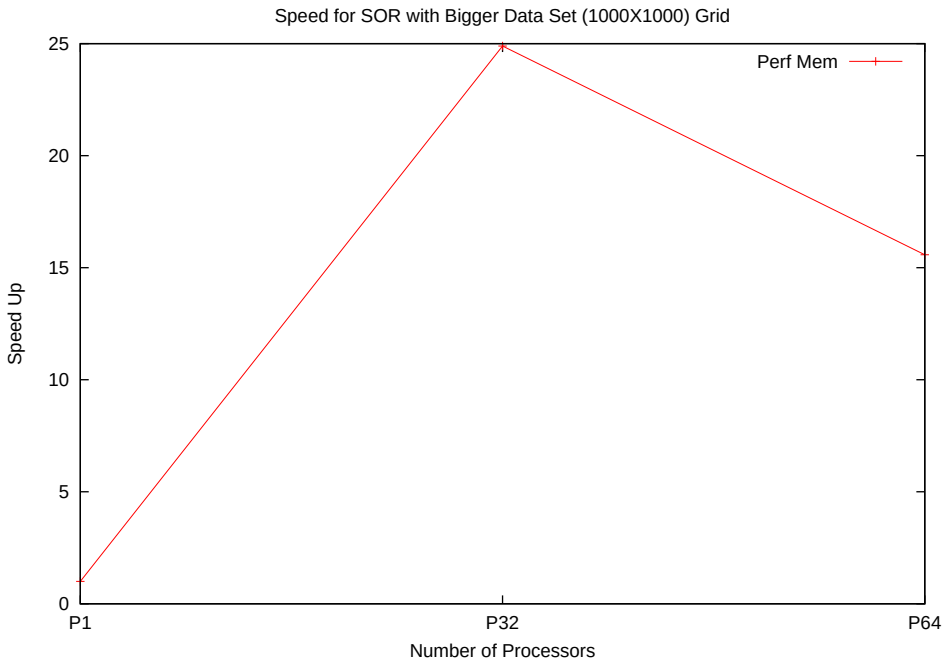

Figure 6.16: Speedup SOR for 1000X1000 with Perfect Memory

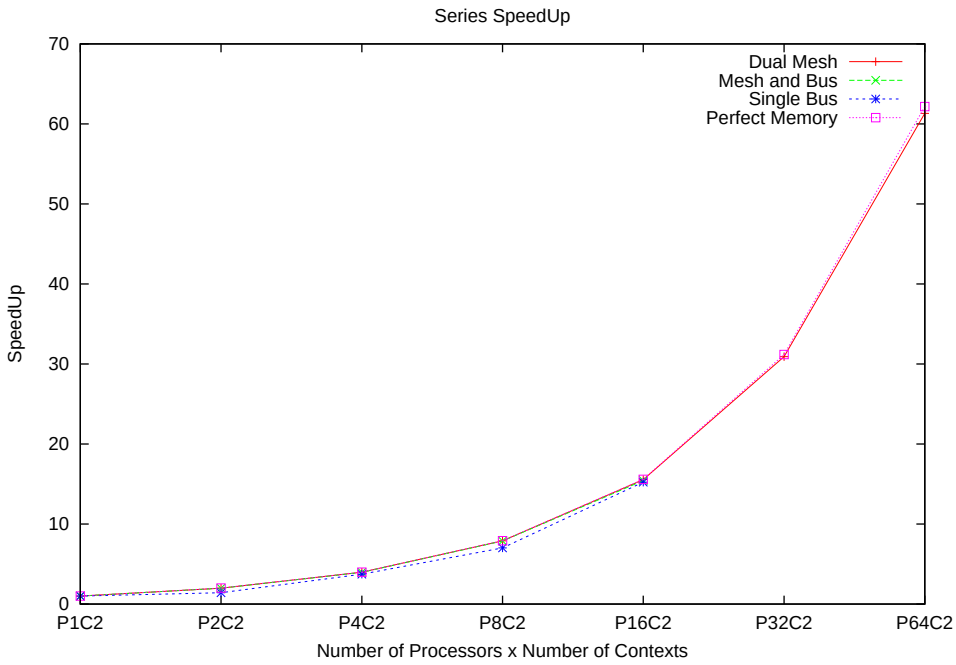

Figure 6.17: Speedup Series

- Sparse

  Similar to Series, the benchmark is inherently scalable as shown by

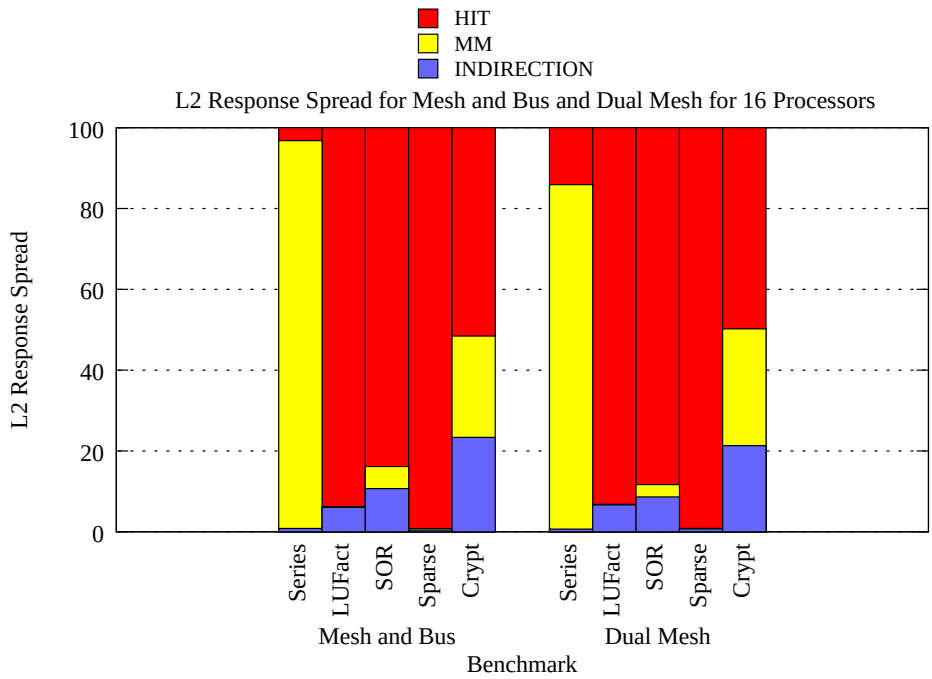

Figure 6.18: % of Indirections, MM acccess and Hits within the L2 for 16 processors: Dual Mesh and Mesh and Bus

the perfect memory graph in Figure 6.19. In terms of the relative performance (Figure 6.2), the mesh and bus based scheme performs better than the dual mesh scheme (1.03x or 0.98x - dividing the execution time from mesh and bus by the execution time from dual mesh) as shown in Figure 6.2. The dual mesh scheme gains from the mesh and bus scheme in terms of arbitration cycles (12%), But, this gain is offset by the increase in number of read and write indirections in the dual mesh scheme, which in turn increases the overall execution time (Figure 6.8).

Although the dual mesh scheme (for 16 processors) was supposed to gain (in terms of the execution time) by not having a central arbiter, it loses out by generating an increased number of indirection messages. On the dual mesh, three out of the five benchmarks-LUFact, Sparse and Crypt, suffer from increased indirection messages, so result in higher execution time and compensate for the savings in arbitration cycles (mesh and bus and single bus). The performance of Series is dominated by main memory access and its execution time on the dual mesh is unaffected by the gain in arbitration cycles (<2% in the mesh and bus scheme), or the increased indirection messages(8%) being generated in the dual

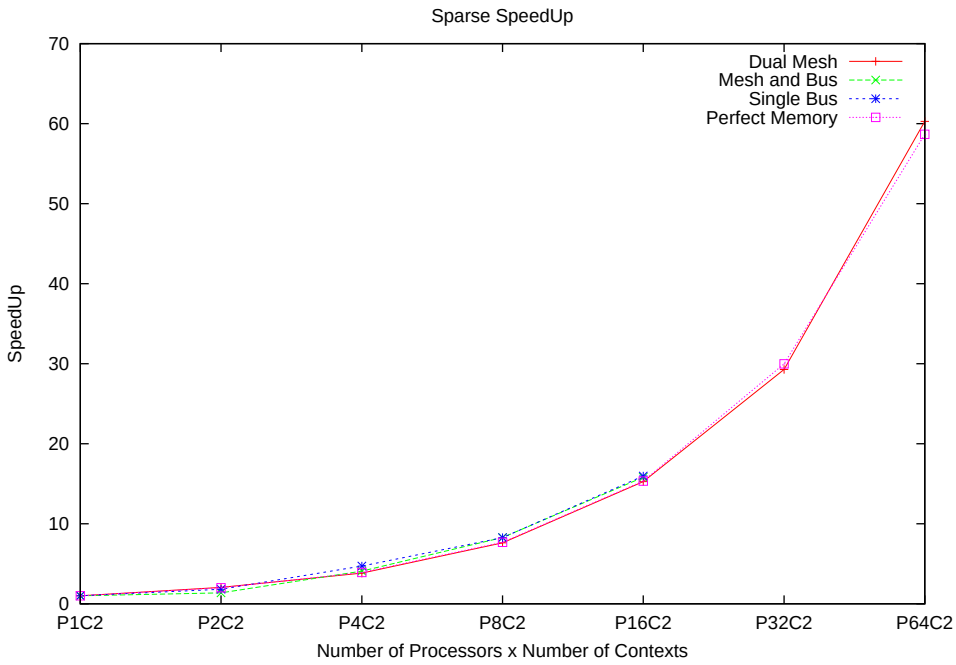

Figure 6.19: Speedup Sparse

mesh scheme. SOR benefits from cycles lost in arbitration within the mesh and bus as well as the lower indirection messages count generated on the dual mesh. However, the benchmark is inherently non-scalable and hence has low returns from running on a multithreaded CMP architecture.

Up to 16 processors, a single bus and a crossbar based CMP [Hor07] performs better than the dual mesh scheme (Figure 6.20), in terms of the relative execution time. In spite of having an arbiter on the single bus, the average read and write delays are bound to fixed number of cycles, in this case 16 cycles. However on the dual mesh scheme or mesh and bus scheme, the average read write delays vary depending on the contention and the hop count for message traversal resulting in an increase in the overall execution time. However, for more that 16 processors, when a single bus becomes unacceptable as an interconnect, mainly due to power, area and clock speed constraints on-chip, the dual mesh scheme is a more appropriate architecture. Its worst case performance is within 33% (Crypt for 64 processors) of the ideal speedup for benchmarks such as, Crypt and LUFact which are affected by the increase in execution time due to the effects of the cache coherency protocol (increased number of indirection messages) and the delays induced by the network.

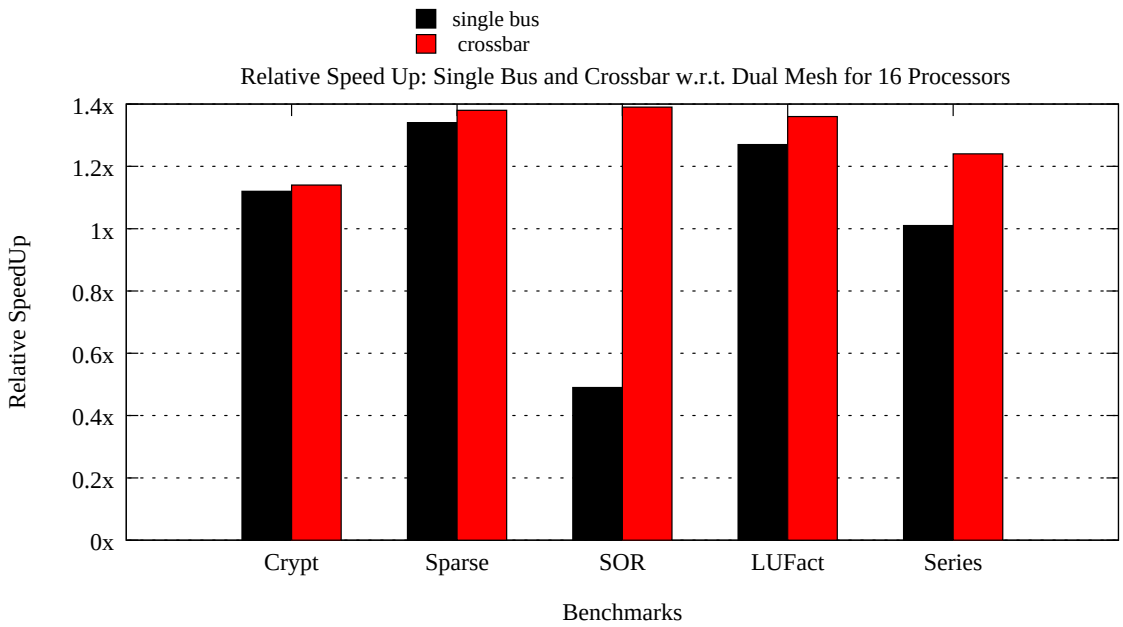

Figure 6.20: Relative Speedup: Using Single Bus over Dual Mesh for 16 Processors

6.4 Summary

This chapter evaluates the performance of the two CMP architectures discussed in this thesis, namely dual mesh and mesh and bus. The absolute performance (absolute speedup) obtained by executing five different kernel benchmarks on these two architectures as well as on a single bus is presented for up to 16 processors. The relative performance (relative speedup) of these architectures is presented by comparing the execution times of the benchmarks for up to 16 processors. The dual mesh scheme is evaluated for greater than 16 and up to 64 processors. Results show that the absolute performance of the dual mesh scheme is dictated by the parallelism inherent in the benchmark (SOR, LUFact - low parallelism; Crypt, Sparse, Series - highly parallel) and the relative performance is dependent on the the delays induced by the network topology and cache coherency protocol (Crypt, LUFact, Sparse).

Chapter 7

Conclusion and Future Work

It is well known that CMPs have already appeared, and will continue to dominate the design of future computing systems. The best possible use of CMP systems comes from extracting high degrees of parallelism from existing applications. Currently, most applications use the shared memory parallel programming paradigm. Sustaining such applications on a CMP system requires implementing cache coherency. Therefore one of the main challenges current and future CMPs face, until the introduction of newer non-shared memory programming models for mainstream applications, is the efficient implementation of cache coherency on CMPs. This thesis addresses this issue by taking advantage of the high density of transistors and wires on current generation silicon chips and explores a completely hardware based architectural solution using two separate interconnection networks on chip.

The design of two CMP architectures, one containing heterogeneous networks (mesh and bus) and the other containing homogeneous networks (dual mesh) was explored in this thesis. The main aim was to investigate the scalabilility of the CMPs using a combination of snoop and limited directory based protocols. The use of two mesh networks was motivated by the increased number of transistor and wire density on-chip as well as the use of shorter wire length interconnects to mitigate the wire delay problem. The snoop protocol was used in order to reduce the write latency encountered in traditional directory based protocols during write invalidates to widely shared data. Cycle accurate simulation of the architectures (mesh and bus and dual mesh) did reveal some of the corner cases, especially w.r.t. the network deadlock issues and synchronization protocol. The CMP architectures were evaluated using kernel based multithreaded benchmarks

from the Java Grande Benchmark suite. Results show that scalability of the architecture is dictated by several factors such as, the inherent parallelism within the application, the size of the data set and the effects of the cache coherency protocol.

Given the results obtained, we conclude that the dual mesh scheme is a promising approach for small and medium sized (up to 64 or even 128 cores) CMP systems. However, very large scale CMP systems (256 or more cores) would require a shift from the shared memory programming model and hence independence from cache coherency protocols to achieve high scalability.

7.1 Future Work

Providing high scalability and lower power dissipation is of prime importance in any CMP design. This section lists some enhancements that could potentially improve on these two aspects of the dual mesh CMP scheme.

1. Adaptive Routing Protocol

   The dual mesh scheme uses a dimension order routing protocol in order to implement read and write message ordering as required by the cache coherency protocol. Figure 7.1 shows the % utilization of vertical links attached to the L2 tiles within a 16 processor dual mesh configuration (Chapter 4, Figure 4.2), every 10 million cycles of the benchmark run. Figure 7.2 shows the % of vertical link utilization for tiles adjacent to the L2. The link utilization is a very small fraction of the total offered bandwidth, implying that the network does not saturate. Comparing Figure 7.1 and Figure 7.2, it is seen that distribution of traffic is uneven within the network, with the L2 tiles subjected to almost 5 times more traffic than that on the adjacent tile links. In the cache coherency protocol implemented, ordering is provided on a single cache line address by using the static dimension order routing protocol. However, the routing protocol is too conservative for request messages that need not be ordered. Therefore, using adaptive routing protocols for request messages and static dimension routing protocol for response messages should provide for potential performance improvement on the dual mesh system. An alternative

and possibly a simpler approach would be to use separate request and response network as used in Tilera [WGHea07].

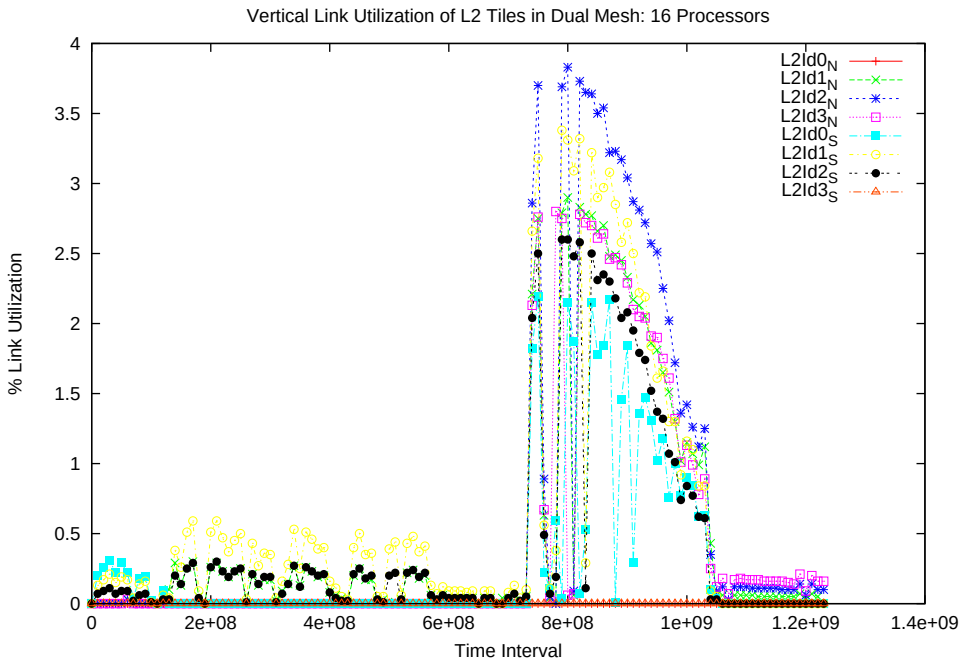

Figure 7.1: Link Activity on Vertical Links on L2 Tiles on a Dual Mesh

2. Formal Analysis of Cache Coherency Protocol

   Formal analysis is commonly used in both hardware and software system verification in order to detect subtle errors, that arise due to the scale and complexity of modern designs. Random benchmark simulation testing is not a solution to complete verification and determining all the bugs (corner cases) present within the design. Most hardware based cache coherent system designed within academia and industry are verified using formal verification tools that exhaustively search the state space of the design [PD97]. In this thesis a random benchmarking approach was taken, however, using formal method analysis for proving the data consistency feature of the cache coherency protocol as well as validating the locking protocol could be another potential area of research in using multiple network based cache coherency scheme.

3. Optimizations to the Cache Coherency Protocol

   The limited directory protocol used in the mesh and bus and dual

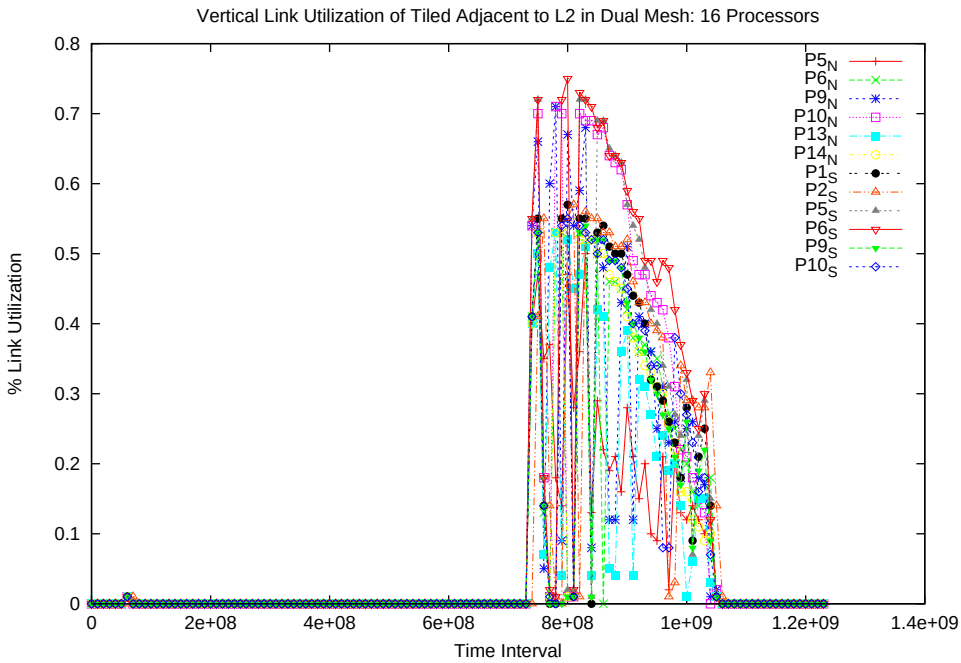

Figure 7.2: Link Activity on Vertical Links on Tiles Adjacent to L2 on a Dual Mesh

mesh scheme, requires the L2 tiles to store the id of a single processor that is either a sharer or owner of the cache line. This storage requires $log_2 N$ bits, where N is the total number of processors within the system. However, previous work on limited directories [ASHH88][MH94][CP99][AGGD05], propose several ways of encoding data within a fixed number of bits in order to accommodate more than one cache line sharer information. Transmitting the sharer information on the broadcast network would help in preventing unncessary invalidations in the dual mesh scheme as mentioned in Chapter 5 Section 5.3.2.

Bibliography

[AB84] J. Archibald and J.L Baer. An economical solution to the cache coherence problem. *SIGARCH Computer Architecture News*, 12(3):355–362, 1984.

[AB86] J. Archibald and J.L. Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, 4(4):273–298, 1986.

[ACJea07] M. Azimi, N. Cherukuri, D.N. Jayasimha, and et. al. Reevaluating amdahl's law. *Integration Challenges and Tradeoffs for Tera-scale Architectures*, 11(3), 2007. http://www.intel.com/technology/itj/2007/v11i3/1-integration/1-abstract.htm: Last Accessed September 2008.

[Aea00] B. Alpern and et. al. The jalepeno virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[AGGD05] M.E. Acacio, J. Gonzalez, J.M. Garcia, and J. Duato. A two-level directory architecture for highly scalable cc-numa multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 16(1):67–79, 2005.

[And90] T.E. Anderson. The performance of spin lock alternatives for shared-moneymultiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[ARM06] ARM. *ARM11 MPCore Processor Technical Reference Manual*, 2006. http://www.arm.com.

[ASHH88] A. Agarwal, R. Simoni, J.L. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of*

the 15th Annual International Symposium on Computer Architecture, pages 280–289, 1988.

[BC91] J.L. Baer and T.F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 176–186, 1991.

[BCGK04] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny. Qnoc: Qos architecture and design process for network on chip. *Journal of Systems Architecture*, 50(2-3):105–128, 2004.

[BD06] J. Balfour and W. J. Dally. Design tradeoffs for tiled cmp on-chip networks. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 187–198, 2006.

[BDH+99] E. Bilir, R. M. Dickson, Y. Hu, M. Plakal, and D. J. Sorin. Multicast snooping: a new coherence method using a multicast address network. In *Proceedings of the 26th annual International Symposium on Computer Architecture*, pages 294–304, May 1999.

[BGC+07] E. Bolotin, Z. Guz, I. Cidon, R. Ginsoar, and A. Kolodny. The power of priority: Noc based distributed cache coherency. In *Proceedings of the First International Symposium on Network-on-Chips (NOCS '07)*, pages 117–126, May 2007.

[Bha05] D. Bhandarkar. Multi multi-core microprocessor chips: Core microprocessor chips: Motivation challenges. Intel 10th EMEA Academic Forum, May 2005. http://www.intel.com/education/highered/research/academicforum.htm.

[BKJN99] V. Berkel, C.H. Kees, M.B. Josephs, and S.M. Nowick. Scanning the technology: applications of asynchronous circuits. *Proceedings of the IEEE*, 87(2):223–233, 1999.

[BKT07] J.A. Brown, R. Kumar, and D. Tullsen. Proximity-aware directory-based coherence for multi-core processor architectures. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 126–134, 2007.

[BSW+99] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance java. In *Proceedings of ACM 1999 Java Grande Conference*, pages 81–88, 1999.

[Cha02] A. Charlesworth. The sun fireplane interconnect. *IEEE Micro*, 22(1):36–45, 2002.

[CMR+06] L. Cheng, N. Muralimanohar, K. Ramani, R. Balasubramonian, and J.B. Carter. Interconnect-aware coherence protocols for chip multiprocessors. In *Proceedings of the 33rd annual International Symposium on Computer Architecture*, pages 339–351, 2006.

[CP99] J.H. Choi and K.H. Park. Segment directory enhancing the limited directory cache coherence schemes. In *IPPS '99/SPDP '99: Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, pages 258–267, 1999.

[CRD07] B. Cuesta, A. Robles, and J. Duato. An effective starvation avoidance mechanism to enhance the token coherence protocol. In *15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP'07)*, pages 47–54, 2007.

[CS06] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. In *ISCA 2006: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 264–276, 2006.

[CSG99] D.E. Culler, J.P. Singh, and A. Gupta. *Parallel computer architecture: a hardware and software approach.* Morgan Kaufmann Publishers, SanFrancisco, California, 1st edition, 1999.

[CSL+06] J.F. Cantin, J.E. Smith, M.H. Lipasti, A. Moshovos, and B. Falsafi. Coarse-grain coherence tracking: Regionscout and region coherence arrays. *IEEE Micro*, 26(1):70–79, 2006.

[Dig92] Digital Equipment Corp. *Alpha Architecture Handbook*, 1992.

[Dig95] Digital(now Compaq/HP). *Shared memory consistency models: a tutorial*, September 1995. http://research.compaq.com/wrl/techreports/abstracts/95.7.html.

[Din06] A. Dinn. JaVM user guide, August 2006. http://intranet.cs.man.ac.uk/apt/intranet/csonly/jamaica/Documents/TechnicalNotes/JaVMUserGuide/JaVMUserGuide.html/ Last Accessed June 2008.

[DS87] W.J. Dally and C.L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, 1987.

[DT01] W.J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *DAC: Proceedings of Design Automation Conference*, pages 684–689, 2001.

[DWKEM05] A. Dinn, I. Watson, C. Kirkham, and A. El-Mahdy. The jamaica virtual machine: A chip multiprocessor parallel execution environment. Technical report, University of Manchester, 2005.

[DYL03] J. Duato, S. Yalamanchilli, and L. Li. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers, San-Francisco, California, 2003.

[EB00] F. Emnett and M. Biegel. Power reduction through rtl clock gating. Technical report, SNUG 2000: Synopsys User Group, San Jose, 2000. www.eng.auburn.edu/ vagrawal/-COURSE/E6270_Fall07/PROJECT/LUO/snug2000.pdf.

[EPS06] N. Eisley, L.S. Peh, and L. Shang. In-network cache coherence. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 321–332, 2006.

[FC08] C. Fensch and M. Cintra. An os-based alternative to full hardware coherence on tiled cmps. In *HPCA 2008: The 14th International Symposium on High-Performance Computer Architecture*, 2008.

[Fen01] K. Fenwick. A performance analysis of java distributed shared memory implementations. Master's thesis, Adelaide University, 2001.

[FPGAD07] R. Fernandez-Pascual, J.M. Garcia, M.E. Acacio, and J. Duato. A low overhead fault tolerant coherence protocol for cmp architectures. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 157–168, 2007.

[GCM+06] P. Gratz, K. Changkyu, R. McDonald, S.W. Keckler, and D. Burger. Implementation and evaluation of on-chip network architectures. In *ICCD 2006. International Conference onComputer Design, 2006.*, pages 477–484, 2006.

[GGV90] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *International Conference on Supercomputing*, pages 354–368, 1990.

[GK08] B. Grot, , and S.W. Keckler. Scalable on-chip interconnect topologies. In *CMP-MSI: 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects, In conjunction with the (ISCA-35), 35th International Symposium on Computer Architecture*, 2008. http://www.cs.utah.edu/cmpmsi08/.

[GKS+07] P. Gratz, C. Kim, K. Sankaralingam, H. Hanson, P. Shivakumar, S.W. Keckler, and D. Burger. On-chip interconnection networks of the trips chip. *IEEE Micro*, 27(5):41–50, 2007.

[GLD06] M. E. Gómez, P. López, and J. Duato. Fir: an efficient routing strategy for tori and meshes. *Journal of Parallel Distributed Computing*, 66(7):907–921, 2006.

[GN92] C. J. Glass and L. M. Ni. The turn model for adaptive routing. In *ISCA: In Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 278–287, 1992.

[Gus88] J. L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[GVA+08] Y. Guo, V. Vlassov, R. Ashok, R. Weiss, and C.A. Moritz. Synchronization coherence: A transparent hardware mechanism for cache

coherence and fine-grained synchronization. *Journal of Parallel Distributed Computing*, 68(2):165–181, 2008.

[GW88] J. R. Goodman and P. J. Woest. The wisconsin multicube: a new large-scale cache-coherent multiprocessor. *SIGARCH: Comput. Architecture News*, 16(2):422–431, 1988.

[GWM90] A. Gupta, W. Weber, and T. Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *International Conference on Parallel Processing*, pages 312–321, 1990.

[HGR07] A. Hansson, K. Goossens, and A. Radulescu. Avoiding message-dependent deadlock in network-based systems on chip. *VLSI Design*, February 2007.

[Hor07] M. J. Horsnell. *A Chip Multi-Cluster architecture with locality aware task distribution.* PhD thesis, University of Manchester, 2007.

[HP03] J. L. Hennessy and D. A. Patterson. *Computer architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 3rd edition edition, 2003.

[HP07] J. L. Hennessy and D. A. Patterson. *Computer architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 4th edition edition, 2007.

[HSXP08] Z. Huang, X. Shi, Y. Xia, and J. Peir. Alternative home: Balancing distributed cmp coherence directory. In *CMP-MSI: 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects, In conjunction with the (ISCA-35), 35th International Symposium on Computer Architecture*, 2008. http://www.cs.utah.edu/cmpmsi08/.

[Inc] Sun MicroSystems Inc. Sun java. http://java.sun.com/. Last Accessed June 2008.

[Int98] Intel Corp. *P6 Family of Processors - Hardware Developer's Manual*, 1998. http://www.intel.com/design/pentiumii/manuals/244001.htm.

[itr05] International technology roadmap for semiconductors - interconnects. Technical report, International Technology Roadmap for Semiconductors, 2005.

[KBD07] J. Kim, J. Balfour, and W. Dally. Flattened butterfly topology for on-chip networks. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 172–182, 2007.

[KBK02] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th annual conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, 2002.

[KFJ+03] R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, and D.M. Tullsen. A multi-core approach to addressing the energy-complexity problem in microprocessors. In *WCED 2003: Workshop on Complexity-Effective Design*, 2003. http://www-cse.ucsd.edu/users/tullsen/wced03.pdf.

[KM08] S. Kaxiras and M. Martonosi. *Computer Architecture Techniques for Power-Efficiency*. Morgan and Claypool Publishers, ist edition edition, 2008.

[KZT05] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *Proceedings of the 32nd International Symposium on Computer Architecture, June 2005*, pages 408–419, June 2005.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executed multiprocess programs. In *IEEE Transactions on Computers*, volume C-28, pages 690–691, September 1979.

[Mat97] D. Matzke. Will physical scalability sabotage performance gains? *IEEE Computer*, 30(9):37–39, 1997.

[MB07] N. Muralimanohar and R. Balasubramonian. Interconnect design considerations for large nuca caches. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 369–380, 2007.

[MBH+05] M.R. Marty, J.D. Bingham, M.D. Hill, A. J. Hu, M.M.K. Martin, and D.A. Wood. Improving multiple-cmp systems using token coherence. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 328–339, 2005.

[MFD+06] A. Mejia, J. Flich, J. Duato, S.-A. Reinemo, and T. Skeie. Segment-based routing: an efficient fault-tolerant routing algorithm for meshes and tori. In *IPDPS: Proceedings 20th IEEE International Parallel amp; Distributed Processing Symposium*, page 84, 2006.

[MH94] S. S. Mukherjee and M. D. Hill. An evaluation of directory protocols for medium-scale shared-memory multiprocessors. In *Proceedings of the 8th ACM-SIGARCH Int'l Conf. on Supercomputing*, pages 64–74, 1994.

[MH08] M.R. Marty and M.D. Hill. Virtual hierarchies. *IEEE Micro*, 28(1):99–109, 2008.

[MHW03] M.M.K. Martin, M. D. Hill, and D. A. Wood. Token coherence: Decoupling performance and correctness. In *Proceedings of 30th International Symposium on Computer Architecture (ISCA-30)*, pages 182–193, May 2003.

[MLC+92] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, 1992.

[MMFC01] A. Moshovos, G. Memik, B. Falsafi, and A.N. Choudhary. JETTY: Filtering snoops for reduced energy consumption in SMP servers. In *Proceedings of the 7th International Symposium on High- Performance Computer Architecture,*, pages 85–96, 2001.

[MPS06] M. Monchiero, G. Palermo, and C. O. Silvano. An efficient synchronization for multiprocessor system on-chip. In *Proceedings of the 2005 workshop on MEmory performance: DEaling with Applications, systems and architecture: MEDEA*, pages 33–40, September 2006.

[MSHW02] M.M.K. Martin, D.J. Sorin, M.D. Hill, and D.A. Wood. Bandwidth adaptive snooping. In *Proceedings of the Eighth IEEE Symposium on High-Performance Computer Architecture*, pages 251–262, 2002.

[NS92] H. Nilsson and P. Stenström. The scalable tree protocol - a cache coherence approach for large-scale multiprocessors. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, pages 498–506. IEEE Computer Society Press, 1992.

[ODH+07] J. D. Owens, W. J. Dally, R. Ho, D.N. Jayasimha, S.W. Keckler, and L. Peh. Research challenges for on-chip interconnection networks. *IEEE Micro*, 27(5):96–108, 2007.

[OS02] V. G. Oklobdzija and J. Sparsø. Future directions in clocking multi-ghz systems. In *ISLPED '02: Proceedings of the 2002 international symposium on Low power electronics and design*, pages 219–219, 2002.

[PD97] F. Pong and M. Dubois. Verification techniques for cache coherence protocols. *ACM Computing Surveys*, 29(1), 1997.

[PIB+01] V. Puente, C. Izu, R. Beivide, J.A. Gregorio, F. Vallejo, and J.M. Prellezo. The adaptive bubble router. *Journal of Parallel and Distributed Computing*, 61(9):1180–1208, 2001.

[Pin06] T. Pinkston. Multicore and multiprocessor interconnection networks, July 2006. Tutorial at ACACES 2006, Second International Summer School on Advanced Computer Architecture and Compilation for Embedded System ,L'Aquila, Italy: http://ceng.usc.edu/smart/slides/appendixE.html.

[PTM96] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed shared memory: concepts and systems. *Parallel Distibuted Technolocy: Systems Applications*, 4(2):63–71, 1996.

[QDT88] D. Quammen, D.K. DuBose, and D. Tabak. A risc architecture for multitasking. In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, pages 230–237, 1988.

[RAG08] A. Ros, M.E. Acacio, and J.M. Garcia. Dico-cmp: Efficient cache coherency in tiled cmp architectures. In *IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008*, pages 1–11, 2008.

[RL97] Y. Rhee and J. Lee. A scalable cache coherent architecture for large-scale mesh connected multiprocessors. In *Proceedings of the 1997 International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN '97)*, page 64, 1997.

[SB77] H. Sullivan and T.R. Bashkow. A large scale, homogeneous, fully distributed parallel machine, i. *SIGARCH Computer Architecture News*, 5(7):105–117, 1977.

[SBB+91] M.D. Schroeder, A.D. Birrell, M. Burrows, H. Murray, and et.al. Autonet: a high-speed, self-configuring local area network using point-to-point links. *IEEE Journal on Selected Areas in Communications*, 9(8):1318–1335, 1991.

[SBO01] L. A. Smith, J. M. Bull, and J. Obdrzalek. A parallel java grande benchmark suite. In *Supercomputing 01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, page 6, 2001.

[SH91] R. Simoni and M. Horowitz. Modeling the performance of limited pointers directories for cache coherence. In *Proceedings of the 18th annual international symposium on Computer architecture (ISCA '91)*, pages 309–319, 1991.

[SKT+05] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. Power5 system microarchitecture. *IBM Journal of Research and Development*, 49(4):505–521, 2005. http://www.research.ibm.com/journal/rd/494/tocpdf.html.

[Sta97] W. Stallings. *Data and Computer Communications*. Prentice-Hall Inc, New Jersey, USA, 1997.

[Ste05] P. Stenstrom. Chip multiprocessors, July 2005. Tutorial at ACACES 2005, First International Summer School on Advanced Computer Architecture and Compilation for Embedded System ,L'Aquila, Italy.

[Sun03] Sun Microsystems, 4150 Network Circle,Santa Clara, CA 95054, USA. *JBus architecture overview*, version1.0 edition, April 2003. http://www.sun.com/processors/whitepapers/JBus_External.pdf.

[Til] Tilera Corporation. *Tile Processor Architecture Technology Brief.* last accessed - August 2008 http://www.tilera.com/pdf/ProductBrief_TileArchitecture_Web_v4.pdf.

[TKMea02] M.B. Taylor, J. Kim, J. Miller, and et. al. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[URS02] T. Ungerer, B. Robic, and J. Silc. Multithreaded processors. *The Computer Journal, BCS*, 3, 2002.

[VAG05] F. J. Villa, M. E. Acacio, and J. M. Garcia. Memory subsystem characterization in a 16-core snoop-based chip-multiprocessor architecture. In *First International Conference on High Performance Computing and Communications (HPCC 2005)*, volume 3726/2005, pages 213–222, September 2005.

[Wal90] D. W. Wall. Limits of instruction-level parallelism. Technical report, Digital Equipment Corp. - Western Research Laboratory, 1990. ftp://ftp.digital.com/pub/Digital/WRL/research-reports/WRL-TN-15.ps.gz.

[WGHea07] D. W., P. Griffin, H. Hoffmann, and et. al. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.

[Wri01] G. M. Wright. *A single-chip multiprocessor architecture with hardware thread support.* PhD thesis, University of Manchester, January 2001.

[Ye03] T.T. Ye. *On-chip multiprocessor communication network design and analysis.* PhD thesis, Stanford University, December 2003.

[ZHWH07] H. Zeng, K. Huang, M. Wu, and W. Hu. Concerning with on-chip network features to improve cache coherence protocols for cmps. In *Asia-Pacific Computer Systems Architecture Conference*, pages 304–314, 2007.