Ph.D. Dissertation

Reconfigurable Communication Architecture in Chip Multiprocessors for Equality-of-Service and High Performance

서비스 균등 분배와 고성능을 위한 다중프로세서칩 상의 재구성형 통신 구조

by

Hanmin Park

February 2016

Department of Electrical and Computer Engineering
College of Engineering
Seoul National University

Reconfigurable Communication Architecture in Chip Multiprocessors for Equality-of-Service and High Performance

서비스 균등 분배와 고성능을 위한 다중프로세서칩 상의 재구성형 통신 구조

지도교수 최기영

이 논문을 공학박사 학위논문으로 제출함

2015 년 11 월

서울대학교 대학원
전기·컴퓨터공학부

박한민

박한민의 공학박사 학위논문을 인준함

2015 년 12 월

위 원 장	채수익	(인)
부위원장	최기영	(인)
위 원	이혁재	(인)
위 원	유승주	(인)
위 원	김윤진	(인)

Abstract

The chip multiprocessor (CMP) era has long begun due to the diminishing return from instruction-level parallelism (ILP) harvesting techniques, the rising power and temperature from frequency scaling, etc. One powerful processor has been replaced by many less-powerful processors forming a CMP. One of the issues arose from this paradigm shift is the management of communication among the processors. Buses, which has been a common choice for the systems with one or several processors, failed to sustain the increased communication burden of CMPs. Many bus-based improvements including hierarchical buses and bus-matrices, were proposed but eventually, network-on-chip (NoC) has become the de facto standard for designing a CMP system, replacing the bus-based techniques.

NoC's strengths over bus mainly come from its capability of conveying multiple transactions simultaneously from different components to the others. The concurrent communications between the cores are conducted by the distributed, yet shared network components, routers. Routers provide cores with services such as bandwidths. One of the design issues in implementing NoC is to distribute these services evenly across all the cores requesting for them. Arbiter is a component that regulates the accesses to shared resources such as channels and buffers. It has the policy under which requests get services in turn from the shared resources so that the

requestors don't fall into deadlock or starvation. One of the common policies for an arbiter is the round-robin, where requests get their grant one by one so that fairness is assured among the requestors. When applied to routers in NoC, it fails to provide the fairness because each request goes through multiple routers, thus multiple round-robin arbiters on a transaction route. The cascaded effect of the round-robin arbitration is that the farther a source is from the destination, the less service it gets from the destination. The first part of this thesis addresses this issue, and proposes thus far the simplest yet the most effective way of providing the fairness to all the nodes on NoC. It applies weighted round-robin scheme where the weights are determined at run-time depending on which cores are allocated to applications or threads running on the CMP. RTL implementation and synthesis are done to show the simplicity of the proposed scheme. Simulation with synthetic traffic patterns and SPEC CPU2006 benchmark applications show that the proposed approach results in outstanding equality-of-service characteristics.

The second part of this thesis deals with the impact of the reconfigurable communication architecture on the performance of a CMP system. One of the pitfalls of NoC is long access latency due to increased hop count between a source and its destination. For example, NoC with mesh topology has its hop count proportional to its size. Because of this, while being a common choice for CMP, mesh topology is said to be inscalable in terms of the number of cores. Some alternatives to mesh topology exist, one of them being high radix NoCs. They replace short and wide channels of mesh with long and narrow ones achieving fewer hop counts. Another

option is to cluster cores so that the dimension of mesh network reduces. The clusters are formed by grouping cores via local communication fabric. The clusters are interconnected by a global communication fabric, often in the shape of mesh topology. Many types of local communication fabric are explored in previous researches, including another NoC with topologies of mesh, ring, etc. However, bus has become one of the most favorable choices for the local connection because of its simplicity. The simplicity leads local communications to be performed with high performance, low chip area, low power consumption, etc. One of the issues in forming core clusters in CMP is their grain size. Tying too many cores into a cluster results in the congestion on the bus, reducing the performance of the local communications. On the other hand, too few cores in a cluster misses the chances of improving system performance by efficient local communications through the bus. It is obvious that the optimal number of cores in a cluster depends on the applications that run on the CMP. Bus reconfiguration with bus segments and switches can be a solution for varying cluster size on a CMP. In addition to the variable cluster sizes, bus reconfiguration has another advantage of processor (not process) migration. Bus reconfiguration can reconnect cores and caches so that the distance between cores and data are reduced dynamically. In this way, data copies and network transactions can be dramatically reduced to improve the system performance. The second part of this thesis addresses this issue and proposes a reconfigurable bus-mesh architecture to accelerate pipelined applications. With the proposed architecture, the data transfer between the successive pipeline stages are done not by data copies but by processor

migrations. Systematic management of bus segments and L1 data caches are required to achieve efficient use of the reconfigurability. The proposed architecture is compared with the baseline architecture, which maintains cache coherence with hardware. Multilayer perceptron (MLP), convolutional neural network (CNN), and JPEG decoder are implemented as example pipelined applications using multi-threaded programming model. The in-house full system simulator is implemented and used to measure the performance improvement of the proposed architecture. The experimental results show that 21.75 %, 14.40 %, and 12.74 % execution cycle reductions are achieved for MLP, CNN, and JPEG decoder, respectively.

Keywords: Weighted round-robin arbitration, fairness, hierarchical network-on-chip, bus reconfiguration, pipelined application

Student Number: 2009-30190

Contents

List of Figures

List of Tables

Part I

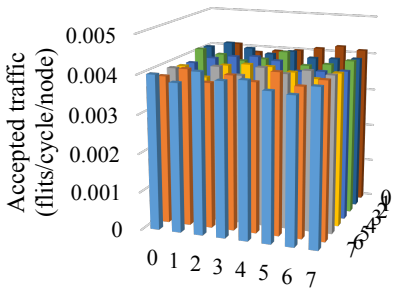
Adaptively Weighted Round-Robin Arbitration for Equality of Service in a Many-Core Network-on-Chip [1]

Chapter 1

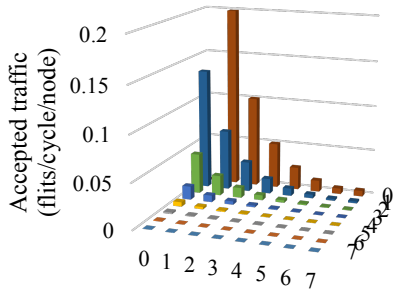
Introduction

Chip multiprocessors (CMPs), which integrate multiple cores on a chip, are now the mainstream of the computer architecture. The trend is mainly led by the slowdown in the improvement of uniprocessor performance arising from diminishing returns in exploiting instruction-level parallelism (ILP) as well as difficulties in increasing clock frequency combined with growing concern over power [2].

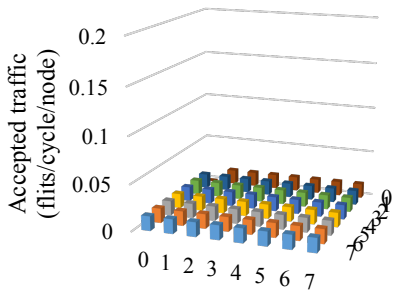
The interconnection architecture has become an important issue as the number of cores in a CMP grows, since it often decides the communication efficiency among the cores and thus the performance of the entire system. Network-on-chip (NoC) [3], being one of the most favorable choices, provides high bandwidth under the unpleasant conditions of decreasing timing margin and the pervasiveness of the bandwidth-hungry applications. The performance improvements come from various



(a) Round-robin arbitration with injection rate 0.001 packets/cycle.



(b) Round-robin arbitration with injection rate 0.05 packets/cycle.



(c) PBWRR with injection rate 0.05 packets/cycle.

Figure 1.1 Impact of network congestion on EoS (4 flits/packet).

sources such as controlled electrical parameters due to structured global wires, the support for concurrent communications, etc.

Equality of service (EoS) [4], as a subset of quality of service (QoS), is to provide equal services to every flow in the network. In general (unless the system has been customized for a specific application), the cores in a CMP access shared resources with the expectation that they will be treated fairly in terms of the amount of services such as memory bandwidth, regardless of where they are located in the

interconnection network. Achieving this goal assures the application programmer that the loads from or stores to the shared memory in any thread that runs on the CMP have the same performance, and that the details of the underlying hardware architecture need not be considered in the programming. This merit results in easy application mapping and load balancing among the cores, and can be beneficial to the overall system performance.

Congestion control [5] is an important issue of EoS because network congestion severely damages the fairness on NoC service distributions. Figure 1.1 shows the impact of congestion on EoS in an 8×8 mesh network with XY routing driven by hotspot traffic [6] for 1 million cycles. In the hotspot traffic, every node in the network sends packets to a common destination node called hotspot. In the figure, the hotspot is at node (0, 0). Each bar represents the number of flits generated by the corresponding node and accepted by the hotspot. With low injection rate (Figure 1.1(a)), the network is not saturated and the accepted traffic (throughput) [7] is evenly distributed across all the nodes. With high injection rate (Figure 1.1(b)) however, the distribution of the service is highly biased toward the nodes close to the hotspot. My previous work, the *position-based weighted round-robin arbitration* (PBWRR) [8], successfully addressed the issue so that the service can be distributed fairly under the hotspot traffic as shown in Figure 1.1(c).

In this part of the thesis, I re-analyze and extend my previous work and propose the *adaptively weighted round-robin arbitration* (AWRR), which provides fair service distributions not only for the hotspot traffic but also for other various traffic

patterns. I also perform experiments on SPEC CPU2006 [9] benchmark applications in multi-programmed manner, and show how EoS is ensured by adaptive adjustments of arbitration weights according to application mappings.

Chapter 2

Previous Work

My discussion on EoS implementation starts from that of QoS, because EoS can be regarded as a special case of QoS where all the requesters are guaranteed to get equal amount of services. One of the most common form of NoC QoS implementation is injection rate regulation [10-12]. They throttle the injection rate of each node according to global information such as network status and the amount of NoC services required by the node. These approaches often suffer from the overhead of the global structure that collects the global information. Performance overhead as well as area and power overhead can be incurred by the pessimistic throttling, the timing gap between the global information gathering and the corresponding throttling control, etc. Therefore, the injection rate regulation schemes are usually suited for real-time applications that need guaranteed QoS support [13].

EoS implementation for best-effort networks [7], on the other hand, is often proposed as the strategy for the network of supercomputer [14], CMP [4, 13], and others [11, 15], where the performance guarantees are not necessary. As noted in [7], fairness between flows is "a presumption of a best-effort class" instead of guaranteeing a specific amount of services to a specific node. In CMP, especially, if location-oblivious task placement [4] is favored due to its simplicity of task mapping, EoS should be supported when the CMP network is saturated by heavy traffic.

One of the representative EoS implementation techniques is the age-based network arbitration [14] designed for CRAY XT3 supercomputer. In this scheme, each packet carries its age (the number of cycles passed since its injection into the network) within its head flit. Packets competing for the router resources, such as the output channel bandwidths, are given their priorities according to their ages—the one with the highest age wins. This arbitration scheme achieves the global fairness across the entire network, but the limited bit-width for packet ages and per-router timestamp makes the algorithm complicated. A per-router timestamp (a free-running counter) acts as a wall clock for calculating how long a packet has stayed in the router. Maintaining timestamps so that rollovers do not cause priority inversion among the packets waiting for network services incurs significant overhead as addressed in [4].

Probabilistic distance-based arbitration (PDBA) [4] is proposed as an approximation of the age-based arbitration. The probability of a packet to be granted in a router increases geometrically as the packet traverses the network. The base of

the geometric increase is dynamically determined as the contention degree at each router. The arbitration should be probabilistic; otherwise the packets from farther nodes might take all the grants and thus make other packets starve. The packets should carry their weights in their head flits. When first proposed [16], its hardware implementation was big and slow because of its need for geometric weight calculation and probabilistic nature of the arbitration, which includes generating random numbers as well as determining the upper bound of the random numbers by taking the sum of weights of the incoming packets. All of them should be done serially. In [4], the new hardware structure is proposed that supports the pre-calculations of random numbers, their scaling according to the corresponding weight of the incoming packet, and summation of the scaled random numbers. In order to exploit the pre-calculation scheme, PDBA is combined with a round-robin (RR) arbiter, so that when the pre-calculated values are not ready, RR is used instead of PDBA. They also try to reduce the bit-width of the weight in a head flit by examining the standard deviation of the accepted traffic with varying weight bit-width. With the aforementioned efforts, though, it still bears a relatively large amount of overhead in terms of chip area and power consumption.

The most recent research on EoS of best effort NoC is [17]. The authors try to tweak PDBA by using remaining hop counts rather than travelled hop counts in the calculation of weights. The arbitration is still probabilistic. Their goal is to change the arbitration policies according to the network status, thus achieving better performance or global fairness as needed. However, they cannot achieve the perfect

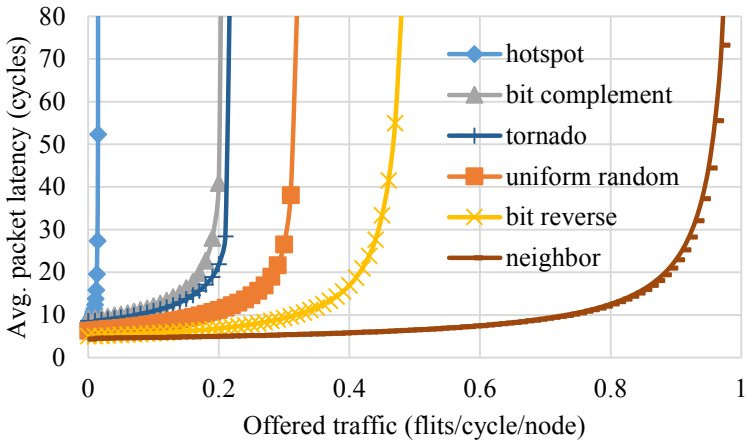
global fairness with the fairness-intended arbitration policy (PPLR: preferentially transferring packets with large remaining hop counts) because (1) it does not consider how long a packet has stayed in the network (in other words, ages are ignored), and (2) it is impossible to accurately predict how much contention a packet will meet in the remaining network traversal path.

Chapter 3

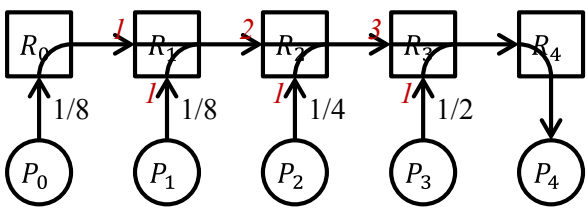
Position-Based Weighted Round-Robin Arbitration

My previous work on the position-based weighted round-robin arbitration achieves the global fairness with much simpler mechanism of weighted round-robin arbitration [7]. The simplicity comes from focusing on the hotspot traffic pattern and exploiting the deterministic natures of commonly used NoCs—mesh topology and XY routing.

The hotspot traffic pattern is in general a traffic pattern that saturates the network fastest. Figure 3.1(a) clearly shows this. It plots the average packet latency versus offered traffic curve of an 8×8 mesh network with XY routing and RR arbitration under various traffic patterns. The left most curve is the behavior of the hotspot



(a) Average packet latency with varying offered traffic under various traffic patterns.



(b) Motivational example of 5-ary 1-mesh.

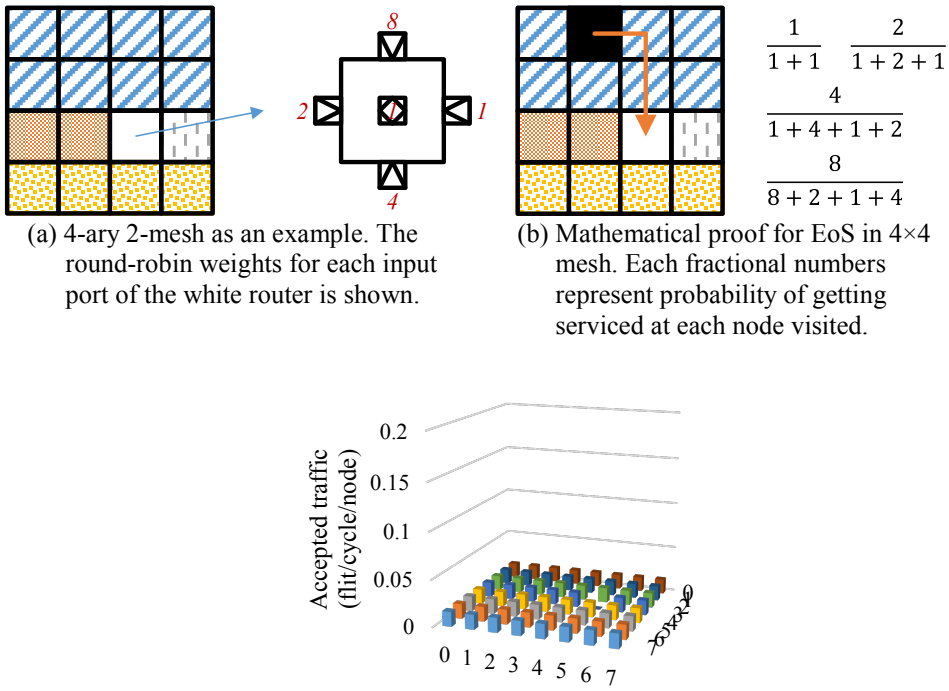
Figure 3.1 Motivation and ideas of the PBWRR arbitration.

traffic pattern. Thus it is natural that the hotspot traffic pattern has been heavily dealt with in various contexts [11, 13, 15].

I also focus on the hotspot traffic pattern in this chapter, starting from the motivational example shown in Figure 3.1(b) where the hotspot is at node 4. Assuming each router utilizes the RR arbitration, the fractional numbers represent the probability that each processor (P_i, $i \in \{0, 1, 2, 3\}$) gets serviced by the hotspot. RR is a locally fair arbitration scheme but does not provide the global fairness. By "local fairness", I mean the fairness within a router, and by "global fairness", I mean

the fairness in a network. It is obvious that the farther the request node is, the less service it gets. The 2D version of the example is shown in Figure 1.1(b). Considering the probability of each node getting serviced as calculated above, I can make them receive equal services by weighting the RR arbitration differently as annotated in Figure 3.1(b) with red italic fonts. For example, in R_2, the packets from the left input port are serviced twice, while the packets from the bottom input port are serviced once. The probability of P_2 getting serviced becomes $\frac{3}{3+1} \times \frac{1}{2+1} = \frac{1}{4}$. Likewise, the probability for P_1 becomes $\frac{3}{3+1} \times \frac{2}{2+1} \times \frac{1}{1+1} = \frac{1}{4}$. In fact, weighting the RR arbitration in this way results in the same probability of $\frac{1}{4}$ for all the four requesting nodes. The weight calculation is simple; they are just the number of nodes that should be served by the corresponding input port of the router. For example, the left input port of R_2 should serve two nodes, P_1 and P_0, thus its weight is two.

This can be easily extended to 2D mesh network that uses XY routing. Figure 3.2(a) shows a 4×4 mesh network as an example. In the white node of the network, the weight for each input port is denoted in red italic fonts. Same as the 1D example above, the weights are just the number of nodes that should be served by each input port. Using this example, mathematical proof for the equality of service in 2D mesh NoC can be done as in Figure 3.2(b). Here a transaction from the black node to the white node is being considered. Each fractional number in the figure represents the probability of getting serviced on the transaction path. Multiplying all the probabilities results in $\frac{1}{15}$, which is the same for all the other nodes in the network



(a) 4-ary 2-mesh as an example. The round-robin weights for each input port of the white router is shown.

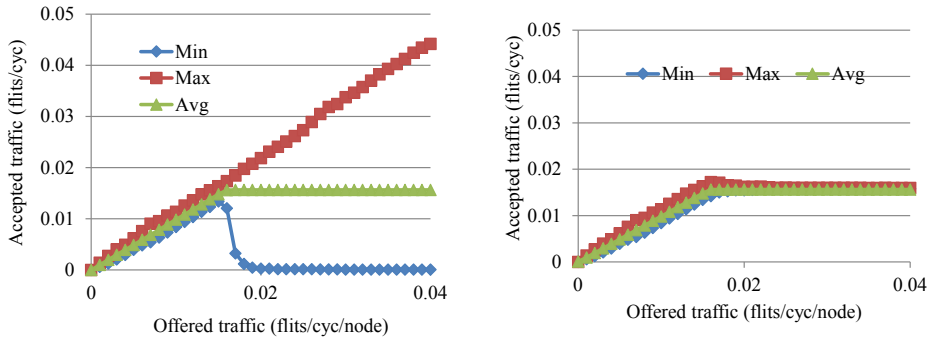
(b) Mathematical proof for EoS in 4×4 mesh. Each fractional numbers represent probability of getting serviced at each node visited.

(c) Accepted traffic distribution for the hotspot at (5, 3) for an 8-ary 2-mesh network. The PBWRR is applied.

Figure 3.2 2D extension for the PBWRR arbitration.

except the white node, which is the common destination node for this example. The simplicity of the congestion control mechanism comes from the exploitation of the deterministic properties of NoC. The arbitration weights are dependent on the position of the router, thus the mechanism is called “position-based”.

Using this scheme, the global fairness is assured for the hotspot located at any node in the network. For example, in Figure 1.1(c), the hotpot is at node (0, 0) whereas in Figure 3.2(c), the hotspot is at node (5, 3). The global fairness is supported in both cases. This is an advantage compared to the buffer sizing schemes



(a) Round-robin (b) PBWRR

Figure 3.3 Accepted traffic versus offered traffic curve. The same behaviors of the average curve in (a) and (b) shows that the throughput sustained by the NoC is not degraded by the PBWRR technique.

for the hotspot traffic [11, 13, 15], because they cannot handle the hotspots other than they are designed for. After the buffers are sized and synthesized, they cannot be reconfigured to the other hotspot locations. Besides, even with the hotspots they are designed for, after the network saturation, the unfairness arises across the entire network.

PBWRR does not require any other information carried within head flits than that required for XY routing such as the position of the destination node. This is an advantage compared to the age-based arbitration and the PDBA, which require age or weight included in head flits.

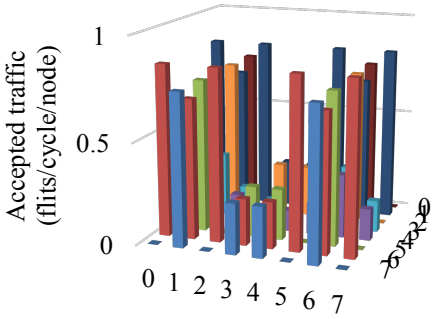
PBWRR does not degrade the total throughput sustained by NoC. Figure 3.3 plots the maximum and the minimum throughputs serviced to a node in 8×8 mesh NoC. It also plots the average throughput serviced to a node in the NoC. As shown in Figure 3.3(a), the round-robin arbitration results in unfair distribution of the

throughput. While the maximum throughput serviced to a node increases continuously, the minimum throughput serviced to a node drops quickly after the offered traffic reaches the saturation throughput of the NoC. On the other hand, with PBWRR, the maximum and the minimum throughput stays the same even after the offered traffic exceeds the saturation throughput of the NoC. Notice the fact that the trends of the average throughput are the same in (a) and (b). This means that the total throughput sustained by the NoC is not degraded by applying the PBWRR technique.

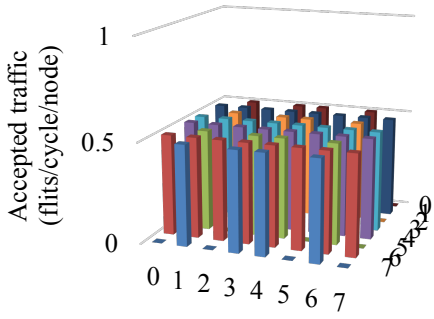
Chapter 4

Adaptively Weighted Round-Robin Arbitration

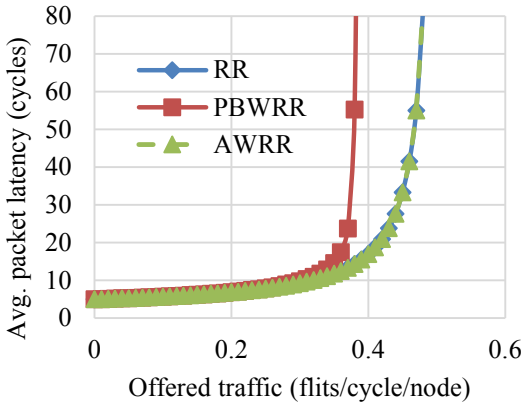
Although PBWRR provides the global fairness under the hotspot traffic pattern, it sometimes fails to do so when the network is saturated by other traffic patterns; Figure 4.1(a) shows the accepted traffic distribution of PBWRR under the bit-reverse traffic pattern. Moreover, as shown in Figure 4.1(c), PBWRR degrades the saturation throughput of the network compared to RR arbitration under some traffic patterns (the figure is for the bit-reverse traffic pattern). If I make the arbitration weights updatable, and devise a proper weight determination and setup mechanism (which I call AWRR), the global fairness can be restored as in Figure 4.1(b) and the saturation throughput can also be restored as in Figure 4.1(c).



(a) Accepted traffic distribution of PBWRR under the bit-reverse traffic pattern.



(b) Accepted traffic distribution of AWRR under the bit-reverse traffic pattern after the adjustment of arbitration weights.



(c) Average latency versus offered traffic curve of PBWRR and AWRR arbitrations under the bit reverse traffic pattern.

Figure 4.1 Behavior of PBWRR and AWRR arbitrations under the bit reverse traffic pattern.

4.1 Hardware Implementation for weight update

The hardware implementation of AWRR is similar to that of PBWRR. The difference is that, in PBWRR, the arbitration weights are fixed at design time so the

values are hard-wired. In AWRR, on the other hand, the arbitration weights are to be reconfigurable so the hard-wired logic is replaced by registers. The bit-widths of the registers are merely those of the arbitration weights in PBWRR, because hotspot traffic pattern would require the maximum number of nodes that each input port should serve when one-to-one communication is concerned. Due to this bit-width constraint, for one-to-many communication patterns such as uniform random traffic, some (or all) of the arbitration weights can be saturated to those of PBWRR.

The update of arbitration weights in AWRR is done using control packets as will be described in Section 4.2. The hardware circuitry for updating a weight is just a simple control packet detector and incrementer/decrementer attached to the weight register. In this section, it is omitted from the figures for clearer explanations, but it is included in the hardware synthesis results shown in Table 5.2.

The hardware implementation of AWRR is shown in Figure 4.2. The overall architecture of a router in 2D mesh network is shown in Figure 4.2(a). Each output port has its own arbiter. Note that some arbiters take four requests while others takes two requests. This is due to the property of the XY routing. For example, an east output port takes requests only from the local and the west input ports as shown in Figure 4.2 (b). The shaded boxes are registers. The input ports of the counters are underlined. As shown in the figure, each AWRR contains an RR arbiter, which can use any existing RR arbiter implementation [18-20].[1] The counter at each input port

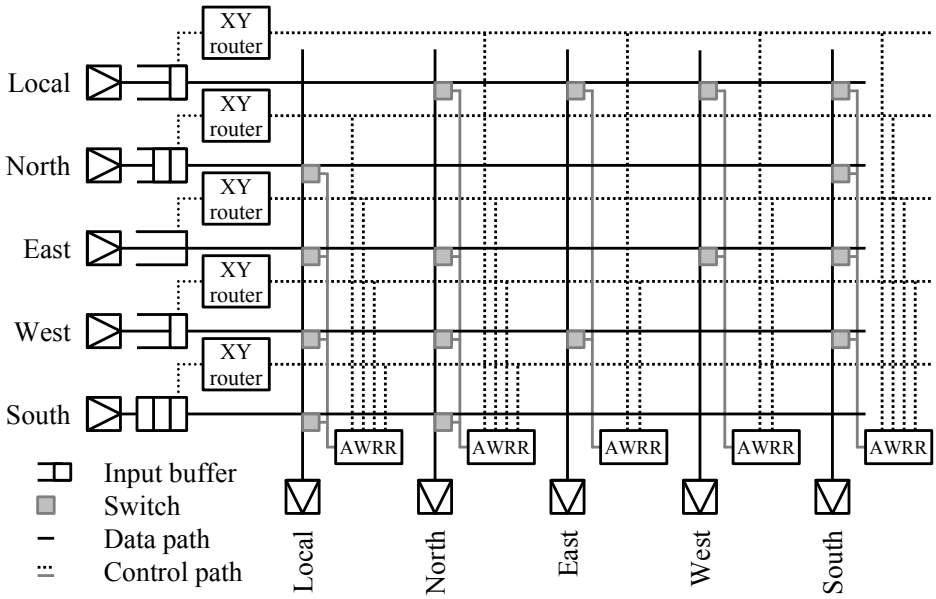
[1] When injection rate is low, RR can achieve EoS as shown in Figure 1.1 and the additional circuitry in AWRR except the RR arbiter can be considered a useless overhead. The low

is preset to the corresponding arbitration weights. Then it is decremented one by one whenever its input port gets granted. If a counter reaches zero, its input port cannot get any grant until the refresh period is reached. The refresh period is tracked by the refresh counter that gets decremented one by one when any grant is asserted. The value of the refresh period is the sum of all the input port arbitration weights.

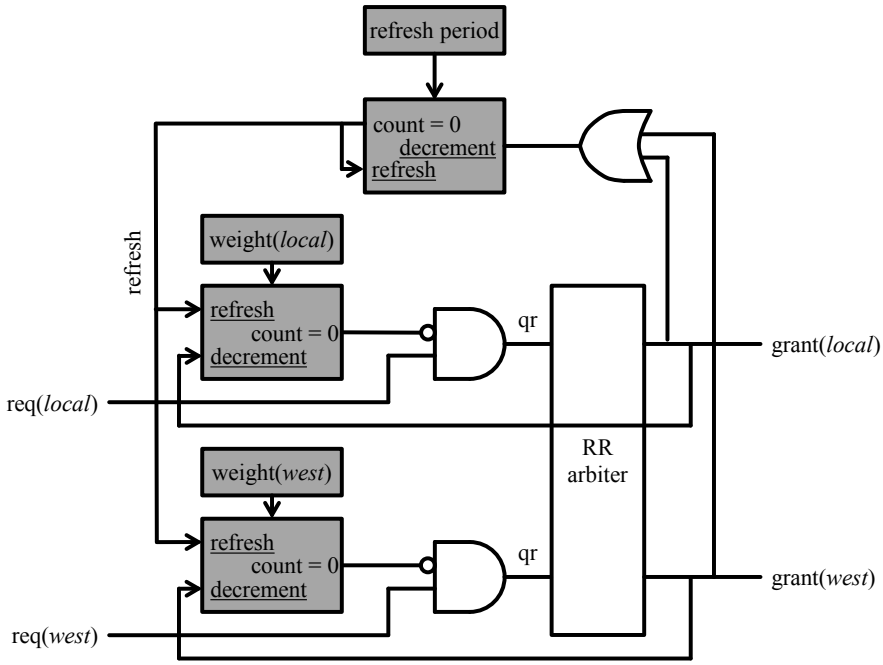
There can be cases where all the requests are from the input ports whose arbitration weights are zero. Instead of not granting any input (because their weights are zero) until the next refresh is accounted, I just apply the round-robin arbitration in those cases. Here the existing RR arbiter is utilized and thus no additional arbiter is needed.

---

injection rate can be detected by using utilization of input buffers, which can be locally determined within each router. The weight management part of an AWRR arbiter can be disabled when none of the input buffers in the arbiter are full, thus lowering the power consumption to the level of RR.

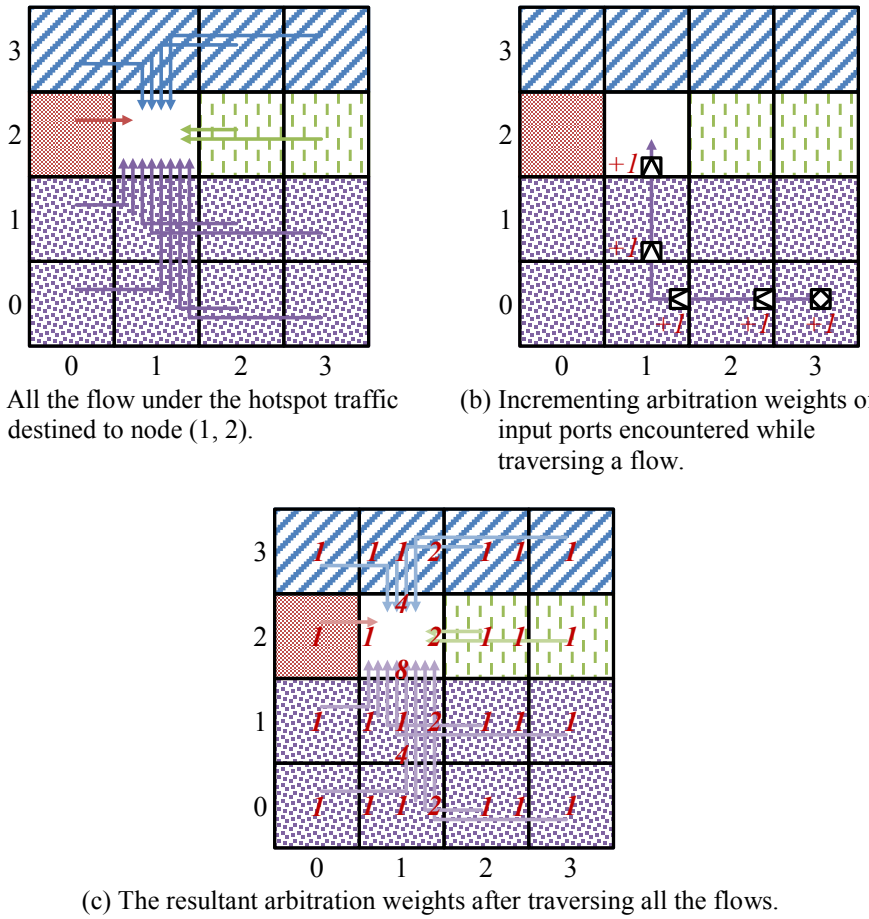


(a) Overall architecture of a router in 2D mesh network.



(b) Schematic diagram of an AWRR arbiter at the at output port.

Figure 4.2 Hardware implementation of AWRR.



(a) All the flow under the hotspot traffic destined to node (1, 2).

(b) Incrementing arbitration weights of input ports encountered while traversing a flow.

(c) The resultant arbitration weights after traversing all the flows.

Figure 4.3 Arbitration weight determination procedure example.

4.2 Arbitration Weight Determination

As describe in Chapter 3, the arbitration weights of PBWRR is determined by counting the number of nodes that should be served by each input port. In order to extend the weight determination procedure for AWRR, I should look at it in a

different way. Figure 4.3 is an example of how the input arbitration weights are determined in the different way when a hotspot is at the white node (1, 2). Figure 4.3(a) shows all the flows for the hotspot traffic. I follow each flow as in Figure 4.3(b) and increment the arbitration weights of all the encountered input ports. Traversing from node (3, 0) toward the hotspot node (1, 2), the arbitration weight of the input port of each router on the path is incremented by one. After repeating the procedure for all the flows, I get the arbitration weights of all the routers as shown in Figure 4.3(c). Note that compared to the weights determined by the original method of counting the nodes to be served, some arbitration weights are not calculated. For example, the arbitration weight of the north port of node (2, 1) in Figure 4.3(c) is not calculated since the port will never be used for this hotspot traffic pattern destined to node (1, 2).

The same concept and approach can be applied to traffic patterns other than hotspot and even to dynamically changing traffic patterns. That is, the aforementioned procedure of following each flow and incrementing the arbitration weights of input ports encountered can be thought of as an adaptive procedure that adjusts the arbitration weights at runtime. Whenever a process is mapped to a node, it sends control packets to each destination node it uses. The routers that the control packets get through increment their input arbitration weights as in Figure 4.3(b). Because XY routing ensures in-order packet delivery, workload packets can immediately follow the control packet without encountering zero input arbitration weight. On the other hand, when the task ends its execution, the node sends another

type of control packets to each destination node it has used to decrement the corresponding input arbitration weights. A program crash might incur missing weight-decrement control packets. Considering the program crashes are rare and non-zero arbitration weights other than their ideal values are acceptable in terms of functional correctness (although the global fairness might degrade), the weight-correction mechanism for this case is hardly necessary. If it is absolutely required for maintaining the global fairness, OS can track the access behavior of a program and correct the weights in those cases without incurring much performance overhead (because the cases are rare). The destination nodes just ignore both types of control packets.

Chapter 5

Experimental Results

5.1 Open-Loop Measurements

I build a cycle-accurate NoC simulator in SystemC [21] with TLM (transaction-level modeling) approach to perform fast experiments. Open-loop measurements are done as described in [7], in a way that packet injection rate is not affected by network congestion. Figure 5.1 presents the accepted traffic distributions of RR, PBWRR, and AWRR under three traffic patterns – bit complement, bit reverse, and tornado. The three traffic patterns are chosen because they are the representative traffic patterns that show the differences of the three arbitration schemes. 8×8 mesh with XY routing is used for the experiments. The AWRR always results in the best global fairness compared with the other arbitration mechanisms. Under the bit complement

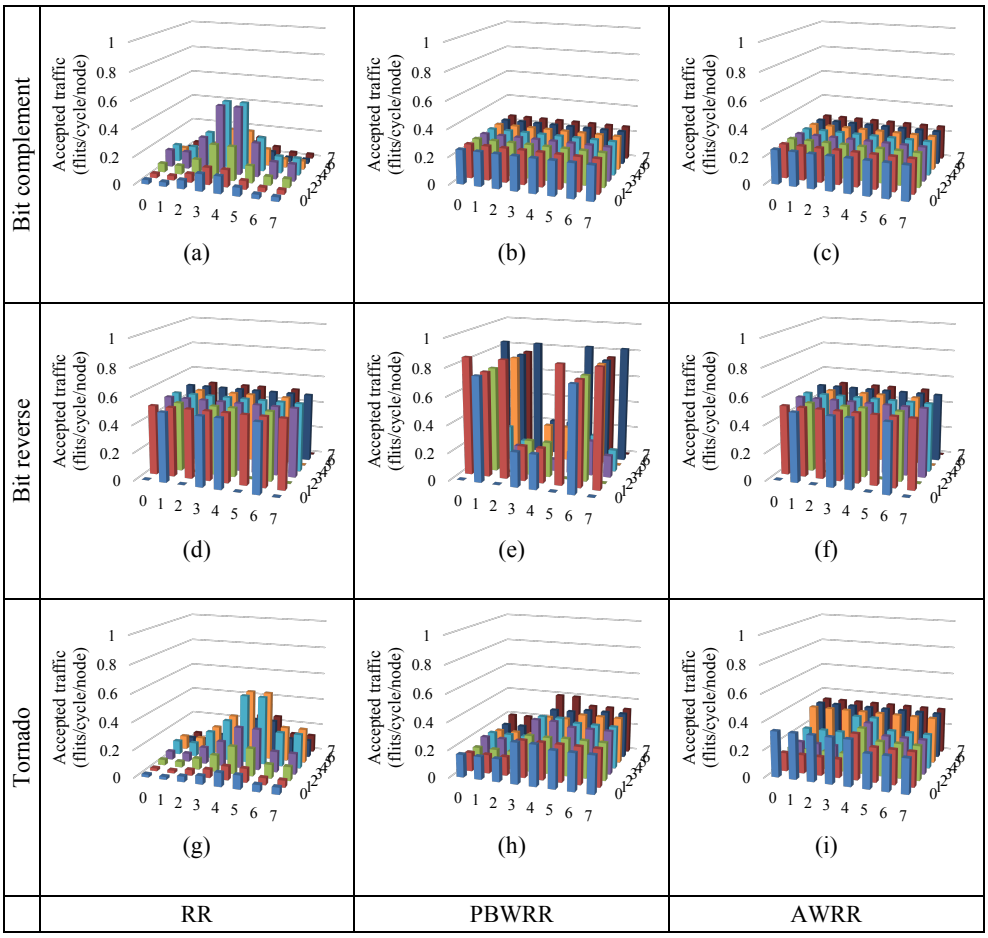


Figure 5.1 Accepted traffic distributions of RR, PBWRR, and AWRR under three representative traffic patterns – bit complement, bit reverse, and tornado. 8-ary 2-mesh, XY routing is used.

and the bit reverse traffic patterns, neither RR nor PBWRR achieves the global fairness, but the AWRR achieves the global fairness in both cases. Under the tornado traffic pattern, none of them achieves the perfect global fairness, while the AWRR provides the best global fairness.

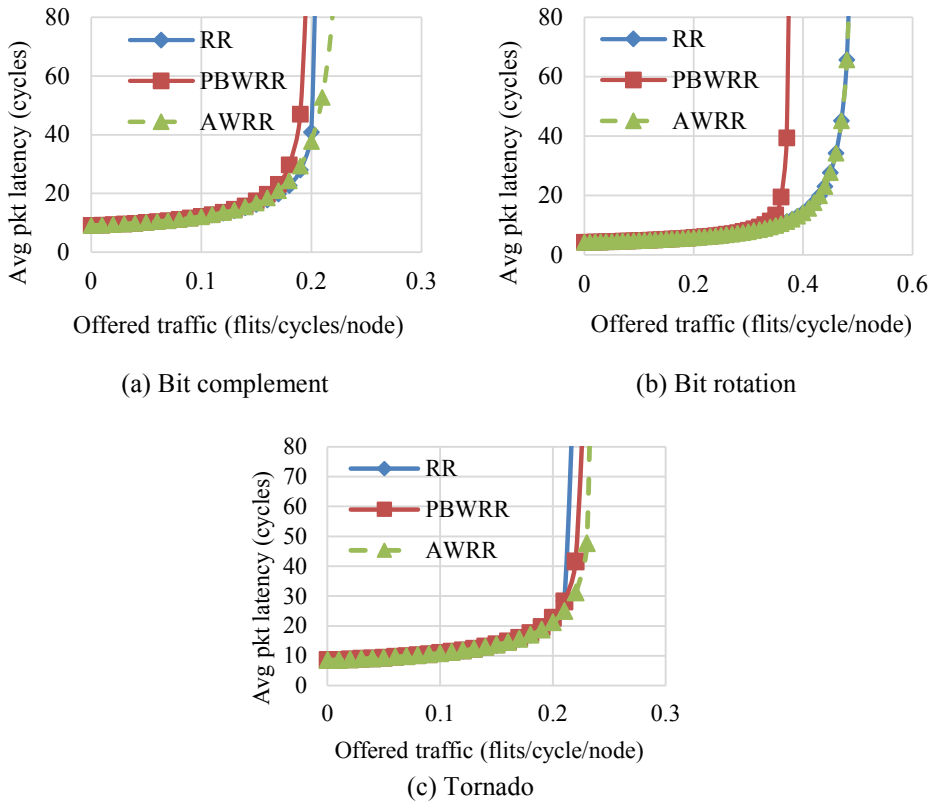


Figure 5.2 Average packet latency versus offered traffic curves for 8×8 mesh network with XY routing. The network is experimented with RR, PBWRR, and AWRR arbitration schemes under the following traffic patterns (For the bit reverse traffic pattern, refer to Figure 4.1(c)).

As mentioned in Chapter 4, PBWRR degrades the saturation throughput under some traffic patterns. Figure 4.1(c) and Figure 5.2(b) show that under the bit reverse and the bit rotation traffic patterns, the saturation throughput decreases by about 20%. This means that for those traffic patterns, the network becomes saturated easier and the system performance might be degraded. By applying the AWRR scheme, the drops in saturation throughput are completely removed as shown in the figures.

I also perform similar experiments under balanced traffic patterns, neighbor and uniform random. They are benign in terms of EoS characteristics of NoCs as can be seen from Figure 5.3. Generally speaking, with the balanced traffic patterns, AWRR does not show any abnormal behavior regarding EoS. Some interesting results worth mentioning are: (1) Under the neighbor traffic pattern, 1 flit/cycle/node from every node in the network can be sustained by all the arbitration mechanisms in concern. (2) For the AWRR arbitration under uniform random traffic pattern, the accepted traffic distribution is the same as that of PBWRR arbitration. This is because as stated in Section 4.1, the bit-widths of the arbitration weight registers are bounded by those of PBWRR, and the uniform random traffic pattern makes them saturated. Average packet latency versus offered traffic curve is also shown in Figure 5.4. Again, no abnormality is found with our approaches.

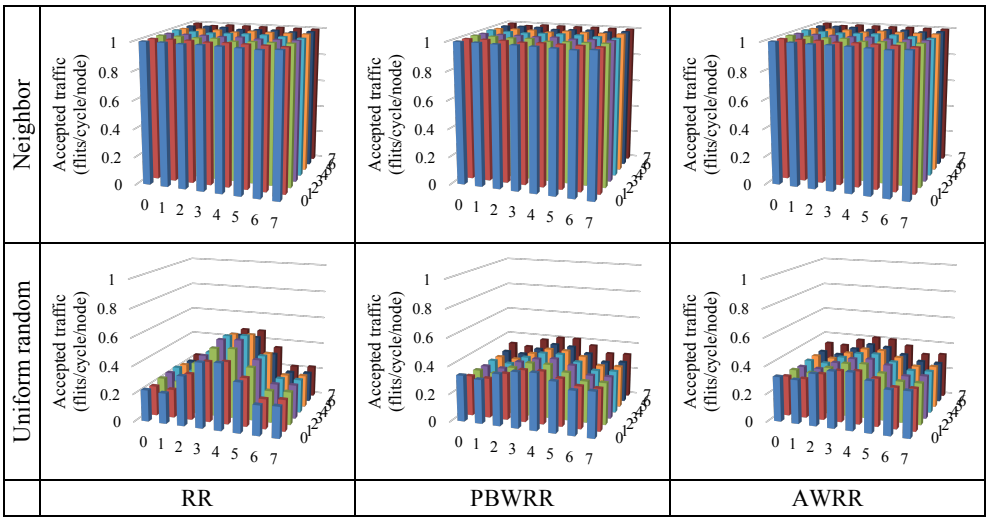
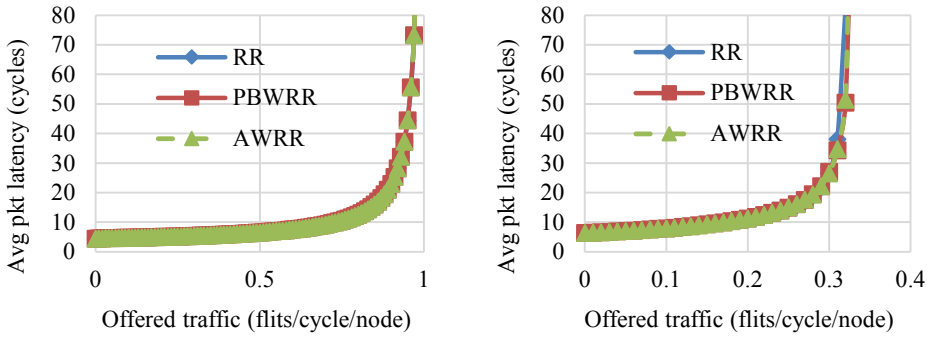


Figure 5.3 Accepted traffic distributions of RR, PBWRR, and AWRR under balanced traffic patterns – neighbor and uniform random. 8-ary 2-mesh, XY routing is used.

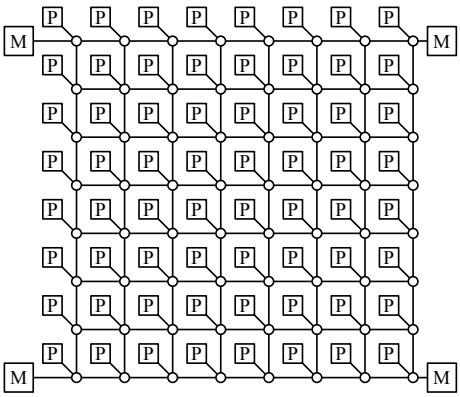
5.2 Closed-Loop Measurements

A CMP architecture assumed for the closed-loop measurements [7] is described in Figure 5.5. Four external memory controllers are attached to the corners of the mesh network similarly with SCC [22] and TILE64 [23]. In order to show the effectiveness of the congestion control, I deliberately choose the system with minimum resources. ZSim [24] is used for generating traces for the network transactions and those traces are used to drive our in-house SystemC NoC model. The NoC model is basically the same with the one used in the open-loop measurements, except that it is modified to reflect the CMP architecture assumed.



(a) Neighbor (b) Uniform random

Figure 5.4 Average packet latency versus offered traffic curves for 8×8 mesh network with XY routing. The network is experimented with RR, PBWRR, and AWRR arbitration schemes under the balanced traffic patterns.



(a) Components connection

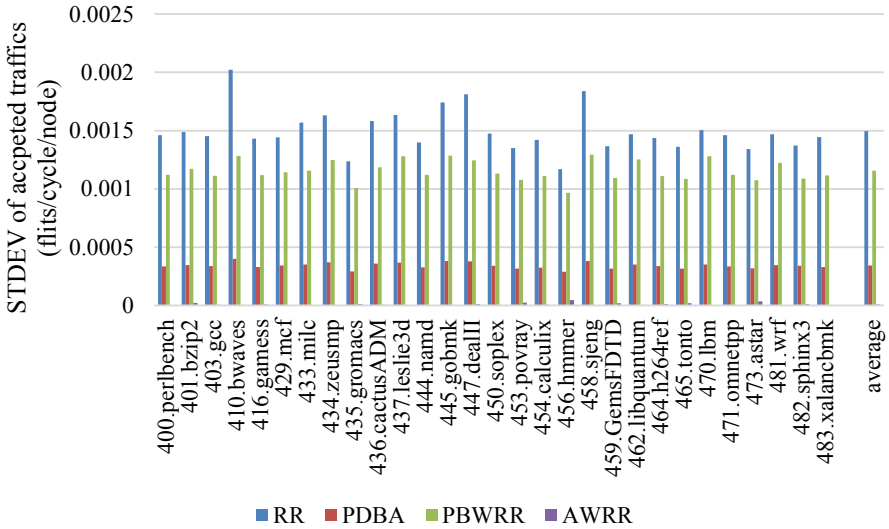
Cores	X86 OoO [24]
L1I/L1D caches	Private 8kB 4-way LRU
Network	8×8 mesh 1 cycle/hop 2 flits/input queue 8 flits/packet
Memory Ctrl	8 reqs/queue FCFS scheduling 100 cycles/access

(b) Configuration of the architecture

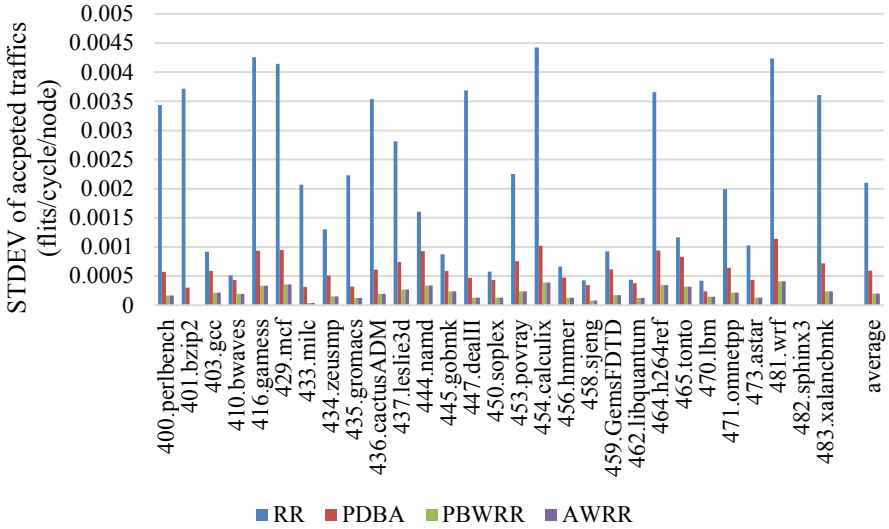
Figure 5.5 CMP architecture for the closed-loop measurements.

I perform experiments with SPEC CPU2006 benchmarks [9] run in multi-programmed fashion. The same benchmark application is run on every node in the CMP architecture. Figure 5.6 shows the standard deviations of the accepted traffic across all the 64 nodes on the network. The results are shown with the four arbitration

schemes – RR, PDBA, PBWRR, and AWRR. In Figure 5.6(a), the memory space of each execution is assumed to be homogeneously distributed across the four external memory banks thus one memory bank serves 16 benchmark executions. On the other hand, in Figure 5.6(b), each benchmark execution spreads its memory access to all the four memory banks to make use of the high memory bandwidth. The arbitration weights are preset according to Section 4.2. The control packets are assumed to be sent prior to the experiment. One control packet per application run is negligible compared to the total number of packets sent during the entire application run. We run each benchmark for 100 million cycles. As shown in the figure, AWRR performs best on EoS. Note that when one memory bank per benchmark execution is assumed, PBWRR results in worse EoS than PDBA, but it behaves better when four memory banks are fully utilized by each benchmark run. Here, we used idealized PDBA where infinite number of bits are used for the arbitration weights. In reality, to reduce the hardware overhead, PDBA uses 4 bits for the arbitration weights, resulting in worse EoS than the one shown in the figure. On the other hand, the results of PBWRR and AWRR arbitration schemes are not under any idealization assumptions where the arbitration weights are at most 6 bits wide for an 8×8 mesh network. Figure 5.6(b) shows the same results for PBWRR and AWRR, because the arbitration weights of AWRR get saturated under this memory mapping configuration. Note that the harmonic means of the accepted throughput in these cases are around 0.0045 so that the standard deviation of 0.0015 for RR indicates that the deviations are around $\frac{1}{3}$ of the average values.



(a) With one memory bank per benchmark execution.



(b) With four memory banks per benchmark execution.

Figure 5.6 Standard deviation of the accepted traffics across all the 64 nodes on the network.

Table 5.1 Network configuration for hardware implementation

Parameters	Values
Topology	8×8 mesh
Routing algorithm	XY routing
Flow control	Wormhole flow control
Input buffer size	3 flits/input port
Routing latency	1 cycle

5.3 Hardware Implementation

I have implemented routers with RR, PDBA, PBWRR, and AWRR in SystemVerilog [25], and compared them in terms of area, power consumption, and critical path delay. The network configuration for the hardware implementations are shown in Table 5.1. Synopsys Design Compiler [26] is used for the syntheses and the measurements of area, power, and critical path delay. For the standard cell library, Synopsys 32/28nm Generic Library [27] available from the Synopsys university program is used. Both the HVT (high threshold voltage) and RVT (regular threshold voltage) libraries are used for the synthesis. The designs are optimized toward maximum clock frequency and the default switching activities are used for the power measurements. With the default switching activities, all the signals are assumed to stay in 0 and 1 state with the same probability of 50 %, and the toggle rates are assumed to be 0.1 where signals change its state every 10 clock cycles on average. The routers of 5 input-output ports are analyzed.

Table 5.2 Synthesis results of routers with various arbitration schemes

		RR	PDBA [4]	PBWRR [8]	AWRR
Cell area (μm^2)		14358.36	24423.11	17511.67	18383.37
Power (μW)	**Dynamic**	225.66	232.53	290.14	245.00
	Leakage	296.41	535.14	362.19	381.58
Critical path delay (ns)		1.16	1.88	1.34	1.31

As the synthesis results in Table 5.2 show, the PBWRR and the AWRR have their characteristics closer to the RR arbiter, while having less area and shorter critical path delay than PDBA. The improved properties come from the fact that PDBA requires multiplications of input weights and random number generation as well as the scaling of the generated random numbers. It also needs separate input buffers for storing pre-calculated scaled random numbers.

Chapter 6

Conclusion

I proposed the adaptively weighted round-robin arbitration algorithm to provide EoS across CMP NoCs. It exploits the deterministic properties of the NoC to evenly distribute the service (e.g., throughput) from the shared resources to the requestors. In contrast to the previous approaches, it does not require any additional information carried in the headers of the packets. The implementation of the proposed arbitration scheme incurs smaller overhead in terms of die area and critical path delay. This is true when compared to the previous fair arbitrations, especially the probabilistic distance-based arbitration that involves multiplication operations and random number generation/scaling. The input arbitration weights can be reconfigured according to the flows on the network. Whenever a new task is mapped on, or an old task is removed from a node, control packets are sent to the destinations it

communicates with, and the arbitration weights of the input ports on the routing paths are updated. The reconfiguration mechanism enables the proposed NoC provide EoS under real applications as well as various synthetic traffic patterns. It also improves the saturation throughput under some traffic patterns compared to the approach that only addresses the network congestion from the hotspot traffic pattern.

Part II

Accelerating Pipelined Applications with Reconfigurable Bus-Mesh Communication Architecture in Chip Multiprocessors

Chapter 7

Introduction

The multi-core era has long begun since diminishing return from ILP exploitation, thermal and power issues due to the frequency scaling etc. As the number of cores on a chip increases, the communication fabric that conveys the messages among them became one of the system performance bottleneck. Buses, which were a common choice for the communication fabric, cannot sustain the heavy volume of the messages. Network-on-chip (NoC) [3] appeared as an alternative to the buses and now became the de factor standard for chip multiprocessors (CMPs). NoC consists of routers which are the distributed components that conveys packets which are the unit of communication among the cores.

One of the main characteristics of NoC is topology, which states in what shape the routers are connected to each other. The choice on topology decides the hop count

between the source and the destination nodes, and the width of a channel between routers, etc., and affects important performance metrics such as communication latency and throughput. Among many candidates, mesh is a common topology choice for CMPs because of its simplicity and regularity. In mesh topology, routers that are physically placed next to each other are connected, forming a 'mesh' when viewed from a distance. Although being the first consideration when designing a CMP, mesh has drawbacks, one of them being much increased hop count as the network size increases. This leads to increased communication latency thus to lower system performance.

There has been a lot of researches that addressed this issue. One of the solutions is to use high radix network topology such as flattened butterfly [28]. In the high radix topologies, physically distant routers are connected by long channels, reducing the number of hops between them. Another solution is to cluster the cores, as in concentrated mesh [29], so that the increase in the size of the network is suppressed as the number of cores in a CMP increases. Clusters are typically formed by connecting some cores together with local communication fabric and then integrating the clusters with a global communication fabric. Many local-global communication fabric pairs are proposed not only for CMPs but also for other forms of SoC. For CMP, buses are the first choice for the local communication fabric, because of its simplicity compared to others such as local NoC [30]. Moreover, practically, buses have been used as a communication fabric so that they are well optimized to connecting small number of cores together.

Based on the above mentioned advantages, [30] shows performance- and energy-wise advantages of bus-mesh communication architecture over the pure mesh, concentrated mesh, and flattened butterfly NoCs. The authors put an emphasis on the importance of efficient local communications, and the local communication is supported by fast and energy efficient local buses. The size of each cluster, which is heuristically decided, is 8 cores. The cluster size should be large enough so that as large portion of the communication as possible is covered by the local buses. On the other hand, it should not be too big to cause bus contention that negatively affects the system performance. The adequate size of the clusters is not clear without considering the applications that are going to be running on the CMP. Depending on the application, the amount of messages passed between the cores varies a lot.

Here rises the need for reconfigurability of the cluster sizes. If the local connections are reconfigurable, cores can be attached to one cluster to another and efficient data accesses are made possible. Segmented bus technique [31] naturally fits into bus-mesh communication architecture to make it reconfigurable. Physically adjacent local buses can be easily joined together or separated from each other by simple switches between them.

With this reconfigurable bus-mesh communication architecture, what remains is the principle for the reconfiguration in order to use it efficiently. I propose a systematic reconfiguration policy on the bus-mesh communication architecture for accelerating pipelined applications to reduce (or eliminate) data copy between the pipeline stages. Besides that, applying the reconfiguration policy reduces the amount

of network transactions and lower-level cache (or DRAM) accesses to further improve the system performance. I implement a full system simulator of the baseline architecture and its extension with bus segment reconfigurability, and show how much the technique improves the system performance.

Chapter 8

Backgrounds and Previous Work

8.1 Segmented Bus

Segmented bus technique is first proposed as an extension to the bus architecture to achieve high performance and low power. Figure 8.1 shows an example of how the technique is applied to a bus-connected system. There are four cores (C_i, $i \in \{0, 1, 2, 3\}$) connected by a bus, which is partitioned into 4 segments with 3 switches (S_i, $i \in \{0, 1, 2\}$). Depending on which switches are open and the others closed, the aspects on the performance and power consumption differ.

Let us assume that two pairs of cores – C_0 and C_1, and C_2 and C_3 – are communicating with each other. When connected by a single bus, the two pairs of cores cannot pass messages simultaneously, because of the bus contention. However,

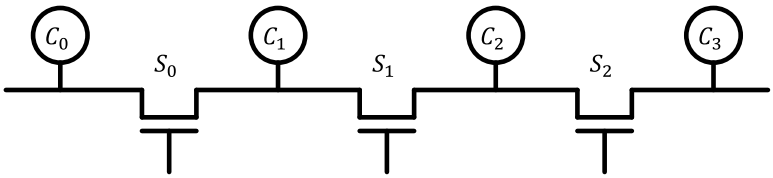


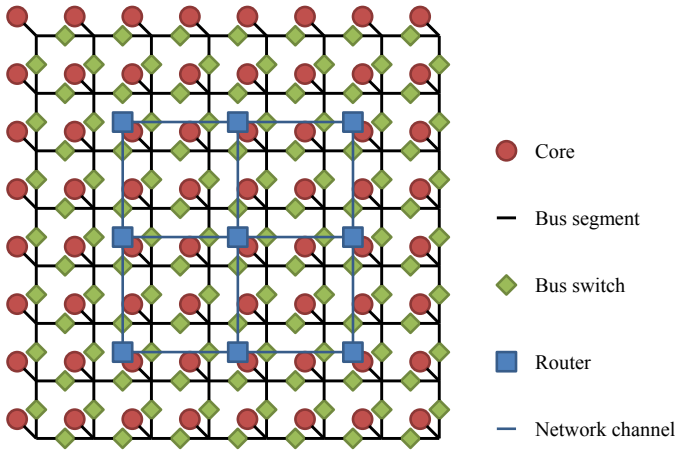
Figure 8.1 An example of segmented bus technique with 4 cores (C_i, $i \in \{0, 1, 2, 3\}$) and 3 switches (S_i, $i \in \{0, 1, 2\}$).

when S_1 is open and the bus is partitioned into two segments, the concurrent communication now can be performed thus increasing the system performance.

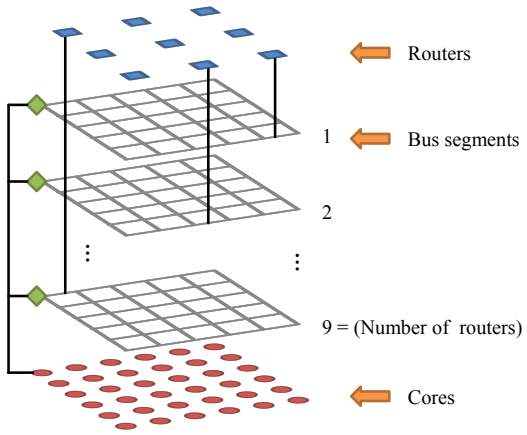
Now let us assume that only one pair of cores, C_0 and C_1, are communicating. The other cores, C_2 and C_3, need not know about the communication, though the signals are still propagated to them through a single bus. The longer the bus is, the more power it consumes because of bigger capacitance and resistance. If S_1 is made open, the signals are blocked to unrelated cores and power consumption decreases.

8.2 CMPs with Reconfigurable Bus-Mesh Communication Architecture

The representative CMPs with reconfigurable bus-mesh communication architecture utilizing bus segment techniques are RAMS [32] and DyaReMA [33]. Their overall architectures are shown in Figure 8.2.



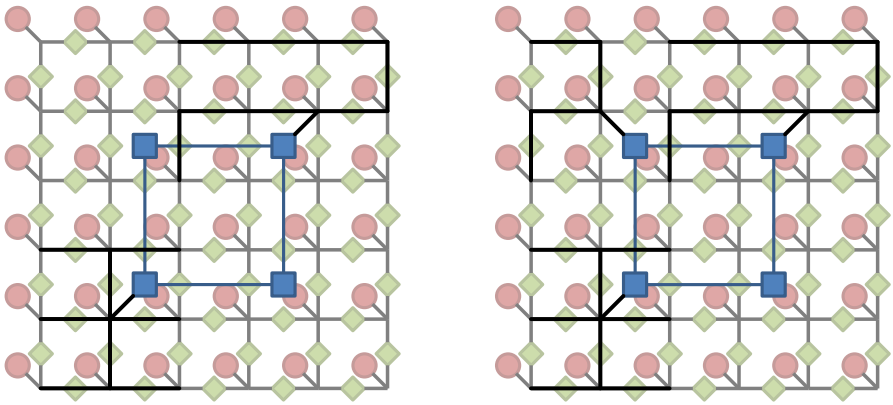
(a) RAMS



(b) DyaReMA

Figure 8.2 Overall architectures of RAMS and DyaReMA.

RAMS, which stands for Reconfigurable Architecture for Multicore Systems, consists of cores, each connected to a different bus segment (Figure 8.2(a)). The bus segments are interconnected with switches, forming a bus matrix. On top of the bus matrix, mesh NoC is implemented to conduct global communications. When a



(a) Before a process is mapped. (b) After the process is mapped.

Figure 8.3 Bus reconfiguration by a new process mapped on a RAMS CMP.

cluster is formed by closing adequate bus switches, it should be connected to a router to access resources of other clusters, such as L2 cache banks distributed across the network routers. In order to maximally utilize the cores on the chip, clustering should be done in a way that no cores are isolated and cannot be included in any cluster.

Bus reconfiguration is done whenever a new process is mapped onto a RAMS CMP. Figure 8.3 shows an example of a process mapping, when two processes are already running on a RAMS CMP (Figure 8.3(a)). The new process requires 5 cores. Two mapping positions are possible, the upper left cores and the lower right cores. On the upper left corner, 6 cores are available while on the lower right corner, 9 cores are available. The available site that has minimum, but more number of cores than the required number of cores is chosen, thus in this case, the new process is mapped onto the upper left cores of the CMP (Figure 8.3(b)).

DyaReMA stands for Dynamically Reconfigurable Multicore Architecture, and consists of one core layer, one NoC layer, and bus-matrix layers (Figure 8.2 (b)). There are the same number of bus-matrix layers with the number of routers in the NoC layer. Each bus-matrix layer is connected to each router in NoC. One bus matrix per router relaxes the configuration constraints on RAMS architecture. In RAMS architecture, when a cluster is formed, it blocks another cluster formation that physically crosses the existing cluster. However, in DyaReMA architecture, each cluster utilizes separate bus matrix so that no cluster blocks the formation of another.

The bus reconfiguration policy of DyaReMA is based on the data access pattern of each core. Each core counts the number of data accesses to each L2 cache bank distributed across the routers of NoC. When the difference between the number of remote L2 cache bank accesses and the number of local L2 cache bank accesses exceeds a predefined threshold, the core is migrated to the remote router that contains the remote L2 cache bank. In order to prevent all the cores from being migrated into a single router, the maximum number of cores per router is predefined. When a core is to be migrated into a cluster with the maximum number of cores, a core in the cluster is first moved out from the cluster and then the new core is migrated into that cluster.

In both RAMS and DyaReMA architectures, the length of buses are not taken into account when forming a cluster. The bus traversal latency increases when a distant cores are grouped together into a single cluster. This decreases the efficiency

of the local communications emphasized in designing a bus-mesh hierarchical communication architecture.

8.3 Near-Threshold Computing

As the name indicates, near-threshold computing (NTC) [34] indicates the operation of digital chips, especially CMPs, with its operating voltage (V_{DD}) dropped close to their threshold voltage (V_t). It achieves low power operation with reasonable performance drop.

A similar operating technique that is often compared with NTC is subthreshold computing (STC). With STC, the optimal operating V_{DD} with respect to the maximum energy efficiency (V_{opt}) is achievable. The faster a chip is, the more dynamic energy is consumed within short execution time while the less leakage energy is required. The slower a chip is, the more leakage energy is consumed while the less dynamic energy is required. The optimal point regarding both the dynamic and leakage energies is at the point where dynamic and leakage power components become nearly balanced [35].

Although STC achieves the maximum energy efficiency (about ~12-16x), the performance drop is significant (about 1000x), allowing its application only to environmental sensing applications and medical sensor applications. On the other hand, NTC targets high performance applications, achieving reasonable performance loss (about 10x) with about 6-8x energy savings [35].

In fact, V_{opt} depends on the actual activity factor (the parameter that denotes how often a circuit switches its value) of the digital circuit in consideration. The higher the activity factor is, the more important the reduction of dynamic energy reduction is, thus the less V_{opt} becomes. [35] addresses this issue and decides its basic cluster size to be four cores integrated with one L1 instruction and data cache. The reason for this decision is that cores have higher activity factor than L1 caches, thus the cores operate at lower operating voltage and frequency than L1 caches do. As the 4 cores are connected with one L1 cache, in the baseline setting (dynamic voltage and frequency scaling (DVFS) technique is employed), cores operate at 4x slower operating frequency than L1 caches do. In this way, the bandwidth requirements from the four cores are met by a single L1 cache. For the similar reason, the basic architecture assumed in this part of the thesis introduced in Chapter 9 has two cores clustered with one L1 data cache.

Chapter 9

Baseline Architecture

The overall structure of our baseline architecture, which resembles that of [35], is shown in Figure 9.1. Two cores are connected by a local bus. The two cores share L1 data cache while having private L1 instruction caches as in the figure. Instruction caches are made private in order to suppress burdens on the local bus. Two cores' sharing an L1 data cache might seem to be lack of bandwidth, but the operating frequencies of the two cores are one half of cache's operating frequency, balancing the request and the support on the data bandwidth. Mesh network integrates 48 clusters to form a CMP. Shared L2 cache consists of 16 banks and it occupies the central 16 sites of the mesh network. The L2 cache banks are controlled by distributed L2 cache controllers which are connected to the mesh NoC via network interface. When L2 miss occurs, L2 cache controller gets the missing data from

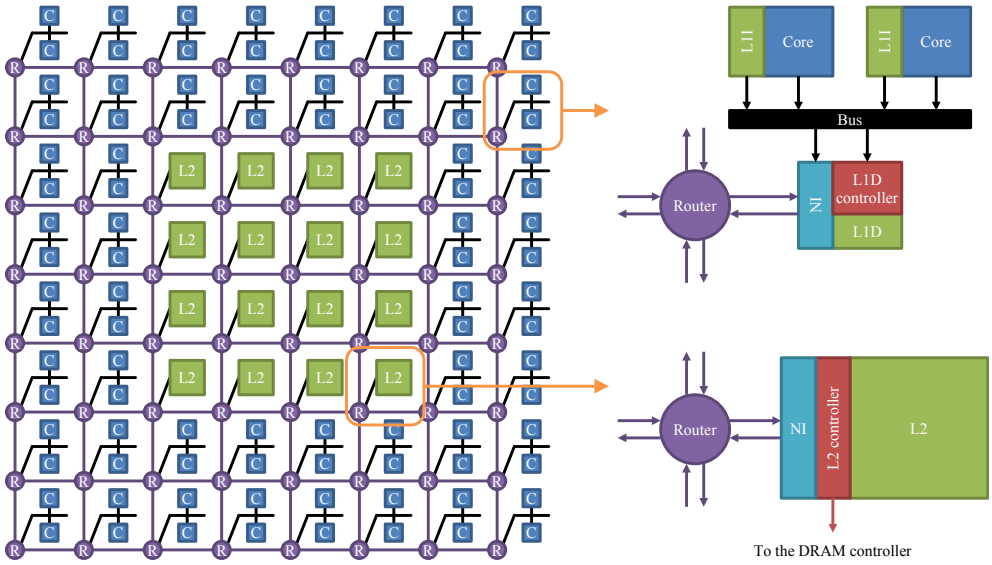


Figure 9.1 The overall structure of the baseline architecture

DRAM through the DRAM controller located outside of the CMP as shown in the figure.

The overall cache coherence protocol follows that of SGI Origin 2000 [36, 37]. Caches are kept coherent by hardware. MESI cache coherence protocol is employed. Each L2 bank keeps a directory to track the status of each cache block. Presence bit vectors are also kept together with the directories to track the existence of cache blocks in private L1 data caches.

Reply forwarding forwards requests to the cache blocks with exclusive directory state in order to reduce the amount of coherence transactions and the transaction latencies. In order to block another transaction that requests for the same cache block from tailoring the forwarded request, busy directory states are used to NACK the

later request. In this way, the requests are not queued either in the local nodes or the home nodes.

When clean cache blocks in private caches (cache blocks with exclusive or shared states) are replaced, replacement hints to the directories are not sent, reducing the transactions on the network. In order to back up the silent drop of clean cache blocks, speculative reads are employed so that the requests to the already-dropped-out exclusive cache block is serviced by L2 cache or DRAM. In the case of invalidation requests to already-dropped-out shared cache block can be just ignored by just replying with invalidation acknowledgment.

Sequential consistency is employed just as SGI Origin 2000. The architecture satisfies the following three sufficient conditions for guaranteeing the sequential consistency [37]:

1. Every process issues memory operations in program order.
2. After a write operation is issued, the issuing process waits for the write to complete before issuing its next operation.
3. After a read operation is issued, the issuing process waits for the read to complete, and for the write whose value is being returned by the read to complete, before issuing its next operation.

Bus snooping is not necessary because the two cores share their L1 data cache making them see the coherent data inherently. Buses are assumed to be 128-bit wide

so that a cache block of 4 words (16 bytes) can be transmitted in one clock cycle. The buses and the mesh network are integrated by network interfaces. The network interface packetizes (depacketizes) the requests from bus side components, such as L1 instruction cache controller and L1 data cache controller (the requests from NoC side components, such as L2 cache controllers and remote L1 data cache controllers). The (de)packetization latencies are assumed to be one clock cycle.

Mesh network also has its channel width of 128-bit, and 3-stage routers with route computation, switch allocation, and switch traversal stages, are used. One cycle per hop is assumed, based on the physical distance between neighboring routers. XY routing is used for routing the packets. No virtual channels are employed in order to keep the network transaction order (For a unique source-destination pair, the packets are received by the receiver in the order they are transmitted by the producer). In order to avoid message-dependent deadlock [38], there should be as many independent NoCs (or virtual channels) as there are request chains in coherence protocol (reply forwarding in this case). However, I assumed infinite depth on the input request queues for every component instead, to simplify the system implementation. Our research focus is not on the performance impact of the request queue depth, so this assumption is harmless on the experimental results of this thesis.

Chapter 10

Motivation

I would like to start the explanation on my proposal with a motivational example of multilayer perceptron (MLP). MLP consists of many neural layers as shown in Figure 10.1. When a stream of test cases needs to be classified, pipelining the MLP may result in higher throughput of the classification results. Let's assume that the pipelined MLP is implemented in software with multi-thread programming scheme. Each thread deals with each layer and synchronizes to the thread that deals with the previous layer to get the input data. It also synchronizes with the thread that deals with the next layer to pass its computation results. When mapped to a CMP architecture, separate core will be allocated to each thread in order to help incurring cores' context switches.

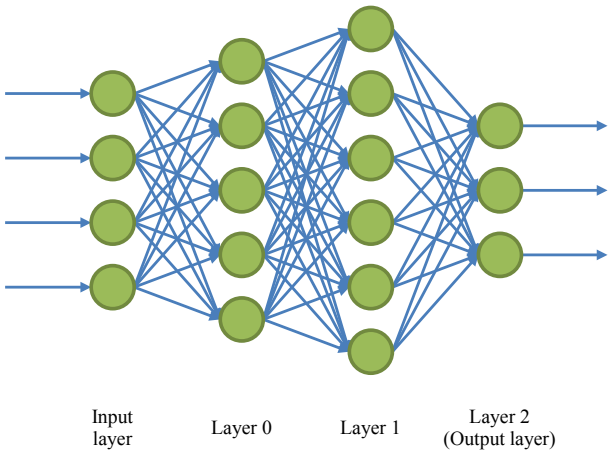
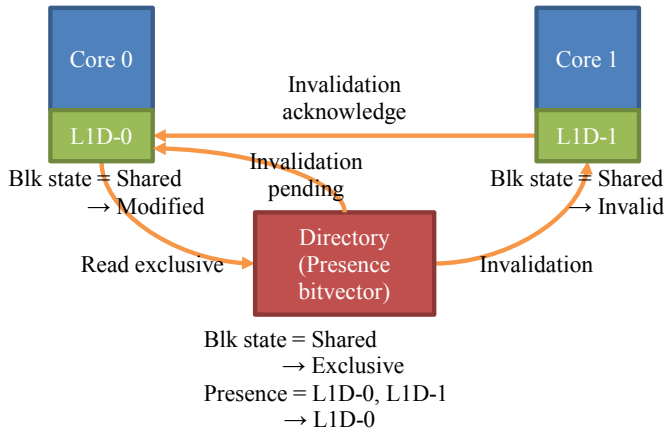


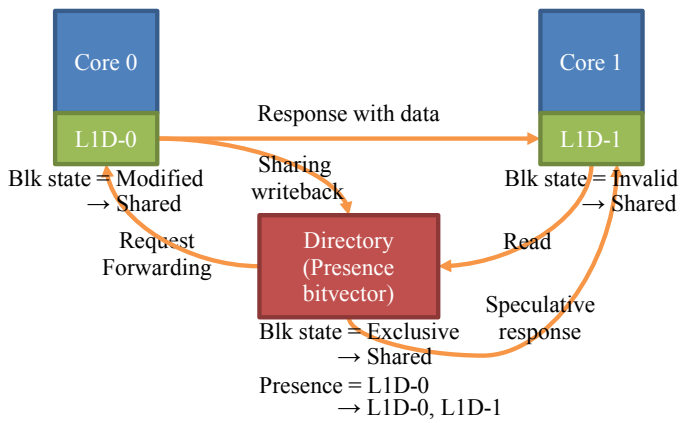
Figure 10.1 Multilayer perceptron (MLP) example.

In the baseline CMP architecture where bus reconfiguration is not supported, the communication between the neighboring threads are done with the closest shared cache, in this case, L2 cache. The cache coherence protocol automatically deals with the communication by cache block invalidation and data forwarding. Figure 10.2 shows this procedure. Let's assume thread-i is mapped to core-i where $i \in \{0, 1\}$. When thread-0 tries to write its computation results to its L1 data cache (L1D-0), the request visits the home directory and invalidates the corresponding cache blocks in thread-1's L1 data cache (L1D-1). After receiving the invalidation acknowledgment message from L1D-1, L1D-0 succeeds to write its computation results to L1D-0.

Now, thread-1 tries to read the result of thread-0. The request propagates to home directory in L2 cache, and finds out that the data is in L1D-0. The request is forwarded to L1D-0. In the meantime, the home directory sends back the speculative reply to L1D-1. Receiving the forwarded request, L1D-0 sends its computation



(a) Core-0's writing the calculation results.

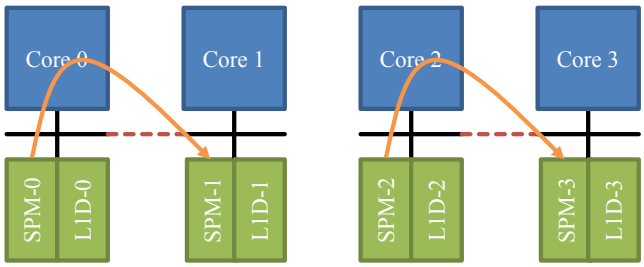


(b) Core-1's reading the calculation results.

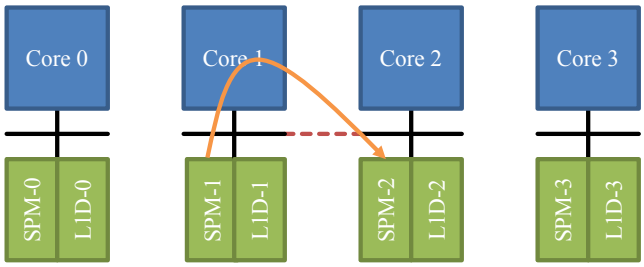
Figure 10.2 Data transfer procedure for neighboring threads.

results to L1D-1. When L1D-1 gets both the speculative reply from the home directory and the reply from L1D-0, the read transaction ends, choosing the appropriate reply between the two, in this case, the reply from L1D-0.

When the bus reconfiguration is supported, this process can be done in much simpler way. Figure 10.3 shows this procedure. Let's assume that thread-*i* is mapped



(a) Bus configuration 0



(b) Bus configuration 1

Figure 10.3 Bus reconfiguration procedure for accelerating MLP.

to core-i where $i \in \{0, 1, 2, 3\}$. In order to apply the technique being proposed, each active thread should have two memory banks in one configuration, one for its input and the other for its output. In Figure 10.3(a), thread-0 has its input memory bank as SPM-0 and the output memory bank as SPM-1. In the same figure, thread-2 utilizes SPM-2 and SPM-3 as its input and output memory bank. In Figure 10.3(b), thread-1 has its input memory bank as SPM-1 and the output memory bank as SPM-2. Note that while thread-0 and thread-2 are active, thread-1 should be inactive because thread-1's input data and output buffer are not ready yet. Thread-1 should wait for thread-0 and thread-2 to finish their work.

The memory banks are scratch pad memory (SPM) separated from L1 data cache. This separation of caches are similar with that of GPUs [39]. When thread-0 and thread-2 are calculating their results, buses are configured as in Figure 10.3(a). When the calculations are finished, the buses are configured as in Figure 10.3(b). As shown in the figure, the output SPM of thread-0 now became the input SPM of thread-1 and the input SPM of thread-2 now became the output SPM of thread-1.

Thread-1 can now perform its calculation without visiting the home node and waiting for the long cache coherence processes. This technique has chances of reducing the amount of data copy and network transactions, and improving the entire system performance.

Chapter 11

Reconfigurable Bus-Mesh Architecture

In this chapter, the motivation will be elaborated and detailed techniques will be introduced to implement the idea.

11.1 Thread Programming Model

The applications to be run on the proposed architecture is coded with thread programming model. Using the thread programming model, each pipeline stage is implemented as separate thread. The threads act asynchronously with respect to each other exploiting the task-level parallelism. The synchronizations of the threads are done when they access the shared memory space. The shared memory space is guarded by mutexes to ensure mutually exclusive accesses. Predicate variables are

```
struct stage1_to_stage2_t {
  bool valid;
  pthread_mutex_t mutex;
  pthread_cond_t cond_valid;
  pthread_cond_t cond_ready;

  uint32_t data[NELEM];
} s1_2;

struct stage2_to_stage3_t {
  bool valid;
  pthread_mutex_t mutex;
  pthread_cond_t cond_valid;
  pthread_cond_t cond_ready;

  uint32_t data[NELEM];
} s2_3;
```

(a) Data structures for the second pipeline stage.

```
 0  void *stage2(void *arg) {
 1    pthread_mutex_lock(&s1_2.mutex);
 2    while (true) {
 3      while (s1_2.valid == false)
 4        pthread_cond_wait(&s1_2.cond_valid,
 5                          &s1_2.mutex);
 6
 7      pthread_mutex_lock(&s2_3.mutex);
 8      while (s2_3.valid == true)
 9        pthread_cond_wait(&s2_3.cond_ready,
10                          &s2_3.mutex);
11
12      // Process the stage 2 here.
13
14      s1_2.valid = false;
15      pthread_cond_signal(&s1_2.cond_ready);
16
17      s2_3.valid = true;
18      pthread_cond_signal(&s2_3.cond_valid);
19      pthread_mutex_unlock(&s2_3.mutex);
20    }
21  }
```

(b) Thread routine for the second pipeline stage.

Figure 11.1 The second pipeline stage written in C++ with Pthread APIs.

used to indicate that the data stored in the shared memory space is valid, or that the shared memory space is ready for accepting the next data. Once a thread is generated to process a pipeline stage, it is blocked until its input shared memory space is filled with a valid data to be processed, and its output shared memory space is ready for accepting the next data. Whatever waiting policy threads follow – polling or waiting for signaling – the predicate variables are checked to see if the input data is ready or the output data memory space is ready.

Figure 11.1 shows an example thread routine for processing the second pipeline stage of a pipelined application and the data structure that describes its input and output shared memory space. C++ programming language and Pthread APIs [40] are used. As described above, each shared memory space data structure

(`stage1_to_stage2_t` and `stage2_to_stage3` in Figure 11.1(b)) contains mutex and predicate (`valid`) as well as shared data (`data`). Condition variables (`cond_valid` and `cond_ready`) are used to signal the blocked adjacent pipeline stages to start their execution. Compiler automatically inserts pads to the `structs` so that false-sharings do not occur.

The thread routine in Figure 11.1(b) also follows the description above. It first waits for the input shared memory space being filled with valid data by the previous pipeline stage thread (lines 1-5). Then, it waits for the output shared memory space being ready for the next data produced by itself (lines 7-10). After possessing both the input and output shared memory space, the thread performs its work. Then, it signals the previous pipeline stage for the input shared memory space (for the previous pipeline stage, this is the output shared memory space) being ready (lines 14-15), and signals the next pipeline stage for the output shared memory space (for the next pipeline stage, this is the input shared memory space) containing a valid data.

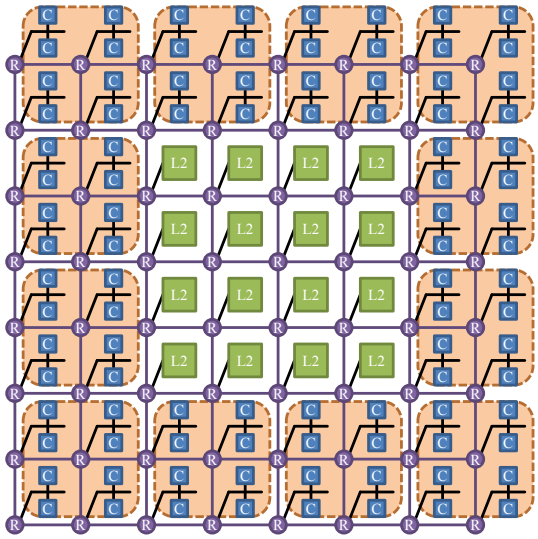
Supporting this programming model in the proposed architecture is as the following. First, addresses of shared memory spaces (`&s1_2` and `&s2_3` in the example in Figure 11.1) should be known by the bus decoders and L1 data cache controllers so that the requests to the data in the shared memory space is properly guided to the corresponding SPM. This can be easily understood with the schematic diagram in Figure 10.3. A bus contains two (L1 data cache, SPM) pairs, and it should decode the request addresses so that they are headed to a correct (L1 data cache,

SPM) pair. After receiving a request, the L1 data cache controller should identify the request if it is for the shared memory space or not. If it is, then the request is satisfied by the SPM, otherwise by the L1 data cache or other memories in the memory hierarchy.

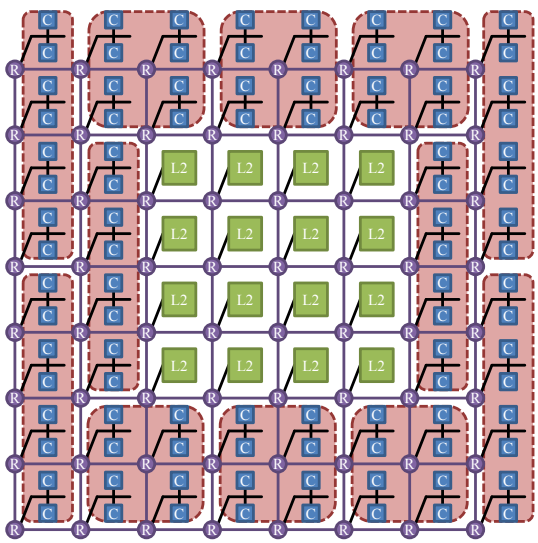
Second, the predicate values should be tracked by the bus reconfiguration controller, so that the bus reconfiguration controller can identify when it is allowed to reconfigure the bus connections. For example, the bus switch between the pipeline stage 1 and the pipeline stage2 can be made open when the work of pipeline stage 1 is over. That can be identified by `s1_2.valid` being true. The same bus switch can be made closed when the work of pipeline stage 2 is over. That can be identified by `s1_2.valid` being false. The tracking by the bus reconfiguration controller is simple. It just snoops on the bus and tracks the values of the predicates.

11.2 Cluster Size

Ideally, as long as input and output SPMs are separated, any shape of clustering is possible for accelerating a pipelined application. However, for the simplicity and regularity, I propose two bus configuration templates as shown in Figure 11.2. Note that this regularity leads to much simpler hardware structure as shown in later sections. The 96 cores are grouped into clusters of 8 cores each. The two templates are iterated while executing a pipelined application. Each cluster is allocated to a thread that processes a pipeline stage. Hetero-grained clustering is left as future work.



(a) Template 0



(b) Template 1

Figure 11.2 Two bus configuration templates for pipelined application acceleration.

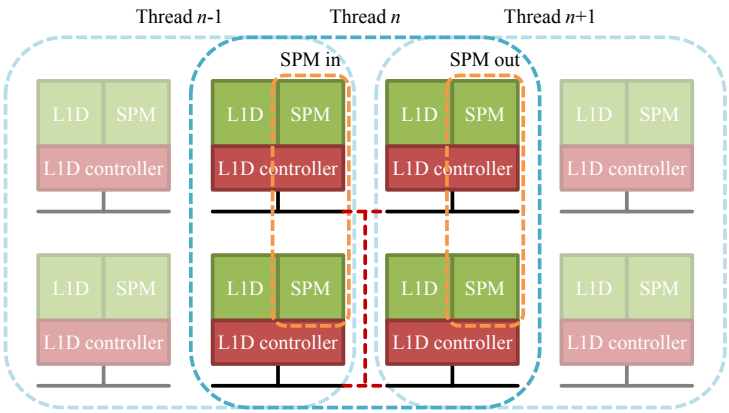


Figure 11.3 SPM address mapping in a cluster.

11.3 Organizing Multiple L1Ds and SPM Banks in a Cluster

In the bus configuration templates introduced in Section 11.2, 8 cores are grouped into a cluster. Because one L1 data cache is shared by two cores in the baseline architecture (Figure 9.1), grouping 8 cores into a cluster results in 4 L1 data cache banks per cluster. In our scheme, L1 data cache is partitioned into L1 data cache and SPM so there are 4 pairs of (L1D, SPM) in a cluster. How to organize and utilize them, specifically how addresses are mapped to each memory bank, greatly affects the system performance.

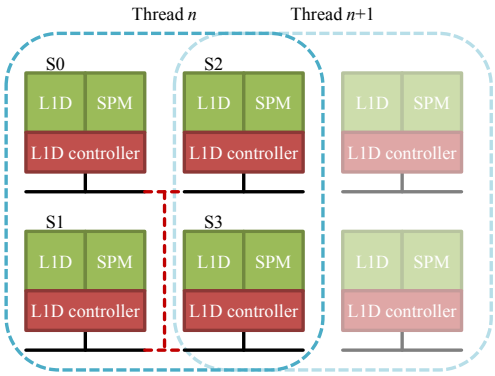
Four SPM banks in a cluster are utilized as shown in Figure 11.3. Before a pipelined application starts running, its shared memory addresses are mapped on the corresponding SPMs. The address mapping follows the physical location of neighboring threads. In Figure 11.3(a), SPM-in is located close to the previous thread

and SPM-out is located close to the next thread so that when bus segments are reconfigured, SPM-in is used by the previous thread and SPM-out is used by the next thread.

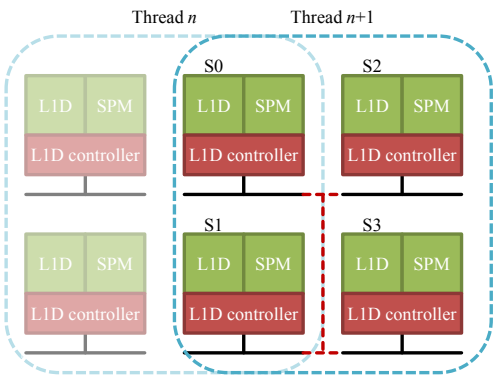
Utilization of L1 data cache banks are a little more complex. In general, if there are multiple cache banks, there are two ways of utilizing them. One is the way extension and the other is the set extension. When the way extension is employed, memory banks are utilized as separate cache ways. If four cache banks are grouped, as in the case of the bus configuration templates in Figure 11.2, together they form a 4-way cache. When the set extension is employed, memory banks are utilized as additional sets. In the case of four cache banks forming a single cache, the integrated cache has four times more cache sets than the original.

There are pros and cons to each technique. With the set extension, to which memory bank a request should be headed can be easily determined, just looking at its address. On the other hand, with the way extension, because single address can be headed to all the four memory banks, every time a request is emitted from the core, all the four memory banks should be checked if the corresponding cache block exists in the integrated cache. If the cache ways are managed all at once, this overhead is just as big as several gate delays, but in my case, because the four banks are physically distant and each has separate controllers, assuming the four memory banks are accessed via bus, the overhead is as big as several bus transaction delays.

Another aspect regarding the two techniques is how tags should be managed. With the set extension, whenever the number of sets changes, the tag bits should be



(a) L1D address (set) mapping in template 0



(b) L1D address (set) mapping in template 1

Figure 11.4 L1D address (set) mappings in bus configuration templates. Note that the sets covered by the two L1Ds in the middle changes from (S2, S3) to (S0, S1) when the template changes from 0 to 1.

changed too. In addition to that, when the number of sets changes, the index of a cache block changes too. That means when the number of sets changes, massive amount of cache blocks may need to be flushed or migrated to another cache bank. However, with the way extension, tags remain the same even if the number of ways changes, eliminating the need for such cache block flushes or migrations.

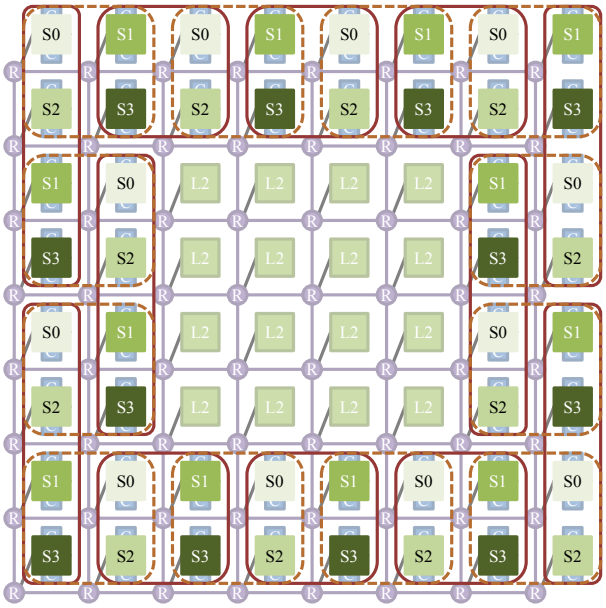


Figure 11.5 L1D address (set) mapping across the entire system so that the template change does not incur L1D data migration / flush.

In the proposed architecture, the set extension is chosen. The primary reason is the ease of determining which cache bank is responsible for a request. The system performance heavily depends on the access latency of L1 cache, and the way extension may increase the L1 data cache latency thus impact negatively on the system performance. Besides, because the cluster sizes are kept constant in the two bus configuration templates (Figure 11.2), L1D cache size per cluster does not change throughout the entire application execution.

One possible case where data flushes or migrations are needed is depicted in Figure 11.4. Let's say each of the four L1Ds ($\mathrm{L1D}_i$ where $i \in \{0, 1, 2, 3\}$) covers a set of cache blocks (S_i where $i \in \{0, 1, 2, 3\}$). For example, when each cache bank

has its size of 100 cache blocks and the direct-mapped policy is employed, then S_0 includes cache blocks whose block indices are $\{[0,100),[400,500),\ldots\}$. S_1 includes cache blocks whose block indices are $\{[100,200),[500,600),\ldots\}$. For S_2 and S_3, the cache block sets are $\{[200,300),[600,700),\ldots\}$ and $\{[300,400),[700,800),\ldots\}$, respectively. As the figure shows, if S_i's are not properly mapped, template change incurs changes of S_i that $L1D_i$ should cover. In the figure, the two L1Ds in the middle changes its set cover from (S_2, S_3) to (S_0, S_1), as the template changes from 0 to 1. I carefully designed the cache set mapping so that the template change does not incur any data flush or migration as in Figure 11.5. The orange dotted lines indicate clustering in template 0 and the red solid lines indicate clustering in template 1 (The coloring are the same as in Figure 11.2). Note that in any template case, every cluster has S_0, S_1, S_2, and S_3 in it, meaning all the memory space are covered by the L1Ds in each cluster.

11.4 L1 Data Cache / SPM Partitioning

When pipelined applications are being accelerated, L1 data caches should be partitioned into L1 data cache and SPM as mentioned in Chapter 10. Three types of partitions are made possible for (L1, SPM) pair, $\left(\frac{1}{4}C,\frac{3}{4}C\right)$, $\left(\frac{1}{2}C,\frac{1}{2}C\right)$, $\left(\frac{3}{4}C,\frac{1}{4}C\right)$, where C is the total size of L1 data cache without partitioning. These types of partitions are necessary in order to fully utilize the valuable cache memory space.

Table 11.1 Synthesis results for bus controllers of bus configuration templates

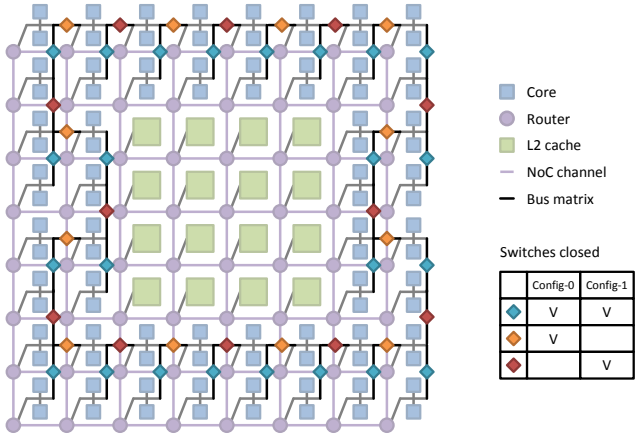
		Arbiter	Decoder
Cell area (μm²)		203.46	5485.00
Power (μW)	**Dynamic**	20.77	406.74
	Leakage	6.07	93.84
Critical path delay (ns)		0.76	0.19

For example, with only $\left(\frac{1}{2}C, \frac{1}{2}C\right)$ partitioning scheme, if the maximum size of data to be transferred between two pipeline stages is much less than $\frac{1}{2}C$, say $\frac{1}{5}C$, then L1 cache space of $\frac{3}{10}C$, which is $\frac{1}{2}C - \frac{1}{5}C$, is wasted without being used and may incur system performance degradation.

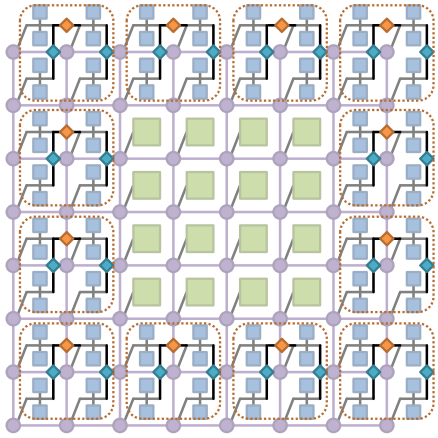
11.5 Reconfiguration Overheads

The reconfiguration overheads are twofold, the chip area overhead and the performance overhead.

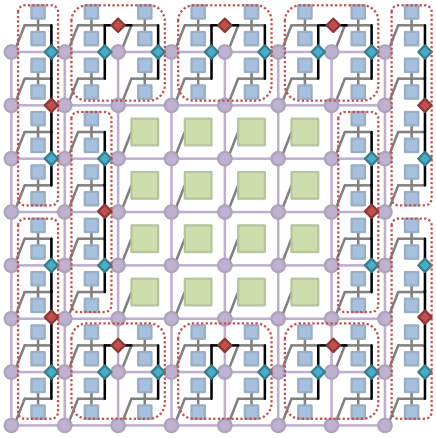
Chip area is increased from the baseline architecture by the bus controllers including arbiters and decoders, and the bus reconfiguration switches. The bus controllers for the templates 0 and 1 are added separately from those of the baseline architecture. Each controller should support 8 masters and 4 slaves corresponding to 8 cores and 4 L1 caches/SPMs in a cluster. I implemented the bus controllers and Table 11.1 shows the synthesis results. The results are for the bus controller of a single cluster. Since each bus configuration template contains 12 clusters, the total area overhead is 12 times bigger than the values in the table. The cores assumed in



(a) Overview



(b) Closed bus reconfiguration switches for the template 0.



(c) Closed bus reconfiguration switches for the template 1.

Figure 11.6 Bus reconfiguration switches

the architecture is ARM Cortex-A5 [41], which has the chip area of 0.27 mm^2. Comparing the chip areas of the bus controllers and the cores, the overheads are minimal.

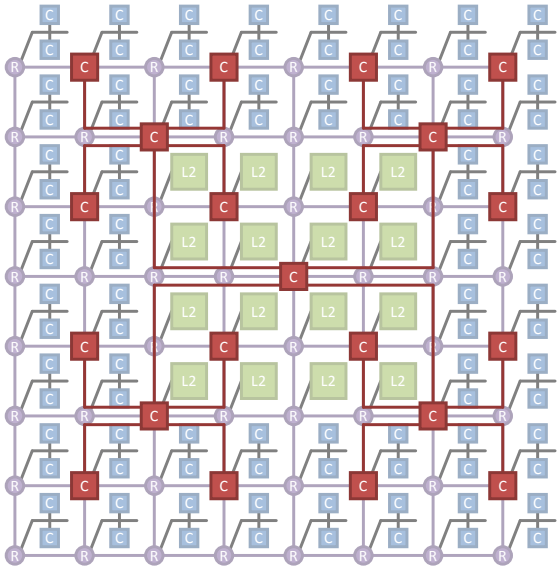


Figure 11.7 Control processors for the proposed architecture. The control processors are connected with tree NoC.

The bus reconfiguration switches needed for the bus configuration templates are shown in Figure 11.6. Figure 11.6(a) shows the overall number and the locations of the switches. The switches are organized as in Figure 11.6(b) and (c) for the bus configuration templates 0 and 1. The bus switches can be implemented with pass transistors, tri-state buffers, or multiplexers and for all the cases, the area overhead is minimal and can be ignored.

The performance overhead is twofold, one from waiting for the completions of the outstanding transactions and the other from thread synchronization control. Whenever bus reconfigurations are performed, there should be no outstanding transactions on going, because if there are some, the bus reconfiguration may cause miss-delivered transactions resulting in the system corruption. This elimination of

outstanding transactions are performed by signaling cores not to emit any transactions until the bus reconfiguration is done. The overhead incurred by this process is not heavy, though, because with sequential consistency, each thread only has single outstanding transaction at most.

The thread synchronization is controlled by the control processors assumed for the proposed architecture. Figure 11.7 shows the total number and the locations of the control processors. The control processors are connected by the 4-ary 2-tree network for the fast communications among them. The control packets usually requires low bandwidth thus tree is a good choice for the connection. The thread completions are locally detected by the control processors at the leaves of the tree network. Then, the completion information is gathered by the control processor located at the root node of the tree network. When all the running threads complete their own pipeline stages and the completion information is gathered by the root control processor, the bus reconfiguration orders are broadcasted to the leaf control processors. Getting the bus reconfiguration orders, the leaf control processors perform the bus reconfiguration. The approximate bus reconfiguration overhead can be calculated as: 4 cycles of detecting thread completion by leaf control processors, 2 times of (de) packetization by root and leaf control processors (2 cycles each), up- and down-ward tree NoC traversal (4 cycles/hop × 2 hops each), and 2 cycles of completion information aggregation by the root control processor. In total, 26 cycles are required for each bus reconfiguration.

Chapter 12

Experimental Results

12.1 Pipelined Applications

The pipelined applications used for the evaluation of the proposed architecture are multilayer perceptron (MLP) [42], convolutional neural network (CNN) [43], and JPEG decoder [44]. For MLP and CNN, only the forward propagation is performed with MNIST dataset [45]. In-house back propagation and Caffe [46] is used for deciding the weights of the neural networks. For JPEG decoder, baseline profile is used. All of them are coded from the scratch following the programming model introduced in Section 11.1. Their pipeline stages are shown in Figure 12.1. The detailed configurations for each application is given in Table 12.1, Table 12.2, and Table 12.3.

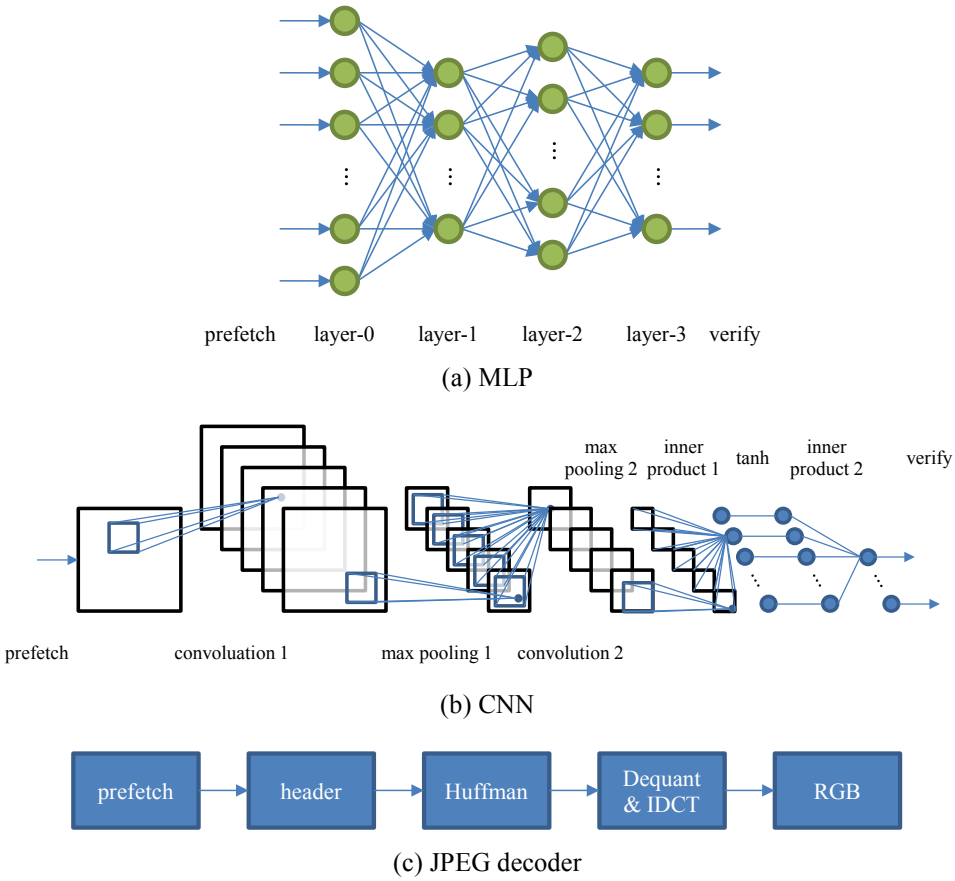


Figure 12.1 Pipeline stages of applications used in experiments.

For all the applications, the configurations are decided so that the size of their shared data does not exceed that of SPMs and the size of their private data does not exceed that of L1 data caches. For MLP, the largest SPM is required for the layer-0, where the biggest number of neurons exist. For the MNIST test set used for the experiments, it has $28 \times 28 = 784$ neurons (MNIST consists of images of handwritten digits whose size is 28×28 pixels). Each neuron has `float` (4 bytes

Table 12.1 MLP Configuration for the Experiments

Test data set	MNIST, 100 test cases
Layer descriptions	• Prefetch ○ Prefetches test set data from DRAM to L1Ds. • Layer 0: 784 neurons • Layer 1: 10 neurons • Layer 2: 20 neurons • Layer 3: 10 neurons • Verify: Verifies the classification results.
Data precision	4-byte `float`
MLP Accuracy	91 %

each) typed activation data, therefore SPM size needed is about 3 KB. In the case of storage requirements for weights, the maximum occurs between the layer-0 and the layer-1. Because the layer-0 has 784 neurons and the layer-1 has 10 neurons, the storage requirement is (each weight is stored in `float` type as activations): $784 \times 10 \times 4$, which is slightly less than 32 KB.

For CNN and JPEG decoder, the similar calculations are done and the experimental configurations are set so that the sizes of the shared data and the private data do not exceed those of SPMs and L1 data caches per cluster.

Table 12.2 CNN Configuration for the Experiments

Reference CNN	LeNet: But the size is reduced to make data fit into SPM and L1D.
Test data set	MNIST, 100 test cases

Layer descriptions	• Prefetch
	  o Prefetches test set data from DRAM to L1Ds.
	• Convolution 1:
	  o Input: 28×28 pixels grey-scale image (MNIST)
	  o 5×5 filter with stride 1
	  o Output: 5 feature maps with 24×24 elements each
	• Max pooing 1:
	  o 2×2 filter with stride 2
	  o Output: 5 feature maps with 12×12 elements each
	• Convolution 2:
	  o 5×5 filter with stride 1
	  o Output: 5 feature maps with 8×8 elements each
	• Max pooling 2:
	  o 2×2 filter with stride 2
	  o Output: 5 feature maps with 4×4 elements each
	• Inner product 1: 40 neurons
	• tanh: 40 neurons
	• Inner product 2: 10 neurons
	• Verify: Verifies the classification results.

Data precision	4-byte `float`
CNN Accuracy	98 %

Table 12.3 JPEG Decoder Configuration for the Experiments

Profile	Baseline
Decoded image	One image of 48×64 pixels

12.2 Simulation Environment

The full system simulator is implemented to conduct experiments. ZSim [24] is used as a functional simulator to get the stream of instruction and data memory traces. Simple core of 1 CPI is assumed so that timing simulation model is simplified. Other structures including bus, cache controllers, coherence controllers, caches, network interfaces, mesh network, and DRAMs are all implemented from the scratch using

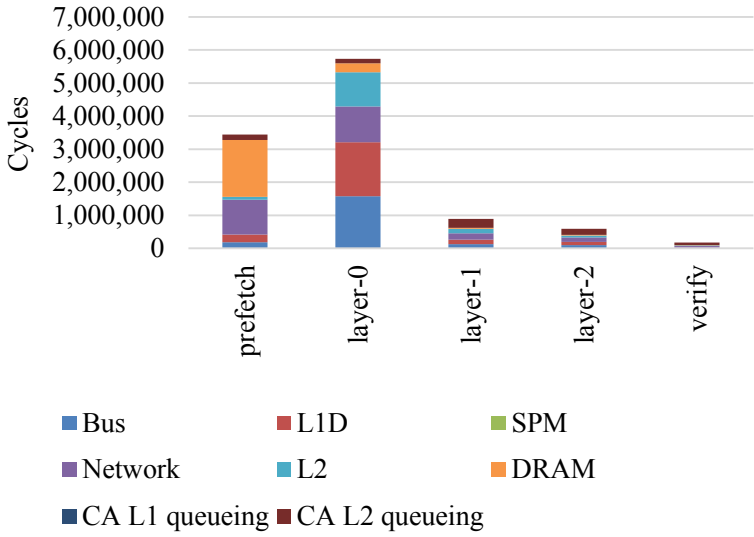
Table 12.4 System Configuration

Core operating frequency	400 MHz
System operating frequency	400 MHz
Cache block size	16 B
L1 instruction cache	Private Size = 32KB Direct mapped 1 cycle/read, 1 cycle/write
L1 data cache	Private Size = 32 KB Direct mapped 1 cycle/read, 1 cycle/write
L2 cache	Unified, Shared Size = 256 KB/bank, 16 banks 8-way set associative 10 cycles/read, 10 cycles/write
DRAM	50 cycles/read, 50 cycles/write

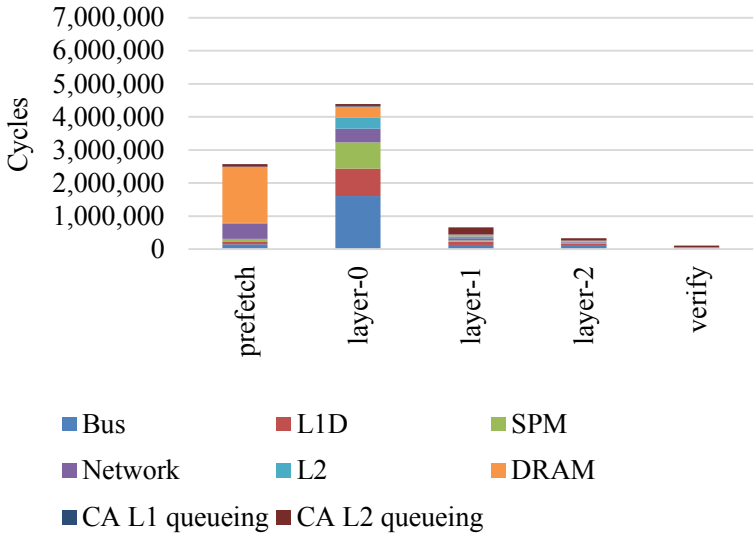
SystemC [21]. TLM technique is used for implementing them. The detailed system specifications are given in Table 12.4.

12.3 Memory Operations' Latency Breakdown

Figure 12.2, Figure 12.3, and Figure 12.4 show the memory operations' latency breakdown of each thread in MLP, CNN, and JPEG decoder, respectively. The execution time reductions are listed in Table 12.5. For MLP, CNN, and JPEG decoder, 21.75 %, 14.40 %, and 12.74 % reduction in execution cycles are achieved. In all the applications, the speed up is achieved by reducing the L2/DRAM accesses. Additional effect of the reduction in L2/DRAM accesses is the reduction in network traversal latencies and queuing delays in L2 cache/coherence controllers.

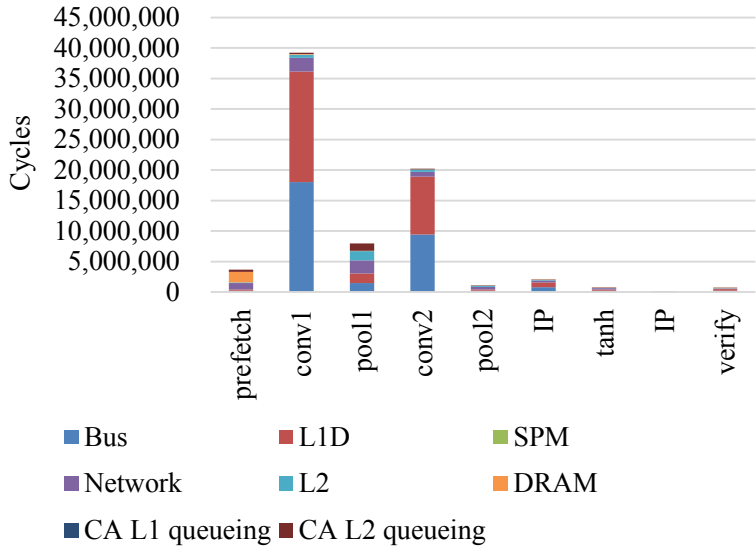


(a) Baseline

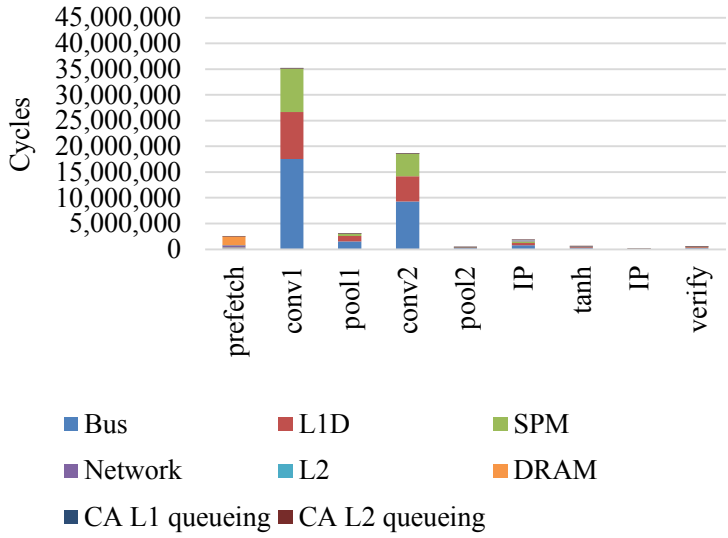


(b) Bus-reconfiguration

Figure 12.2 MLP – Memory operations' latency breakdown.

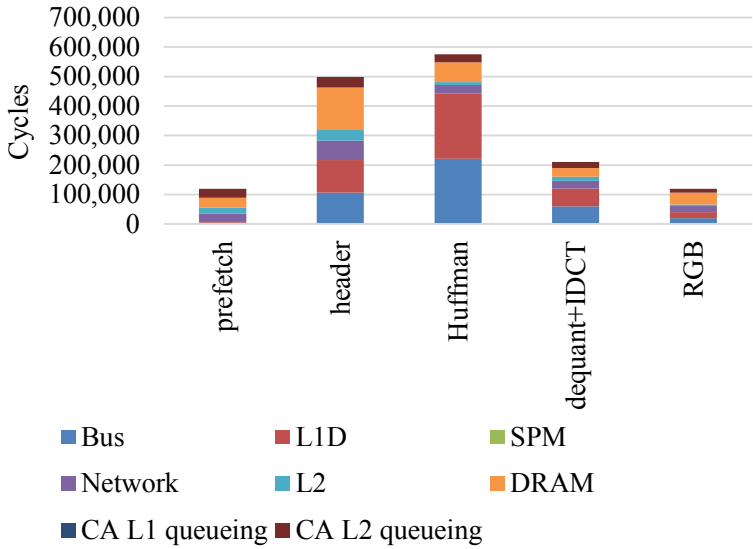


(a) Baseline

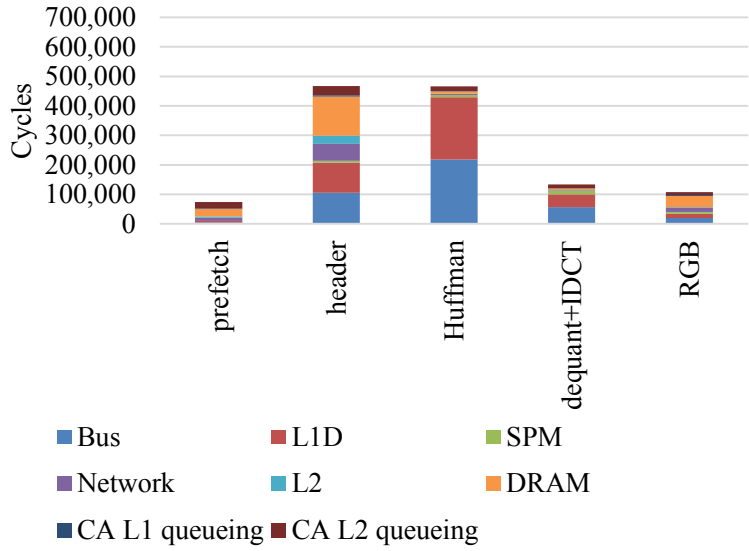


(b) Bus-reconfiguration

Figure 12.3 CNN – Memory operations' latency breakdown.



(a) Baseline



(b) Bus-reconfiguration

Figure 12.4 JPEG decoder – Memory operations' latency breakdown.

Table 12.5 Changes in Total Execution Cycle

Application	Baseline	Bus reconfiguration	Improvement
MLP	11,233,692	8,789,851	21.75 %
CNN	70,238,226	65,753,718	14.40 %
JPEG decoder	1,980,846	1,728,505	12.74 %

The expected reduction in network and L2/DRAM access latencies can be found in threads that process layers of MLP and CNN. The Huffman, dequant+IDCT, and RGB threads of JPEG also show some improvements. In the proposed architecture, reduced L1 data cache access latencies are moved to SPM access latencies.

The prefetch stage of MLP and CNN has L2/DRAM latencies not so much reduced by the proposed architecture. This is because the prefetch stages read the input data from DRAM and writes the out data to L2 caches (except for the first cold miss cases) in the baseline architecture. The L2/DRAM latency portion of the baseline architecture that is to be reduced by the proposed architecture is the portion with respect to the output data, which L2 cache accesses. The L2 cache accesses take much less time than DRAM accesses, resulting in small latency decrease in the prefetch stages. The prefetch stage of JPEG decoder, however, has decrease of the L2/DRAM latency in the proposed architecture to half of the baseline architecture. This is because in this case, the prefetch stage reas the input data from DRAM and writes the out data to DRAM. MLP and CNN processe 100 test cases while JPEG decoder processes 1 image. Therefore, while the cold misses of MLP and CNN's prefetch stages are amortized, the JPEG decoder's prefech stage does not have chance to use temporal data locality.

In CNN, threads that deal with convolution layers have much more accesses to L1 data caches than L2/DRAM inherently. Therefore there is less latency decrease compared to MLP application. This is because convolution layer reuses small convolution filter over and over again. These convolution layers became the bottleneck in accelerating CNN with the proposed technique.

The header stage of the JPEG decoder has its network and L2/DRAM latencies nearly not reduced, because it utilizes stack memory space a lot. Especially, the formation of the quantization tables and Huffman tables require a lot of stack memory space, due to the way they are stored in the JPEG file and the zig-zag reordering of the decoded tables in the header stage.

The Huffman stage of the JPEG decoder has much more L1 data cache accesses than L2/DRAM accesses, and this results in small decrease of the overall memory operations' latency in that stage. The reason is that the Huffman stage reuses small Huffman tables over and over again.

Chapter 13

Conclusion

I proposed the reconfigurable bus-mesh architecture to accelerate pipelined applications programmed in multi-thread programming model. Each pipeline stage is programed as a separate thread, exploiting the task-level parallelism. In the proposed architecture, the data that should be transferred from one pipeline stages to the next are not copied to the next stage's private cache. Rather, the cache of the current stage is physically handed over to the next pipeline stage by bus reconfiguration. In this way, unnecessary data copies are removed. Considering the way data copies between the successive pipeline stages are done in conventional cache coherent architectures, the proposed architecture has additional advantages. In the conventional cache coherent architectures, to read the data that is in dirty state in previous stage's private cache, the data first should be copied into a lower level

cache/memory that is shared by the pipeline stages. After updating the directory information in the shared lower cache, the data is transferred from the previous stage's private cache to the next stage's private cache. This coherence maintenance process involves not only data copies among various memories in the memory hierarchy but also network traversals. The network traversals can also be removed by the proposed architecture. The simulation model and the example pipelined applications – multilayer perceptron (MLP), convolutional neural network (CNN), and JPEG decoder – are implemented for the experiments. The experimental results show that 21.75 %, 14.40 %, and 12.74 % performance improvements are achieved for MLP, CNN, and JPEG decoder, respectively by applying the proposed technique.

Bibliography

[1] H. Park and K. Choi, "Adaptively weighted round-robin arbitration for equality of service in a many-core network-on-chip," *IET Computers & Digital Techniques,* vol. 10, no. 1, pp. 37-44, 2016.

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufmann Pubilshers, 2011.

[3] W. J. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," in *Design Automation Conference (DAC)*, Las Vegas, Nevada, USA, 2001, pp. 684-689.

[4] G. Kim, M. M.-J. Lee, J. Kim, J. W. Lee, D. Abts, and M. Marty, "Low-overhead network-on-chip support for location-oblivious task placement," *IEEE Transactions on Computers,* vol. 63, no. 6, pp. 1487-1500, 2014.

[5] G. D. Micheli and L. Benini, *Networks on Chips: Technology and Tools*, Morgan Kaufmann Publishers, 2006.

[6] G. F. Pfister and V. A. Norton, ""Hot spot" contention and combining in multistage interconnection networks," *IEEE Transactions on Computers,* vol. C-34, no. 10, pp. 943-948, 1985.

[7] W. J. Dally and B. Towles, *Principles and Practices of Interconnection Networks*, Morgan Kaufmann Publishers, 2003.

[8] H. Park and K. Choi, "Position-based weighted round-robin arbitration for equality of service in many-core network-on-chips," in *International Workshop on Network on Chip Architecture (NoCArc)*, Vancouver, British Columbia, Canada, 2012, pp. 51-56.

[9] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News,* vol. 34, no. 4, pp. 1-17, 2006.

[10] J. W. Lee, M. C. Ng, and K. Asanović, "Globally-synchronized frames for guaranteed quality-of-service in on-chip networks," in *International Symposium on Computer Architecture (ISCA)*, 2008, pp. 89-100.

[11] N. Alfaraj, J. Zhang, Y. Xu, and J. Chao, "HOPE: Hotspot congestion control for Clos network on chip," in *International Symposium on Networks-on-Chip (NOCS)*, 2011, pp. 17-24.

[12] Y. Yao and Z. Lu, "Fuzzy flow regulation for network-on-chip based chip multiprocessors systems," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014, pp. 343-348.

[13] S. Foroutan, A. Sheibanyrad, and F. Pétrot, "Cost-efficient buffer sizing in shared-memory 3D-MPSoCs using wide I/O interfaces," in *Design Automation Conference (DAC)*, San Francisco, California, 2012, pp. 366-375.

[14] D. Abts and D. Weisser, "Age-based packet arbitration in large-radix *k*-ary *n*-cubes," in *Supercomputing (SC)*, 2007, pp. 1-11.

[15] J. Hu and R. Marculescu, "Application-specific buffer space allocation for networks-on-chip router design," in *International Conference on Computer-Aided Design (ICCAD)*, 2004, pp. 354-361.

[16] M. M. Lee, J. Kim, D. Abts, M. Marty, and J. W. Lee, "Probabilistic distance-based arbitration: Providing equality of service for many-core CMPs," in *International Symposium on Microarchitecture (MICRO)*, 2010, pp. 509-519.

[17] P. Wang, S. Ma, H. Lu, Z. Wang, and C. Li, "Adaptive remaining hop count flow control: Consider the interaction between packets," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2015, pp. 54-60.

[18] P. Gupta and N. McKeown, "Designing and implementing a fast crossbar scheduler," *IEEE Micro,* vol. 19, no. 1, pp. 20-28, 1999.

[19] G. Dimitrakopoulos and E. Kalligeros, "Dynamic-priority arbiter and multiplexer soft macros for on-chip networks switches," in *Design, Automation & Test in Europe (DATE)*, 2012, pp. 542-545.

[20] H. F. Ugurdag and O. Baskirt, "Fast parallel prefix logic circuits for n2n round-robin arbitration," *Microelectronics Journal,* vol. 43, no. 8, pp. 573-581, 2012.

[21] Accellera Systems Initiative, *SystemC*, Available: http://www.accellera.org/.

[22] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl*, et al.*, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," in *International Solid-State Circuits Conference (ISSCC)*, 2010, pp. 108-109.

[23] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, *et al.*, "TILE64 processor: A 64-core SoC with mesh interconnect," in *International Solid-State Circuits Conference (ISSCC)*, 2008, pp. 88-598.

[24] D. Sanchez and C. Kozyrakis, "ZSim: fast and accurate microarchitectural simulation of thousand-core systems," in *International Symposium on Computer Architecture (ISCA)*, Tel-Aviv, Israel, 2013, pp. 475-486.

[25] D. I. Rich, "The evolution of SystemVerilog," *IEEE Design & Test of Computers,* vol. 20, no. 4, pp. 82-84, 2003.

[26] Synopsys, *DC Ultra*, Available: http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DCUltra/.

[27] Synopsys, *Synopsys 32/28nm Generic Library*, Available: http://www.synopsys.com/Community/UniversityProgram/Pages/32-28nm-generic-library.aspx.

[28] J. Kim, J. Balfour, and W. Dally, "Flattened butterfly topology for on-chip networks," in *International Symposium on Microarchitecture (MICRO)*, 2007, pp. 172-182.

[29] J. Balfour and W. J. Dally, "Design tradeoffs for tiled CMP on-chip networks," in *International Conference on Supercomputing (ICS)*, Cairns, Queensland, Australia, 2006, pp. 187-198.

[30] R. Das, S. Eachempati, A. K. Mishra, V. Narayanan, and C. R. Das, "Design and evaluation of a hierarchical on-chip interconnect for next-generation

CMPs," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2009, pp. 175-186.

[31] J. Y. Chen, W.-B. Jone, J.-S. Wang, H.-I. Lu, and T.-F. Chen, "Segmented bus design for low-power systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 7, no. 1, pp. 25-29, 1999.

[32] A. Avakian, J. Nafziger, A. Panda, and R. Vemuri, "A reconfigurable architecture for multicore systems," in *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010, pp. 1-8.

[33] A. Avakian, N. Agrawal, and R. Vemuri, "Reconfigurable multicore architecture for dynamic processor reallocation," in *International Symposium on Applied Reconfigurable Computing (ARC)*, 2012, pp. 329-334.

[34] B. Zhai, R. G. Dreslinski, D. Blaauw, T. Mudge, and D. Sylvester, "Energy efficient near-threshold chip multi-processing," in *International Symposium on Low Power Electronics and Design (ISLPED)*, Portland, OR, USA, 2007, pp. 32-37.

[35] D. Fick, R. G. Dreslinski, B. Giridhar, G. Kim, S. Seo, M. Fojtik*, et al.*, "Centip3De: A cluster-based NTC architecture with 64 ARM Cortex-M3 cores in 3D stacked 130 nm CMOS," *IEEE Journal of Solid-State Circuits,* vol. 48, no. 1, pp. 104-117, 2013.

[36] J. Laudon and D. Lenoski, "The SGI Origin: a ccNUMA highly scalable server," in *International Symposium on Computer Architecture (ISCA)*, Denver, Colorado, USA, 1997, pp. 241-251.

[37] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel computer architecture: A hardware/software approach*, Morgan Kaufmann Publishers, 1999.

[38] Y. H. Song and T. M. Pinkston, "A progressive approach to handling message-dependent deadlock in parallel computer systems," *IEEE Transactions on Parallel and Distributed Systems,* vol. 14, no. 3, pp. 259-275, 2003.

[39] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro,* vol. 30, no. 2, pp. 56-69, 2010.

[40] D. R. Butenhof, *Programming with POSIX Threads*, Addison-Wesley, 1997.

[41] ARM, *Cortex-A5 Processor*, Available: http://www.arm.com/products/processors/cortex-a/cortex-a5.php.

[42] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature,* vol. 521, no. 7553, pp. 436-444, 05/28/print 2015.

[43] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE,* vol. 86, no. 11, pp. 2278-2324, 1998.

[44] *Information Technology - Digital Compression and Coding of Continuous-Tone Still Images - Requirements and Guidelines*, International Telecommunication Union T.81, 1992.

[45] Y. LeCun, C. Cortes, and C. J.C. Burges, *The MNIST Database of Handwritten Digits*, Available: http://yann.lecun.com/exdb/mnist/.

[46] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick*, et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *ACM*

international conference on Multimedia, Orlando, Florida, USA, 2014, pp. 675-678.

국문초록

명령어 수준 병렬성 이용의 한계와 동작 주파수 증가에 따른 동작 전력과 온도 증가로 인해 다중프로세서칩 시대의 시대가 시작 된지 오래이다. 하나의 강력한 프로세서는 여러 개의 덜 강력한 프로세서로 대체되었다. 그러한 프로세서 간의 통신을 관리하는 방법에 관한 연구는 다중프로세서칩으로의 패러다임 전환으로 인해 야기된 주제 중 하나이다. 기존의 시스템에서 통신 구조 선택시의 첫 고려 사항은 버스였다. 그러나 버스는, 그 적은 대역폭으로 인해 여러 프로세서 간의 통신을 유지하는 것에 실패하였다. 계층적 버스나 버스 매트릭스 등, 버스를 기반으로 한 기술들이 등장하였으나 결국에는 네트워크-온-칩 (NoC)이 버스를 대체하고 다중프로세서칩을 위한 새로운 표준이 되었다.

버스에 대한 NoC 의 강점은 주로, 여러 트랜잭션들을 동시에 지원할 수 있는 것에 기원한다. 그러한 동시적인 트랜잭션들은 네트워크에 분포되어 공유되는 라우터에 의해 수행된다. 네트워크 내의 라우터들은 코어들의 요청을 공평하게 수행하여 각 코어가 같은 양의 서비스 (예를 들어 대역폭)를 받을 수 있도록 해야 한다. 아비터는 라우터 내에 위치하여, 공유되는 자원인 채널이나 버퍼들이 공평하게 분배되도록 하는 구성요소이다. 아비터가 자주

이용하는 분배 원칙은 라운드-로빈이다. 라운드-로빈 방법에서 각 트랜잭션 요청은 한 번씩 순서대로 돌아가면서 공유 자원을 이용한다. 그런데 이러한 라운드-로빈 방법이 NoC 에 적용되었을 경우, 불공평한 서비스 분배가 일어날 수 있다. 그 이유는 하나의 트랜잭션이 여러 개의 라우터를 거치면서 라운드-로빈 법칙을 여러 번 적용 받게 되어, 가까이 있는 프로세서 간의 통신이 멀리 떨어진 프로세서 간의 통신보다 많은 양의 서비스를 받게 되기 때문이다. 이 논문의 첫 part 는 이 문제를 다룬다. NoC 서비스의 균등 분배를 위해 weighted 라운드-로빈 방법이 이용되었으며, 제안된 방법이 지금까지의 연구들과 비교했을 때 가장 간단하면서도 효과적인 방법임을 보인다. 라운드-로빈의 weight 들은 다중프로세서칩 상의 어플리케이션 배치에 따라 동적으로 조절된다. 라우터의 RTL 을 구현하여 그 간단함을 보이고, synthetic traffic 과 SPEC CPU2006 벤치마크를 이용한 시뮬레이션 결과로 그 효력을 보인다.

이 논문의 두 번째 part 는 재구성 가능한 통신 구조를 통해 시스템 성능의 향상을 꾀한다. NoC 의 결함 중 하나는, 코어 수 증가에 따른 트랜잭션 발신자와 수신자 사이의 거리 증가이다. 특히 다중프로세서칩 상에서 가장 많이 쓰이는 그물망 형태의 NoC 는 코어 수 증가에 따른 확장성이 결여되어 있는 구조이다. 그물망 형태를 대체하기 위한 구조로서 high-radix NoC 와 클러스터링 방법이 있다. High-radix NoC 에서는 멀리 떨어진 코어 간을 잇는 채널을 설치한다. 그 결과, 물리적으로 멀리 떨어진 코어 간의 네트워크 홉

수가 줄어들게 된다. 클러스터링 방법에서는 코어들을 먼저 지역통신구조로 묶어 클러스터를 만든 후에 그 클러스터들을 광역통신구조로 다시 묶는다. 이 방법을 이용하면 광역통신구조의 크기가 줄어들어 코어 간의 빠른 통신이 가능해진다. 지역통신구조를 위한 통신 구조로, 그물망형, 링형 등도 고려되었으나, 이들 지역 NoC 대신, 기존에 적은 수의 코어들을 위한 통신 구조로서 강점을 보인 버스가 가장 많이 이용된다. 버스는 그 간단함을 기반으로 적은 칩 면적과 전력을 소비하면서 빠른 지역 통신을 가능하게 한다. 이 때 대두되는 주제 중 하나는 얼마나 많은 코어들을 버스로 묶을 것인가 하는 것이다. 너무 많은 코어들을 묶을 경우, 버스가 붐비게 되어 효율적인 지역 통신이 불가능해진다. 너무 적은 코어들을 묶을 경우, 버스의 강점을 제대로 이용하지 못하게 된다. 이 상황에서의 자연스러운 선택은 버스를 재구성 가능하게 하는 것이다. 즉, 여러 개의 버스 조각들을 스위치로 연결하거나 분리시킴으로써 클러스터의 크기를 동적으로 조절할 수 있게 하는 것이다. 이러한 클러스터의 동적 크기 변경 외에도, 버스 재구성의 강점은 존재한다. 그 중 하나는 버스 재구성을 통한 코어와 캐시의 연결 상태 변경과 이를 통한 코어와 데이터 간의 거리 단축이다. 이를 이용하면, 코어 간 데이터 통신이 필요할 때 데이터 복사와 네트워크 트랜잭션을 동시에 없앰으로써 시스템 성능을 향상시킬 수 있다. 이 논문의 두 번째 part 에서는 이를 응용하여 파이프라인된 어플리케이션의 성능 향상을 도모한다. 제안된 구조는 버스

재구성과 L1 캐시를 조직적으로 활용하여 이웃하는 파이프라인 단계 간의 데이터 전달을 효율적으로 수행한다. 이 논문에서 제안된 구조는, 기존의 캐시 일관성을 하드웨어적으로 유지하는 구조와 비교된다. 다층 퍼셉트론 (MLP), convolutional 신경망 (CNN), JPEG 복호 어플리케이션들을 멀티쓰레드 파이프라인 구조로 제작하여 실험에 이용하였다. 기존의 하드웨어 캐시 일관성 구조와 제안된 구조의 전체 시스템 시뮬레이션 모델들을 구현하여 실험한 결과 MLP, CNN, JPEG 복호 어플리케이션들에 대해 21.75 %, 14.40 %, 12.74 % 의 성능 향상을 얻었다.

주 요 어: Weighted 라운드-로빈, 균등성, 계층적 네크워크-온-칩, 버스 재구성, 파이프라인된 어플리케이션

학　　번: 2009-30190