I, Annie Avakian, hereby submit this original work as part of the requirements for the degree of Doctor of Philosophy in Computer Science & Engineering.

It is entitled:
Reducing Cache Access Time in Multicore Architectures Using Hardware and Software Techniques

Student's name: Annie Avakian

This work and its defense approved by:

Committee chair: Ranganadha Vemuri, PhD
Committee member: Karen Davis, PhD
Committee member: Wen Ben Jone, PhD
Committee member: Carla Purdy, PhD
Committee member: Karen Tomko, PhD
Reducing Cache Access Time in Multicore Architectures Using Hardware and Software Techniques

A dissertation submitted to the

Division of Research and Advanced Students
of the University of Cincinnati

in partial fulfillment of the requirements
for the degree of

Doctorate of Philosophy

in the School of Electronics and Computing Systems
of the College of Engineering and Applied Sciences
University of Cincinnati
March 2012

by

Annie Avakian

B.E. Computer Engineering
Lebanese American University, Byblos, Lebanon
February 2005

Dissertation Advisor and Committee Chair: Ranga Vemuri
Abstract

One of the challenges of multicore design is providing data quickly to all the processor cores running on a system. This has led to the development of architectures based on Network on Chips (NoC) due to their flexibility and scalability. In the NoC architecture, data is distributed among cache banks connected to different routers, therefore data access time varies by location. If data is accessed by two or more cores at the same time, then placing that data in the vicinity of those cores significantly improves overall access time.

Recent proposals of hybrid and reconfigurable interconnect architectures try to take advantage of data locality to a certain extent by grouping processors that work on the same data set. In this dissertation, we propose migrating processor cores instead of data lines to take advantage of data locality. This is done either at static time where cores are assigned to routers before the runtime of the process with the introduction of the Reconfigurable Architecture for Multicore Systems (RAMS). In the static architecture, previous knowledge of the process characteristics is essential in determining a good configuration. We extend the idea further with the introduction of a dynamic reconfiguration entitled Dynamically Reconfigurable Multicore Architecture (DyaReMA) where cores are reassigned on the fly based on the pattern of data requests.

A good test platform is essential to verify the validity of proposed architectures. We developed a complete modular hardware model that can be used to synthesize and simulate a range of architectures.

The architectures proposed in literature have a homogeneous distribution of caches to routers. We propose a heterogeneous configuration of cache blocks to routers and show that the performance is significantly improved. We take this idea one step further and introduce the reconfigurable cache architecture that reassigns cache blocks to neighboring routers based on data access
Hardware implementation alone cannot extract the full potential of multicore architectures. Efficient data migration and cache coherency protocols are needed that further improve cache access time. Since data access is determined at run time, data migration has become an important part of multicore design. However, traditional data migration is expensive both due to the high cost in keeping track of all data accesses and the actual migration of data lines across multiple routers. In this dissertation, we propose a stepwise data migration scheme that improves efficiency reducing hardware cost.

Data sharing introduces the need for cache coherency protocols. We propose a hybrid cache coherency protocol that combines the advantages of both broadcast and directory protocol to implement an efficient and scalable coherency method. Two coherency protocols are proposed, the first takes advantage of the characteristics of the hybrid/bus architecture. The second is targeted for NoC architectures.

The proposed architectures along with data migration and cache coherency can improve data access time for L2 cache misses and help to significantly reduce the runtime of different processes in the realm of multicore architecture.
To,

Mom and Dad
Acknowledgements

First and foremost I would like to thank Dr. Vemuri for his leadership and guidance. You are a great mentor and I grew both as a researcher and a person under your supervision.

I would like to thank Dr. Iyad Ouaiss for teaching me to love research. Thank you for your support and your efforts in helping me pursue a PhD career.

Thank you Dr. Karen Davis, Dr. Wen-Ben Jone, Dr. Carla Purdy and Dr. Karen Tomko for agreeing to be on my committee and your valuable input.

Thank you to my master’s students Jon, Aishwariya, Natwar for your contribution to this work. Without your help, I could not have explored the breadth of areas as I have done. It was a pleasure working with all of you. Karthik, I wish you success and I will continue to collaborate with you until you meet your goals.

Thank you Shubhankar, Almitra, Angan and Hao. You welcomed me in the DDEL-ite family with open arms and helped me take my first steps as a PhD student. Special thanks to Mike and Jon. I looked forward to our coffee hours every day. You made the lab fun. And thank you to all the DDEL-ites. You were my home away from home.

Lana, Tim, Katie, Joe, Ryan, Nishan, thank you for making work enjoyable.

Last but not least, thank you mom, dad and Taline for your continuous love and support. I would not have reached this far without your encouragement. I know it has been a long journey, but we made it here together. I love you.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of Contents</td>
<td>vii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>x</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xiii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 The Evolution of the Processor</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Challenges of Multicore Architecture</td>
<td>2</td>
</tr>
<tr>
<td>1.2.1 Communication Bottleneck</td>
<td>3</td>
</tr>
<tr>
<td>1.2.2 Cache Coherency</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Research Approach and Overview</td>
<td>6</td>
</tr>
<tr>
<td>1.3.1 Hardware Techniques</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Test Platform Generation</td>
<td>10</td>
</tr>
<tr>
<td>1.4.1 Hardware Simulation Platform</td>
<td>10</td>
</tr>
<tr>
<td>1.5 Heterogeneous Cache Architecture</td>
<td>10</td>
</tr>
<tr>
<td>1.5.1 Software Techniques</td>
<td>11</td>
</tr>
<tr>
<td>1.6 Dissertation Outline</td>
<td>12</td>
</tr>
<tr>
<td>2 Reconfigurable Architecture for Multicore Systems: RAMS</td>
<td>13</td>
</tr>
<tr>
<td>2.1 Motivation</td>
<td>13</td>
</tr>
<tr>
<td>2.1.1 Exploiting Data Access Characteristics</td>
<td>14</td>
</tr>
<tr>
<td>2.1.2 Determining Number of Cores per Bus</td>
<td>17</td>
</tr>
<tr>
<td>2.1.3 Adaptability of Architecture to the Next Generation</td>
<td>19</td>
</tr>
<tr>
<td>2.1.4 Reliability</td>
<td>19</td>
</tr>
<tr>
<td>2.2 Proposed Architecture - RAMS</td>
<td>19</td>
</tr>
<tr>
<td>2.2.1 Architecture Features</td>
<td>19</td>
</tr>
<tr>
<td>2.2.2 Configuration</td>
<td>21</td>
</tr>
<tr>
<td>2.3 Experimental Results</td>
<td>27</td>
</tr>
<tr>
<td>2.3.1 Benefits of the Hybrid Model</td>
<td>27</td>
</tr>
<tr>
<td>2.3.2 Benefits of Reconfigurability</td>
<td>29</td>
</tr>
<tr>
<td>2.4 Conclusion and Future Work</td>
<td>35</td>
</tr>
</tbody>
</table>
### 3 Reconfigurable Multicore Architecture for Dynamic Processor Reallocation  
3.1 Introduction ................................................. 36  
3.2 Proposed Architecture  
  3.2.1 DyaReMA ............................................. 39  
  3.2.2 DyaReMA with Reduced Number of Meshes .......... 46  
  3.2.3 Segmented DyaReMA ................................. 46  
3.3 Experimental Results and Analysis  
  3.3.1 DyaReMA Simulation Results ......................... 50  
  3.3.2 Synthesis Models ................................... 52  
3.4 Conclusion .................................................. 55

### 4 Hardware Simulation Platform  
4.1 Introduction and Motivation .............................. 57  
4.2 Possible Architectures  
  4.2.1 Bus Based Architecture ............................. 58  
  4.2.2 NoC .................................................. 62  
4.3 Hybrid Bus/NoC Architecture ............................ 64  
4.4 Reconfigurable Hybrid Architecture .................... 66  
4.5 Evaluation of the Different Architectures ............. 67

### 5 A Heterogeneous Cache Distribution with Reconfigurable Interconnect  
5.1 Introduction and Motivation .............................. 70  
5.2 Static Distribution  
  5.2.1 Heterogeneous Cache Distribution ................... 72  
  5.2.2 Proposed Architecture ............................. 73  
  5.2.3 Experimental Results ............................... 74  
5.3 Dynamic Distribution  
  5.3.1 Proposed Architecture ............................. 74  
  5.3.2 Cache Reassignment Policy ......................... 75  
  5.3.3 Experimental Results ............................... 77  
5.4 Conclusion .................................................. 79

### 6 Alternate Methods to Reduce Cache Access Time  
6.1 Data Migration ............................................ 80  
  6.1.1 Introduction ........................................ 80  
  6.1.2 Motivation .......................................... 82  
  6.1.3 Data Migration ..................................... 83  
  6.1.4 Experimental Setup .................................. 88  
  6.1.5 Results ............................................. 90  
  6.1.6 Conclusion and Future Work ......................... 93  
6.2 Address Lookup Methodologies ......................... 93  
  6.2.1 Introduction and Motivation ......................... 93  
  6.2.2 Proposed Distributed Address Lookup Methodology .... 95  
  6.2.3 Experimental Results ............................. 97  
  6.2.4 Conclusion .......................................... 98
## Contents

7 A Hybrid Cache Coherency Protocol for Hybrid and NoC Multicore Architectures 102
   7.1 Introduction and Motivation .............................................. 103
   7.2 Hybrid Cache Coherency Protocol for Hybrid Architectures .......... 105
      7.2.1 Hybrid Protocol .................................................. 106
      7.2.2 Experimental Results ............................................ 109
   7.3 Hybrid Cache Coherency on Network on Chip .......................... 113
      7.3.1 Hierarchical Directory Broadcast Protocol ....................... 114
      7.3.2 Experimental Results ............................................ 117
   7.4 Conclusion .............................................................. 121

8 Conclusion 123
   8.1 Hardware Methods ...................................................... 123
      8.1.1 Reconfigurable Architecture for Multicore Systems: RAMS ......... 123
      8.1.2 A Dynamically Reconfigurable Multicore Architecture: DyaReMA .... 124
      8.1.3 Test Platform Generation ......................................... 125
      8.1.4 Heterogeneous Cache Architecture ................................ 125
   8.2 Software Methods ........................................................ 125
      8.2.1 Data Migration and Address Lookup ................................ 126
      8.2.2 Cache Coherency ................................................... 126
   8.3 Future Work .............................................................. 127

Bibliography 128
List of Figures

1.1 NoC Architecture .................................................. 5
1.2 Hardware and Software Techniques ................................. 7
1.3 Test Platform ..................................................... 9

2.1 NoC Architecture .................................................. 15
2.2 Hybrid Architecture ............................................... 16
2.3 \{Number of Routers, Number of cores on each bus\} \{4x4,4\} .............................................. 17
2.4 \{Number of Routers, Number of cores on each bus\} \{5x5,4\} .............................................. 18
2.5 Average wait time based on number of cores per bus and memory intensity .................................. 18
2.6 Hybrid Reconfigurable Architecture ............................... 20
2.7 Cores that cannot be Assigned ................................... 22
2.8 Cores that cannot be Assigned ................................... 23
2.9 Before and After New Assignment ................................. 25
2.10 Example of Mapped Cores ....................................... 27
2.11 Comparison of 3x3 and 9x9 NoCs ................................ 28
2.12 Comparison of 4x4 and 11x11 NoCs .............................. 28
2.13 Comparison of 6x6 and 16x16 NoCs .............................. 29
2.14 Average Number of Hops to Access Data on each L2 Slice ........................................ 30
2.15 Average Number of Hops to Access Data on each L2 Slice ........................................ 30
2.16 Average Number of Hops to Access Data on each L2 Slice ........................................ 31
2.17 Average Number of Network Hops for 64 Cores ................. 31
2.18 Average Number of Hops to Access Data on each L2 Slice ........................................ 32
2.19 Average Number of Hops to Access Data on each L2Slice ........................................... 32
2.20 Average Number of Hops to Access Data on each L2 Slice ........................................ 33
2.21 Average Number of Network Hops for 128 Cores ............... 33

3.1 Proposed Architecture .............................................. 40
3.2 Router Connectivity ............................................... 42
3.3 Distance Calculation ............................................... 43
3.4 DyaReMA with Reduced Meshes .................................. 47
3.5 Bus Interface ..................................................... 47
3.6 4x4 NoC Switchbox Architecture .................................. 48
3.7 Implementation of Switchbox .................................... 49
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.8</td>
<td>Results for 64 Core</td>
<td>52</td>
</tr>
<tr>
<td>3.9</td>
<td>Results for 128 Core</td>
<td>53</td>
</tr>
<tr>
<td>4.1</td>
<td>Bus Architecture</td>
<td>60</td>
</tr>
<tr>
<td>4.2</td>
<td>Processor Design</td>
<td>61</td>
</tr>
<tr>
<td>4.3</td>
<td>NoC Architecture</td>
<td>62</td>
</tr>
<tr>
<td>4.4</td>
<td>Router Design</td>
<td>63</td>
</tr>
<tr>
<td>4.5</td>
<td>Hybrid Architecture</td>
<td>65</td>
</tr>
<tr>
<td>4.6</td>
<td>Area Comparison</td>
<td>69</td>
</tr>
<tr>
<td>5.1</td>
<td>Average Latency for Heterogeneous and Homogeneous Cache Distribution</td>
<td>72</td>
</tr>
<tr>
<td>5.2</td>
<td>Static Heterogeneous Cache architecture for 6x6 Router Network</td>
<td>73</td>
</tr>
<tr>
<td>5.3</td>
<td>Average Latencies of Homogeneous vs Heterogeneous Configuration 6x6 Network</td>
<td>75</td>
</tr>
<tr>
<td>5.4</td>
<td>Average Latencies of Homogeneous vs Heterogeneous Configuration 16 Cores [64]</td>
<td>76</td>
</tr>
<tr>
<td>5.5</td>
<td>Average Latencies of Homogeneous vs Best Heterogeneous Configuration 6x6 Network</td>
<td>78</td>
</tr>
<tr>
<td>5.6</td>
<td>Average Latencies of Homogeneous vs Best Heterogeneous Configuration 16 Cores</td>
<td>78</td>
</tr>
<tr>
<td>5.7</td>
<td>Average Latencies of Homogeneous vs Best Heterogeneous Configuration 128 Cores</td>
<td>79</td>
</tr>
<tr>
<td>6.1</td>
<td>Hybrid NoC Single Router Layout</td>
<td>83</td>
</tr>
<tr>
<td>6.2</td>
<td>North/South Neighbors</td>
<td>84</td>
</tr>
<tr>
<td>6.3</td>
<td>East/West Neighbors</td>
<td>84</td>
</tr>
<tr>
<td>6.4</td>
<td>Access Counter Distances</td>
<td>85</td>
</tr>
<tr>
<td>6.5</td>
<td>Directional Migration Cache Structure</td>
<td>88</td>
</tr>
<tr>
<td>6.6</td>
<td>Percent Difference in Cycles with Variable Threshold Values</td>
<td>91</td>
</tr>
<tr>
<td>6.7</td>
<td>Percent Difference in Cycles to Retrieve Data</td>
<td>91</td>
</tr>
<tr>
<td>6.8</td>
<td>Percent Difference in Distance to Retrieve Data</td>
<td>92</td>
</tr>
<tr>
<td>6.9</td>
<td>Percent Difference in Local L2 Accesses</td>
<td>92</td>
</tr>
<tr>
<td>6.10</td>
<td>Global Lookup Table</td>
<td>96</td>
</tr>
<tr>
<td>6.11</td>
<td>Distributed Lookup Tables</td>
<td>96</td>
</tr>
<tr>
<td>6.12</td>
<td>Size of Lookup Table - Threshold 8</td>
<td>98</td>
</tr>
<tr>
<td>6.13</td>
<td>Average Cache Access Time Gain - Threshold 8</td>
<td>99</td>
</tr>
<tr>
<td>6.14</td>
<td>Size of Lookup Table - Threshold 64</td>
<td>99</td>
</tr>
<tr>
<td>6.15</td>
<td>Average Cache Access Time Gain - Threshold 64</td>
<td>100</td>
</tr>
<tr>
<td>6.16</td>
<td>Size of Lookup Table - Threshold 256</td>
<td>100</td>
</tr>
<tr>
<td>6.17</td>
<td>Average Cache Access Time Gain - Threshold 256</td>
<td>101</td>
</tr>
<tr>
<td>7.1</td>
<td>Hybrid Bus/Network Architecture</td>
<td>107</td>
</tr>
<tr>
<td>7.2</td>
<td>Directory vs Hybrid Protocol Area Overhead</td>
<td>108</td>
</tr>
<tr>
<td>7.3</td>
<td>Hybrid Architecture with Hybrid Cache Coherency Protocol</td>
<td>108</td>
</tr>
<tr>
<td>7.4</td>
<td>Increase in Cache Access Time for Synthetic Benchmarks</td>
<td>112</td>
</tr>
<tr>
<td>7.5</td>
<td>Increase in Cache Access Time for PARSEC Benchmarks</td>
<td>113</td>
</tr>
<tr>
<td>7.6</td>
<td>Hybrid Cache Coherency Protocol for NoCs</td>
<td>115</td>
</tr>
<tr>
<td>7.7</td>
<td>Directory vs Hybrid Protocol Area Overhead for NoCs</td>
<td>116</td>
</tr>
<tr>
<td>7.8</td>
<td>Read/Write Procedure for Hybrid Cache Protocol</td>
<td>116</td>
</tr>
</tbody>
</table>
7.9  Increase in Cache Access Time for Synthetic Benchmarks - NoC . . . . . . . . . . 120
7.10 Increase in Cache Access Time for PARSEC Benchmarks - NoC . . . . . . . . . . 121
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Priority of Cores</td>
<td>26</td>
</tr>
<tr>
<td>2.2</td>
<td>Experimental setup</td>
<td>34</td>
</tr>
<tr>
<td>3.1</td>
<td>Example of Counters</td>
<td>44</td>
</tr>
<tr>
<td>3.2</td>
<td>Experimental setup</td>
<td>51</td>
</tr>
<tr>
<td>3.3</td>
<td>Implemented Components</td>
<td>54</td>
</tr>
<tr>
<td>3.4</td>
<td>Synthesis Setup</td>
<td>54</td>
</tr>
<tr>
<td>3.5</td>
<td>Synthesis Results</td>
<td>55</td>
</tr>
<tr>
<td>6.1</td>
<td>Simics Simulator Setup</td>
<td>89</td>
</tr>
<tr>
<td>6.2</td>
<td>NoC Simulator Setup</td>
<td>89</td>
</tr>
<tr>
<td>6.3</td>
<td>Average Migration Percent Difference to Best Placement</td>
<td>90</td>
</tr>
<tr>
<td>7.1</td>
<td>Area Overhead of Directory and Hybrid Protocol for Hybrid Architectures</td>
<td>108</td>
</tr>
<tr>
<td>7.2</td>
<td>Experimental Setup</td>
<td>110</td>
</tr>
<tr>
<td>7.3</td>
<td>Characteristics of the Synthetic Benchmarks</td>
<td>111</td>
</tr>
<tr>
<td>7.4</td>
<td>Area Overhead of Directory and Hybrid Protocol for NoC</td>
<td>114</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

According to David House’s extension of Moore’s Law [6], demand on computing power doubles approximately every 20 months. As the industry continuously reacts to meet this growing demand, new ideas must be explored. Traditionally this has been met by increasing the clock speed of individual processors. Unfortunately, the diminishing returns in increasing processing speed after passing the 3 GHz plateau have forced exploration of other alternatives. Over the years, processor design has evolved to keep up with processing demand and different design alternatives have come forth, some showing more promise than others. This has lead to a new trend in increasing the total number of processor cores in a single chip [36] [18]. In September of 2006, Intel announced the development of a single processor design containing 80 processor cores [7] [33]. While simply increasing the total number of processor cores on a chip is straight-forward, connecting them together to provide a harmonious system is proving to be among the most critical design tasks.

1.1 The Evolution of the Processor

Pipelined architectures were adopted early on by most mainstream processor manufacturers. At that time, the effort of architects was targeted towards increasing clock speed by designing
Chapter 1: Introduction

faster and faster components. When more fundamental changes were called for, architects from Intel responded with hyper-threading technology [53]. In hyper-threaded processors, there exists one physical core, but some components within the core are duplicated. The operating system sees the processor as two logical components and thus the two threads can run on the same core at the same time. When one thread is not using certain components, the second thread can take advantage of those resources and use them for its needs.

As software becomes more and more complex, the need for more processing power becomes evident. After the success of hyper-threading, the industry started moving towards multicore architectures [60]. In multicore architectures, more than one physical processing unit is designed on a single chip. The first multicore architectures had two cores, but the number of cores found on a single die is increasing exponentially and expected to reach up to a thousand cores in the next decade [4]. The different cores that exist on a single chip can either be homogeneous, where all the cores perform the same function [60], or heterogeneous, where each processing core has a unique feature and can be used for a specific application [44].

As the number of processing cores increases, the challenge becomes not increasing the speed of the cores, but rather speeding up the communication between the cache and the different cores [20].

1.2 Challenges of Multicore Architecture

Multicore design is not without its challenges. Data is shared among the different cores found on the system that leads to communication bottleneck and cache coherency issues.
1.2.1 Communication Bottleneck

Communication bottleneck has been a continuous challenge in processor design and the introduction of multicore architectures has further exacerbated the problem. In a single core design, the core accesses the data found in the cache using a dedicated connection. In multicore systems, each core has a private cache, but the last level cache is shared among all the cores. In dual-core or quad-core designs, the cores are usually connected to a bus and the bus in turn is connected to the cache. Each processor core maintains a unique L1 or even L2 cache while the last level cache is shared among all the processing cores. The cores then need to be granted bus access to be able to send data requests to the last level cache. This scheme works well when the number of cores is small, but as the number of cores increases, the bus quickly reaches saturation point. Therefore this is not a viable method for developing a system with more than a handful of processor cores.

Different approaches have been tried in order to reduce communication latency, and so far Network-on-Chip (NoC) has come out the winner [16]. In the NoC scheme, processor cores are connected to routers and the communication between cores is done using protocols similar to a computer network. While processor cores have a private L1 cache, the L2 cache is distributed and shared among all the cores. The NoC architecture is shown in Figure 2.1

Concentrating on the hardware aspects alone cannot yield a good architecture. As we move toward parallel computation, each process is expected to be parallelized and composed of several threads working on the same data set. If the L2 cache is distributed among routers [70], then the latency to obtain the data varies depending on the network hops required in the NoC architecture. Physical proximity plays an important role in this scheme. The closer the L2 cache, the fewer hops is needed to obtain the requested data.

To reduce communication latency in an NoC architecture, several proposals have been introduced. One such proposal is a global sub-mesh interconnect [19]. The authors propose using hierarchical rings for global routing and local rings for local routing. Using this method, Bourd-
uas and Zilic showed they can achieve significant gains in number of hops for large mesh sizes. Another approach to reduce interconnect latency was proposed by Bolotin et al. [17]. The authors proposed reducing cache access time by introducing priority support at the hardware level. They also proposed using virtual invalidation ring, or an efficient way to store and forward short messages. Knauerhase et al. [43] offer a different approach. They show the operating system can take advantage of the underlying architecture better if it learns from the information garnered from runtime task behavior. While these methods reduce cache access time, they do not take advantage of the characteristics of parallel applications. On the other hand, Das et al. [28] suggest having eight processor cores on a bus connected to a router as a hybrid bus-mesh architecture. While this approach does take advantage of the inherent properties of multithreaded applications, the architecture is not flexible.

1.2.2 Cache Coherency

Having a distributed shared cache level presents a new problem; Cache Coherency. When multiple cores are working on the same data, then the different cores can have the same copy of a data line. When cores are performing reads, this causes no issues. But during write operations, multiple cores can update the same data line. Special care needs to be taken to ensure that all the cores have the correct copy of the data.

In a bus architecture, this problem is solved by implementing the bus snooping policy. In the bus snooping scheme, the cores are all connected on a single bus, therefore the different cores monitor the traffic on the bus and check for data writes. For every data write, they check to see if they have a copy of that data. If they do, they invalidate their copy of the data.

In the NoC architecture, the solution is not as simple. The routers have access only to the packets that pass through them. Furthermore, in order to check for data writes, the routers need to depacketize each packet passing through. There are different protocols that are commonly used to
Figure 1.1: NoC Architecture
solve this problem. The first is the broadcast protocol where if a data line is shared, a broadcast message is sent out on the network so that each core can check to see if it should invalidate its data line. Although this method is simple, it can quickly saturate the network. The second method is the directory protocol where each data line keeps track of which cores have a copy of that data line. When the data is updated, it sends a message only to those cores to invalidate their copy. The issue with this method is the large overhead needed to keep track of all the data shares.

An efficient cache coherency method is essential to reduce overall network traffic without a large overhead, otherwise the control messages can themselves cause a bottleneck and slow down the entire network.

1.3 Research Approach and Overview

In order to meet the challenges of multicore architectures, we propose several hardware and software techniques to improve performance. Figure 1.2 shows the different approaches tackled to meet those challenges.

1.3.1 Hardware Techniques

The simplest way to design multicore architectures is to connect processing units to a bus which is then connected to the L2 cache. Although this is simple and works for small number of processing units, as the number of processing cores increases, so does the wait time to obtain a bus grant. Therefore, architecture designers have looked for other ways to improve data access time. In recent years, Network on Chip (NoC) has garnered popularity among designers and is seen as a way to improve performance time. Scalability is important while designing multicore architectures since the number of cores on a chip is expected to increase exponentially in the coming years reaching hundreds of processing cores in one chip. In order to account for that, hybrid architectures have
Figure 1.2: Hardware and Software Techniques
been proposed in recent years [28]. In hybrid architectures, more than one core is connected to a bus that is then connected to a router. We propose having a reconfigurable architecture, where the number of cores connected to one bus can be changed either at static or dynamic time.

**Reconfigurable Architecture for Multicore Systems: RAMS**

Our studies indicate that while a hybrid architecture is preferable, the optimal number of processors on each bus varies based on the application. This number appears to vary between 1 and 8 depending on the communication requirements of the application. Furthermore, various applications simultaneously executing on the same system require differing number of processors on each bus-based subsystem to minimize the overall throughput time.

We present a new reconfigurable NoC architecture which allows scalable bus-based multiprocessor subsystem on each node in the NoC. Following configuration, the system provides a multi-bus execution environment where each processor is connected to a bus and the bus-based subsystems communicate via routers connected in a mesh-style configuration. The system can be reconfigured to vary the number of bus subsystems and the number of processors on each subsystem. Each processor contains a Level 1 (L1) cache and each bus, connected to a router, has access to a Level 2 (L2) cache. The L2 cache is distributed across the network and together forms a large virtual L2 that can be shared by all the processors in the system via the router network.

We present the architecture in detail, discuss a configuration algorithm, and show the experimental results (using the NS2 and SIMICS simulators) on standard and synthetic benchmarks indicating the performance advantages of the proposed architecture.

**A Dynamically Reconfigurable Multicore Architecture: DyaReMA**

Although RAMS is effective in reducing cache access time by grouping processing cores that work on the same data onto the same bus, the design is static. Cores can only be assigned
Figure 1.3: Test Platform

prior to the execution of the application. Therefore, prior knowledge is needed to determine the optimal location of processing units. Furthermore, once a core is assigned, it is not changed during the lifetime of the application. If the characteristics of data accesses change for a processing unit, the architecture does not allow for a change in core assignments to cater to the changing needs of the processing unit. We propose dynamically migrating processors to take advantage of data locality. This is realized by implementing a reconfigurable interconnect that allows reassignment of processor cores to different routers at runtime. We present the proposed architecture in detail, show a segmented hardware implementation of the proposed architecture, and discuss experimental results using PARSEC benchmarks showing the performance gains of the proposed architecture. Our results show a gain in average L2 access time of up to 24% when implementing the proposed architecture compared to a hybrid architecture without reconfiguration. Finally, we present area and performance data based on a detailed Verilog model and synthesis of the proposed architecture to show the cost of the proposed architecture.
1.4 Test Platform Generation

When testing new architecture designs, the simulation platform that is used to test the new architecture is vital. Figure 1.3 shows the components needed to build a comprehensive test platform.

1.4.1 Hardware Simulation Platform

The proposed architectures as mentioned above are simulated using Simics. However, Simics is not capable of providing data related to power consumption, heat dissipation and hardware faults. We developed a hardware platform where the hardware components are modular and can be combined to setup and simulate different kinds of architectures. The following architectures are some of the ones that can be simulated using the Hardware Simulation Platform:

- Bus Based Architecture
- NoC Architecture
- Hybrid Architecture
- Segmented DyaReMA

1.5 Heterogeneous Cache Architecture

In the standard NoC, the last level cache is distributed equally among all the cores. We propose having a heterogeneous distribution, where the threads that are data intensive are assigned to routers with larger L2 caches while threads which are not data intensive are assigned to routers with less memory. Two methods are proposed, static and reconfigurable.
1.5.1 Software Techniques

Data Migration

In the NoC architecture, data is distributed among cache banks connected to different routers, therefore data access time varies by location known as Non-Uniform Cache Access (NUCA). If data is accessed by two or more cores at the same time, then placing that data in the vicinity of those cores significantly improves overall access time. Since data access is determined at run time, data migration has become an important part of multicore design.

Data migration tries to take advantage of temporal locality. It keeps track of data accesses by processing units and tries to move the data closer to the processing cores accessing that data line. Data migration can significantly improve data access time since all future accesses will be quicker. However, the overhead associated with it is substantial. We propose a Directional Migration that does stepwise data migration with little overhead and can still come close to the optimal values in cache access time that Data Migration can achieve.

Aside from temporal locality, we propose a solution that takes into account spacial locality. If a particular data line is being accessed by certain processors, it is reasonable to assume that they will access subsequent data lines as well due to spacial locality. Therefore, if a data line is migrated, then migrating neighboring lines can improve data access time since those lines will be migrated themselves in the near future. Active Neighbor Migration further reduces the time it takes to access L2 cache.

Cache Coherency

Cache coherency is essential to ensure data correctness. The overhead associated with cache coherency in terms of either hardware overhead or network latency can negatively impact the performance of the system. We propose a new cache coherency protocol that uses hierarchical
structure to minimize network latency and reduce hardware overhead by as much as 87%.

1.6 Dissertation Outline

This dissertation is organized as follows: Chapter 2 introduces the static reconfigurable architecture (RAMS). Chapter 3 adds on to RAMS to create a dynamically reconfigurable architecture where cores can be reassigned to routers at runtime. Chapter 4 presents the hardware simulation platform that we developed to test different architectures. Chapter 5 introduces static and dynamic heterogeneous cache architectures. Chapter 6 proposes incremental data migration technique and address lookup methodology. Chapter 7 introduces a new cache coherency protocol. Chapter 8 concludes.
Chapter 2

Reconfigurable Architecture for Multicore Systems: RAMS

To meet the challenges described in the introduction chapter, we propose a reconfigurable hybrid bus-network architecture, where the number of processor cores attached to each bus is reconfigurable and is dependent on the needs of the active processes and applications. By configuring switches, multiple buses are created each with varying number of processor cores. The buses are then connected to the routers. The configuration of the switches remains the same during the lifetime of the running process.

2.1 Motivation

In this section, we will present experimental analysis of several simulated benchmark data sets to understand the benefits of bus-based vs. NoC architectures. We set four goals for the proposed architecture as follows:
2.1.1 Exploiting Data Access Characteristics

The first step in developing a multiprocessor architecture is to understand the data access patterns of typical processes and applications executing on the architecture. Exploiting the temporal and spatial localities is essential. Therefore, recognizing that threads of the same process will use the same data, grouping those threads becomes the first priority. Having a hybrid model of bus-network setup, where threads communicate locally via a bus and globally via a network can greatly reduce communication time between caches and processor cores. Buses are faster when data accesses are not intensive, and furthermore the area is much smaller when compared to assigning one router to each processor core. To show the benefits of a hybrid model, we simulated both the NoC architecture 2.1 and the hybrid architecture 2.2. Both simulations were done in C++. In the NoC architecture, each router has a processor core. The cores have private L1 caches, while the L2 cache is shared and is equally distributed among the routers.

In this scenario, a processor sends a request for memory access. The router acknowledges the request, and sends a packet to the router with the corresponding address. The memory request process time is measured as the number of hops the request travelled to reach the destination plus the number of hops for the return path. In our example, routers can process one request at a time, therefore if a router has pending requests from other processor cores, then the request is queued in the order of arrival. This delay is also added to the overall request process time. All links are considered uniform and have a unit time delay. The overall result of the simulation is the average processing time of all the memory requests generated by the processor cores.

In the hybrid bus-NoC model, $p$ number of processor cores are connected to a bus. The bus in turn is connected to the router. The L2 cache is still distributed shared L2. However, the size of the L2 cache in each router is $p$ times larger than the first case, since $p$ processor cores are connected to the bus. Whenever a core generates a memory request, the processor in this case needs to wait for a bus grant [56] in order to send the request. Once the router receives the request, it
Figure 2.1: NoC Architecture
processes it and sends it back on the bus. If the request needs to be sent to a different router, then it is packetized and sent on the network. The memory request process time is measured as the time it takes to get the bus grant, plus the time it takes the router to process the request.

Several benchmarks have been simulated on both architectures. The sets of data generated are produced by synthetic benchmarks. The numbers 5, 10, 20, 30, 40, 50 on the x-axis in Figures 2.3 and 2.4 represent the percentage of the time cores generate memory requests. We have a random process that generates a number between 0 and 1. If the number is less than the percent value, then the processor core generates a memory request. The destination of the request, meaning the router to whom the request will reach, is also generated randomly. The first bar is the result of the simple NoC model. The rest of the bars represent the hybrid model. The numbers 5, 10 and 20 represent the percentage of the time the processor cores request a memory address found on a remote router instead of the local router. The average wait time then is calculated from the time the request is generated, to the time the data reaches the source again. It includes wait times in the router queues, number of hops and process time. We considered a unit time for all delays.

In the tables and corresponding figures, it is evident that the hybrid model performs better than the NoC model. But as seen from Figure 2.5, we cannot add as many processor cores as we
please to the bus since the bus saturates faster than the network on chip. But for small number of processor cores, having a hybrid model reduces the overall delay. It is also interesting to see how the average wait time increases significantly when the memory requests exceed 20%. The reason for that sudden jump is the saturation of the system.

2.1.2 Determining Number of Cores per Bus

Now that we established the need for a hybrid bus-network model, the second step is determining how many cores need to be assigned to each bus. But then, why should we make the number static? If the number of processor cores on a bus is fixed, it simplifies the architecture. But making it variable makes the architecture more flexible. As seen in Figure 2.5 different memory access rates have different needs. Therefore we propose having a reconfigurable interconnect, where the number of cores per bus is variable and assigned by the operating system at run time. The proposed architecture will be introduced in Section 2.2.
Figure 2.4: {Number of Routers, Number of cores on each bus} \{5x5,4\}

Figure 2.5: Average wait time based on number of cores per bus and memory intensity
2.1.3 Adaptability of Architecture to the Next Generation

A third step is studying how well the architecture can adapt to new designs. If design engineers decide to put functional units alongside processor cores on a single chip or have asymmetric processor cores rather than symmetric ones [34], then the architecture should be able to use the functional units without major redesign. In a static architecture, it would be difficult to incorporate other functional units alongside processor cores without redesigning the entire architecture. But in the reconfigurable model, functional units can be connected to processor cores without much overhead. Therefore the need for reconfigurable interconnect becomes a necessity.

2.1.4 Reliability

An important final step is determining the reliability of the architecture. Having a large number of processor cores with reconfigurable hardware also means core failures can be tolerated gracefully. The architecture should be able to identify non-functional processor cores, and adapt to the existing architecture. Reconfigurability guarantees the continuation of operation without sacrificing performance.

Based on the four points mentioned in this section, we propose Reconfigurable Architecture for Multicore Systems (RAMS), the details of which are explained in the following section.

2.2 Proposed Architecture - RAMS

2.2.1 Architecture Features

From the section 2.1, the need for a reconfigurable hybrid architecture is clear. Based on the aforementioned observations, we propose the architecture shown in Figure 2.6. We will explain how this architecture covers the needs mentioned in the motivation section.

In the figure, processors are represented by circles, routers are rectangular, and the dia-
Figure 2.6: Hybrid Reconfigurable Architecture
monds are bus switches that have 2 states, on and off. Each processor core has a local L1 cache, and the L2 cache is shared and distributed equally among the routers. As seen, the routers are connected as a mesh. Communication between routers is packet switched. The processor cores are also connected as a mesh. But each segment of the bus is controlled by a switch. Based on the configuration of the switches, buses can be created dynamically. Once the buses are formed, they need to be connected to the routers. The connection point between routers and the buses are shown in the bubble. Each router has 4 access points to the bus. The alternative was to dedicate one processor core to each router as a starting point, but having four allows the bus to expand in any direction. The number of switches needed to implement this architecture is $2p(p - 1) + 4R$ where $p$ is the number of processors on each row and $R$ is the number of Routers in the system.

Therefore, the architecture provides a hybrid bus-network model. It also gives the flexibility of connecting as many processor cores on a bus as needed. If there are different functional units in the design, they can be connected to different processes on demand via reconfiguration. Finally, adapting to core failure is simple. The OS can disregard the dead processor core and remove it from the available cores list.

### 2.2.2 Configuration

The switches need to be configured so as to form a bus and connect the bus to a router to form a hybrid bus-NoC instance. In order to map cores to routers, several constraints need to be taken into consideration while configuring the system. Listed below are some of those constraints:

- No processor should be connected to more than one router at the same time.
- No core should be stuck isolated and unusable.
- All cores need to be shared equally between routers on a need basis. No router can use all the available cores, starving other cores in the process.
Any mapping of cores to routers should be possible.

A core can become unusable if the cores around it are assigned to routers and it does not have a free path to a different router. Since the cores that are on the edge of the mesh have a higher probability of becoming stuck, the algorithm should try to assign those cores first. But routers can only get to the boundary cores starting from their local neighbors. An example of this scenario is shown in Figure 2.8. If cores 2 and 5 are assigned, core 1 cannot be assigned to any other bus until either core 2 or core 5 is released.

In order to meet these constraints, the proposed algorithm assigns a priority to each core. The priority can be based on several factors. In the first case, the priority can be based on the distance of the core relative to each router. The ones close to the routers would have high priorities, and the ones that are further away would have lower priorities. This insures that the buses are short and the cores assigned to each router are condensed. The disadvantage of this approach is that the cores that are on the extremities will have less likelihood of being assigned therefore becoming unusable.
Another approach is to have R priorities for every core, each corresponding to the distance of each router and its position in the Mesh. Meaning, if a core is on the lowest left corner, then its priority relative to the top right router will be low, but relative to the cores in the center of the mesh, the priority would be high.

However, since this algorithm will be used at run time, speed of execution and the amount of data stored is very important. To simplify this proposed method, we used one priority for every core based on its location in the Mesh. The priority values start at $p$ which is equal to the number of cores per row. The cores on the boundary have the highest probability of getting stuck, therefore the priority of those corner cores are set to the highest. Following that, the priority decreases while going inward. The priorities for the cores at the center of the mesh are set to zero as they do not have a high probability of being blocked. Additionally, simulations showed that setting these values as all zeroes instead of a linear decrease including the values of three, two and one provided for greater flexibility and granted the inner groups the ability to grow in a balanced way.
An example of the priority values for an 8x8 core is shown in Table 2.1. Because of assigned priorities, the routers at the corners of the mesh will have a tendency to connect to the cores at the corners and edges of the mesh, thereby minimizing the risk of cores being unusable. The algorithm is shown in Algorithm 1.

**Algorithm 1: Mapping Algorithm**

As seen, the algorithm first assigns priorities to each processor core. When a process asks for NP cores, the algorithm lists the routers have at least NP cores available. It then chooses the router that has the least amount of available cores that is greater than NP. The greedy approach is used in this aspect. Once the router is chosen, the algorithm then starts forming the bus. It enqueues the available neighboring cores in terms of priority. The processor core connected to the router is
Figure 2.9: Before and After New Assignment
Table 2.1: Priority of Cores

<table>
<thead>
<tr>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

the first core in the queue. It then starts queuing the neighbors of the cores in the priority queue. It continues this process until NP cores are used to form a bus.

In Figure 2.9, we can see an instance of the mapping algorithm. If we need to map 5 cores to a new bus, then we see that R1 has 6 cores available and R2 has 12 cores available, then the algorithm will choose R1. The smaller one is chosen in case in the next cycle another process needs more than 6 cores. Once R1 is chosen, the bus is formed starting with the nearest cores of highest priority.

The result of the mapping algorithm is shown in Figure 2.10. The numbers at the top of each core represent the router to which a particular core is assigned. Negative one means that a particular core is not assigned.

Of course there is always room for improvement in order to ensure better mapping. However, this algorithm guarantees an assignment if a path exists since it checks the number of available cores for every router. Furthermore, if a hardware failure should occur, then the algorithm can assume the failed processor core is always assigned, and continues operation without major disruption. The algorithm was implemented in Java. The run time is less than 1 second for architectures with over a hundred cores.
2.3 Experimental Results

In the experimental analysis, we show two sets of data. The first one is to show the benefits of the hybrid model using the Network Simulator (NS2) tool [3]. The second set shows the run-time improvement of the reconfigurable architecture using the SIMICS simulator [5] [52].

2.3.1 Benefits of the Hybrid Model

In RAMS, since processor cores that communicate regularly are on the same bus, the requests sent on the network are fewer when compared to the NoC implementation since in the NoC implementation, all communication between cores is done using the network.

To compare the two models, we simulated two architectures. The first is a 9x9 NoC implementation and the second is a 3x3 hybrid bus-network implementation with 8 cores per bus. In
the NoC implementation requests are generated every 1 microsecond, while in the hybrid implementation, 2 scenarios are shown, one when requests are generated by the nodes every 4us and the other every 8us. The result computed is the average wait time it take to process a request and is dependent on the number of hops of the mesh. The bandwidth of the links is 10MB with a propagation delay of 10us.

In both cases of the hybrid implementation, process requests were received faster when compared to pure NoC. The delay was computed as the number of hops needed to get from source to destination. Queuing delay is included in the results. This result was expected based on the C++
2.3.2 Benefits of Reconfigurability

To verify the merits of reconfigurability, we simulated the architecture in Simics. The simulation variables used for the experimental results are shown in Table 7.2. We used the benchmark suite PARSEC since it was developed solely for multicore architectures [15] [14].

The number of hops is evaluated based upon the time it takes to access a physical cache addresses tied to an individual router. In the non-reconfigurable model, processor cores were attached to routers in an even allotment. Meanwhile in the RAMS model, cores were attached to routers using the reconfigurable approach. The results generated from the x264, streamcluster and fluidanimate PARSEC benchmarks with a 3 x 3 L2 Mesh and 64 cores are shown in Figures 2.16, 2.15 and 2.14. Figures 2.17 and 2.21 show the average number of network hops per L2 access for
Figure 2.14: Average Number of Hops to Access Data on each L2 Slice

Figure 2.15: Average Number of Hops to Access Data on each L2 Slice
Figure 2.16: Average Number of Hops to Access Data on each L2 Slice

Figure 2.17: Average Number of Network Hops for 64 Cores
Figure 2.18: Average Number of Hops to Access Data on each L2 Slice

Figure 2.19: Average Number of Hops to Access Data on each L2 Slice
Figure 2.20: Average Number of Hops to Access Data on each L2 Slice

Figure 2.21: Average Number of Network Hops for 128 Cores
all the L2 slices on 64 processors and 128 processor architectures, respectively. These plots show the average one-way networks hops consumed in accessing each of the 9 L2 slices. After comparing the Non-Reconfigurable and RAMS approaches, there is a clear trend showing that overall network activity on a per router basis is greatly reduced in RAMS. This is further evidenced in Figure 2.21 where each benchmark showed a minimum average improvement of at least 45% hops per cache access.

Figures 2.19, 2.20 and 2.18 shows a nearly identical experimental setup, however this simulation involved 128 total cores. Again the Average Network Hops per L2 access are greatly reduced when using the RAMS approach. On every benchmark and for nearly every L2 slice, the overall network activity per access is significantly reduced. All three benchmarks saw the Average Network Hops per L2 Access reduced by a minimum of 38% hops per access This significant increase is evidence of the usefulness and need for a reconfigurable approach such as RAMS.
2.4 Conclusion and Future Work

Hybrid bus-based and NoC architectures are essential to gain the performance benefits of both. In such hybrids, the superiority of bus-based architectures for small number of processors can be combined with the scalability of NoC architectures for large number of processors. Further, configurability introduces additional flexibility in allocating optimal bus sizes for each individual application, maximizing processor utilization and enhancing reliability. In this chapter, we have presented a new hybrid, configurable architecture named RAMS. Preliminary studies show that the proposed RAMS architecture is superior to a fixed bus or a fixed topology NoC architecture.

Further study is needed to fully understand the cost of reconfigurability. Various cache optimizations can be used to find further enhancements to the RAMS architecture. Operating system level procedures to further target placement in local caches proved useful in reducing both network traffic and cache contention in [26] and [12].

All of these experiments could be implemented in variations on the RAMS architecture to provide additional enhancements. The most significant improvement that can be made to enhance the flexibility of the architecture is to allow cores the ability to be assigned to any router. This ensures that no core can ever get stuck in an unusable corner. It also assures that each core will be placed or connected to its ideal router. By building an architecture that can allow cores to be assigned to any router without affecting other cores, opens up the doors to dynamic reconfiguration as will be explained in the following chapter.
3.1 Introduction

The number of processor cores in a chip is increasing with every generation. What was once a concept is quickly turning into reality as Intel recently prototyped the 48 core chip for laptops [66]. Envisioning dies with hundreds of simultaneous cores is not a difficult task, but these advances are not without challenges. One of the main issues with having multiple cores running simultaneously is providing data as quickly as possible to all the cores. The common bus architecture is not feasible because the saturation threshold is too low. Network on Chip is gaining popularity due to its scalability and efficiency [45].

In the NoC scheme, each processor core has a private L1 cache and the L2 cache is divided among different routers on the network. Processors can access remote routers by sending data requests with the destination address. The shortcoming of this approach is the vast difference in access time between the different cache banks since they are dependent on the location in the mesh...
architecture. Long wires can significantly hamper access time when transferring data from the L2 cache to the processor. This approach is called Non Uniform Cache Access (NUCA) [42] [22] [25]. There are two kinds of NUCA, the static [42] and the dynamic [41]. In D-NUCA run time data migration is possible unlike S-NUCA.

Different mapping strategies have been tested to reduce the overall access time. Foglia et al. in [30] show that the characteristics of the application running on a system along with the mapping protocol used, significantly effect the overall access time. They also discussed migration schemes suitable for different topologies. Dybdahl and Stenstromr in [29] propose dynamically changing the amount of private cache that cores can utilize on a need basis. Increasing the private cache allows for more data to be stored locally thus reducing overall access time.

Data migration and replication are also used to reduce the discrepancy in data access time from different cache banks. In data migration, data is moved from one cache bank to another based on the frequency of accesses by different processor cores. But moving data to the processor accessing it may not suffice if more than one core is accessing said data. Kandemir et al. [40] place the cache line in an optimal location convenient to all the cores accessing that data. Accessing data found in a local cache is significantly faster than one found on a remote cache. Therefore, although this approach reduces access time, it cannot place the cache line in the local bank of any processor since multiple processors are accessing the data at the same time.

Hao et al. [32] take advantage of spatial locality of data references. They use Prediction based L2 cache data Migration algorithm where they predict the data that will be used next by a processor based on sequential locality and migrate that data before the processor requests it. This improvement is valid in the case where one processor core is accessing a cache line but this approach does not address data shared among multiple cores. Another approach is data replication. Jin and Cho [39] propose analyzing the behavior of the program at compile time and placing data initially based on that analysis. Beckmann et al. [13] propose Adaptive Selective Replication where data is
replicated only if the benefit outweighs the cost.

Others have taken on a completely different approach altogether. They have proposed changes in the architecture to reduce cache access time. Das et al. [28] propose a hybrid Bus/Network architecture where eight processor cores are connected to a bus and the bus is connected to a router. In the first chapter we introduced a reconfigurable architecture where the number of cores connected to a bus is variable and dependent on the characteristics of the executing application. The reconfiguration in this proposed architecture is limited and is statically done before the runtime of the application.

The idea of reconfiguration is similar to thread migration [27]. Thread migration can move threads to cores closer to the frequently accessed router, but that impacts two threads at the same time since another thread needs to be moved to the original core. Furthermore, if many cores are accessing the data in the same router, switching the migrating thread with another may not necessarily improve data access time since the cache access time of the reallocated thread will be adversely affected. The main difference is that with reconfiguration, the number of resources on each bus can be changed, while with thread migration, the number of resources on each router is fixed at design time.

In the previous chapter, we introduced the RAMS architecture that allowed cores to be connected to a limited number of routers. That architecture was limited in flexibility and it could only assign cores to routers prior to the execution of the process. In this chapter, we propose changing the mapping of cores to routers at runtime. To achieve this goal, we propose a hybrid bus-network architecture with reconfigurable interconnect, where the number of processor cores attached to each bus is reconfigurable and is dependent on the needs of the active processes and applications executing on the system. Unlike in RAMS where configuration is done statically, the number of cores attached to each router is changed dynamically at runtime depending on the cache accesses of every core. This allows the reallocation of resources based on the needs of the executing
3.2 Proposed Architecture

In order to dynamically assign cores to routers, two aspects need to be considered. The first is that more than one processor can be connected to a router at the same time, and the second is that the architecture allow dynamic reconfiguration. In this section, we present the proposed architecture and then show a segmented variation of the proposed architecture.

3.2.1 DyaReMA

DyaReMA allows the dynamic reassignment of processor cores based on the history of cache accesses. Processor cores that access remote data more frequently than the local data are reassigned to the remote router. Any processor core can be assigned to any router and processor assignments can be changed dynamically during runtime. DyaReMA is shown in Figure 3.1.

1. Packet Switched Layer In the figure, the squares at the top are the routers connected in a mesh. There are $k \times k = R$ routers. The last level cache (L2 in our case) is a dynamic NUCA and is distributed among the different routers. In order to access data found on a cache bank, the request is sent to the local router. Once the router determines the destination address, the request is packetized and sent on the packet switched network. The request is moved from one router to the other using the XY routing algorithm.

2. Set of Processors The circles at the bottom of the figure are the processors. There are $p \times p$ processors. The processors have private L1 cache banks. When the data is not in the L1 cache, the request is sent to the router to which the processor is connected. Since more than one processor core can be connected to the router, we introduce a circuit switched routing layer.
Figure 3.1: Proposed Architecture
3. **Circuit Switched Routing Layer** The circuit switched routing layer is used to form buses that in turn connect to the routers. Since each router needs a dedicated bus, there are $R$ routing layers. The processors connect to the routers by programmable switches. Every processing core has $R$ programmable switches, and each switch is connected to one circuit switched routing layer. Since processors can be assigned to any router, this may lead to long buses being formed and the wire delay it ensues may be significant. We propose using buffers at every cross-section of the mesh or a pipelined bus. This way the clock frequency is not hampered by long wire delays.

4. **Programmable Switches** The processors are connected to all the circuit switched routing layers by programmable switches. If the switch of a processor is *on* for layer $j$, that means the processor is connected to router $j$. Processors can be connected to one router at a time, therefore, only one programmable switch of a processor can be *on* at one time. In Figure 3.2, the red boxes signify that the switch is *off* and the green ones signify that the switch is *on*. In this example, processors 2, 3, 4, 6 and 7 are connected to the router. Assigning a core to another router is a simple step, but deciding when to reassign a processor is the challenge and it depends on the history of data accesses as explained in the LRU Policy Section.

5. **Router Bus Connectivity** Each of the routing layers are connected to one router. These are assigned at design time and cannot be changed.

**Distance Model**

In order to ensure that our analysis takes into consideration the delays that the circuit might incur, we included distance calculation while analyzing the benchmarks. The distance model is shown in Figure 3.3. The distance between 2 processes is 1 unit time. In the ITRS roadmap [4], it is stated that bus delay across 10mm is expected to be 12 clock cycles, meanwhile the new
processors are around 140\textit{mm}$^2$ [2]. Therefore, assuming the cores span over 100\textit{mm}$^2$, a one cycle delay between cores on the bus is a reasonable assumption. Assuming regular architecture, the routers should be placed equidistant from one another. The distance between routers is calculated as $\frac{p-1}{r}$ where $p$ is the number of processor cores on each row and $r$ is the number of routers on each row. This ensures that the routers are equidistant. We then setup a distance matrix that represents the distance from every processor core to every router. When calculating the number of hops a core needs to access a remote router, we add the bus distance between the core and the local router of the circuit switched layer and network hops needed to get to the remote router on the packet switched layer.

**Processor Reallocation Policy**

The processors are reallocated to different routers based on the history of memory accesses. The idea is to take advantage of spatial and temporal locality which states that if a data is accessed, it is very likely that it will be accessed in the future, also if a data is accessed, the surrounding data will be accessed as well. The difference between this method and data migration is that if the processor is reassigned, it will have access not only to the data it was accessing, but to
Figure 3.3: Distance Calculation

\[ \text{Distance} = \frac{(p-1)}{r} \]

- \(p\): Number of Cores on each row
- \(r\): Number of Routers on each row

1 Unit Distance
Table 3.1: Example of Counters

<table>
<thead>
<tr>
<th>Router Accessed</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>R2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>R4</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>R1</td>
<td>1</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

all the data found in that cache bank taking advantage of spatial locality. Another advantage of the proposed method is that there is no need for fine grained book-keeping of cache line accesses but rather the entire bank found on the destination router, thereby reducing the overhead.

In order to keep track of the data accessed, each processor has $R$ counters where $R$ is the number of routers in the system. The idea is to compare data accessed between the local cache and a remote one. If a processor core accessed a remote router more often than it accessed the local router, then that processor is a good candidate to be reassigned to the remote router.

To keep track of the count, every time the processor accesses a cache line, the counter corresponding to the target address is incremented if the target is in a remote location to determine the pattern of behavior of each processor core. If a processor core accesses the local cache, then all the counters are decremented to reflect only the difference of data accesses in the counters. An example is shown in Table 3.1.

In this example there are four routers. The processor with the corresponding counters is connected to Router 1. Every time it accesses Router 1, it is considered a local access. When it accesses any other router it is considered a remote access.

The first column represents the router accessed by the processor core. As seen in row 1, the processor accessed Router 2 and therefore, the corresponding counter C2 is incremented. In the last row, the processor accesses the local router, and therefore, the counters are all decremented.

But moving all the processor cores to their ideal location may result in all the cores being assigned to one router if the data on that router is heavily used by all other cores. Therefore, in the proposed method, after moving a processor core ($MP$) and assigning it to the target router
Chapter 3: Reconfigurable Multicore Architecture for Dynamic Processor Reallocation

If the number of cores on the target router reaches a certain threshold $N$, then another victim processor $(VP)$ from the target router is moved to the router $(RO)$ that hosted the moving processor core $MP$. The processor core that is chosen to be moved from $RT$ is the one that is Least Recently Used meaning the core that accessed data found on $RT$ last. In order to keep track of the LRU, the algorithm assigns counters to each processor. When a core accesses the local router, the counter of the other routers are incremented, and its counter is set to 0 since it just accessed the local counter. The algorithm is shown in Algorithm 2.

Input: List of Routers $R$, List of Processors $P$

Threshold = $N$

$Memory_{Target} = Destination_{Router}$

if $Memory_{Target} == Local_{Router}$ then
  for $i = 1$ to $R$ do
    Counter$(i) ←−$
  end for
  for $i = 1$ to $P$ do
    LRU$(i) + +$
  end for
  LRU$(Current\ Core) ← 0$
else
  Counter$(Memory_{Target}) + +$
  if Counter$(Memory_{Target}) == N$ then
    Find Core with Highest LRU
    Swap cores
    LRU$(ReplacedCore) ← 0$
  end if
end if

Algorithm 2: LRU Algorithm

There can be variations to the proposed method. For example, the processor core $(VP)$ that is chosen to be swapped is not moved to the same place as the original processor $(MP)$ but rather to a different location close to its original location if the neighboring routers have enough space to accommodate this processor $(VP)$. 
3.2.2 DyaReMA with Reduced Number of Meshes

Having \( R \) number of meshes can be very costly. An alternative is to reduce the number of meshes to less than the number of routers as shown in Figure 3.4. The number of meshes needed can range between 1 and \( R \) based on the characteristics of the applications for the target architecture. This means that the routers have to share the mesh layers as shown in the figure. Figure 3.5 shows two routers connected to one mesh. Processors 2, 3, 4, 6, and 7 are connected to Router 1, while processors 9, 10, 11 and 12 are connected to Router 2. The buses are then connected to the router. The rest of the processor cores can then form another bus and connect to a different router as long as a path exists between the core and the router. Therefore, routers can share the mesh, but this can limit the ability of the processors to connect to all routers.

If the number of buses is less than \( R \), then a routing algorithm needs to be used to find a path from the core to the target router on the available layers. If a path cannot be found due to other assignments, then a core cannot be reassigned to the router. Reducing the number of meshes available does reduce the area, but the reassignment time is increased since a path needs to be constructed between the core and the target router. On the other hand, having \( R \) meshes consumes more area, but reassignment is a one step process. Therefore, an analysis should be done on the importance of area versus timing for the target architecture to determine the number of meshes that should be available in the architecture.

3.2.3 Segmented DyaReMA

Ideally, all cores should be able to connect to any router to achieve maximum potential. But with current technology, allowing all cores to be connected to any router is very costly. An alternate would be to allow some cores the flexibility to migrate while others are fixed to certain cores. Figure 3.6 shows one such architecture. It is a 4x4 mesh NoC architecture with 128 cores. Some cores are migrating cores meaning they can be reassigned to different routers, while others are
Figure 3.4: DyaReMA with Reduced Meshes

Figure 3.5: Bus Interface
non-migrating meaning they are assigned to one particular router and cannot be migrated. Each 2x2 NoC has a switchbox that controls the migration of the cores. In this scheme, migration happens only among those 4 routers, cores cannot be migrated to routers connected to different switchboxes.

Figure 3.7 shows the implementation of the switchbox. Only the connection for one of the nodes is shown in the figure. The migrated masters on the bus are empty locations where the migrating masters (cores) can reside. The AHB bus signals of the migrating masters are multiplexed to drive the signal of the migrated master. Since the migrating master can occupy only one slot, the register file is used as the select signal of the multiplexer to determine which of the migrating cores from other nodes can migrate. The register file keeps track of the location of the migrated masters.

\[
\text{Number of multiplexers} = \text{Number of migrating masters from each node} \times 4 \times \text{no of signals in the bus} \times 2
\]

\[
\text{Number of input for each multiplexer} = \text{Number of migrating master in a node} \times 4
\]
Figure 3.7: Implementation of Switchbox
The bus is implemented as AHB bus protocol [1]. If data is not found in the local router, the processor request cannot be fulfilled quickly. When the peripheral does not have the data ready, it can terminate the request by the master with a split transaction. The bus is then granted to another master to fulfill its data request while the data request of the first core is being processed by the packet switched network. The AHB arbiter is responsible for assigning the control of the bus to the masters sitting on the bus. It multiplexes the request signals from the masters and drives the common bus by the signals of the selected master. The design of the AHB arbiter is based on the Round Robin scheme. The master gets the control of the bus for N number of clock cycles at a time, this can be programmed at static time. In the design, the maximum number of clock cycles required to switch the grant between two masters is (N-1), where N is number of masters sitting on the bus.

3.3 Experimental Results and Analysis

3.3.1 DyaReMA Simulation Results

To verify reduction in access time when reassigning cores from one router to another, we simulated the architecture in Simics [52]. The simulation variables used for the experimental results are shown in Table 7.2. We used the benchmark suite PARSEC [15] since it was developed solely for multicore architectures.

Figures 3.8 and 3.9 show the results of the benchmarks having 64 and 128 cores respectively. We used 3x3 and 4x4 routers. Each core has a private L1 cache and the L2 cache is shared among the routers, so it is essentially divided into 9 or 16 cache banks. The threshold number that is increasing from 4 to 16384 is the difference of data accessed between the local router and the remote routers before making a swap. Meaning, if a processor core accesses the remote router N times more than it accessed the local router, then that processor core is reassigned to the remote router. In this case, we did implement the LRU mechanism where when a processor core is assigned
Table 3.2: Experimental setup

<table>
<thead>
<tr>
<th>Parameter</th>
<th>value used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of routers</td>
<td>9 or 16</td>
</tr>
<tr>
<td>Number of cores</td>
<td>64 or 128</td>
</tr>
<tr>
<td>L2 Size in KB</td>
<td>$3072 \times \text{number of processors}$</td>
</tr>
<tr>
<td>L2 Line Size</td>
<td>128B</td>
</tr>
<tr>
<td>L2 # of Cache Lines</td>
<td>$L_2 \text{ Size} \times 1024 \times L_2 \text{ Line Size}$</td>
</tr>
<tr>
<td>L2 Associativity</td>
<td>8-way</td>
</tr>
<tr>
<td>L2 Replacement Policy</td>
<td>LRU</td>
</tr>
<tr>
<td>L1 Size in KB</td>
<td>64</td>
</tr>
<tr>
<td>L1 Line Size</td>
<td>128B</td>
</tr>
<tr>
<td>L1 # of Cache Lines</td>
<td>$L_1 \text{ Size} \times 1024 \times L_1 \text{ Line Size}$</td>
</tr>
<tr>
<td>L1 Associativity</td>
<td>2-way</td>
</tr>
<tr>
<td>L1 Replacement Policy</td>
<td>LRU</td>
</tr>
<tr>
<td>Benchmark</td>
<td>PARSEC</td>
</tr>
</tbody>
</table>

to a new router, a victim processor is chosen from the target location based on the access history and assigned to the router that lost a processor core. This assures that all the routers have the same number of cores that were previously assigned to them.

The figures show a significant improvement garnered from assigning processors to routers. The unit measured is the percentage gain compared to non reconfigurable model in average number of hops that a request needs to access data found in a remote cache bank. As seen, there is a reduction of up to 24% of average hops required to access a remote data. This is compared to the hybrid model that in itself has shown an improvement in access time compared to the regular NoC model. The chosen threshold has a significant impact on the output. If it is too small, then in some cases, premature reassignments can be made thus introducing large overhead. If it is too large, then cores will not be reassigned to other routers. In our case, a threshold beyond 256 has an adverse effect on the 64 core architecture, and a threshold larger than 2048 has an adverse effect on the 128 core architecture. So choosing a good threshold value is paramount to obtaining significant gains.

The simulation results show that reassigning cores to routers is a viable method to reduce data access time and should be considered while designing multicore architectures.
Of course the proposed architecture introduces a large overhead due to the bus layers. Since the proposed architecture is a hybrid architecture, the number of routers needed for the 128 core architecture is 16 instead of 128. For a generic router of size $0.3748 \text{mm}^2$ in 90nm technology [49], the area savings is around $42 \text{mm}^2$. For the bus layers, the length of the bus needed is assumed to be $12 \times 10 \text{mm}$ to reach all the cores for each layer since the cores are arranged approximately $11 \times 11$. In 90nm technology, wire width of 3 lambda and spacing of 4 lambda, a 128 bit bus has an area of $4.8384 \mu \text{m}^2$. For 16 mesh layers the area needed is $77 mm^2$. Therefore, the area increase is around $35 \text{mm}^2$. If there are $k$ metal layers, then the area increase due to the mesh layers is reduced by a factor of $k$ since each mesh can be assigned to a different metal layer. With transistors getting smaller and 3D chips on the horizon [49] [62], this area overhead will be acceptable in the near future.

### 3.3.2 Synthesis Models

To obtain realistic estimates of the area and performance of DyaReMA, we have developed a complete Verilog model of the Segmented DyaReMA explained in section 3.2.3. The com-
Components that we developed are shown in Table 3.3. The developed model is highly parametrized. The components can be used to synthesize different architectures ranging from bus based, to NoC, to hybrid NoC, and reconfigurable hybrid NoC architectures. More details can be found in Chapter 4.

From the synthesis results, we observed that in the segmented DyaReMa architecture shown in Section 3.2.3, five clock cycles are needed to access data found on the local router. Meanwhile, on average, data obtained through the network with idle traffic takes about 80 clock cycles. Therefore, migrating the core to a remote router gives sixteenfold performance improvement. If the design with core migration has maximum operating frequency less than that of the design without core migration due to the worst case timing path added by the switch boxes, the bus can be pipelined. If a 5 stage pipeline is added it will still give an improvement factor of 8.

Table 3.4 shows the synthesis parameters that were used for the segmented DyaReMA. We used Synopsis DC Compiler to obtain the area. The gate count in Table 3.5 is calculated in terms of NAND gate area for X1 drive strength which is 0.798.

We synthesized 3 different architectures, NoC, hybrid architecture, and segmented DyaReMA
Table 3.3: Implemented Components

<table>
<thead>
<tr>
<th>Components</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AHB Master Interface</td>
<td>Connects processor and L1 cache interface to AHB Bus</td>
</tr>
<tr>
<td>AHB Slave</td>
<td>Interface between AHB Bus and Peripherals</td>
</tr>
<tr>
<td>AHB Arbiter</td>
<td>Assigns control of the bus to the masters connected to the bus</td>
</tr>
<tr>
<td>Processor</td>
<td>Simple processor to generate data requests</td>
</tr>
<tr>
<td>L1 Cache Controller</td>
<td>write-through, no-allocate policy and up to 4-way set associative</td>
</tr>
<tr>
<td>Router</td>
<td>Connects local node to the neighboring nodes</td>
</tr>
<tr>
<td>L2 Cache Interface</td>
<td>Arbitrates the L2 cache request from local and remote nodes</td>
</tr>
<tr>
<td>L2 Remote Router Interface</td>
<td>Interface between local and remote L2 cache request</td>
</tr>
<tr>
<td>Round Robin Arbiter</td>
<td>Implements Round Robin scheme</td>
</tr>
<tr>
<td>Packetizer</td>
<td>Packetizes the request to remote L2 cache</td>
</tr>
<tr>
<td>Depacketizer</td>
<td>Forms the packet from the flits generated by the packetizer</td>
</tr>
<tr>
<td>Request Queue</td>
<td>Stores the request from the local bus for remote node L2 cache access</td>
</tr>
<tr>
<td>Performance Counter Logic</td>
<td>Used to analyze the performance of the system</td>
</tr>
<tr>
<td>Split Request Completion Logic</td>
<td>Interface between Common Bus and Packetizer remote L2</td>
</tr>
<tr>
<td>Switch Box</td>
<td>Explained in section 3.2.3</td>
</tr>
</tbody>
</table>

Table 3.4: Synthesis Setup

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Non Migrating Core per node</td>
<td>3</td>
</tr>
<tr>
<td>Number of Migrating Core per node</td>
<td>2</td>
</tr>
<tr>
<td>Number of Migrated Core support per node</td>
<td>2</td>
</tr>
<tr>
<td>Register File Depth</td>
<td>19</td>
</tr>
<tr>
<td>Processor PC Load Address Register Number</td>
<td>16</td>
</tr>
<tr>
<td>Register File Address Width</td>
<td>5</td>
</tr>
<tr>
<td>Switch Data Width</td>
<td>8</td>
</tr>
<tr>
<td>Switch Select Width</td>
<td>3</td>
</tr>
<tr>
<td>Migrating Core Enable Register Number</td>
<td>18</td>
</tr>
</tbody>
</table>

using the components shown in Table 3.3. The results showed that a 2x2 NoC architecture with 4 processor cores has a gate count of $258K$ ($64.5k$ per core). A hybrid architecture with 2x2 routers and 8 processing cores per router has a gate count of $382K$ ($11.9K$ per core). A reconfigurable hybrid NoC with 2x2 routers and 7 processing cores has a gate count of $354K$ ($12.6K$ per core).

The processor cores have limited instruction set. The goal of the HDL platform is to evaluate the cache access time of different multicore architectures. Therefore, the processors have memory instructions to generate memory requests that cover the entire address space.
Table 3.5: Synthesis Results

<table>
<thead>
<tr>
<th>NoC</th>
<th>L2</th>
<th>Data/Addr Width</th>
<th>Flits</th>
<th>Cell Area</th>
<th>Total Area</th>
<th>Gate Count(K)</th>
<th>Max Freq(MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x2</td>
<td>64</td>
<td>8</td>
<td>4</td>
<td>185187</td>
<td>234063</td>
<td>232</td>
<td>250</td>
</tr>
<tr>
<td>2x2</td>
<td>256</td>
<td>10</td>
<td>4</td>
<td>283271</td>
<td>355837</td>
<td>355</td>
<td>250</td>
</tr>
</tbody>
</table>

This shows that a hybrid architecture or a reconfigurable hybrid architecture have similar gate counts per processing unit, meanwhile a simple NoC architecture has a lot more overhead per core. Therefore, the proposed reconfigurable architecture is a viable option to reduce cache access time.

### 3.4 Conclusion

In multicore systems, Non Uniform Cache Access [42] is a reality and reducing the access time is a challenge all designers face. Data replication and data migration are two common methods that have been studied to reduce access time. With data replication, cache coherency becomes an issue that can become too burdensome for the network to handle. With data migration, the challenge is keeping track of where data migrated and how the optimal location of that cache line is found on the network. Thread migration is an alternative, but that requires reallocating threads away from local resources. The proposed method tries to reduce access time by reassigning cores to routers instead of or in addition to data migration. When cores are reassigned to routers, then the cores have access to not only the data they are currently accessing but to the neighboring data as well taking advantage of spatial and temporal locality. If multiple cores access the same data, then the cores can all be assigned to the same router thus benefiting from accessing the data locally. In order to achieve this, we proposed a hybrid bus/network with reconfigurable interconnect. Our experimental results using Simics showed up to 24% improvement in average number of hops when accessing remote data. We also introduced a hardware implementation of a simplified version of the proposed architecture to show that the area penalty is not significant.
The idea is not to use processor reassignment as a standalone method. This approach can be combined with data migration to further reduce cache access time. If data is being accessed by one processor core, and that processor core is accessing only a few line of the remote cache, then data migration is sufficient without the need to reassign the router and the overhead involved in that process. But if multiple processor cores are accessing a remote data continuously, then placing those cores on the same router can reduce access time significantly. Another scenario where this method is useful is when a processor core is accessing many cache lines of the remote L2 cache, then instead of moving all the cache lines to the local router, the processor core can be moved to the remote router. These methods used in combination with each other can take advantage of the provided architecture and improve the overall runtime of a process running on the system.

We will discuss alternate methods of reducing cache access time in Chapters 6 and 7.
Chapter 4

Hardware Simulation Platform\textsuperscript{1}

4.1 Introduction and Motivation

Simulation tools are not flexible when trying to simulate different architectures. The most widely used simulation platform for multicore architectures is Simics. While Simics is a powerful simulation tool, it has its limitations. First and foremost it's a software simulation tool, meaning it only mimics hardware simulation. It can give cycle accurate information, but it gives no insight about hardware cost, power dissipation or hardware delays. Furthermore, the basic features provided by Simics does not have the capability to simulate NoCs. Simics sees the L2 cache as one cache block and the parallelism of the multicore architecture is only an imitation. The instructions on the different cores are executed in a round robin fashion to ensure parallelism. The data is seen as a contiguous address space where data accesses are uniform. Though this gives an accurate estimation of delays and congestion on a bus based design, the mainstream multicore architecture is headed towards NoC designs. The main distinguishing feature in NoCs and Bus architectures is the Non-Uniform Cache Access that is introduced due to the nature of the mesh architecture of NoCs. Race

\textsuperscript{1}This chapter contains collaborative work with Natwar Agrawal [9]. Further details are included in [8]
conditions can occur while multiple cores access the same data, but this phenomenon cannot be computed or predicted in Simics. An add-on to Simics was developed by researchers at University of Wisconsin Madison called GEMS [54], that implements NoC on top of Simics to create the mesh architecture and introduce the real delay of the cache data accesses. Since it’s based in software, it fails to model the hardware fault that might cause damage to flits thus requiring requests to be resent. Furthermore, both Simics and GEMS are software simulators, they cannot give any information on area or power dissipation. Another shortcoming of this model is the fact that simulations are slow, they can last in access of 3 days to simulate large benchmarks.

The shortcomings of software simulators open up the need for better simulation platforms. In order to generate the hardware area cost, heat dissipation, power consumption, the simulator should entirely be based in hardware. It might be feasible to estimate area cost on paper, but other features such as heat dissipation or power consumption require hardware implementation of the architecture. The Hardware simulator tool should be able to cater to the needs of a large range of architectures. Therefore, the components should be modular so they can be used as building blocks to construct larger architectures. The basic components that any basic multicore hardware simulator should have are bus, router, processor, cache blocks, and the detailed sub components that are needed by these basic components.

4.2 Possible Architectures

The components designed in our Hardware Platform can be used to form a bus based architecture, an NoC Architecture, a Hybrid bus/NoC architecture, and the segmented DyaReMA presented in Chapter 3.
4.2.1 Bus Based Architecture

For a bus based design, multiple processors are connected to one bus and the bus in turn is connected to the L2 cache. A bus based design is shown in Figure 4.1. The components needed to ensure the correct operation of the bus based design are as follows:

- **AHB Arbiter**: Since only one processor can write on the bus at one time, a controller is needed to manage the communication between the processor cores and the L2 cache. The AHB Arbiter grants the bus to the processors in a round robin manner. Once the arbiter grants the bus to the processor, the processor sends its data request to the cache. If the data is not found in the L2 cache, the data is fetched from Main Memory. During this time, the bus is idle and can be used to process data requests of other processors. To provide this capability, the AHB Arbiter can perform a split transaction where this request is put on hold and the bus grant is passed on to the next processor.

- **Round Robin Arbiter**: Round Robin Arbiter is used to implement the round robin policy of the AHB Arbiter. It gives the bus grant to the first processor then passes on the grant to subsequent processors in series.

- **AHB Master Interface**: The AHB Master Interface provides the platform where the processors and the Arbiter can communicate with each other. For a bus based design, multiple processors will be connected to one bus. Therefore, an interface is needed to control the communication between the processors and L1 caches and the AHB Arbiter.

- **AHB Slave**: The AHB Slave provides the interface between the AHB Bus and its peripherals.

- **Processor**: A processor is needed in any hardware platform tool. The design of the processor though is highly dependent on the needs of the architecture designers. The processor design can range from a simple ALU to a highly pipelined, multithreaded processor. To implement
Figure 4.1: Bus Architecture
real life benchmarks, the processors should have a solid instruction set to fulfil the needs of the applications running on it. For our needs, the main areas that need to be tested are the bottlenecks on the bus or the mesh. Therefore, a simple processor that generates data at certain intervals is sufficient to test the different designs and do a cost benefit analysis of the different architectures. The designed processor in our Hardware Simulation Platform is a simple processor that can generate data. The addressing mode allows it to generate requests for any address in the given address space. The design of the processor is shown in Figure 4.2.

- **L1 Cache Controller**: The designed L1 cache has write-through policy and a no-allocate policy where data is written directly to the main memory on a cache miss for write request. This reduces the overhead of bringing data into the L1 cache if not needed in future. The L1 cache is implemented as a 4-way set associative cache.
4.2.2 NoC

In the NoC architecture, the processor and the L1 cache controller are the same as before. But the interface between the processor and the L2 cache is done through the mesh architecture instead of the bus architecture. The NoC architecture is shown in Figure 4.3. The following components are needed to implement the mesh architecture:

- **Router**: In the NoC architecture, the L2 cache is split among the routers. Therefore, each router has a section of the L2 cache. When the processor generates a request, the data might be in any of the routers found on the mesh. The routers are used to transfer the request from source to destination. In our design, the flits are transferred between routers in the XY direction. Because of the segmented nature of the NoC architecture, the delay in accessing
data is non-uniform. The design of the router is shown in Figure 4.4.

- **Packetizer**: Read or write requests generated by the processors to the L2 cache tend to be many bytes long. In order to speed up the delivery process, requests are split into flits at the source router. Flits are then sent on the network to the destination router. The packetizer creates the request by setting the address in the header flit based on which the flow path is determined.

- **Depacketizer**: Since the requests are sent as flits that combined form a packet, the destination router needs to reassemble the packet than depacketize it to decode the request of the processor. Once the depacketizer decodes the request, it sends the information to the L2 cache to be
processed.

- **L2 Cache Interface**: The L2 bank found in the router can be accessed both by the processor that is connected to the router and the remote processors that send their request on the packet switched layer. The L2 cache interface arbitrates between the requests that are local to the router or the ones that are sent remotely. It implements the round robin scheme to arbitrate the requests.

- **Split Request and Completion Logic**: When a data line is not found in the local L2 cache bank, the request is sent to the target router to access the cache line. In order to send the request to the remote router, the request generated by the processor and sent to the local router is packetized. Once the request is packetized, it is sent on the packet switched layer to the remote router. The split request logic acts as the interface between Common Bus and Packetizer logic that packetizes the request to be sent to the remote router.

### 4.3 Hybrid Bus/NoC Architecture

In the Hybrid NoC/Bus Architecture, \( n \) processors are connected to a bus, and the bus in turn is connected to a router. The architecture is shown in Figure 4.5. The components to realize this architecture are a combination of the ones in the bus architecture and the NoC architectures. The bus that links all the processors to the router uses the AHB protocol. All the processors that are connected to the bus are the masters. When a processor needs data, it will set its bus grant request signal to low. The AHB arbiter uses the Round-Robin scheme to pass the bus control between the processors. The router is also similar to the one described in the NoC architecture. The cache blocks sit on the router, and the requests that come to the router are queued and processed in the order of arrival. The router also acts as the pass through for requests that are destined for routers further down the line.
Figure 4.5: Hybrid Architecture
The main difference between the bus and the hybrid Bus/NoC architecture though is how the bus communicates with the L2 cache block. While before the bus directly communicated with the L2 cache, in the hybrid model the bus sends the request through the router instead of sending it directly to the L2 cache block. The reason is that the L2 cache block processes requests not only from the local bus, but rather from all the others routers found on the network. On the simple bus architecture, requests to the L2 cache are sent one at a time since it is dependent on the bus grant. There is no need for queues. In the hybrid model, requests need to be queued and processed one at a time. Hence the need for a request queue.

In the Hybrid architecture, more than one processor exists on the bus. When a processor sends a memory request, the fulfilment of that request may take time. In the meantime, other processors do not sit idle. Instead, they are granted the bus while the first processor waits for memory. Since the router needs to process more than one request at a time, the request queue is setup to keep track of all the requests generated by the different processors that are connected to the bus.

4.4 Reconfigurable Hybrid Architecture

Since DyaReMA, the architecture proposed in 3.2.2, is very costly, the hardware platform tool proposed here offers the components to simulate the segmented DyaReMA introduced in 3.2.3. In the full DyaReMA, all cores can be connected to any router at any given time. The cores are chosen to be moved based on the access patterns between the core and the cache blocks. The cores are moved closer to the routers to decrease the latency of accessing data found in the L2 cache blocks. Ideally all the cores should be able to connect to any router, but as mentioned previously, that can be very costly in terms of hardware. A more realistic model is to allow the cores the ability to migrate to neighboring routers.
In the segmented DyaReMA, the base system is still a hybrid bus/NoC architecture. The cores are connected to a bus that implements the AHB arbiter. Each bus is connected to a router and the requests from the bus to the router are stored in the request queue. The routers are connected as a mesh on the packet switched network.

In addition to the base hybrid system, the processing cores have the ability to be reassigned to neighboring routers. In order to accomplish this task, a switchbox is used. The detailed explanation of the switchbox is shown in 3.2.3.

4.5 Evaluation of the Different Architectures

The performance of an architecture is mainly determined by the execution time of the processes running on the architecture. To compare two architectures, it is imperative to compare execution time. With processor cores getting faster and faster, the main impediment of the overall execution time is data accesses. Therefore, any hardware simulation tool should be able to measure the delay in accessing data. The data can be local in the L1 cache, it can reside in the L2 cache, or it is retrieved from main memory. In the bus architecture, the delay is dependent on the delay of the bus grant. In the NoC architecture, the delay is dependent on the queue sizes and traffic on the packet switched layer. Therefore, any performance metric that is measured should take into account the different architectural needs.

Based on these observations, we determined that the best way to determine data access time is to keep a counter in the processors. Every time a request is sent to access data not found in the local L1 cache, the counter is activated. The counter counts the number of clock cycles between the time when data is requested and the time data is received. This time varies and is dependent on the architecture and the location of the requested data. Once the overall data access time is determined by the counter, the average wait time can be determined by dividing the clock cycles
measured by the counter with the number of memory requests generated by the processor core.

The area comparisons of the different architectures is shown in Figure 4.6.
### COMPONENTS

<table>
<thead>
<tr>
<th>ADDRESS_WIDTH</th>
<th>LOC L2 CACHE SIZE</th>
<th>NUMBER OF PROCESSOR PER NODE/ROUTER</th>
<th>NOB</th>
<th>PAKET FIELD DIVISION (padding + header + src_adr + mno + type+ data+ addr)</th>
<th>DATA WIDTH</th>
<th>COMBINATIONAL AREA</th>
<th>NON COMBINATIONAL AREA</th>
<th>Total Cell Area</th>
<th>Total Area</th>
<th>GATE COUNT IN TERMS OF X1 NAND GATE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PURE NOC</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2x2</td>
<td>8</td>
<td>64</td>
<td>1</td>
<td>6x4+2</td>
<td>x+1</td>
<td>x+3</td>
<td>8+8</td>
<td>8</td>
<td>47857</td>
<td>74709</td>
</tr>
<tr>
<td>2x2</td>
<td>10</td>
<td>256</td>
<td>1</td>
<td>4x2+4</td>
<td>x+1</td>
<td>x+3</td>
<td>10+10</td>
<td>10</td>
<td>83265</td>
<td>122689</td>
</tr>
<tr>
<td>3x3</td>
<td>10</td>
<td>64</td>
<td>1</td>
<td>7</td>
<td>x+4</td>
<td>x+1</td>
<td>x+3</td>
<td>10+10</td>
<td>10</td>
<td>142095</td>
</tr>
<tr>
<td>3x3</td>
<td>12</td>
<td>256</td>
<td>1</td>
<td>3</td>
<td>x+5</td>
<td>x+4</td>
<td>x+1</td>
<td>x+12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>4x4</td>
<td>10</td>
<td>64</td>
<td>1</td>
<td>7</td>
<td>x+5</td>
<td>x+4</td>
<td>x+1</td>
<td>x+10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>4x4</td>
<td>10</td>
<td>256</td>
<td>1</td>
<td>5</td>
<td>x+5</td>
<td>x+4</td>
<td>x+1</td>
<td>x+12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>8x8</td>
<td>12</td>
<td>64</td>
<td>1</td>
<td>5</td>
<td>x+6</td>
<td>x+6</td>
<td>x+1</td>
<td>x+12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>8x8</td>
<td>14</td>
<td>156</td>
<td>1</td>
<td>5</td>
<td>x+5</td>
<td>x+6</td>
<td>x+1</td>
<td>x+14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td><strong>HYBRID NOC</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2x2</td>
<td>8</td>
<td>64</td>
<td>4</td>
<td>5</td>
<td>x+4</td>
<td>x+2</td>
<td>x+3</td>
<td>8+8</td>
<td>8</td>
<td>60178</td>
</tr>
<tr>
<td>2x2</td>
<td>8</td>
<td>64</td>
<td>4</td>
<td>4</td>
<td>x+4</td>
<td>x+2</td>
<td>x+3</td>
<td>8+8</td>
<td>8</td>
<td>82295</td>
</tr>
<tr>
<td>2x2</td>
<td>10</td>
<td>256</td>
<td>4</td>
<td>1</td>
<td>x+4</td>
<td>x+2</td>
<td>x+3</td>
<td>10+10</td>
<td>10</td>
<td>98460</td>
</tr>
<tr>
<td>2x2</td>
<td>10</td>
<td>256</td>
<td>8</td>
<td>0</td>
<td>x+4</td>
<td>x+2</td>
<td>x+3</td>
<td>10+10</td>
<td>10</td>
<td>125930</td>
</tr>
<tr>
<td>3x3</td>
<td>10</td>
<td>64</td>
<td>4</td>
<td>5</td>
<td>x+5</td>
<td>x+4</td>
<td>x+2</td>
<td>10+10</td>
<td>10</td>
<td>176291</td>
</tr>
<tr>
<td>3x3</td>
<td>10</td>
<td>64</td>
<td>8</td>
<td>5</td>
<td>x+5</td>
<td>x+4</td>
<td>x+3</td>
<td>10+10</td>
<td>10</td>
<td>238067</td>
</tr>
<tr>
<td>3x3</td>
<td>12</td>
<td>256</td>
<td>4</td>
<td>5</td>
<td>x+5</td>
<td>x+4</td>
<td>x+2</td>
<td>12+12</td>
<td>12</td>
<td>276530</td>
</tr>
<tr>
<td>3x3</td>
<td>12</td>
<td>256</td>
<td>8</td>
<td>5</td>
<td>x+5</td>
<td>x+4</td>
<td>x+3</td>
<td>12+12</td>
<td>12</td>
<td>350453</td>
</tr>
<tr>
<td>4x4</td>
<td>10</td>
<td>64</td>
<td>4</td>
<td>5</td>
<td>x+5</td>
<td>x+4</td>
<td>x+3</td>
<td>10+10</td>
<td>10</td>
<td>313443</td>
</tr>
<tr>
<td>4x4</td>
<td>10</td>
<td>256</td>
<td>8</td>
<td>5</td>
<td>x+5</td>
<td>x+4</td>
<td>x+3</td>
<td>10+10</td>
<td>10</td>
<td>423271</td>
</tr>
<tr>
<td>4x4</td>
<td>12</td>
<td>64</td>
<td>4</td>
<td>5</td>
<td>x+5</td>
<td>x+4</td>
<td>x+3</td>
<td>12+12</td>
<td>12</td>
<td>491636</td>
</tr>
<tr>
<td>4x4</td>
<td>12</td>
<td>64</td>
<td>8</td>
<td>5</td>
<td>x+5</td>
<td>x+4</td>
<td>x+3</td>
<td>12+12</td>
<td>12</td>
<td>623069</td>
</tr>
<tr>
<td><strong>HYBRID NOC WITH CORE MIGRATION</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2x2</td>
<td>10</td>
<td>256</td>
<td>3,2,2</td>
<td>5</td>
<td>x+5</td>
<td>x+2</td>
<td>x+3</td>
<td>14+10</td>
<td>14</td>
<td>153638</td>
</tr>
<tr>
<td>2x2</td>
<td>8</td>
<td>64</td>
<td>3,2,2</td>
<td>4</td>
<td>x+4</td>
<td>x+2</td>
<td>x+3</td>
<td>8+8</td>
<td>8</td>
<td>76577</td>
</tr>
<tr>
<td>2x2</td>
<td>10</td>
<td>256</td>
<td>3,2,2</td>
<td>4</td>
<td>x+4</td>
<td>x+2</td>
<td>x+3</td>
<td>10+10</td>
<td>10</td>
<td>118519</td>
</tr>
</tbody>
</table>
Chapter 5

A Heterogeneous Cache Distribution with Reconfigurable Interconnect\(^1\)

In NoC multicore architectures, the last level cache is distributed equally among the routers in the network. In this chapter, we propose having a heterogeneous distribution and show that it outperforms the homogeneous distribution. Furthermore, we propose reconfigurable cache banks that can be assigned to different routers based on the needs of the processes connected to the router.

5.1 Introduction and Motivation

Reducing cache access time is one of the foundations of a good architecture in a multicore environment. Hammoud et al. [31] proposed changing the global nature of L2 cache slices. Instead of having one shared L2 cache, sections of the cache are set as private and assigned to processors. Other sections of the cache can be shared by multiple processors based on predetermined assignments. Although effective, this method has large overhead in terms of cache coherency and data search.

\(^1\)This chapter contains collaborative work with Aishwariya Pattabiraman [64]. Further details are included in [63]
OS level page allocation using page coloring techniques proposed by Cho and Jin [26] is commonly used. This method ensures that data is stored close to the process accessing it, but it cannot cater to the needs of data intensive applications.

In this chapter, we show that a heterogeneous cache distribution of caches to routers is superior to the homogeneous distribution. Two methods are proposed as follows:

- **Static Heterogeneous Architecture:** In this architecture, the cache distribution is done at design time and cannot be changed at runtime. Extensive experimentation is needed to determine which routers should be allocated more cache banks. Furthermore, the scheduling algorithm needs to take into consideration the needs of the processes. Data intensive applications need to be assigned to processors that have access to larger local cache banks.

- **Dynamic Reconfigurable Heterogeneous Architecture:** In this architecture, some cache banks do not belong to any one router. Instead, they have connections to four routers and can be reassigned to one of the four neighboring routers based on the needs of the applications running on it.

To show that a heterogeneous distribution can outperform the homogeneous distribution, we simulated synthetic benchmarks having different access patterns. The architecture used is the hybrid bus/network architecture. For the heterogeneous distribution, cache banks were first distributed equally among the routers. Then they were reassigned to routers based on access patterns. The end result was a heterogeneous distribution of cache blocks.

The simulation results of a 6x6 network is shown in Figure 5.1. In all cases, the heterogeneous distribution outperformed the homogeneous distribution.
5.2 Static Distribution

In traditional designs, caches are distributed equally among the routers. However, in the previous section we showed that the heterogeneous distribution of caches is better than the homogeneous distribution.

5.2.1 Heterogeneous Cache Distribution

PARSEC benchmarks [15] provide a snapshot of emerging workloads. Looking at the different applications included in the suite, it becomes clear that data size of the different applications differ significantly from one another. Therefore, one size fits all cache distribution may not be the best architecture for multicore systems. Heterogeneous distribution of cache blocks across the routers can cater to the needs of the varying applications better than the homogeneous model. A generic distribution of large and small cache block is needed to accommodate workloads with varying degrees of data intensity.
5.2.2 Proposed Architecture

The placement of the larger cache blocks plays a pivotal role in determining overall cache access latency. Extensive experimentation led to our proposed heterogeneous cache distribution for 6x6 mesh size is shown in Figure 5.2. Three sizes of cache banks large(B), medium(M) and small(S) are distributed along the network. Since large cache blocks can cause congestion and small blocks can cause network hops, the small and large cache block are alternated to balance the two. If a process is assigned to a small cache bank, and during runtime it requires more memory, a larger cache bank exists at a distance of at most one hop. Another reason for the alternating sizes is to balance the network congestion around the larger cache blocks. The large cache banks only exist in the center for greater accessibility.
5.2.3 Experimental Results

We developed a C++ tool to create the synthetic benchmarks and simulate the NoC architecture.

The generated synthetic benchmarks have different characteristics. The benchmarks emulate multiple workloads working on different data sets. Several variables are used to create the different characteristics as follows:

- **Number of requests:** This variable represents the number of requests generated by each processor. The processors generate requests at different intervals based on data intensity until they reach the number of requests.

- **Rate of requests:** Represents the number of data requests in a given time interval. If the rate is 10%, then a data request is generated for every 10 instructions by the processor.

- **Data intensity of the Process:** The parameter represents the size of the address space from which processors access data. Processors can generate any address request bound by the size of this parameter.

Network sizes of 4x4, 5x5 and 6x6 were simulated with synthetic benchmarks. The benchmarks were simulated both having homogeneous distribution and heterogeneous distribution of cache banks. Figure 5.3 shows the average cache access time between the heterogeneous and homogeneous architectures. The sizes of the B, M and S were set to 6MB, 4MB and 2MB respectively. The figure shows that the average cache access time is reduced in all cases by up to 20.3%.

5.3 Dynamic Distribution

In this section, the proposed static heterogeneous architecture is extended to include a reconfigurable aspect where the number of cache blocks assigned to each router is not static and can
be changed before the execution of processes.

5.3.1 Proposed Architecture

The proposed architecture is shown in Figure 5.4. The reconfiguration of cache blocks to routers is done through the green switches shown in the figure. The cache blocks shown in the figure are only the ones that can be reconfigured. In addition to these cache blocks, the routers have fixed cache blocks assigned to them at design time. The switches can be turned on or off based on the needs of the applications running on a router. When an application is known to be data intensive, more cache is assigned to the router before the runtime of the application.

The cache block still has one addressing space. The routers need to keep track of where data is located. For every reconfiguration of caches to routers, a control signal is sent to the routers to update their mapping list of caches to routers.

5.3.2 Cache Reassignment Policy

If processes belonging to one application span multiple routers, then different placements of data lines and hence assignment of caches to routers can significantly impact cache access time.
Therefore, a method is needed to determine the best placement of cache blocks.

Different processes accessing shared data place a force on that data. The force exerted by one processor is the number of data lines and the frequency of data requests on the cache bank. The optimal location is somewhere in the middle of the different forces with a net force of zero. Therefore, the Force Directed Heuristic [65] is ideal for this computation.

The cache banks are moved to their zero sum location and locked for the current iteration. The zero sum location can be computed by the following equations:

\[ x_i = \frac{\sum_r w_{ir} \cdot x_r}{\sum_r w_{ir}} \]  
\[ y_i = \frac{\sum_r w_{ir} \cdot y_r}{\sum_r w_{ir}} \]

Where:
\[ r = \text{Number of Routers} \]
\[ W_{ir} = \text{No. of requests from router } r \text{ to cache } i \]
\[ X_r = X \text{ co - ordinate of router } r \]
\[ Y_r = Y \text{ co - ordinate of router } r \]

Since the assignment of all cache blocks to one router is not desired, a maximum number of cache banks is allowed for each router. If the number of cache banks in the zero sum location of the cache block is less than the maximum, then the cache block is assigned to that router. Otherwise, the cache block is assigned to the router and a free cache block belonging to that router is selected next for movement. If all the cache blocks are locked, then the cache block is assigned to the closest router that has available slots. The process is repeated until all the cache blocks are reassigned to their ideal location. The process is repeated for a few iterations until no more improvements can be made.

### 5.3.3 Experimental Results

The proposed method was tested using both the synthetic benchmarks and the PARSEC benchmarks. The simulation platform is similar to the ones mentioned in previous chapters.

The results of the synthetic benchmark are shown in Figure 5.5. There are 16 synthetic benchmarks with different parameters. The heterogeneous cache access time shown is the best result obtained after multiple iterations of FDH. For most cases, the heterogeneous configuration outperforms that of the homogeneous distribution of caches to routers.

The results hold true for the PARSEC benchmarks. Figures 5.6 and 5.7 show 16 and 128 cores respectively. For both cases, the heterogeneous configuration is better than the homogeneous configuration by up to 61%.
Figure 5.5: Average Latencies of Homogeneous vs Best Heterogeneous Configuration 6x6 Network

Figure 5.6: Average Latencies of Homogeneous vs Best Heterogeneous Configuration 16 Cores
5.4 Conclusion

In this chapter we showed that the heterogeneous distribution of cache blocks to routers outperforms the standard homogeneous distribution. Based on initial experimentation, we proposed two variations of the NoC architecture. The first is the static heterogeneous architecture where the size of cache blocks is varied across the network at design time. We showed the ideal distribution of large, medium and small cache blocks. In the second architecture, we added a layer of reconfigurability to the architecture where a subset of cache blocks can be assigned to neighboring routers based on the needs of the applications running on the processors. In the first case, we obtained gains in average cache access time of up to 20%. In the second case the gain was significantly higher reaching as much as 61%.
Chapter 6

Alternate Methods to Reduce Cache Access Time

Data migration can play a key role in reducing cache access time. During data migration, the mapping of physical addresses to cache addresses can be changed. Therefore, a method is needed to keep track of data locations in the network. In this chapter, we will discuss a new data migration algorithm and then introduce different address lookup methodologies.

6.1 Data Migration

6.1.1 Introduction

In any processor design, reduction of execution time is always considered a priority. With the saturation of the speed of processing cores, alternate methods have been proposed to reduce overall execution time. Multicore processors have become mainstream and the most widely used to speed up execution time of applications. But multicore design is not without its challenges. The

---

1This section contains collaborative work with Jon Nafziger [57]. Further details are included in [58]
most widely used architecture for multicore design is the NoC as mentioned in previous chapters. The most challenging task in multicore design is dealing with the Non Uniform Cache Access nature of the L2 cache along with speeding up data transfers between L2 cache banks and individual processing cores.

In previous chapters we proposed fundamental changes in the architecture design to meet this challenge. While those methods are highly effective, they include a hardware cost overhead that might not be feasible to implement in the near future. Other methods to reduce data access time with less hardware overhead might be desirable. One such method is the idea of data migration.

In previous generations of chip multiprocessors, the L2 cache was a single unit with either a single or double read and write ports. With the increase in the number of processing cores requesting data, this method is no longer feasible. Any proposed architecture in the near future will have some aspects of Network on Chip. Since data in the NoC scheme is distributed among the different routers, accessing data from different locations has different retrieval time. The associativity mapping of a single L2 cache system can still apply. If data has a specified location in the distributed cache system based on its address, then it is called static NUCA. Not much can be done in this scheme since data cannot be migrated to a different location. The alternative is the Dynamic NUCA where data can be stored in multiple locations. The D-NUCA is more flexible, but it may lead to increase in cache access time since locating the data might be time consuming.

In this chapter we will present a solution for the Two Dimensional Post Office Problem called Directional Migration. The proposed method takes advantage of the characteristics of the D-NUCA to migrate data. The proposed method will be tested on both NoC and hybrid models to illustrate the benefits of using this system. Furthermore, an improvement is made on the Directional Migration to include prediction based data migration where neighboring data are migrated along with the original data to take advantage of spatial locality. The proposed methods are suboptimal since the best placement method requires too much overhead. We will show that the suboptimal
solutions are very close to the best placement ones by comparing the results with the Best Placement Migration Algorithm method.

### 6.1.2 Motivation

In multicore design where cache accesses are non-uniform, initial placement of the data and the processing core plays a major role in determining overall execution time. If the data is placed close to the processing core accessing it, then the overall execution time is greatly reduced due to spatial locality. A set of data is accessed a multitude of times during the execution of the process.

Many suggestions have been proposed to place data in its ideal location. Cho and Lin [26] proposed using OS level cache allocation to place data closer to the processing core accessing it. On the other hand, Chen et al. [24] propose mapping threads to processors at compile time to ensure threads belonging to the same process are placed close to one another. The data that these threads access is then placed on a router near these processing elements. Both methods are effective in reducing cache access time if the OS has previous knowledge of the behavior of the processes running on the system. A more robust system is needed that can alter the location of pages based on the accesses observed during the runtime of the application.

With applications becoming more multi-threaded, data sharing becomes an important characteristic that needs to be considered by any design architect. Placement of data plays a vital role in overall access time. Therefore, algorithms that were based on the assumption that one or two cores access data at the same time will no longer be relevant Cache Data Migration. Instead the placement of data has become a weighted average problem where cores from different locations put a force on a data line and therefore that data line is moved towards the center of the weighted sum of those forces. This idea was first introduced by Kandemir et al. [40]. Although this approach is capable of finding the ideal placement of any data line, this method requires the system to keep
track of every data access on each data line by each processing core. The overhead associated with it is large and will become infeasible as the number of cores increases.

To solve the Two-Dimensional Post Office problem without garnering a large overhead, we propose a Data Migration technique that implements a step wise migration. The main idea is to still migrate data without the need to keep a fine grained accounting of all data accesses. The details of the proposed method will be explained in the following section. We further expand on the idea to implement a prediction based data migration that takes into consideration spacial locality.

### 6.1.3 Data Migration

**Directional Migration**

The main priority of the method proposed for data migration in this section is scalability. Any proposed method should still be relevant and usable a few years from now when the number of cores on a chip reaches the scale of hundreds. Therefore, instead of keeping track of data accesses
from every core to every line in the L2 cache, only the general direction of the request is tracked. In any mesh system, a router is connected to at most four other routers. The request for data can come from any one of those four general directions. Instead of keeping track of which core requested the data, we keep track of which direction the request came from. The four directions are dubbed as North, East, West, and South. Figures 6.3 and 6.2 show which routers are considered as North, East, West, or South with respect to router $R_{10}$. Each data line therefore needs to keep track of these four values. Furthermore, the number of counters needed to keep track of data accesses can further be reduced if only the difference in the two directions is taken into consideration. Meaning, the two counters will represent the difference of data access requests in the North/South direction and the East/West direction. The variables used to represent these two counters are $AC_{NS}$ and $AC_{EW}$.

Since one counter is used in each of the directions North/South and East/West, a signed value is used to determine the direction where the pressure is coming from. In this case negative values mean the force is coming from the North or the East direction and a positive value means it’s coming from the West or the South direction.

The architecture that is targeted by this algorithm is a 2D-Mesh architecture. The distance between any two routers is easily calculated by measuring the Manhattan distance between them. The Manhattan distance of two routers is composed of two segments, the horizontal direction and
Chapter 6: Alternate Methods to Reduce Cache Access Time

Figure 6.4: Access Counter Distances
the vertical direction. Knowing the source and destination routers, both horizontal and vertical segments can be computed separately since the position of each router is known beforehand. As previously described, the two counters that keep track of the pressure on any data line coming from the different routers are signed numbers. Therefore, once the two vertical and horizontal segments are computed, it is imperative to determine the direction from which it came. The horizontal line needs to be identified as West or East, and the vertical line needs to be identified as North or South. If the request is coming from the Northern or Eastern sides, the value of the Manhattan distance is subtracted from the counters $AC_{NS}$ or $AC_{EW}$, and if it is coming from South or West, it is added to said counters.

Once the values of the counters are computed after each data access, they are then compared to the migration threshold. The migration threshold is hard coded in the system. The value of the migration threshold is based on statistics from simulation analysis of a wide range of benchmarks. The threshold value will not be ideal for all applications, but the value is chosen as it best fits the majority of applications. Once the counter values are compared with the threshold migration, a migration is triggered if they exceed this threshold. Since the value of each counter is signed, the absolute value of the counter is compared to the threshold value. If a migration is necessary, then the sign is checked. If the sign of the counter is positive, then the migration will occur in the West or South direction based on the counter being tested $AC_{EW}$ or $AC_{NS}$. If the value is negative, then the migration occurs in the East or North direction.

Since migrated data lines need to be stored in the destination cache block, a data line in the target cache needs to be moved to accommodate the migrated data line. To determine which data line will be the victim, Least Recently Used methodology is used. The data line in the source router is replaced with the LRU data line in the destination router.
Active Neighbor Migration

The inherent characteristics of temporal and spacial locality of L2 caches can be manipulated to speed up cache accesses. The first section in this chapter Directional Migration dealt with temporal locality. The main purpose of migrating data is the belief that data that has been requested previously will be requested again in the near future. But that method does not take into consideration spacial locality.

Spacial locality is the idea that if a data line is requested, data lines that are close to the previous one will be requested as well. Therefore, it is intuitively clear that when data line \( i \) is migrated, then migrating data lines \( i - 1 \) and \( i + 1 \) might be beneficial. Active Neighbor Migration aims at implementing an algorithm that takes advantage of spacial locality and migrates data lines in a structured method.

Active Neighbor Migration is an add-on to Directional Migration. Once a data line is selected for migration, its immediate neighbors are considered for migration as well. The counters of the neighboring data lines are checked to see if they follow a similar pattern as the migrating data line. This is done so that data lines are not migrated in vain. The counters of the data lines adjacent to the migrating data line will have a value that is less than that of the migration threshold, otherwise those lines would have been migrated in previous iterations. Therefore, it is necessary to determine the threshold value at which the neighbor lines should be migrated. The formula used here is

\[
\left( \frac{\text{MigrationThreshold}}{NR} \right) \leq AC_{Dir}
\]

The value of \( NR \) plays an important role in determining the effectiveness of Active Neighbor Migration. If the value of \( NR \) is too large, then the threshold value will be too small which may trigger unnecessary migrations. If the value is too small, then the neighboring data lines will not be migrated early enough to have an effect at reducing the overall access time.

Once the neighboring lines \( i + 1 \) and \( i - 1 \) are migrated, the process checks for subsequent
Chapter 6: Alternate Methods to Reduce Cache Access Time

88

Figure 6.5: Directional Migration Cache Structure

lines $i - 2$ and $i + 2$ to see if they are eligible for a migration as well. This process is continued until the data lines down the line do not meet the requirements for data migration.

**Hardware Requirements**

In order to implement Directional Migration and Active Neighbor Migration, additional hardware components are needed to keep track of data accesses by the different routers. The simplest way to implement the counters is an addition to the cache block themselves. The number of bits required for each counter is $\log_2$ of the Migration threshold. An additional bit is needed as a flag to signal that a data line is being migrated. So a total of $1 + 2 \times \log_2(MigrationThreshold)$ are needed for each data line. If each data line is 128 bytes and the Migration Threshold is chosen as $256 = 8$ bits, then the cache size is increased by 1.66%. This addition is not significant when compared to the benefits of implementing data migration. Figure 6.5 shows the hardware implementation of the cache.

### 6.1.4 Experimental Setup

To verify the benefits of implementing Directional Migration and Active Neighbor Migration, we simulated the results in Simics using the PARSEC benchmarks. Table 6.1 shows the parameters used to setup the architecture.
Chapter 6: Alternate Methods to Reduce Cache Access Time

<table>
<thead>
<tr>
<th>Simics Parameter</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>Linux 2.16.1</td>
</tr>
<tr>
<td>Number of Cores</td>
<td>128</td>
</tr>
<tr>
<td>L2 Size (KBytes)</td>
<td>3072 * # Cores</td>
</tr>
<tr>
<td>L2 Line Size (Bytes)</td>
<td>128</td>
</tr>
<tr>
<td>L2 Associativity</td>
<td>8-Way</td>
</tr>
<tr>
<td>L2 Replacement Policy</td>
<td>LRU</td>
</tr>
<tr>
<td>L1 Size (KBytes)</td>
<td>64</td>
</tr>
<tr>
<td>L1 Line Size (Bytes)</td>
<td>128</td>
</tr>
<tr>
<td>L1 Associativity</td>
<td>2-Way</td>
</tr>
<tr>
<td>L1 Replacement Policy</td>
<td>LRU</td>
</tr>
<tr>
<td>PARSEC Benchmarks</td>
<td>Blackscholes, Bodytrack, Facesim, Ferret, Fluidanimate</td>
</tr>
</tbody>
</table>

Table 6.1: Simics Simulator Setup

<table>
<thead>
<tr>
<th>NoC Simulator Parameter</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Migration Threshold</td>
<td>256</td>
</tr>
<tr>
<td>Interconnect Speed (Msgs per Cycle)</td>
<td>1</td>
</tr>
<tr>
<td>Router/Cache Nodes</td>
<td>16</td>
</tr>
<tr>
<td>Core Density (Cores per Router)</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 6.2: NoC Simulator Setup

Same as before, post processing was done to determine the number hops in the NoC architecture as shown in Table 6.2. We compared the results to the best placement migration for the Two-Dimensional Post Office problem in [40]. The best placement method maintains a counter for each core sharing each data line. This results in an increase of 12.5% in cache size while maintaining a migration threshold of 256. The difference between the two methods is that the Best Placement Migration migrates the data line to the optimal location based on data accesses up to the point the counter hits the migration threshold, while the Directional Migration migrates the data line by one hop only. Although the Best Placement Migration places the data line in its ideal location, it is more costly in hardware to implement and computationally intensive.
Chapter 6: Alternate Methods to Reduce Cache Access Time

![Table 6.3: Average Migration Percent Difference to Best Placement](image)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Data Request Time</th>
<th>Data Request Distance</th>
<th>Local Cache Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>DM-ANM0</td>
<td>78.25%</td>
<td>13.27%</td>
<td>15.61%</td>
</tr>
<tr>
<td>DM-ANM2</td>
<td>71.24%</td>
<td>13.27%</td>
<td>15.43%</td>
</tr>
<tr>
<td>DM-ANM4</td>
<td>70.05%</td>
<td>13.65%</td>
<td>15.79%</td>
</tr>
<tr>
<td>DM-ANM8</td>
<td>73.45%</td>
<td>13.45%</td>
<td>15.72%</td>
</tr>
</tbody>
</table>

6.1.5 Results

As mentioned in 6.1.3, the value of the Threshold Migration plays a decisive role in determining the benefits of implementing Data Migration. To determine the optimal value to use, we varied the threshold migration from 4 to 32768 as shown in Figure 6.6. As expected, as the migration threshold increases, so does the number of hops. This is due to the fact that as the threshold migration increases, data lines are not migrated in a timely manner to be effective.

Once it was determined that a Migration Threshold value of 256 was sufficient, we setup an experiment keeping that number constant and varying the value of NR for Active Neighbor Migration. The results were compared to both the Best Placement Migration and Directional Migration with Active Neighbor Migration. Directional Migration without Active Neighbor Migration is the one with DM – ANM set to 0. Figure 6.7 shows that network traffic is reduced.

As expected, the Best Placement Migration outperforms our proposed Directional Migration. At its best the difference is 17%. In the worst case scenario, the difference is 70%. This shows that the Directional Migration method is not always beneficial and is highly dependent on data access patterns. As seen, the value of NR plays an important role as well. For some benchmarks, a NR value of 2 is more beneficial while for others a NR value of 4 is required. A larger set of benchmarks can be analyzed to determine the best value for NR that is suitable for most benchmarks.

Table 6.3 shows the percentage difference between average migration and optimal values.

Figure 6.8 shows the distance L2 data requests are traversed between the L2 cache and
Chapter 6: Alternate Methods to Reduce Cache Access Time

Figure 6.6: Percent Difference in Cycles with Variable Threshold Values

Figure 6.7: Percent Difference in Cycles to Retrieve Data
Chapter 6: Alternate Methods to Reduce Cache Access Time

Figure 6.8: Percent Difference in Distance to Retrieve Data

Figure 6.9: Percent Difference in Local L2 Accesses
the requesting processing core. As seen, the distance to access data is increased compared to the optimal value. Again this is to be expected since the best placement migration method places the data in its optimal location from the first migration, while the DirectionalMigration migrates data one hop at a time. The difference though is 10-17% which is an acceptable increase compared to the hardware savings of DirectionalMigration over BestPlacementMigration.

Since the experimental setup is a hybrid system, increasing the local L2 accesses is desired. Figure 6.9 shows the percentage difference in accessing local L2 cache in the two methods. As seen, BestPlacementMigration performs better than DirectionalMigration due to the same reasons mentioned above. The difference between the two methods fall in the range of 14-18%.

6.1.6 Conclusion and Future Work

Reducing L2 cache access time is a priority for any multicore architecture design. Data migration can play a key role in reducing data migration. The hardware costs associated with BestPlacementMigration is large and may not be a good design alternative. But DirectionalMigration uses minimal hardware overhead less than 87% of that of BestPlacementMigration. DataMigration along with ActiveNeighborMigration are capable of reducing data cache access time to within 16% of the best placement method.

6.2 Address Lookup Methodologies

6.2.1 Introduction and Motivation

In standard processors, the last level cache is a single memory bank with one or more ports. To locate data in the cache, physical address information can be used for data mapping. In NoC designs, the same principle applies for most cases. Physical address can be used to find the location in the cache, then that value is used to locate the router that contains the data location.
However, data migration violates the principle of direct mapping. When data is moved from one cache bank to another, direct mapping cannot be used to locate the data line in the cache. Therefore, alternate methods are needed to maintain data location.

Broadcast method can be used to locate data. For every data transaction, a processor can broadcast a message on the network to determine the location of a particular data line. This method has no area overhead, but can saturate the network. Another method is for every processor to keep track of data locations in a local lookup table. This requires large area overhead and is not scalable. Global lookup table is also an option where each processor sends a request to the global table to determine address location.

To find alternate solutions, Lira et al. proposed HK-NUCA [51]. In HK-NUCA, the L2 cache is k-way associative and each set is located in a different cache bank. When a data is requested, it is saved in one of the memory cache banks. During migration, data can be relocated to a different bank and mapped to its associative location. In the proposed method the associative degree is 16, 8 local and 8 central. When a processor requests data, it needs to determine the location of that data. It sends individual requests to the 8 local ones first and if not found, it sends requests to the 8 central ones. If not found, the request is then forwarded to main memory.

Feihui et al. [50] propose a completely new architecture. They propose a 3D L2 cache architecture to reduce cache access time. They show that a 3D design performs much better than a 2D mesh design with data migration.

In this section, we propose a new distributed address lookup methodology. Instead of one global table, multiple tables exist across the network to reduce network congestion and overall cache access time without too much area overhead.
6.2.2 Proposed Distributed Address Lookup Methodology

In the proposed method, instead of having one global lookup table as shown in figure 6.10, we propose having distributed tables across the network. In the global table scheme, for every data access, the requesting processor sends a request to the global table to find the target router given a physical address. This can cause two major issues. The first is that for every control request generated, the request is queued along with control or data requests from other routers. Therefore, these control requests cause additional traffic on the network and can delay the cache access time of all data requests. The second issue is the additional delay of the requesting processor. When data is migrated, the processor has to first send a control signal to the global table, then once it receives the target router id, it can send the data request on the network. If the global table is located too far from the requesting processor, the delay caused by the control signal can be significant.

Another option is for each individual router to maintain a local copy of all migrated data. The advantage of this method is that no control signals are needed to determine the data location. When a processing core sends a request to a remote router, the source router first checks whether that data line exists in the list of migrated lines. Based on the result, it can set the target router when packetizing the request. The drawback of this model is the amount of hardware overhead needed. Data lines are shared across the network, this means multiple duplicates of the same information can exist among the routers.

Therefore, we propose a distributed address lookup methodology, where instead of having one global table for all migrated data, we have multiple tables across the network that serve a subset of the routers. This method is a compromise of the global and individual options. Figure 6.11 shows one such example. In a 4x4 network, we can have four local tables, each serving four routers. In this scenario, routers 10, 11, 14 and 15 send their request to the table connected to router 15. Therefore, the distance travelled by the control signal is shorter compared to the global router. The disadvantage of having distributed tables is the duplicate values that may exist across the tables.
Chapter 6: Alternate Methods to Reduce Cache Access Time

Figure 6.10: Global Lookup Table

Figure 6.11: Distributed Lookup Tables
6.2.3 Experimental Results

In order to test our hypothesis, we simulated the PARSEC benchmarks mentioned in previous sections. The target architecture is the hybrid Bus/Network architecture where each bus has eight processors connected to it. We simulated the benchmarks for three different values of the threshold 8, 64 and 256. When the threshold value is lower, data is migrated more often and therefore, more control signals are generated on the network to locate migrated data.

Figures 6.13, 6.15 and 6.17 represent the percentage difference in cache access time between the global address lookup and the distributed address lookup methods when compared to the individual lookup method. As seen, the average cache access time increases in both cases since control signals are needed to determine address location. However, comparing the global and distributed method, it becomes clear that the time increase of the distributed method is much less than that of the global method. The reason is that in the global method, all the control signals are directed at one router that creates long queues and therefore large delays. In the distributed method, the congestion is distributed instead of being centralized.

Figures 6.12, 6.14 and 6.16 show the percentage increase in area when comparing the distributed and individual models to that of the global lookup method. For the distributed and the individual methods, the size of the table is the cumulative size of all the tables across the network. In the case of the distributed model, there are 4 tables across the network. In the individual model 16 tables are needed, one for each router. The size of the tables are more than that of the global model because of duplicate values for target addresses found across the different tables. In some cases, the cumulative size of the tables is more than double that of the global tables. Looking at the figures it becomes clear that the size of the individual tables is much more than that of the distributed model. This is because all the cores in the network are sharing data and therefore, multiple duplicates exist across the tables.

A trade-off needs to be made between cache access time reduction and increase in area.
If the area of the overall die is an issue, then a global table or the distributed model can be sufficient to locate the migrated data. If there exists enough space on the chip, then multiple tables can be set in place to implement either the distributed method or the individual method to reduce overall processing time.

The size of the tables is set at design time. Therefore, if the distributed method is to be implemented, the cumulative size of the tables need to be more than that of the global table and can be set based on the characteristics of target applications.

### 6.2.4 Conclusion

In this section, we showed that a distributed table can reduce overall access time compared to the global model and reduce area overhead compared to the individual model. In some cases, the cache access time was reduced by as much as 85%. However, the area overhead is larger compared to the global lookup table. Having multiple applications running on a system, it is expected that processes that share data are assigned to routers close to one another. If all processes sharing data
Chapter 6: Alternate Methods to Reduce Cache Access Time

Figure 6.13: Average Cache Access Time Gain - Threshold 8

Figure 6.14: Size of Lookup Table - Threshold 64
Figure 6.15: Average Cache Access Time Gain - Threshold 64

Figure 6.16: Size of Lookup Table - Threshold 256
are assigned to processors that share the address lookup table, then duplicates need not exist across the network. Therefore, with a good thread assignment policy, the cumulative size of the distributed tables can be maintained the same as that of a global lookup table.
Chapter 7

A Hybrid Cache Coherency Protocol for Hybrid and NoC Multicore Architectures

With multicore architectures becoming mainstream, multithreaded applications are following suite to take advantage of the many cores found on a single chip. Having multithreaded applications introduce its own set of challenges, mainly shared data among multiple threads. As processing cores have their own private L1 caches, maintaining cache coherency among the cores is a necessity and can be achieved by utilizing either hardware or software techniques. Broadcast and directory protocol can be used but at a cost of time or area overhead. In this chapter, we introduce two cache coherency techniques; the first is unique to hybrid Bus/Network architectures where the area overhead can be reduced by 87% for 128 core chip over the directory protocol and cache access time is reduced by an average of 44% compared to broadcast method depending on the characteristics of the application. The second method is a hybrid protocol for NoC multicore architectures where data overhead is reduced by 89% compared to the directory overhead having
comparable cache access times.

7.1 Introduction and Motivation

In the traditional Network on Chip multicore architecture [45], each processing core has a private L1 cache and the L2 cache is shared among all the processors on the packet switched network. This method works great if each processing core accesses and more importantly updates its own data set. Unfortunately, the aim behind the introduction of multicore architectures is to implement applications in parallel. This means multiple cores access a common data set and more than one core can have a copy of the data set in their private L1 cache. Any core can read or update the data found in its own private cache. The problem arises if multiple cores try to update the same data set. In the write through policy, once the data is updated in the L1 cache, it is also updated in the L2 cache. A race condition can occur if more than one core tries to update the same data line at the same time. Furthermore, if a data line is updated in one cache block, then it means that the data line found in other cache blocks should be updated to assure that all the processing cores have a copy of the correct data. This problem is defined as the cache coherency problem. Cache coherency protocols exist to coordinate the issue of multiple copies of data found on the network.

One such protocol is the Directory Protocol [21] [69] [48]. In the directory protocol scheme, each data line is responsible of keeping track of all the processing cores accessing that data line. The simplest method to implement that is to add $p$ bits to each data line and set that bit to 0 or 1 if that processing core has a copy of that data line. When a data line receives a write request, it checks to see if a copy of that data line exists anywhere else on the network. If it exists, then a control signal is sent to each processing core that has a copy of that data line in their L1 cache. The processing cores in turn invalidate their copy of the data and the next time they need to access that shared line, an L1 cache miss occurs and the processor requests a new copy of the updated data line.
Directory protocol sends control messages to only those processors that share a data line. This method does not introduce unnecessary traffic on the network. However, for a 128 byte data line and 128 processing cores, each data line needs 128 bits to represent each processing core to keep track of all the cores accessing that cache line. To implement directory based protocol, the cache size is increased by 12.5% to keep track of all the processing cores. For 256 cores, 25% overhead is needed for bookkeeping. Looking at the numbers, it becomes evident that although efficient, directory protocol cannot scale for large number of processing cores.

The other option is implementing the Broadcast Protocol [23]. In the broadcast protocol, when a data line is updated in the shared L2 cache, a broadcast signal is sent out on the network to all the processing cores. The processing cores on the network check to see if they have a copy of that data line. Then any processor that shares that data line invalidates its copy in its L1 cache. The advantage of the broadcast method is that it has no area overhead. The only overhead is 1 bit per data line to determine if that data line is shared. The reason for that shared bit is to avoid sending unnecessary control signals for every write request if a data line is not shared. However, the broadcast protocol has two main disadvantages. The first is that if a large percentage of data lines are shared among processing cores, then the number of broadcast messages sent can overwhelm the network and introduce large delays. The second issue is that a broadcast message is sent to all the processing cores even if only a handful of cores share that data line introducing unnecessary traffic. Therefore, the broadcast method is scalable in terms of hardware overhead at a cost of network congestion.

In order to reduce the amount of overhead needed for cache coherency, researchers have looked at many alternatives. Jerger proposes SigNet [37], an architecture that uses coarse vector directories, but at the same time tries to reduce the number of requests sent on the network before the requests reach their destination. If a cache line need not be invalidated on a certain path, the control signal is halted midstream. Meanwhile Kurian et al. [46] have designed a new multicore
architecture that takes advantage of optical interconnect. They implemented ACKwise, a cache coherency protocol that utilizes the characteristics of their proposed architecture to create a fast and efficient broadcast network. On the other hand, Shield et al. [68] try to reduce cache coherency overhead and increase data accesses by applying asymmetric cache coherency protocols for non-uniform workloads.

From these examples, it becomes evident that cache coherency is highly dependent not only on the data patterns, but on the architecture as well. Taking advantage of the different architectures to reduce the overhead introduced due to cache coherency is a must. That overhead can be either hardware overhead or latency overhead due to the control signals that are sent to invalidate cache lines found in different processing cores. We take advantage of the inherent characteristics of the hybrid architecture introduced in [28] [11], in order reduce the number of control signals that are sent on the network to maintain cache coherency. Furthermore, we extend the idea of a hybrid cache coherency protocol to NoC architectures and propose a hybrid cache coherency protocol that tries to find a median between directory and broadcast protocols.

### 7.2 Hybrid Cache Coherency Protocol for Hybrid Architectures

As seen in chapter 2 hybrid architecture of bus network configuration is a viable method to reduce overall cache access time. Furthermore, the architecture can scale efficiently as the number of processing cores increase on a single chip. With the advance of multicore architecture, software should adapt to take advantage of the many cores found on a chip. As software become multithreaded, data sharing among threads will become more paramount. Hybrid and the hybrid/reconfigurable architectures try to take advantage of the characteristics of multithreaded systems and assign threads belonging to one process to the cores that are connected on a single bus and on a single router. The cores that are connected on a single bus are more likely to share data than
cores that are found on different buses connected to different routers. Figure 7.1 shows the hybrid bus/network architecture where multiple cores are connected to a bus that is in turn connected to a router.

### 7.2.1 Hybrid Protocol

We propose that the cores that are connected on a single bus implement the bus snooping protocol [38]. Bus snooping has been used successfully in multiprocessor environments. In the bus snooping protocol, the cores that are connected on the bus will monitor the data read or write requests transmitted on the bus. If a processing core sends a write request to a data line, then the other processors on that bus will check whether they have a copy of that data line. If one does, then it will invalidate its own copy of the data. It is important to note that the bus is connected to the router, so any transmission on the bus is not in the form of packets and can be easily decoded by the processors on the bus.

Since processing cores connected to different routers can share data, a cache coherency protocol still needs to be implemented at the network level. Cache coherency is maintained using the MESI protocol [61]. Directory protocol is used on the network, but instead of keeping track of the processing cores that access a data line, we keep track of the routers requesting that data line. That request can come from one or more of the processing cores connected to the router, but only the router identity is saved in the directory protocol.

If the data line is 128 bytes and the number of processors are 128, we can connect 8 processing cores to a bus and hence, the number of routers needed to support all the buses is 16. In the cache line, 16 additional bits are needed to maintain the directory protocol. That is a 2 Byte increase in each cache data line. The percentage overhead is reduced from 13% to 1%, a major reduction. For 256 processing cores, the overhead increases to 3%. Although the overhead significantly increases as the number of processing cores increase, it increases at a much lower rate.
compared to directory protocol. Figure 7.2 shows the percentage overhead for the directory protocol versus the hybrid protocol. Table 7.1 shows the number of bytes needed to keep track of shared data for directory and hybrid protocols.

Once a data line is updated in the L2 cache, if processing cores on different routers share that data, then a control signal is sent to each router that has a processing core with that data line in its private L1 cache. Once the router receives this message, it sends a broadcast message on the bus so that the processors that have a copy of that data line can invalidate their copy of the data. Figure 7.3 shows the design of the proposed cache coherency protocol. As seen, the processors have a cache snooping logic that monitor the requests sent on the bus. The L2 cache lines in the routers have DP: Directory Protocol portion, where they keep track of the routers that have a copy of that data line. The routers also have a CC logic which is the cache coherency logic that generates the control signals. There are two types of control signals, one that is directed to other routers that have a copy of the data line. The second is a control signal that is broadcast on the bus so that cores local to that router can invalidate their copy of the data.
Figure 7.2: Directory vs Hybrid Protocol Area Overhead

Figure 7.3: Hybrid Architecture with Hybrid Cache Coherency Protocol

Table 7.1: Area Overhead of Directory and Hybrid Protocol for Hybrid Architectures

<table>
<thead>
<tr>
<th># of Cores</th>
<th>Directory Protocol</th>
<th>Hybrid Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>256</td>
<td>32</td>
<td>4</td>
</tr>
<tr>
<td>512</td>
<td>64</td>
<td>8</td>
</tr>
<tr>
<td>1024</td>
<td>128</td>
<td>16</td>
</tr>
</tbody>
</table>
7.2.2 Experimental Results

To compare the different protocols, enumerating the number of control signals does not give us accurate results. The control signals sent for the directory protocol and the hybrid protocol can be compared since both deal with point to point communication. However, broadcast control signals send a message over the entire network, therefore, it cannot be directly compared with hybrid or directory protocols. To determine the impact of using the different protocols, we simulated the architecture using the different protocols and compared the average cache access time of the processing cores. The cache access time is calculated as the number of clock cycles lapsed between the time a processor sends a request to an L2 cache block until the time it receives the request. The average cache access time is equal to the total wait time for cache accesses divided by the total requests generated by all the processors.

The time it takes to receive a cache request depends on many factors. Time spent waiting for the bus grant, time spent in the router queues, and the distance from the processor to the destination router. The different coherency protocols effect the traffic on the network, therefore, a request can remain in the queue for longer periods based on network traffic.

To verify the benefits of using the coherency protocol, we simulated the architecture using both synthetic benchmarks and PARSEC [15]. However, PARSEC benchmarks run as a single application. To simulate the characteristics of multiple applications running on the system at the same time, we simulated synthetic benchmarks with different characteristics.

We used the Simics simulator for the purpose of this experiment [52]. The simulation variables used for the experimental results are shown in Table 7.2.

The output of the Simics simulator was a log of all cache transactions. Once the log was created, we performed post processing similar to [35] [59] to determine the number of control signals for the different cache coherency protocols.

The number of processing cores are 128. We assigned 8 cores to a single bus, and the bus
Table 7.2: Experimental Setup

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cores</td>
<td>64,128</td>
</tr>
<tr>
<td>L2 Size in KB</td>
<td>3072*number of processors</td>
</tr>
<tr>
<td>L2 Line Size</td>
<td>128B</td>
</tr>
<tr>
<td>L2 # of Cache Lines</td>
<td>L2 Size * 1024 * L2 Line Size</td>
</tr>
<tr>
<td>L2 Associativity</td>
<td>8-way</td>
</tr>
<tr>
<td>L2 Replacement Policy</td>
<td>LRU</td>
</tr>
<tr>
<td>L1 Size in KB</td>
<td>64</td>
</tr>
<tr>
<td>L1 Line Size</td>
<td>128B</td>
</tr>
<tr>
<td>L1 # of Cache Lines</td>
<td>L1 Size * 1024 * L1 Line Size</td>
</tr>
<tr>
<td>L1 Associativity</td>
<td>2-way</td>
</tr>
<tr>
<td>L1 Replacement Policy</td>
<td>LRU</td>
</tr>
<tr>
<td>Benchmark</td>
<td>PARSEC</td>
</tr>
</tbody>
</table>

in turn is connected to a router. There are 16 routers overall connected as 4x4 mesh architecture.

The bus protocol is AHB. The routers are on a packet switched layer and use XY routing to transfer packets between routers.

For the directory protocol, a hash table was created to keep track of all the different processing cores accessing each data line. Every time a write request is sent to a data line, the hash table is examined to see if other processing cores have a copy of the same data line. If they do, then a control signal is sent to each of the processing cores to invalidate their copy of that data line. Therefore, for each write transaction, the number of control signals sent is equivalent to the number of processing cores that have a copy of the data line.

For the broadcast protocol, it does not matter how many processing cores access a particular data line. Even if only 2 processing cores share the data line, a control message is broadcast over the entire network since there is no method to determine which processing core has a copy of that data.

For the hybrid protocol, instead of keeping track of the processing cores accessing the different data lines, the cache coherency logic keeps track of the routers from which the requests were sent. In this case, there are 16 routers and thus, 16 bits are needed to keep track of the access
Table 7.3: Characteristics of the Synthetic Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Clock Range</th>
<th># Requests</th>
<th>% R/W</th>
<th># Data Sets</th>
<th>Data Overlap</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>3,000,000</td>
<td>1,600,000</td>
<td>50</td>
<td>16</td>
<td>No overlap in data sharing</td>
</tr>
<tr>
<td>B2</td>
<td>3,000,000</td>
<td>1,600,000</td>
<td>50</td>
<td>16</td>
<td>No overlap in data sharing</td>
</tr>
<tr>
<td>B3</td>
<td>3,000,000</td>
<td>1,600,000</td>
<td>50</td>
<td>16</td>
<td>50% overlap in data sharing</td>
</tr>
<tr>
<td>B4</td>
<td>3,000,000</td>
<td>3,000,000</td>
<td>50</td>
<td>16</td>
<td>50% overlap in data sharing</td>
</tr>
<tr>
<td>B5</td>
<td>3,000,000</td>
<td>3,000,000</td>
<td>50</td>
<td>16</td>
<td>20% overlap in data sharing</td>
</tr>
<tr>
<td>B6</td>
<td>600,000</td>
<td>1,000,000</td>
<td>50</td>
<td>16</td>
<td>20% overlap in data sharing</td>
</tr>
</tbody>
</table>

patterns. When a write request is sent to the L2 cache, the cache coherency logic checks to see if processing cores on any router have a copy of the updated data line. If they do, one control signal is sent to each router that has that data instead of sending individual control signals to all the cores on the network.

In all cases, when a control message is sent to the processing cores to invalidate their copy of the data, the shared flag is cleared in the data line.

**Synthetic Benchmarks**

The synthetic benchmarks were created to simulate the characteristics of multiple applications running on the system. The benchmark characteristics are shown in Table 7.3.

The cores are divided into groups of 16. No overlap in data sharing means that each group works on its own set of data. Data overlap means that each group accesses data assigned to it, and in addition to that, it accesses data lines in the address space assigned to the adjacent groups. For example, a 50% data overlap means that a processor in group $i$ requests data from the address blocks of its assigned data space $i$, plus the upper 50% of addresses from $i-1$ and the lower 50% of addresses assigned to the group $i+1$.

Figure 7.4 shows the results of the synthetic benchmarks. The figure represents the percentage increase in cache access time of the broadcast and directory protocols over the hybrid proto-
col. As expected, as the number of requests increase and the network is more congested, the average access time of the broadcast protocol increases exponentially. This is to be expected since for every data share conflict, a message is sent on the entire network. The hybrid protocol outperforms the directory protocol by 10% since it requires less control signals to maintain cache coherency. While directory protocol sends a control signal to each of the processors accessing the data line, the coherency hybrid sends a single control signal to the routers. Therefore, a decrease in cache access time is to be expected.

**PARSEC Benchmarks**

The simulated benchmarks span between 4 and 5 million clock cycles. The percentage of read requests to the total request ranged from 12% to 33%.

As seen in Figure 7.5, the hybrid method is superior to the directory protocol. The average cache access time is reduced on average by 44%. The directory protocol performs better than the broadcast method and that is to be expected since the directory protocol is a point to point communication. The percentage gain between the hybrid protocol and the directory or broadcast
protocols varies between the different benchmarks. The improvement in average cache access time highly depends on the characteristics of the data pattern of the benchmarks, specifically percentage of data shared among the processing cores. Initial placement of the processing cores to routers also plays an important role in cache access time.

The line size in this example is 128 bytes. The number of processing cores is 128, therefore, 16 bytes are needed to keep track of all data accesses. This means the overhead of the directory protocol is 12.5% while the overhead for hybrid protocol is 1.56%.

It is evident from the data analysis that the hybrid coherency protocol not only reduces the hardware overhead significantly, it also improves cache access time of the processing cores since only one signal is sent for a group of routers sharing a data line.

### 7.3 Hybrid Cache Coherency on Network on Chip

The cache coherency presented in the previous section is unique to *Hybrid Architectures*. The results highlight the need for a hybrid coherency method to reduce overall cache access time.
while maintaining a scalable protocol. The main advantage of the hybrid architecture is the natural grouping of cores that belong to the same process and therefore share data. That same idea can be extended for the NoC multicore architecture. The aim behind the proposed method is to implement a hierarchical system to reduce area overhead. For the hybrid architecture, one bit represented a group of processing cores connected to a single router. The idea behind the proposed method has a similar approach for simple NoCs where one bit represents a group of cores contained in a region.

### 7.3.1 Hierarchical Directory Broadcast Protocol

Figure 7.6 shows the layout of the proposed cache coherency protocol. As before, each router has a processing core with a private L1 cache and a shared L2 cache distributed over the packet switched network. The network has 144 routers and each router has a single processor connected to it.

The network is divided into regions, in this example each region has 9 routers. In each region, one router is assigned as the router in charge of that region known as the broadcast router. In Figure 7.6, the red colored routers are tasked with broadcasting to their region. Each cache line keeps track of the data shares by referencing the region the request came from. In this example, if routers R2, R48 and R61 are accessing a data line in R58, then the Hybrid Directory Cache Protocol keeps track of regions 0 and 4 as having a copy of the particular data line. Next to the cache line, the corresponding bits for regions 0 and 4 have the control bit set to 1 while all other regions have control bit set to 0. If that cache line is updated, then router R58 sends control signals to Routers...
Figure 7.6: Hybrid Cache Coherency Protocol for NoCs
Chapter 7: A Hybrid Cache Coherency Protocol for Hybrid and NoC Multicore Architectures

Figure 7.7: Directory vs Hybrid Protocol Area Overhead for NoCs

Figure 7.8: Read/Write Procedure for Hybrid Cache Protocol
13 and 49. When those routers receive the control signal, they broadcast the message to the routers in their region. For example, router 13 sends the message to routers 0,1,2, 12, 13, 14, 24, 25 and 26. It is important to note here that the designated routers do not keep track of the data accesses in their region and therefore cannot determine which processing cores in their region have a copy of that data line. Therefore, a broadcast message is sent to all the routers in that region.

The method of the division of routers to regions has a direct impact of the effectiveness of the proposed method. The regions cannot be chosen randomly. When a broadcast message is sent in a region, the control signals should reach their destination with minimal impact on network traffic. Therefore, it is preferred that routers in a region be at most a few hops away. In the example above, the routers are at most 2 hops away from the designated broadcast router in their region.

To keep track of the mapping of routers to regions, each router can have a single hash table. The number of entries in the hash table is equivalent to the number of routers found on the network. Therefore, the size of the table is negligible compared to the size of the cache.

Table 7.4 shows the number of bytes needed to keep track of the data lines that are shared among different processing cores. As seen, the number of bytes needed increases exponentially for directory protocol with the increase in the number of cores. Figure 7.7 shows the percentage increase in area overhead as the number of cores increases. The data line in the simulated benchmarks is 128 bytes. As seen, for 1024 cores, the cache area overhead for maintaining directory protocol is 100% while for coherency protocol it is 11%. Therefore, the hybrid protocol can scale at a better rate as the number of cores increase in future generation CMPs.

7.3.2 Experimental Results

In order to assess the effectiveness of the proposed hybrid cache coherency protocol, we examined the average cache access time of the processor cores. The benchmarks used were the same as in Section 7.2.2. However, instead of having 16 routers connected as a 4x4 mesh architecture, we
have 144 cores connected as a 12x12 mesh architecture. Only one processing core is connected to each router. Since we have 128 cores and 144 routers, some routers are not assigned any cores. The L2 cache is shared among all the cores and is distributed on all 144 routers.

For the broadcast protocol, the simulator only keeps track if a data is shared by at least two cores. If a data line is shared, then for every write request on that shared line a broadcast message is sent on the network. When processors receive the control message, they check to see whether they have a copy of that data line. If they do, then they invalidate their copy of the data.

For the directory protocol, each cache line has an additional 128 bits representing the processing cores. For every data write, a control signal is sent to each of the processing cores that have a copy of that data.

As for the hybrid cache protocol, Figure 7.8 shows the procedure for read and write accesses. Each router has a single table representing the mapping of routers to regions. For every data read access, the cache coherency logic checks to see the region mapping for that processor. If the bit corresponding to the region is not set, then it sets the shared bit of that region to 1. If different processor cores that belong to the same region access the shared line, only one bit is set to represent all the cores in that region.

The write cache request is a two step process. When the cache access is a write request, then the router checks to see the regions that have a shared bit of 1. Then for every set bit, the coherency logic sends a control signal to the corresponding broadcast router for that region. In step 2, the broadcast router receives the control signal. It then sends a broadcast message to the cores in its region. In this example, each broadcast router is responsible for 9 routers.

Since the hybrid protocol is likely to send control signals to routers that do not share a data line, the traffic generated on the network due to broadcast control signals will increase when compared to the directory protocol. However, the area overhead needed for the hybrid architecture is less than that of the directory protocol. In this example, we have 128 byte L2 cache line. The
directory protocol needs 16 bytes as overhead, while the hybrid cache coherency needs 1.8 bytes, an area reduction of 89%.

**Synthetic Benchmarks**

The synthetic benchmarks used for this experiment are the same ones described in Section 7.2.2. In this case, the groups of processing cores accessing the same data set that were used to generate the synthetic benchmarks are assigned to the same broadcast router.

Figure 7.9 shows the results of the synthetic benchmarks. The percentage values show the difference in cache access time between that of the proposed hybrid cache coherency and that of directory and broadcast protocols. The results show that the broadcast protocol is infeasible as the number of data lines shared increases and the network is more congested.

The main comparison is between directory and hybrid protocols. In most cases, the hybrid coherency protocol performs better than the directory protocol. In one case, the directory protocol is better than the coherency protocol by 0.3%, an acceptable compromise when taking into consideration the vast reduction in hardware overhead between the two protocols. It is evident from the results that the increase in the number of control signals sent due to the broadcast aspect of the proposed hybrid architecture is counterweighted by the decrease in the control signals sent from the shared data line to the broadcast routers instead of all the processing cores sharing that data line. In the directory protocol, control signals are sent to individual cores, thus they travel longer distance to reach their destination. This causes packets to be queued in more routers along the way causing more traffic. The broadcast control signals of the coherency protocol are at most 2 hops away. This explains the percentage gain in cache access time of the hybrid protocol.
PARSEC Benchmarks

Figure 7.10 shows the percentage difference in cache access time for PARSEC benchmarks. As seen, the increase in network traffic due to hybrid cache coherency did not significantly impact overall cache access time of the processing cores. The broadcast protocol in this case is not that different than the directory or the hybrid protocols, the reason is due to the fact that data sharing in these benchmarks is limited and the number of control signals that were sent on the network was less than 1%. Therefore, the control signals did not have a significant impact on the overall cache access time. As shown in the synthetic benchmarks, the broadcast protocol is not a feasible solution as the number of data lines shared among cores increases.

Looking at the results of both the synthetic and PARSEC benchmarks, it becomes evident that a hierarchical solution is a feasible method to reduce area overhead without significantly impacting overall cache access time.
7.4 Conclusion

As seen, neither the broadcast protocol nor the directory protocol for cache coherency is scalable. The overhead of the directory protocol is too large for large networks. The traffic introduced by the broadcast protocol is too large and can saturate the network. In this chapter, we proposed a hybrid cache coherency protocol that takes advantage of the hybrid architecture to find a good medium. The overhead of the hybrid protocol is less than that of the directory protocol by 87%. Furthermore, the cache access time decreased for the hybrid protocol when compared to directory or broadcast protocol since the number of control signals sent on the network is less than that of the other protocols. Average cache access time decreased by an average of 44%.

Although the hybrid method was introduced as part of the hybrid network, the same principle can apply on simple NoC architectures. We also proposed a hybrid cache coherency protocol for NoC architectures. In the proposed method, the network is divided into regions. Each region has a broadcast router in charge of that region. The cache coherency protocol then keeps track of only the routers that are assigned regions and send control signals to only those routers. The routers in
charge then send a broadcast signal to the routers in their region. As shown, the area overhead can be reduced by as much as 89% when compared to the directory protocol without significant increase in network congestion and therefore average cache access time.

As the number of cores increase, a hierarchical method will be essential to overcome the challenges introduced by the increased number of cores in future generation CMPs. Coherency protocols need to take advantage of the characteristics of the applications as well as unique architectural features to implement efficient cache coherency protocols.
Chapter 8

Conclusion

The aim of the proposed work in this dissertation is to reduce average cache access latency in multicore system. To accomplish this goal, several hardware and software techniques were proposed.

8.1 Hardware Methods

Three architectures were proposed to reduce cache access time.

8.1.1 Reconfigurable Architecture for Multicore Systems: RAMS

We studied hybrid architectures where each node in a mesh-style packet-switched NoC architecture contains a bus-based subsystem with a small number of processors. Experimental results using select benchmarks demonstrated that these hybrid architectures offer superior performance when compared with purely bus based or purely NoC style architectures.

Further studies indicated that while a hybrid architecture is preferable, the optimal number of processors on each bus subsystem varies based on the application. This number was shown to be between 1 and 8 depending on the communication requirements of the application. This observation,
in addition to the observation that different processes have different number of threads, led to the proposal of a reconfigurable architecture for multicore systems.

RAMS is a new reconfigurable NoC architecture which allows scalable bus-based multiprocessor subsystems on each node in the NoC. Following configuration, the system provides a multi-bus execution environment where each processor is connected to a bus and the bus-based subsystems communicate via routers connected in a mesh-style configuration. The system can be reconfigured to vary the number of bus subsystems and the number of processors on each subsystem. Each processor contains a Level 1 (L1) cache and each bus, connected to a router, has access to a Level 2 (L2) cache. The L2 caches distributed across the network together form a large virtual L2 that can be shared by all the processors in the system via the router network. Experimental results using the SIMICS simulator with PARSEC and synthetic benchmarks showed the performance advantages of the proposed architecture.

### 8.1.2 A Dynamically Reconfigurable Multicore Architecture: DyaReMA

Extending on the idea of RAMS, we proposed migrating processors to any router in the NoC architecture. This is realized by implementing a reconfigurable interconnect that allows reassignment of processor cores to different routers at runtime. We presented the proposed architecture in detail. The architecture has a set of buses that are connected to all the routers. The processors then can be reassigned to any router by changing the assignment of processor to buses. The results showed significant improvement in cache access time. The cache access time was decreased by as much as 24% when compared to the hybrid architecture with no reconfigurability.

Since the proposed architecture is costly with current technology, we showed a segmented hardware implementation of the proposed architecture. In the segmented version, processors can be reassigned to neighboring routers only. This limits the flexibility of the proposed architecture, but shows significant gain in terms of area overhead. We presented area and performance analysis based
on a detailed Verilog model and synthesis of the proposed architecture.

8.1.3 Test Platform Generation

To test various architectures, we created a modular design platform that can cater to a variety of architectures. The components created can be used to create different architectures. The architectures tested were bus, network, hybrid and segmented DyeRaMA. The created library of modules can be used to create a multitude of different architectures. The modules are synthesizable and can give accurate area and power analysis.

8.1.4 Heterogeneous Cache Architecture

The architecture proposed in literature all have a homogeneous distribution of cache blocks to routers. We proposed a heterogeneous distribution of caches to routers to improve cache access latency. Since cache blocks are assigned to routers at design time, we analyzed the best distribution of cache blocks to implement a generic architecture that suits most applications. The best architecture has the following configuration: large and small caches alternated in the center of the network, while routers on the vicinity have middle sized cache blocks. This implementation showed improvements of up to 20% compared to the homogeneous model.

Extending on this idea, we implemented a reconfigurable cache architecture, where a subset of cache blocks are reconfigurable and can be assigned to neighboring routers based on the needs of the application running on it. This implementation showed improvements of up to 61%.

8.2 Software Methods

Apart from the hardware methods mentioned above, we improved on current software policies to reduce cache access time.
8.2.1 Data Migration and Address Lookup

The success of multicore architecture is contingent on getting data as quickly as possible to the processors accessing it. Data migration keeps track of data accesses and moves the data lines closer to processors requesting it. Data migration is costly since every data line needs to keep track of all data accesses. We proposed an incremental data migration where data lines are moved one router at a time. The data lines are moved based on the number of accesses from North/South East/West direction. Furthermore, we implemented prediction based data migration where neighboring data lines of migrated data are moved as well based on threshold values. This takes advantage of spacial locality.

Data migration violates the rules of data mapping. Therefore, a method is needed to search for data in the network. We proposed a distributed lookup table where several routers across the network keep track of migrated data and serve a subset of routers. We showed that having distributed lookup table instead of global lookup method can reduce cache access time by as much as 87%. This is due to the distribution of congestion of control signals and the reduction in distance between a router and an address lookup table. The cumulative size of the distributed tables however are larger than that of the global table due to the existence of duplicate values across the tables.

8.2.2 Cache Coherency

Having private L1 caches and shared L2 caches, cache coherency becomes a necessity. Different cache coherency protocols can be used. Broadcast methods broadcasts a control signal on the network for every data write of shared memory. Directory protocol keeps track of all the processors accessing a data line at a cost of area overhead. We introduced two cache coherency techniques; the first is unique to the hybrid Bus/Network architecture where we showed that the area overhead is reduced by 87% for 128 core architecture over the directory protocol and cache access time is reduced by an average of 44% compared to broadcast method. The second protocol is
a hybrid protocol for NoC multicore architectures where we showed that data overhead is reduced by 89% compared to the directory overhead having comparable cache access times.

8.3 Future Work

The methods proposed in this work do not take into consideration thermal issues. When dealing with hundreds of cores, heating can become an issue. Overheating can destroy parts of the chip [55] [47]. Thermal aware scheduling and resource allocation can go a long way to elongating the lifetime of the multicore processor.

Thread migration is another potential improvement that has not been considered [67]. Thread migration is the act of moving threads from one processor core to another. This feature would allow process mapping to be dynamic and more threads to run simultaneously on the chip.

Efficient scheduling and resource mapping [24] methods are needed to take advantage of multicore systems. Software is moving towards multithreaded applications. Different threads belonging to the same application need to be bound to processor cores that are close to one another. Threads belonging to the same application are expected to share data, hence, data belonging to the same application can reside in a cache block close to the assigned threads.

A study combining two or more of the software and hardware techniques presented in this work should be performed. Combining different methods can further improve overall cache access time in multicore systems.

Finally, a hardware test setup must be implemented that can give accurate timing, area and power analysis. The components that are included in section 4 are synthesizable. Those components can be used to create different architectures and implement them on FPGAs for further analysis.
Bibliography


