A DIRECTORY ENHANCED NETWORK ON CHIP FOR FPGA

BY

JACOB S TOLAR

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Associate Professor Deming Chen

ABSTRACT

This thesis presents and evaluates a directory enhanced network on chip for FPGA, with the goal of improving the performance of cores generated by FCUDA, a translation tool enabling CUDA code to be run on FGPAs. NoCs are an inherently scalable platform, as aggregate system bandwidth increases with the number of nodes in the system. This work enhances an existing NoC to include a directory protocol capable of tracking the location of on-chip data stored in core-local BRAMs. By tracking the location of on-chip data, requests that would normally be satisfied by off-chip memory can be fulfilled by on-chip sources, allowing performance gains. Simulation results show a directory-enhanced NoC gains of up to 40% in speed over an ordinary NoC for some applications. In addition, simulation and synthesis results show potential for increased overall application performance over a bus-based system, despite the significant area overhead of NoC routers.

ACKNOWLEDGMENTS

The completion of this thesis and my graduate work at Illinois would not have been possible without the support and assistance of the following individuals and groups.

I am especially grateful to my advisor, Dr. Deming Chen, for his continued support, encouragement, and guidance throughout my time as a student in his research group. Many thanks also to former student Alexandros Papakonstantinou, who invested significant time and energy into helping my project succeed. I would also like to thank Swathi Gurumani of ADSC and Dr. Kyle Rupnow of NTU for their advice and suggestions on this project, as well as Dr. Rakesh Kumar for his help and guidance during my time as a teaching assistant.

I would like to acknowledge both Intel Corporation and the Gigascale Systems Research Center for their generous support of this work.

Thanks to Leslie Hwang, Wenxun Huang, and Liana Nicklaus for their contributions to the work presented in this thesis.

I am also indebted to the other members of Dr. Chen's research group for ideas, comradery, and quite a few sugary snacks. In particular, thanks to Wei Zuo and Keith Campbell, who were occasional sounding boards for my project. Many thanks to as well as to JP, our wonderful secretary who ensures that everything runs smoothly in our group.

Finally: I am forever grateful to my family, for helping to keep me sane; to my friends, for helping me to enjoy my time as a graduate student; to my girlfriend, Abby, for her constant love and encouragement; and to my Lord, for life itself.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

ASIC	Application specific integrated circuit
BRAM	Block RAM
CUDA	Compute Unified Device Architecture
DSP	Digital signal processing
FIFO	First in, first out
FPGA	Field programmable gate array
GPU	Graphics processing unit
HLS	High-level synthesis
LUT	Look-up table
NoC	Network on chip
NRE	Non-recurring engineering [costs]
PLD	Programmable logic device
SM	Streaming multiprocessor
SoC	System on chip

CHAPTER 1

INTRODUCTION

In the past several years, there has been a dramatic shift in computing trends toward concurrent computation. For decades, integrated circuits scaled in size according to Moore's law [1], with manufacturers cramming an ever-increasing number of ever-shrinking transistors onto a single integrated circuit. While Moore's law continues to hold true, manufacturers have faced significant challenges in power [2] and heat dissipation due to the breakdown of Dennard scaling [3].

In an attempt to continue to deliver improved performance despite an increasingly constrained power budget, in the last decade manufacturers have turned to multicore processors [4], [5], [6] (which, notably, do not solve the fundamental problem of reaching the limits of Dennard scaling [7], [8]). Unfortunately, multithreaded programs for multicore processors are notoriously difficult to write [9] and require additional cost and effort when compared to sequential programming models [10].

In light of this, new programming models have emerged in order to allow programmers to more easily exploit the performance offered by highly parallel hardware. Nvidia's CUDA [11], [12] provides a programming model for their massively parallel GPUs; similarly, OpenCL [13] has emerged as a framework for exploiting parallelism in heterogeneous systems. FCUDA [14] is a design flow that enables CUDA programmers to easily exploit the inherent parallelism of an FPGA using ordinary CUDA code.

This thesis presents an automatically generated, intrinsically scalable network on chip (NoC) for FCUDA, which enables efficient point-to-point communication both among cores and between cores and other devices (such as off-chip memory). A directory system has been added to the NoC, which allows requests for off-chip data to be fulfilled by on-chip sources if possible. The goal of this work is to improve overall application performance in FCUDA by improving communication efficiency and enabling automatic exploitation of data locality.

CHAPTER 2

BACKGROUND

This chapter provides some brief background on several relevant topics: FPGAs, FCUDA, NoCs, and coherence protocols.

2.1 FPGAs

ASICs, or application-specific integrated circuits, have long been the manufacturing technology of choice for custom digital hardware. With hundreds of millions of gates, an ASIC can be designed to implement any arbitrary design, with the benefits of low cost at very high scale. However, ASICs face many limitations, including high NRE costs, increasing mask costs [15], and long turnaround time and time-to-market. Additionally, ASICs are high-risk: if a bug is found in an ASIC after fabrication, there is no recourse - completely new devices must be fabricated and shipped to angry customers.

Field programmable gate arrays, or FPGAs, are a class of programmable logic device (PLD) that originally became commercially viable in the mid-1980s [16]. FPGAs overcome many of the barriers of ASIC design by enabling users to quickly design and reconfigure hardware, without the high initial cost (in terms of both man-hours and dollars) of ASIC designs. FPGAs do have some disadvantages when compared to ASICs, such as higher unit cost at volume, significant power/area drawbacks, and lower speed. However, they are becoming increasingly popular for prototyping and low-volume products due to their advantages in turnaround time, startup cost, and reconfigurability [17].

For designers, the architecture of FPGAs forces difficult trade-offs. Certain hardware structures in FPGAs are plentiful, while others are limited to a significant degree. Thus, a designer must take care to take advantage of the strengths of the FPGA platform while working around the weaknesses.

For example, FPGAs are generally over-provisioned with wire and register resources. However, an FPGA generally has a limited number of DSP blocks or slices [18], [19]. These hard macros have significant performance and power advantages over equivalent circuits implemented in LUTs. However, they are both limited in number and flexibility, and careful trade-off analysis is required to get optimal performance.

Similarly, FPGAs contain hard memory blocks known as block rams (BRAMs) [20], [21]. These BRAMs are significantly more efficient than synthesized memories, but are limited in number and are inflexible: even the newest FPGA designs from popular vendors are limited to 2-port RAMs. Expensive custom techniques must be used to achieve greater numbers of ports [22].

2.2 High-Level Synthesis and FCUDA

High-level synthesis (HLS) is a technique for translating a high-level design description (for example, in the C programming language) into a synthesizable RTL (Verilog or VHDL) hardware description [23]. HLS aims to improve designer productivity by increasing the abstraction level at which the designer works [24], although this often has a significant performance cost when compared to a manual design [25].

FCUDA [14], [26] is a design flow that enables CUDA [11] kernels to be mapped onto FPGAs using a high-level-synthesis tool, Vivado [27] (previously AutoPilot [28]). Using FCUDA, programmers familiar with the CUDA programming model can quickly explore a large design space and efficiently utilize the resources of the FPGA. The output of the FCUDA flow is a set of computation cores at the RTL level (with a core roughly performing the role of a CUDA SM) onto which the original computation is mapped.

2.3 Network-on-Chips

Network-on-chips (e.g., see Figure 2.1), or NoCs, have become increasingly popular in both academia and industry as transistor size has decreased. Designer productivity has not kept pace with the exponentially increasing number of transistors that fits on a single chip [29]. Additionally, traditional communication architectures (such as shared buses) do not scale well as the

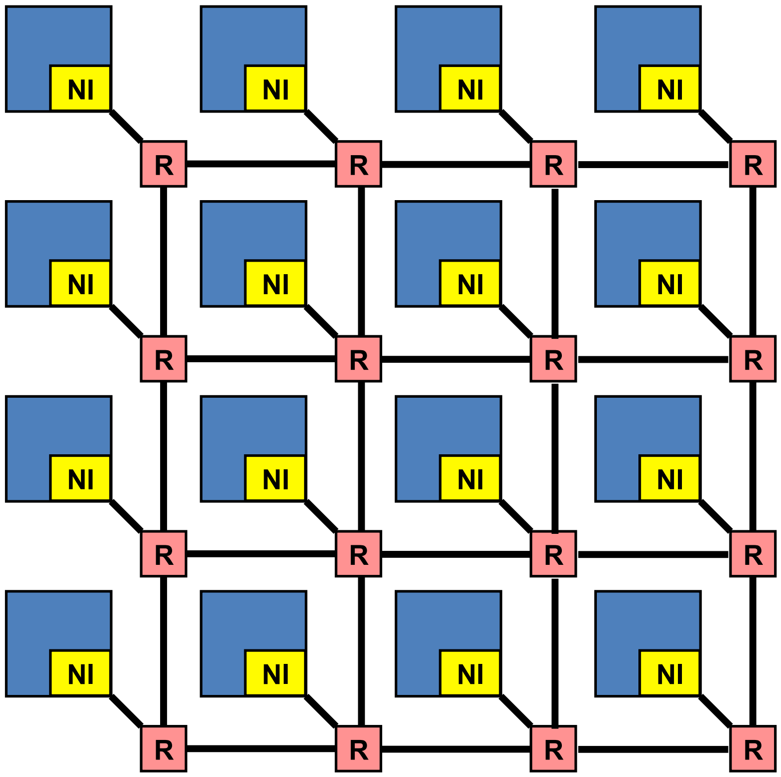


Figure 2.1: Sample network of 16 routers and 16 compute nodes.

number of nodes increases. A network-based fabric provides potential solutions to both of these issues.

First, NoCs provide a regular, structured, on-chip communications fabric. This means that NoCs offer potential for improved designer productivity and design predictability [30], which is extremely important as transistor count continues to increase.

Additionally, NoCs are inherently scalable [31], allowing multiple clients to communicate simultaneously and with greater aggregate bandwidth than a bus [32]. When additional nodes are added, additional network resources can be added as well, increasing aggregate bandwidth. Shared buses, on the other hand, scale poorly by definition, as a bus simply shares the same resource among more devices as additional nodes are added to the design [33]. Thus, buses are generally limited to a small number of masters until arbitration becomes very costly [31].

NoCs offer other advantages as well. For example, NoCs have been shown to use less power-per-bit than buses [34]. The regular structure of a NoC can make factors like crosstalk easier to predict and design around [32]. Similar to bus protocols such as AMBA, CoreConnect, or Wishbone, NoCs also offer modularity by defining a standard interface with which modules may communicate. In industry, both Intel [35] and Tilera [36] have fabricated chips based on NoCs.

While some of this reasoning applies specifically to ASICs, NoCs are also attractive on FPGAs for many of the reasons previously listed, including potential for improved productivity, greater design regularity, greater bandwidth, and better scalability. Several designs have been proposed for future *hard* FPGA architectures based on the NoC paradigm [37], [38]. Others have proposed *soft* NoCs for current FPGA architectures [39], [40], [41], [42].

2.4 Coherence

In a multiprocessor system in which each processor has its own cache in front of a shared memory, it is possible for several copies of the same data to exist: for example, one copy in the main memory and one copy in each local cache. When one processor changes its local copy of the data, the other caches containing this data should eventually be updated. The process of ensuring that all caches contain updated data is known as *cache coherence*, and can be implemented in a variety of ways.

2.4.1 Snoopy coherence

In a snoopy coherence protocol [43], [44], [45], [46], a bus is shared between all processors in the system. Each processor monitors the bus for activity in order to update the coherence state of its local cache, as shown in Figure 2.2. However, as with any shared bus solution, scaling problems begin to appear as the number of processors increases.

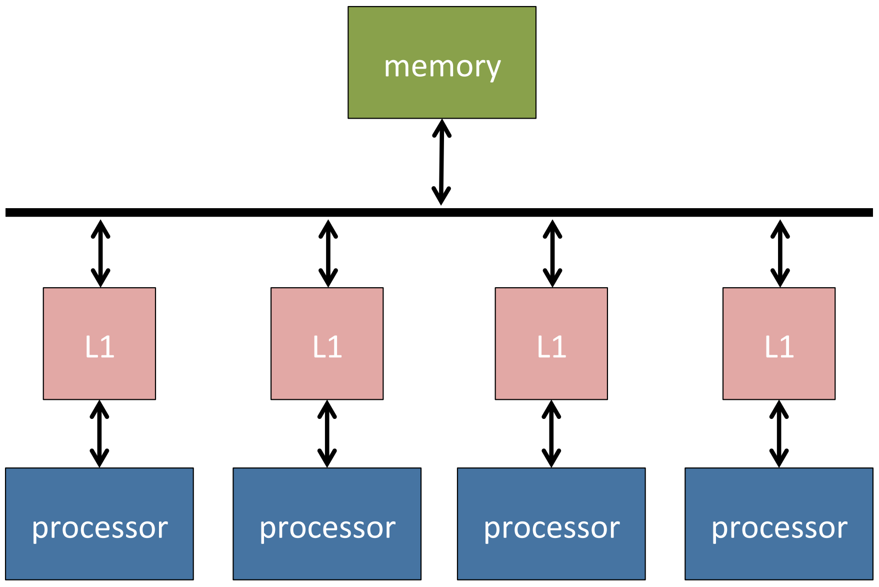


Figure 2.2: System using a bus-based snoopy coherence scheme.

2.4.2 Directory coherence

Directory-based cache coherence, introduced separately by Tang [47] and Censier and Feautrier [48], is a method of ensuring coherence that does not require snooping on a shared bus. Thus, directory coherence becomes significantly more efficient than snoopy coherence at scale, as it communicates using point-to-point traffic rather than a shared-bus broadcast.

The earliest directory coherence protocols are known as *full-bit* and consist of extra status bits as well as per-processor presence bits for every block in main memory (where a block is the size of a cache block). The purpose of a directory is to track owners of a block within a system. When a read miss occurs, data can be fetched from an owning cache on chip; writes cause invalidations to be sent to only sharing processors. The directory can be either centralized or distributed; if distributed, each address in the system has a statically defined "home node" which tracks the directory information for that address.

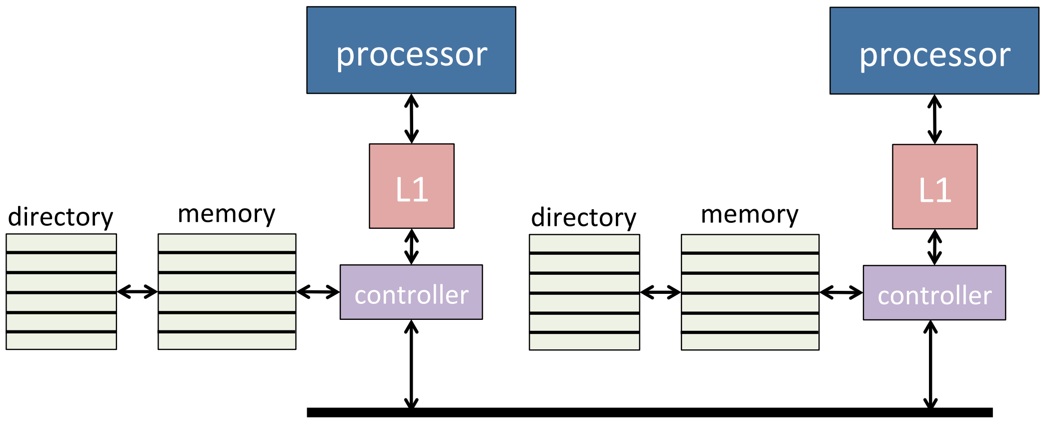


Figure 2.3: System with two nodes using full-bit directory coherence.

Full-bit directory coherence has scaling issues with large numbers of nodes, as the storage overhead required for the directory increases with each additional processor in the system. With enough nodes, this scheme eventually becomes untenable: the size of the directory could exceed the total size of memory!

Several studies have shown that a write generally causes very few invalidations (that is, a given block of memory is normally shared between only a small number of nodes when a write occurs) [49], [50]. This indicates that the full-bit directory is significantly over-provisioned (most bits in the presence vector will be set). To reduce overhead, *limited pointer* directory scheme [51] tracks only a small set of sharers per block. Instead of a bit per processor per line, the limited pointer scheme tracks a small number of sharers by node ID. For example, a limited pointer directory scheme may choose to keep track of 3 sharers; in a system with 256 nodes, this means a total of 24 bits of storage required per line, as opposed to 256 bits/line for the full-bit scheme. When the directory reaches capacity for a given cache line, there are two basic strategies: (1) force the line to broadcast on invalidation, or (2) invalidate a single share (eviction). Other proposals have been made that attempt to improve on these methods [52], [53].

Other directory coherence proposals include *dynamic* and *sparse* directories. A dynamic directory stores pointer to an entry in a pool [54] for each block in memory, along with additional status bits. Instead of preallocating directory space to track every block in memory - which will likely go underutilized - a smaller amount of dynamically allocatable space is used. The goal is to allocate less total space for the directory, while improving the utilization of the available space (as not all blocks in memory will be present in cache all the time). A sparse directory [55], [56] stores tracking data within the cache itself. Sparse directories exploit the fact that caches are much smaller than main memory and that directory tracking is only required for data that is present in the cache.

CHAPTER 3

IMPLEMENTATION

For this work, an initial open-source packet-switched NoC for FPGA [41] was chosen to be used as a baseline system. This NoC was first enhanced to be more easily configurable, and a generator was written to automatically create a mesh network.

A directory system was added to the network to enable routers to track the location of on-chip data. The directory system can be used to improve performance by allowing some requests to be handled on-chip rather than off-chip.

Several enhancements were made to the directory system in order to improve performance. These enhancements include allowing bypassing of directory stages to improve latency, as well the ability to track outstanding memory requests and capture duplicate requests.

Finally, a set of scripts and hardware modules were designed to enable FCUDA cores to communicate with the NoC, as well as to aid the user in designing a complete system.

3.1 Baseline Open-Source NoC

As an initial step, an open-source packet-switched NoC was chosen to be used as an initial implementation [41]. This NoC was designed for synthesis on FPGAs (specifically, the Xilinx Virtex IV) and was thus ideal for this application.

This NoC uses statically generated routing tables to route packets through the network. Given a specific router within the network, an arrival port, and a destination, there is only a single entry in the routing table specifying the next hop the packet should take. Critically, the next hop is actually generated one hop ahead of time (the next hop computed in a given router determines the output port to be chosen at the *next* router along the path).

The chosen NoC supports flow control via back-pressure status bits sent through the network. Each router has (for example) N inputs and N outputs. A given output port can transmit a maximum of one packet per cycle; thus, if all N input ports attempt to communicate with the same output port, some buffering will be required. When buffers reach a high-water point (almost full), neighboring routers are notified to stop sending packets via a back-pressure signal.

3.2 NoC Enhancements

The original open-source implementation of the NoC lacked configurability; almost all aspects of the NoC were statically defined. Of the essential aspects of the NoC router, only the data width of a packet was configurable.

For this work, the original NoC was enhanced to be almost entirely configurable via a single user-provided header file. Thus, the NoC can now be configured with a variable data width, variable FIFO sizes, variable number of router inputs, and more. The original implementation was limited to 32 total nodes in the system; that limitation has been removed (there is no limit to the size of the network).

In addition, the NoC was enhanced to remove the use of Xilinx primitives in the design. Previously, the use of vendor-specific components used meant the design could only be used for Xilinx FPGAs. All vendor-specific portions of the design were rewritten in generic Verilog, enabling the design to be synthesized on other platforms.

Finally, a script is provided that produces a connected NoC from an input network topology. In addition to setting up all connections between routers and nodes, the script generates correct routing tables for each router, a task that is quite painstaking to do manually.

3.2.1 NoC packet format

In order to support additional features at the network level, the original NoC packet format was expanded to include several additional fields. A packet in the system includes the fields shown in Table 3.1.

Table 3.1: Listing of all possible NoC packet types

Type	Description
sendokbit	Backpressure bit; implements congestion control
sendbit	Valid bit: set to 1 if the packet is valid
nexthop	The output port this packet should be routed to
lastbit	1 if the packet is the last in a sequence of flits
dest	The destination node of the packet
src	The source node of the packet
type	The type of the packet (see Table 3.2
data	Data payload of the packet
addr	Memory address for data requests / responses

Each packet sent through the network specifies a packet type, indicating the function of the packet; these packet types are listed in Table 3.2. Of these types, only memory reads (TYPE_REQUEST) and directory updates (TYPE_RESPONSE_ADDR) require access to the directory; all other packets pass through the directory unchanged.

Table 3.2: Listing of all possible NoC packet types

Type	Description
TYPE_REQUEST	A memory read request
TYPE_RESPONSE_ADDR	A directory update
TYPE_RESPONSE_DATA	A response containing requested data
TYPE_C_REQ	A request that has been redirected to a core
TYPE_WRITE	A memory write
TYPE_OUTSTANDING	An outstanding request match (see Section 3.4.1)

3.3 Directory System

The NoC provides a framework and platform on top of which services may be provided to the system. As the goal of implementing the NoC is to improve overall performance, several services which provide potential performance enhancements were considered. This thesis presents a directory system added to the NoC in order to improve memory access time. This directory protocol is similar in spirit to the directory schemes discussed above, but has several key distinctions. Significantly, unlike ordinary coherence schemes, the NoC directory is not required for correctness.

Coherence is not required because this project uses cores which adhere to the CUDA programming model [11]. CUDA programmers have no expectation of coherence between threadblocks in the same kernel; thus, there is no need for a coherence protocol to track multiple active sharers of data. Thus, by implementing a simpler protocol, some key savings can be achieved over a true coherence protocol. The implementation and protocol are described in the following sections.

3.3.1 Directory protocol

As in a distributed directory coherence protocol, each unique address in the system is statically assigned a *home node*, which is a router within the system. The home node of an address defines which directory is used to track the state of that address. For example, if a given address has a home node with ID 7, then the directory within router 7 tracks the state of that address.

Each directory consists of a RAM (implemented by one or more BRAMs on the FPGA) of configurable size. Each directory is similar in principle to a direct-mapped cache. An input address is split into tag, index, and offset fields; the resulting index maps each address in the system to a single directory index. An update to the directory simply erases the prior information that was stored at a given index.

Each directory entry contains three fields: a tag field, a data field, and a valid bit. As in a cache, a tag field is needed to distinguish between distinct addresses which happen to map to the same directory index. The data field contains the network address of the core containing a given address. The valid bit simply indicates whether or not the entry contains valid data and is set to 0 initially.

When a read request for a given address is made by a compute node, the request is first routed to the home node for that address (a router). Upon arrival at the home node, the directory is accessed using the index field of the input address. By comparing the tag field of the input address with the tag in the accessed directory entry, the router can determine whether or not a copy of the data corresponding to the given address resides on chip. If the data is on chip (there is a tag match), then the packet is redirected to the node expected to have a copy of the data (that is, the node specified by the directory data field). If there is no tag match, then there is no known copy of the data on chip and the request is routed to the memory controller to be fulfilled by the off-chip memory.

Cooperation from the memory controller is required in order to update the directory. When the memory controller is ready to respond to a request with the requested data, it is required to return two packets to the network. The first packet, which contains the data, is routed directly to the destination compute node. Since this packet is not necessarily routed through the home node for the requested address, an additional packet is sent from the memory controller directly to the home node for the requested address. This second packet is used to update the directory to point to the on-chip location of the data that has just arrived from memory.

Note that no invalidations or updates are ever sent based on actions that occur at the compute nodes themselves; updates only occur when data returns from the memory controller. This means that requests for missing data could easily arrive at compute nodes. Therefore, a compute node must be able to determine, given an input address, whether or not it contains the data corresponding to that address.

3.3.2 Advantages vs. true coherence

The purpose of the directory system, as in a directory used in a more typical coherence scheme, is to track the location of on-chip data. In this case, however, the data tracked is not in a cache but rather in FCUDA scratchpad memories: BRAMs local to individual FCUDA cores which serve as temporary storage or as CUDA `__shared__` memory blocks. These scratchpad memories are often used as fast user-directed local caches of memory: for example, to store a single tile of input in a tiled computation like matrix multiplication.

Conceptually, the protocol is similar to a limited pointer scheme: sharers are tracked by ID and only a limited number of sharers are tracked; an eviction policy applies once the directory can track no more sharers. Savings achieved by the simpler version of the protocol are described in detail here.

First, note that the directory system is able to completely ignore memory writes: because CUDA cores have no notion of coherence between threadblocks, programmers do not expect writes to propagate from one core to another. Specifically, if a CUDA program reads and writes the same memory location from two separate threadblocks, the behavior is undefined. Memory writes can quite obviously not be ignored in a true coherence scheme; these schemes exist solely to update or invalidate sharers in case a shared line is written. The ability to ignore writes immediately simplifies the protocol (no updates or invalidates are ever sent) as well as significantly reduces the hardware cost of implementing the directory scheme.

The directory system is used solely as a means to exploit data locality: the directory tracks data locations on chip and enables memory requests to be satisfied from on-chip sources instead of off-chip DDR. Because the directory is used simply as a performance optimization and not for correctness, the directory structures themselves can be much more flexible. Several advantages are gained from this:

1. The system assumes that the cost any on-chip access is significantly less than that of an off-chip access. Therefore, unlike a limited bit directory scheme which tracks a small number of sharers and reverts to broadcast or eviction when the directory entry is full, this scheme tracks only a single on-chip sharer.

2. The directory size is configurable and need not cover the entire memory space. This significantly expands the available design space by allowing the designer to trade directory precision for total overhead of the directory system. If desired, these directories can be implemented as small caches which only track a small portion of the total address space.

   For varying numbers of nodes, Figure 3.1 shows that a large directory for the proposed protocol (512 entries) compares quite favorably in terms of storage requirements to both full-bit and limited-pointer traditional directory schemes. For example, note that with 256 nodes and 4 GB of addressable memory, the proposed directory scheme requires around 122 kB of storage, while the smallest traditional scheme (dynamic) requires 8 MB as a *low* minimum bound (this does not include the size of the directory entry pool). Memory resources are scarce on the FPGA; avoiding the cost of the traditional coherence schemes is very beneficial!

   Of course, ordinary directory schemes and the proposed scheme are not

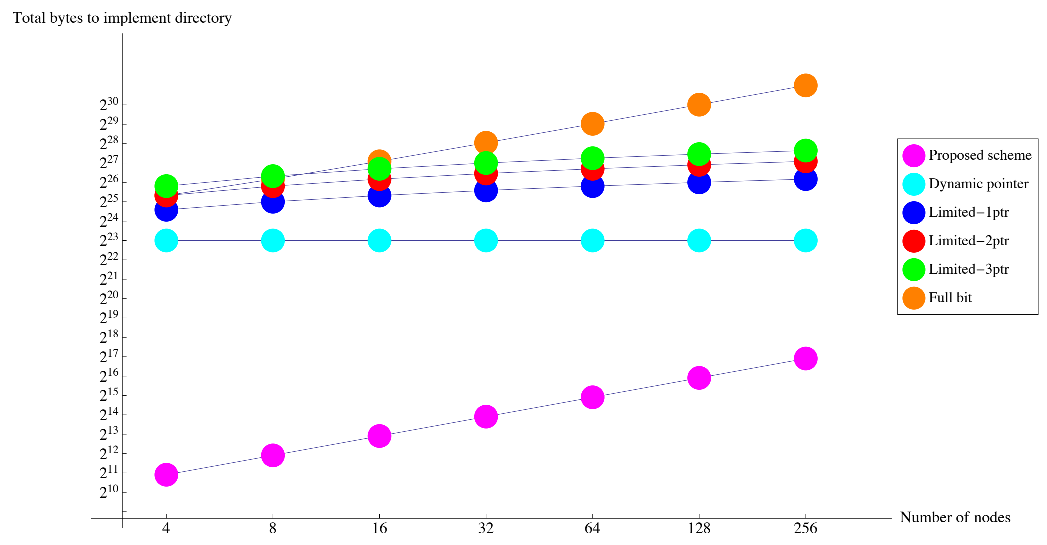


Figure 3.1: Log-log plot showing combined comparison graph. Proposed scheme uses 512 entries in each directory.

truly comparable; the proposed scheme has significantly less precision. For example, it may not be possible to track the state of all on-chip data with the proposed scheme. However, in this system, that simply means slightly longer request latency and does not lead to incorrect behavior. Of course, the designer may always choose to increase the directory size in order to improve performance.

3. Finally, in this scheme, it is not problematic for the directory to contain incorrect information. This is clearly in contrast with traditional coherence protocols using directories for the purpose of correctness. Allowing incorrect data in the directory grants additional savings in terms of the simplicity of the protocol. For example, no invalidations need to be sent to update directories when data is evicted from local core memories, reducing network traffic and preventing excess load on the directory.

3.3.3 Design

The directory system was designed to be overlaid on top of the existing NoC: the core NoC router design does not change. Instead, the (optionally enabled) directory is integrated into a router at its input ports; thus, a packet flows through the directory on its normal path through the network. The directory can redirect a packet by updating the packet's routing information before it reaches the core of the router.

Only two types of packets may access the directory: address requests and directory updates. Additionally, each memory address in the system has a home directory; a given packet only accesses the directory at its home node. Thus, a given packet will require access to at most one directory on its path through the network.

Each router has a single directory module, shared by all input ports. The directory is implemented as a three-stage pipeline just before each router input; Figure 3.2 shows a simplified block diagram of the directory pipeline for a single router port.

The first pipeline stage is for directory access. The directory—configurable in size and implemented using BRAM resources on the FPGA—is similar in principle to a direct-mapped cache. The directory contains three fields: a valid bit, a tag field, and a data field. These arrays are indexed simultaneously using a bit field of the memory address carried in the packet. As in a cache, the valid and tag fields are used to determine whether a given

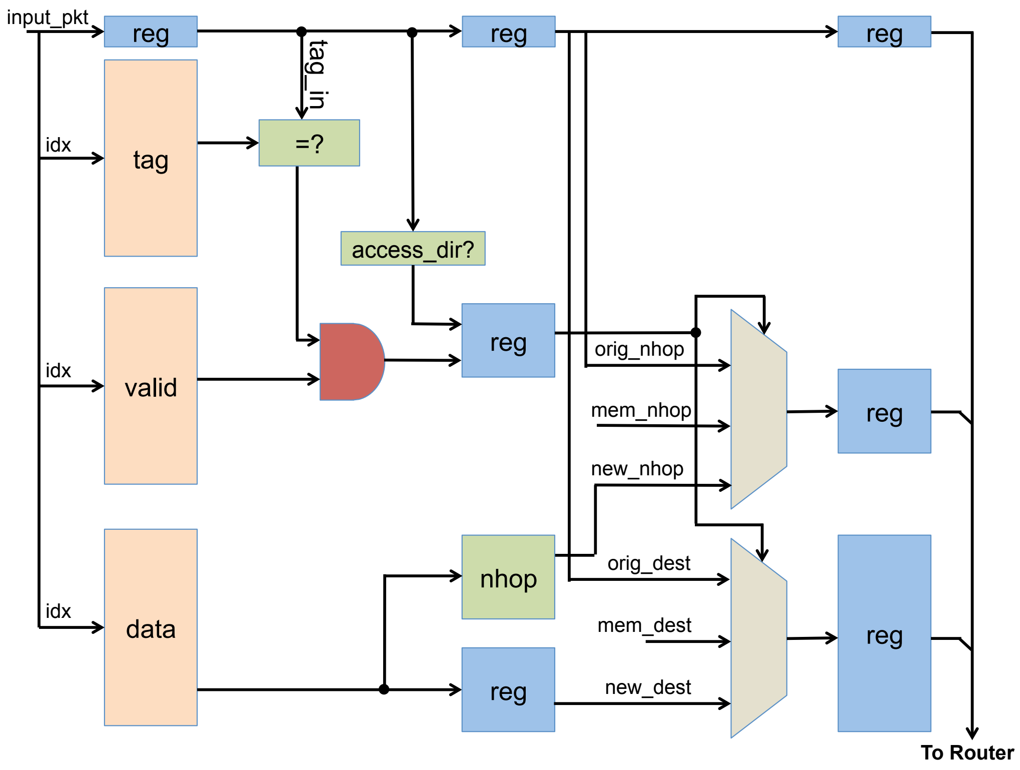


Figure 3.2: Simplified block diagram of a single directory port. Logic to update directory not shown.

directory entry matches the address requested by the input packet. The data field contains the directory mapping and corresponds to the system node ID of the compute node expected to contain the requested data.

Updates to the directory also occur in the first pipeline stage; the directory write signal is generated based on the packet type (must be TYPE_RESPONSE_ADDR) and the packet home node (must be the current node). When a write occurs, information previously stored at the specified directory index is simply overwritten.

The second directory pipeline stage is used for route lookup. NoC routers expect routing information to be generated by the previous hop; thus, in order for the packet to be correctly redirected on a directory hit, the next hop information must be generated and updated before the packet enters the router core.

The final stage is a selection stage, seen on the far right of Figure 3.2. Based on whether the directory had a hit or miss, and whether or not the packet actually accessed the directory, the packet destination and next hop are chosen to be one of three possible sources:

1. The original values, if the directory was not accessed;

2. The next hop and destination address of the memory controller, if the directory was accessed but no on-chip source was found; or

3. The next hop and destination address of the compute node expected to contain the data.

3.3.4 Directory BRAM arbitration

The NoC directory implementation described above requires BRAMs to implement the directories in each router; a directory BRAM is central to all input ports of the router. A router generally contains 5 input ports; thus, the directory should ideally have 5 read/write ports so that each input port can read or write concurrently. Unfortunately, as discussed in Section 2.1, this is simply not possible: BRAMs have a maximum of 2 ports. A workaround must be implemented or the system will be unsynthesizable for existing hardware.

Instead of emulating multiple-ported BRAMs [22] (a possible future direction), BRAM port access is arbitrated to multiple input ports using a simple round-robin scheme. The router inputs are divided into two groupings; for a five-port router, this results in one group of two ports and one group of three. Each group is allocated a single BRAM port to share.

Within a single group, two access priorities exist: read and write. Write requests are always given priority over read requests; that is, if a read request and write request arrive simultaneously, the write request is allowed to access the directory and the read request is dropped (note that dropping requests does not cause incorrect results in this scheme). When two requests of equal priority arrive at the same time, priority between the requests is determined by a small round-robin counter. Thus, over time, each port gets approximately equal priority. This priority logic is quite complex to encode for even a small number of ports and thus is generated by a Boolean minimizer, Bfunc [57], not unlike the Espresso logic minimizer [58].

A simplified block diagram of this priority scheme is shown in Figure 3.3. By partitioning the inputs into two separate sets, the arbitration logic overhead decreases; however, it can cause arbitration to be imperfect. For example, in a router with five ports, each port in the grouping of two has a significantly better chance of accessing the directory in a given cycle than the ports in the grouping of three. The arbitration is also potentially wasteful: if both *i3* and *i4* in Figure 3.3 require access to the directory in a given cycle while none of *i0*, *i1*,or *i2* require access, one of the BRAM ports is wasted and a directory action is needlessly dropped.

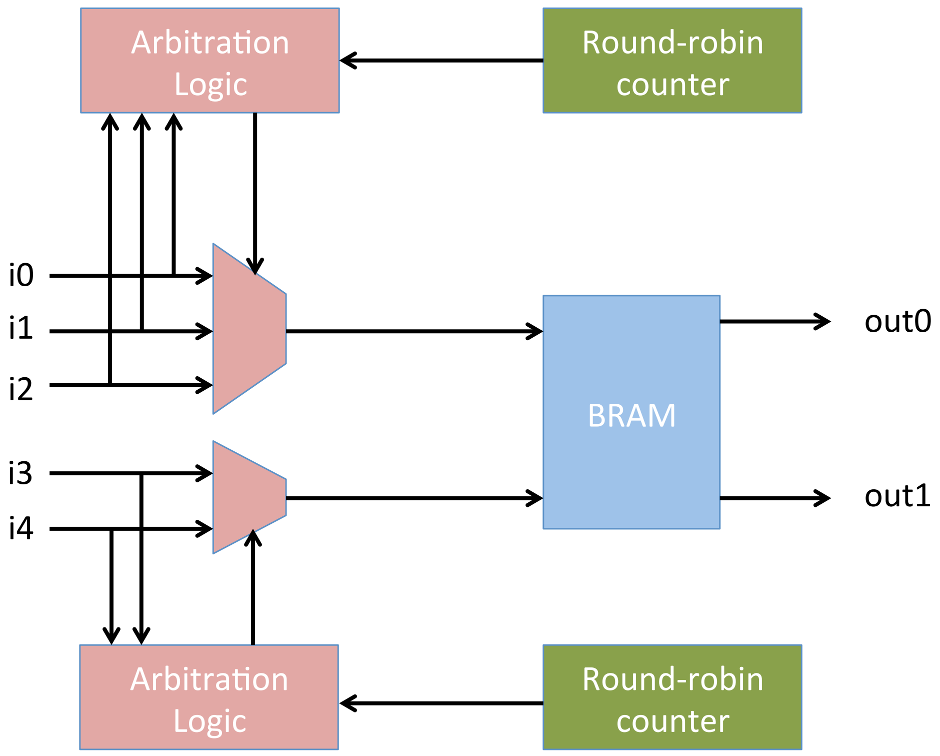


Figure 3.3: Simplified block diagram of directory arbitration scheme for a router with 5 NoC input ports.

3.4 Directory Enhancements

As described in Section 3.3.3, the directory system has no issues with correctness; it is used only to improve performance by exploiting read-only sharing data between cores. However, the protocol adds significantly to request latency and has no capability of capturing and redirecting outstanding requests. The following sections describe methods to handle both of these issues.

3.4.1 Merging outstanding requests

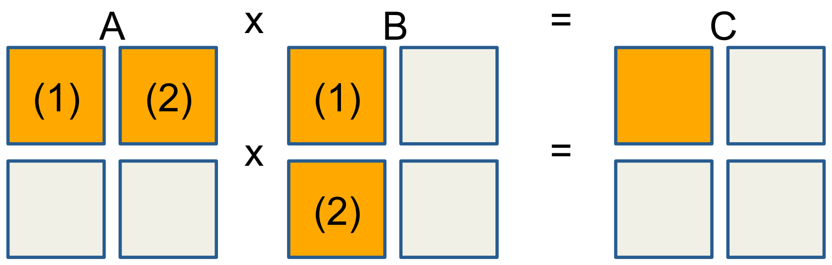
Consider a tiled matrix multiplication kernel. In this kernel, separate cores will often make requests for the same address at very nearly the same time, as shown in Figure 3.4.

This presents a problem: the directory system only tracks data that is *currently* on chip. If multiple requests for the same address occur simultaneously, no directory hits will occur and all requests will be fulfilled by off-chip memory. To solve this problem, the directory system can be augmented to track information about outstanding requests.

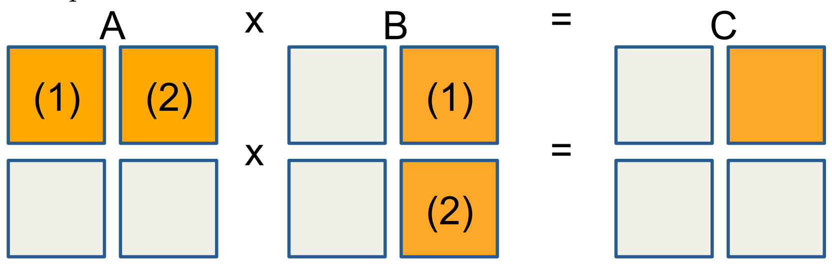
For example, when the first request for a given address is redirected to memory by its home node, the home node directory can be updated (using an extra bit within the directory) to indicate that the given address is not presently on chip but is expected to return from memory soon. Additional requests for that address can be either retried shortly in the future, or enqueued to wait for the original request to return.

Unfortunately, as discussed in Section 3.3.4, directories within the network are significantly limited by the number of ports available on a BRAM. Thus, extra reads and writes to the directory should be avoided if at all possible.

An alternate solution is to add additional state to routers to mimic greater directory port count without actually requiring additional utilization of the



(a) Order of tile requests for core #1 of tiled matrix multiplication.



(b) Order of tile requests for core #2 of tiled matrix multiplication.

Core 1	**Core 2**
Req A1[0]	Req A1[0]
Req B1[0]	Req B2[1]
Req A1[1]	Req A1[1]
Req B1[1]	Req B2[1]
…	…
Req A1[255]	Req A1[255]
Req B1[255]	Req B2[255]
// compute	// compute
Req A2[0]	Req A2[0]
Req B3[0]	Req B4[0]
// pattern continues	// pattern continues

(c) Timeline of requests made from the two cores, with concurrent requests for the same data highlighted.

Figure 3.4: Tile request patterns for a sample tiled matrix multiplication kernel. Cores #1 and #2 both request the same tile of matrix A at the same time. Because the two cores execute concurrently, both cores request identical addresses at very nearly the same time.

directory BRAMs. This state can instead be implemented with a small set of registers, as shown in Figure 3.5.

For each input port to a router, a single register is added, used to track outstanding requests that arrive on that port. When a request arrives at its home node, all of the *outstanding registers* at that router are checked to see if any matching requests (requests to the same address) are currently outstanding. If there are no outstanding requests found, routing continues as normal (directory lookup occurs, packet is redirected to either memory or to compute node containing data). If there is an outstanding request for the same address, the current request is immediately redirected to the compute node where the data is expected to arrive.

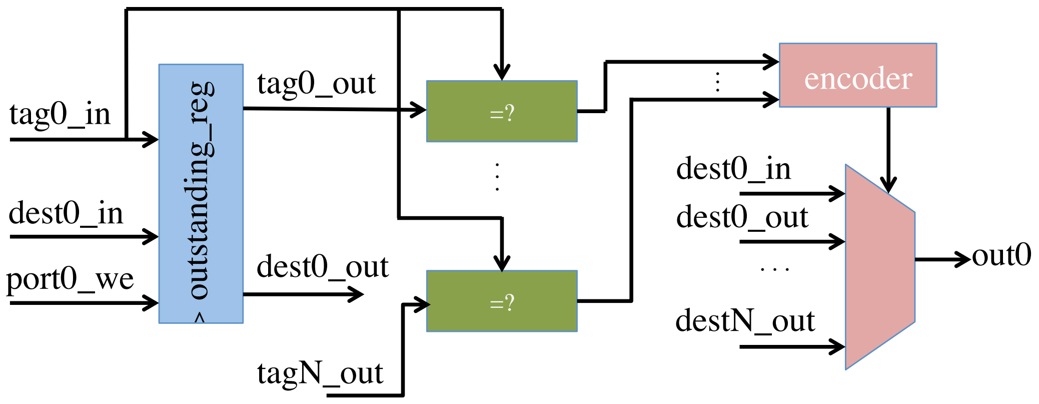


Figure 3.5: Basic block diagram showing a single port of the outstanding register scheme.

Once requests are arrive at the destination compute node, an important issue arises: How long should the request wait for the outstanding data to be returned? For instance, it is possible that upon arrival, the data has already been evicted from the compute node; this would cause the request to wait indefinitely. In a pathologically bad situation involving stale directory information, two compute nodes could deadlock by waiting on each other for the same data that neither has requested.

A simple approach is taken to solve this problem. When a packet is redirected to wait for an outstanding request, it snoops on the input channel of the destination for a configurable number of cycles. After reaching a maximum wait time, the packet times out and is redirected to main memory, effectively preventing deadlock in the system. The queueing and snooping logic at each core is shown in Figure 3.6.

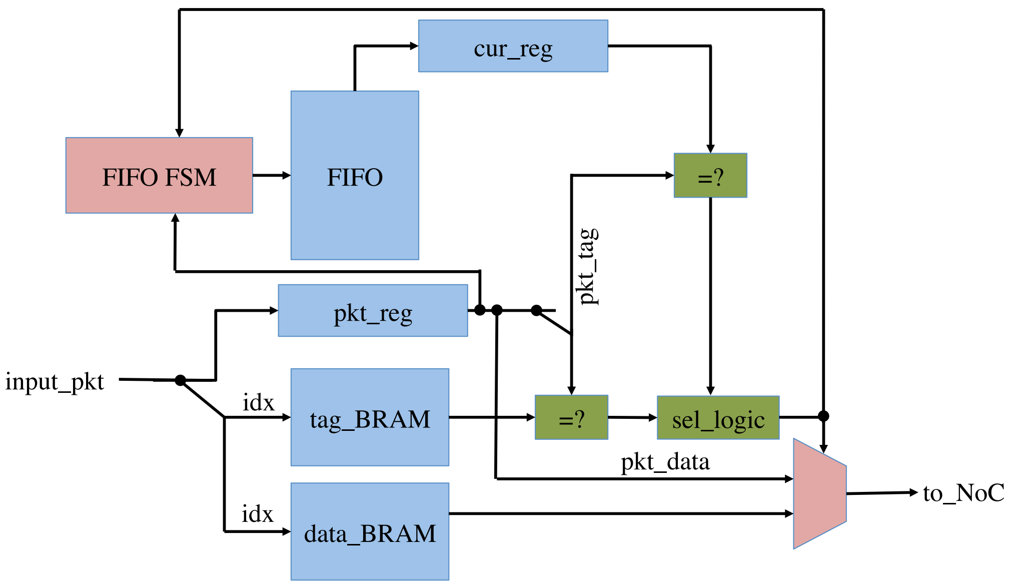


Figure 3.6: A simplified block diagram showing snooping and queuing at a core for outstanding requests.

3.4.2 Directory bypassing

The latency of a NoC can be critically important to application performance. Thus, it is important to reduce system latency as much as possible. Directories add a nontrivial delay of 3 cycles to the path of a router; incurring this additional delay at every router on a path can thus significantly impact performance.

However, packets need not always access a given router's directory. Note, for example, that memory read requests only need to access the directory of the home node; no directory access is needed in other routers. Some packets (e.g., memory writes) never access the directory system. In general, a given packet will require access to at most one directory on its path from source to destination.

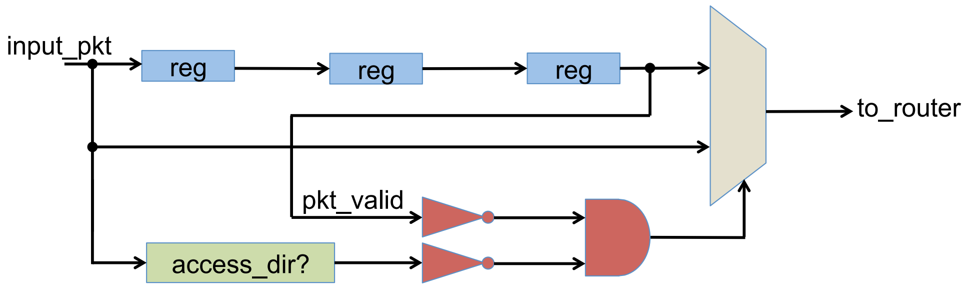


Figure 3.7: Directory bypassing for a single port.

Thus, by adding bypass paths for packets to skip the directory (shown in Figure 3.7), the latency of an individual packet can be significantly improved. A packet requires 3 cycles to pass through an uncongested router; thus, there is potential for a 50% improvement in latency (from 6 to 3 cycles) for packets that do not access directories. Packets which do require directory access incur only a 3-cycle penalty at a single router.

Bypassing is quite simple to implement. Upon arrival at the router, combinational logic is used to determine whether or not the packet needs to access the directory; this depends on the destination, type, and home node of the packet. If the packet does not need to access the directory and there is no valid packet at the final directory stage, the packet is immediately forwarded. Note that the packet must also be cancelled at the input (logic not shown in Figure 3.7) in order to avoid sending duplicate packets through the network.

3.5 Integrating FCUDA with the NoC

As stated previously, the goal of this work is to integrate this NoC with the FCUDA compilation flow. This is much easier said than done; the cores generated by FCUDA require several changes in order to be connected to the NoC and make use of the directory system. As an artifact of using a high-level synthesis tool to perform the transformation from C to RTL, the generated cores can only implement what can be described at the C level. However, in order to interact with the NoC, cores must be able to handle behavior that cannot be described in synthesizable C code: specifically, the NoC makes asynchronous requests for data that the cores must be able to respond to. Transformations to the cores are made at the C level and external logic is added to each core in order to make integration with the network and directory possible. The major changes required are detailed in the following sections.

3.5.1 Memory port combining

FCUDA cores almost always read and write memory in the course of kernel execution. For example, in an implementation of matrix multiplication, the two input matrices and single output matrix will be stored in memory. A programmer implementing matrix multiplication in CUDA would therefore use three pointer parameters in the device function, as seen in Figure 3.8.

```
void matrixMul ( float * A, float * B, float * C ) {
 // use A, B, C to do matrix multiplication
}
```

Figure 3.8: Ordinary matrix multiplication kernel.

FCUDA utilizes Vivado to translate CUDA cores to synthesizable RTL. Thus, each portion of the original C code (including function parameters) must eventually be translated into physical hardware. Vivado can translate function parameters into several different defined interface types; for memory ports, we choose to use the `ap_bus` protocol, a simple and general bus communication protocol that can be trivially connected to a NoC router port. Since Vivado implements each memory parameter as a *separate* `ap_bus` port, a kernel with three memory parameters will require a total of three NoC ports for every core in the system. This effectively triples the number of NoC that must be available in the system. Unfortunately, NoC routers do not scale well with the number of input ports per router, and get significantly more expensive with even a single additional port.

There are two potential solutions to this problem. One solution is to build a custom arbiter and multiplexor system that sits just outside of the FCUDA core. This extra logic could arbitrate access from many memory ports to a single NoC router port. Thus, to the high-level synthesis tool, the three interfaces are separate; outside of HLS, custom hardware is implemented in order to reduce the number of required router ports. As custom hardware is generally difficult to design, debug, and maintain, a software solution is instead chosen.

Instead of a hardware solution, a second solution is to take advantage of the high-level synthesis tool—already used as part of the FCUDA flow—and offload the work of arbitration to Vivado. A simple source-level transformation is sufficient to combine several memory ports into one. The matrix multiplication kernel shown above can trivially be transformed into the code seen in Figure 3.9.

```
void matrixMul(float * memport) {
  float * A = &memport[0];
  float * B = &memport[4096];
  float * C = &memport[8192];
  // use A, B, C
}
```

Figure 3.9: Code transformed to use a single memory port.

3.5.2 Making BRAMs visible

FCUDA translates CUDA shared memory to BRAMs on the FPGA. The directory system is built to take advantage of those BRAMs by redirecting off-chip requests to on-chip sources: FCUDA BRAMs that cache read-only memory.

Unfortunately, the cores generated by high-level synthesis are a black box: there is no way to access the data structures within a core. This is expected: it corresponds to the notion of scope in C. And indeed, in CUDA, shared memory is visible only across a single threadblock, not multiple threadblocks. An example of C code using local memories can be seen in Figure 3.10.

In addition to the problem of the FCUDA BRAMs being completely hidden from external view, there is also the problem of asynchronous access: in order for the directory system to work properly, the core must accept requests at any time for its data and respond immediately. In order to model this behavior at the C level, techniques such as interrupts or threads with shared memory must be used; none of these constructs are synthesizable by high-level synthesis tools.

```
void matrixMul(float * memport) {
  float s_a[16][16];  // store one tile of A
  float s_b[16][16];  // store one tile of B
  // compute
}
```

Figure 3.10: Code using local, hidden BRAMs.

Therefore, an additional software transformation is needed. Instead of using local BRAMs, described at the C level as local arrays, these arrays are transformed into function parameters, as seen in Figure 3.11. Using the `ap_mem` interface pragma, Vivado will synthesize a fast interface to these BRAMs while allowing outside logic to also access them.

```
void matrixMul(float * memport, float s_a[16][16],
   float s_b[16][16]) {
  // compute
}
```

Figure 3.11: Code using external BRAMs.

3.5.3 Address mapping

There is a slightly more subtle issue than allowing external access to local memories. Cores generated by Vivado do not need to keep track of a mapping from BRAM index to memory system address; this mapping is inherent in the structure of the program. For example, if a core copies data from off-chip memory into a local BRAM, the exact memory contents that reside in the local BRAM are simply a function of the state of the program; no extra information is required to be stored in order to retrieve and/or verify this data.

However, the NoC *does* require this mapping information for correct operation. The NoC directory requires two major features to be implemented by each core: First, given an address, at what index would it be stored within the core's BRAM? Second, is the data within a BRAM valid for a particular address (or does it correspond to some other address which happened to map to the same index)?

The second issue of verification can be solved by storing a tag entry for each data BRAM entry. When a memory request returns to the core, external logic can store which address was requested in a small FIFO. On the next core BRAM write, the association between BRAM index and tag is stored.

The first issue (mapping) is somewhat more difficult to solve. The mapping of an index to address could depend on both parameters of the kernel (e.g. number of threadblocks, threadblock dimensions, etc.) as well as the precise current state of the core. For example, if a BRAM is written multiple times in a single kernel, then it could be impossible to determine a mapping function statically. There are two potential solutions to this problem: the mapping can be determined using extra combinational logic, or extra memory could be used to track the mapping.

We first consider using extra memory to track the mapping as the data within the FCUDA BRAM changes. This solution requires an additional data structure to track the mapping of address to index. For example, an address bit field could be used to index into a secondary array; the secondary array stores the index into the data array. However, using an extra BRAM to store this mapping is not ideal: BRAMs are limited on an FPGA. Additionally, this scheme can lose information over time: for example, if multiple addresses map to the same 'tag BRAM' entry but *separate* data BRAM indexes, this scheme will only be able to track the most recently update address.

For some kernels, we can instead determine a static mapping function that can be trivially implemented in logic. This scheme is less general as it does not work for all kernels, but is advantageous because it avoids the extra memory cost and is able to perfectly map a memory address to a BRAM index (with no loss of information). As an example, the mapping function for matrix multiplication is given in Figure 3.12.

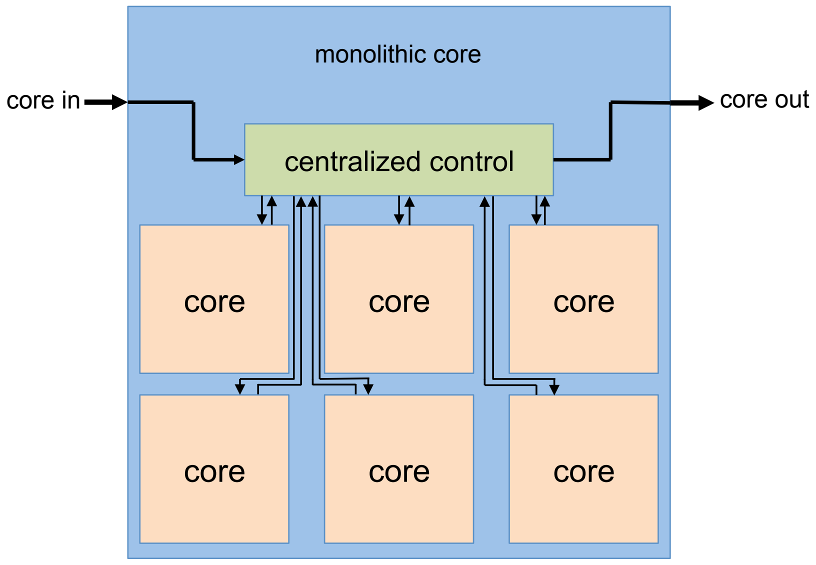
```
int get_index(int address) {
  return (address % BLOCKDIM_X) + BLOCKDIM_X *
     (((address) /
    (BLOCKDIM_X * GRIDDIM_X)) % BLOCKDIM_Y);
}
```

Figure 3.12: Mapping function from memory address to BRAM index for tiled matrix multiplication.

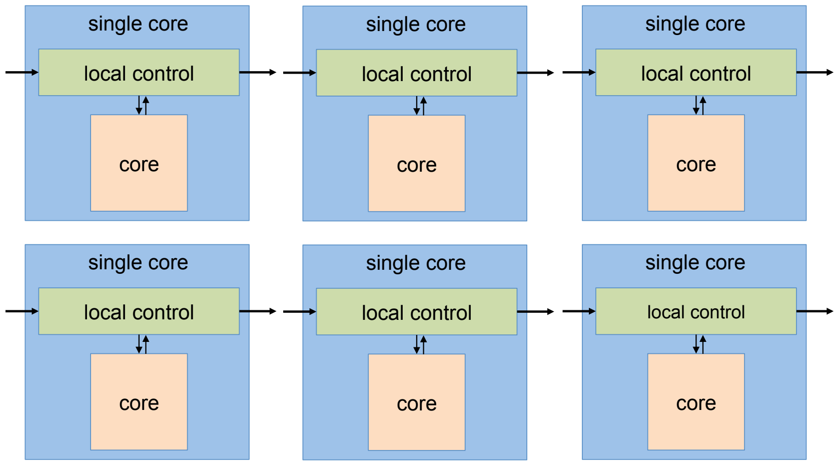
3.5.4 Decentralized control

The final transformation that must occur involves the way that FCUDA generates multicore kernels. The output of FCUDA is a single C file, which is eventually synthesized by HLS into a single opaque top-level module that includes several core instantiations. These core instantiations are internal to the top-level module and cannot be accessed. Thus, in order to connect these cores to the NoC, they need to be split into separate files at the C level and synthesized separately; an image showing the necessary changes is presented in Figure 3.13. While the top-level module originally generated by FCUDA includes centralized control (this control logic is used to map the computation onto the separate cores), for decentralized cores the control must be distributed - each core must have its own control, able to operate completely independently.

This transformation can be implemented by simply generating a single C file per core, rather than a single monolithic C file. A sample of the source-level conversion that this transformation entails is given in Figure 3.14 (original code) and Figure 3.15 (transformed code).



(a) Centralized cores: ordinary output of FCUDA



(b) Set of individual cores, as required by FCUDA+NoC

Figure 3.13: Cores before and after transformation.

```
// matmul.c
void matrixMul( /* args */) {
  // ...
  for(i = 0; i < gridDim.x; i += 3 )  {
    mask1 =  i + 0 < gridDim.x ;
    mask2 =  i + 1 < gridDim.x ;
    mask3 =  i + 2 < gridDim.x ;
    core(mask1);
    core(mask2);
    core(mask2);
  }
}
```

Figure 3.14: Kernel with centralized control.

3.6 NoC Tool Flow

In order to enable improved programmer productivity, the solutions presented above have all been automated and integrated into a single cohesive toolchain. Figure 3.16 shows the work flow for a user of the system from CUDA code to a running simulation or synthesis. This section details the transformations that are made and what is required of the user in order to complete the flow.

3.6.1 CUDA code

The input to the system, as with FCUDA, is CUDA code, written to be executed on an NVIDIA GPU. No extra annotations are required on the part of the programmer in order to use the flow.

```
// core1.c
void matrixMul_core1(/* args */) {
  for(i = 0; i < gridDim.x; i += 3 )  {
    mask =  i + 0 < gridDim.x ;
    core(mask);
  }
}
// core2.c
void matrixMul_core2(float * memport, float
   s_a[16][16], float s_b[16][16]) {
  for(i = 0; i < gridDim.x; i += 3 )  {
    mask =  i + 1 < gridDim.x ;
    core(mask);
  }
}
// core3.c
void matrixMul_core3(float * memport, float
   s_a[16][16], float s_b[16][16]) {
  for(i = 0; i < gridDim.x; i += 3 )  {
    mask =  i + 2 < gridDim.x ;
    core(mask);
  }
}
```

Figure 3.15: Kernel with decentralized control.

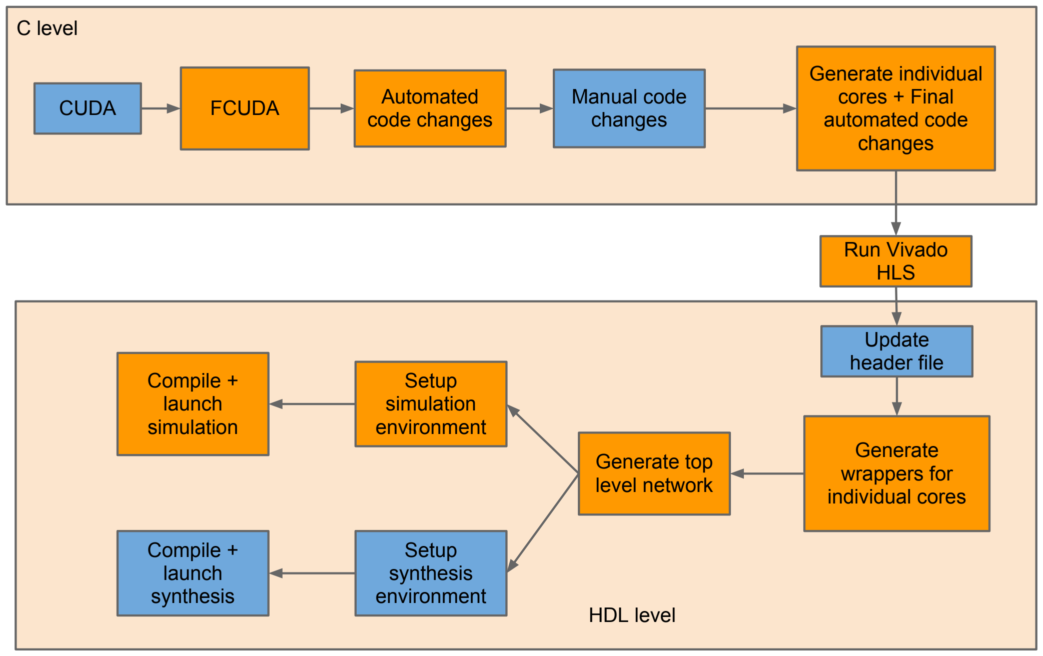

C level
CUDA
FCUDA
Automated code changes
Manual code changes
Generate individual cores + Final automated code changes
Run Vivado HLS
Update header file
Generate wrappers for individual cores
Generate top level network
Setup simulation environment
Compile + launch simulation
Setup synthesis environment
Compile + launch synthesis
HDL level

Figure 3.16: System flow. Orange tasks are completely automated; blue tasks require some user intervention.

3.6.2 FCUDA

Given a few required user inputs, such as the input kernel name and the number of cores desired, FCUDA is automatically launched and generates C code suitable for high-level synthesis.

3.6.3 Automated code changes

In order to support communication with the NoC, several changes must be made to the code prior to synthesis. With minimal user input, these changes can be made automatically. For example, BRAMs are automatically converted from local memories to top-level function parameters (see Section

3.5.2); this requires the user to specify which BRAMs need to be externalized.

These changes are currently made by programs external to FCUDA and were written in this way for convenience and development speed; integrating these changes into FCUDA itself will make the process significantly more stable.

3.6.4 Manual code changes

At this point in the flow, a small number of manual code changes are required to be done by the user. The following changes are required at the C level:

1. The user must create the memory access pointers necessitated by combining memory ports (see Section 3.5.1).

2. The user must delete existing local BRAM instantiations.

Note that once the changes mentioned in Section 3.6.3 are integrated into the FCUDA source-to-source compiler, the manual changes listed here will no longer be necessary; the process will flow completely automatically.

3.6.5 Generate cores

As discussed in Section 3.5.4, in order to connect FCUDA cores to the NoC, the cores must be separated into independent cores rather than contained in a single module. This transformation is done automatically.

3.6.6 Run Vivado

After cores are generated at the C level, a high-level synthesis tool, Vivado, is automatically run, generating several RTL cores. This process is completed with no user intervention.

3.6.7 Network configuration

Almost all parameters of the NoC are configured via a Verilog header file. While most parameters are automatically generated, the user must provide some key parameters in order for the system to work correctly. The user is also required to specify the BRAM indexing function as discussed in Section 3.5.3. Finally, the user must also specify the network topology and layout.

3.6.8 Core wrappers

RTL-level wrappers are automatically generated for each compute core generated by Vivado. This is required in order to enable the cores to communicate with the NoC. Each core wrapper instantiates the required shared BRAMs for that core, instantiates the required network interface and arbitration hardware, and connects all these together in a single top-level module. A high-level block diagram of a wrapper is shown in Figure 3.17; the two critical portions are the BRAM controller (BCTRL), which allows the NoC to access the core BRAMs, and the network interface arbiter (NARB), which arbitrates access to the output NoC line between the core and the BCTRL module.

3.6.9 Top-level network

The final step before simulation or synthesis is to generate a top-level network. Using the user-provided topology, an entire network is generated, including the necessary routing tables for packets to flow correctly. All compute cores are connected to the network, as well as a memory controller. The top-level module additionally includes the necessary pinouts for a connection to off-chip DDR memory.

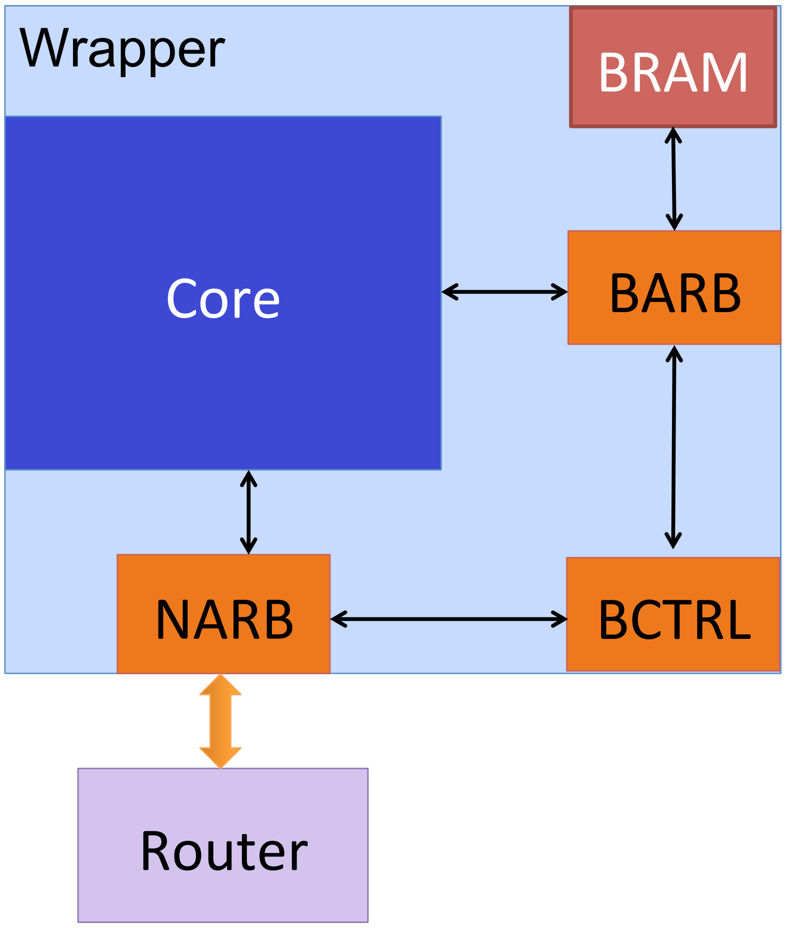


Figure 3.17: High-level block diagram of single core and wrapper.

3.6.10 Simulate

The simulation flow is essentially complete and requires little user intervention. A basic testbench instantiates the network and connects it to a simulated off-chip memory. The user may specify data in order to initialize memory; then, all necessary files are compiled automatically and the simulation is launched.

3.6.11 Synthesis

The synthesis portion of the flow is still incomplete; as such, the process of synthesis must be completed by the user. However, the critical portion of the system - the NoC router and directory - has been confirmed to pass through low-level synthesis, place and route, mapping, and timing analysis. Thus, synthesis should be a fairly straightforward process (and, given time, can be automated).

3.6.12 Ideal flow

Requiring user intervention in the NoC project flow is obviously not ideal; the flow should be as automatic as possible. Given enough time, almost all of the tasks required to be completed by the user can be automated. With additional development on the NoC toolchain, the user will eventually only be required to insert a small number of annotations in the original source code, and define the BRAM indexing functions mentioned in Section 3.6.7. All other portions of the flow can be automated.

3.7 Examples

Included here are a few walk-through examples detailing the operation of the directory protocol.

3.7.1 Directory miss

Consider a small system with three compute nodes, one memory node, and four routers. Let one node be connected to each router, and the routers be connected in a 2x2 mesh. Such a system can be seen in Figure 3.18.

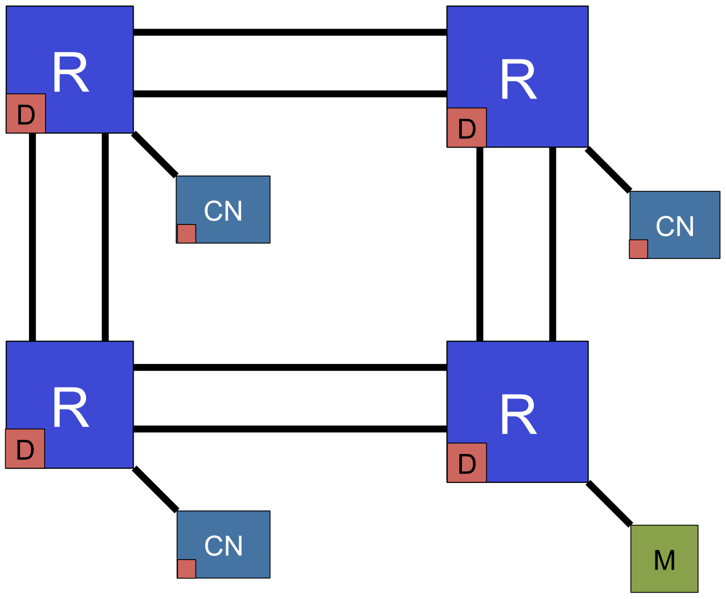


Figure 3.18: Sample network of 4 routers and 3 compute nodes, with 1 memory node

This example will walk through the sequence of events that occurs when a request is made and the request cannot be resolved by the NoC directory.

1. Node A requests an address, say, address `0x1234`. This request is translated from a Vivado HLS ap_bus request [27] to a NoC packet with an initial destination of the home node of address 0x1234. Note that the home node of a given address is determined statically; in this case, assume the home node is determined by the bit field `address[3:2]`, or `01` - router 1 - in this case.

2. The packet is placed on the output network interface of node A, where it is accepted by one of the five network inputs of router 0. Because the packet is destined for router 1, router 0 simply forwards the packet to the next output port and no directory actions occur. In this case, the eastern output port of router 0 will be chosen, placing the packet at the input of router 1.

3. The packet arrives at router 1. The first step in routing is the directory check; by dividing the address into index/tag fields, the directory verifies whether or not the requested data is on chip. In this case, it is not and the packet destination is changed to the memory node (node M). The packet is then routed through router 1 and router 3 on its path to reach M.

4. At node M (the memory controller), the request is enqueued, sent to the off-chip memory, and data eventually returned. In order to update the directory system and tag BRAMs, the response from the memory controller must include the address corresponding to that data. For this reason, a FIFO stores the memory address, along with the eventual NoC destination port, to send the request back through the network.

5. When the data returns from off-chip memory, the memory controller finally sends two packets in response to the network. One packet is destined for the source node – the node that requested the data in the first place. The second packet is sent to the home node for the requested address so that the directory may be updated.

3.7.2 Directory hit

This example walks through the process of a directory hit.

1. As in the above example, say node A requests address `0x1234`. The initial request is sent to the home node.

2. At the home node, the directory lookup occurs as above. In this case, the directory returns a 'hit'. Instead of being redirected to memory, the request is instead redirected to the node containing the data corresponding to address `0x1234` (as returned from the directory).

3. Say the directory specifies that node C contains the data. The packet destination and next hop is changed to direct the packet through the network to node C.

4. When the packet arrives at node C, the packet expects the node to contain the data corresponding to address `0x1234`. However, the data could potentially not be at this node; it could have been very recently overwritten, for example. Thus, a lookup involves: (1) determining the BRAM index to look up (see Section 3.5.3) and (2) comparing the resulting tag to the input tag.

5. If there is a tag match (hit), then the data has been found on-chip. A response can be sent to the originating node (A) containing the data. If there is no hit at the node, the packet is simply redirected to memory and the request continues as in the previous example.

3.7.3 Outstanding array hit

This example discusses the sequence of events that occurs when two requests for the same address are issued at nearly the same time.

1. As above, assume core A requests address `0x1234`. At the same time, core B requests the same address.

2. The request from core B arrives at the home node (router `1`) first. As described previously, each input port to the router has a corresponding register which tracks outstanding requests which arrived on that port. Because router `1` is the home node for this request, the register corresponding to the port on which the request from core B arrived is updated to denote an outstanding request for address `0x1234`. The request travels through the router and is eventually rerouted to the memory node to fetch the data.

3. The request from core A arrives at home node - also router `1`. When a request arrives at its home node, the registers which track outstanding requests are all checked for outstanding requests of the same address. In this case, the packet sent by core B has set one of these registers to denote that there is an outstanding request for address `0x1234`. The packet from core A is then redirected to core B; at core B, the request snoops on the input bus until a fixed timeout is reached.

4. The memory controller sends its response to core B, containing the data at address `0x1234`. The request from core A notices this response

on the NoC input to core B and uses the data in the response packet to respond to core A.

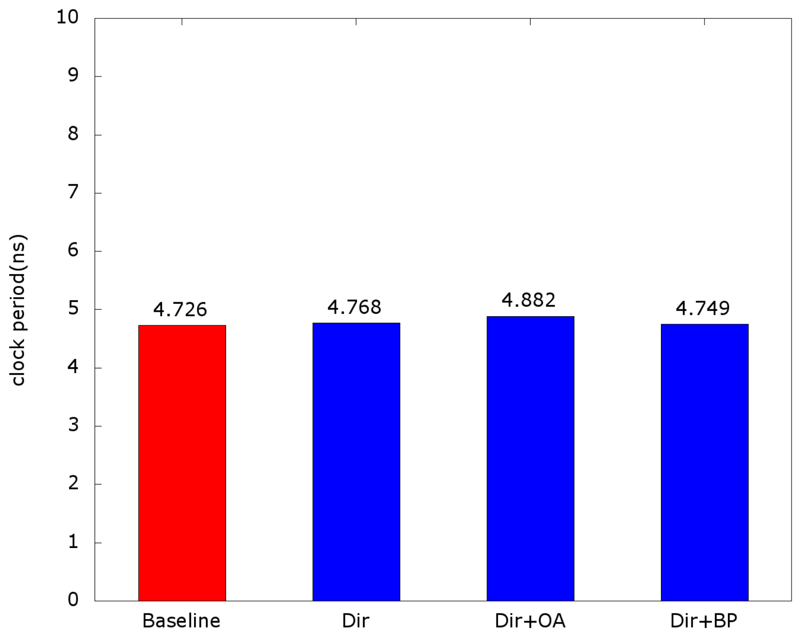
CHAPTER 4

EXPERIMENTAL RESULTS

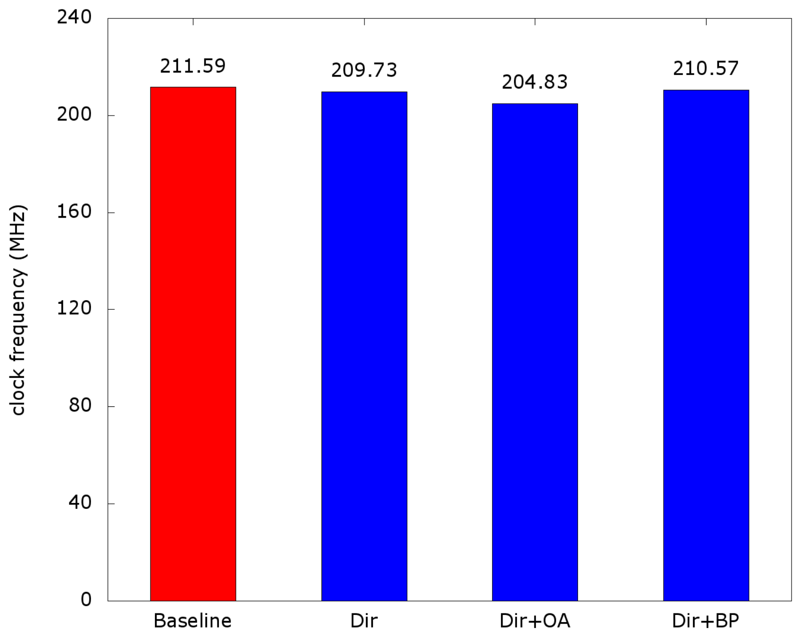
Experimental results were gathered in several ways in order to quantify the characteristics of the NoC and directory system. First, area and frequency results were obtained using reports from ISE 14.1, Xilinx's synthesis tool. Cycle-accurate simulation results were obtained using ModelSim simulations of small systems for two benchmarks. Finally, comparisons to previous work are calculated for a larger system using area and performance models. Benchmarks used include tiled matrix multiplication and 1D convolution. For RTL simulations, input sizes are limited to 48x48; in the calculated portion of the results, the input size is significantly larger.

4.1 Synthesis Results

ISE 14.1 was used to synthesize, map, place, and route the design for a Xilinx Virtex 7 690T FGPA, a large FPGA with 106,000 slices, 3600 DSP blocks, and 2940 18Kb BRAMs. Because portions of the tested system are not designed to be synthesized (in particular, the memory controller), the main components of the system were synthesized separately.



(a) Change in clock period with various directory features enabled. Baseline has no directory.



(b) Variation in frequency with directory features enabled/disabled.

Figure 4.1: Router operating frequency and period with various features enabled. *Key: Baseline: No directories; Dir: basic directories enabled; OA: Outstanding request tracking enabled; BP: Bypass enabled.*

4.1.1 Router synthesis

First, extensive results for synthesis of the NoC routers are presented. Figures 4.1a and 4.1b show the variation in clock period and frequency with various directory parameters enabled in the synthesized router. ISE is able to exceed the target clock period constraint of 5 ns on every design explored; there is very little variation. When this design was synthesized for a smaller Virtex 5 FPGA, the synthesis tool had less freedom to trade area for speed; on the Virtex 5, the router required significantly fewer slices but also had a much lower maximum frequency.

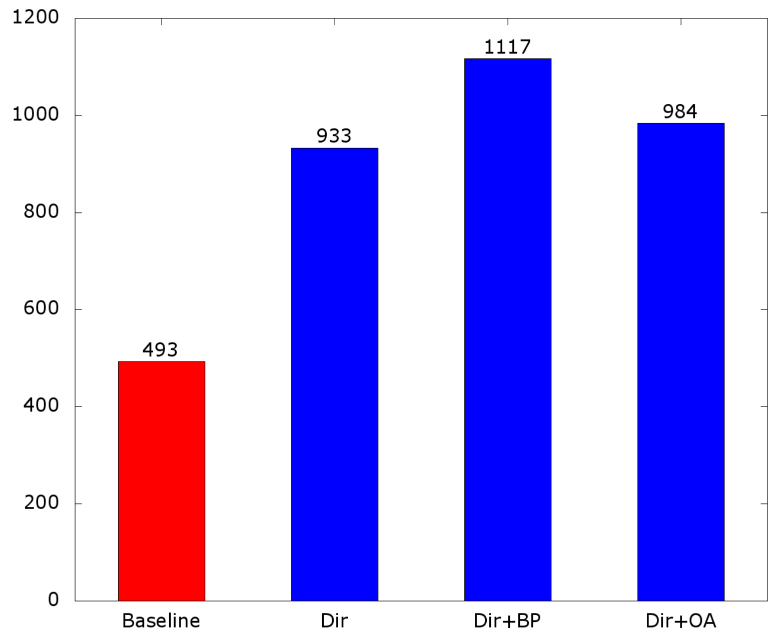


Figure 4.2: Number of slices required to implement baseline router, compared to router with directory and various enabled features.

Figure 4.2 shows the area overhead of various features for directory-based routing for a 5-input router. The baseline shown is a NoC router with no directory system; this router requires 493 slices. A basic directory system

Table 4.1: Area and Frequency results for router

System	Slices	Registers	LUTs	BRAMs	Min. period
baseline	493	944	1370	0	4.726 ns
dir128	1002	1751	2480	1	4.86 ns
dir256	933	1746	2464	1	4.768 ns
dir512	956	1741	2483	1	4.964 ns

(256 entries), with no support for request bypassing or for handling outstanding requests, comes at a cost of 440 additional slices, almost double the original number of slices required. Both bypassing and tracking outstanding requests are also expensive, costing 184 and 51 additional slices respectively. Both additional features require a large number of muxes to be added to the design; unfortunately, wide muxes on FPGAs require a large number of LUTs. Not included in the slice results are the additional BRAMs required by directory-based implementations; a 256-entry directory requires a single BRAM, while the baseline router does not require any BRAM resources.

Finally, Table 4.1 shows variation in synthesis results as the directory size changes; all are within 10% of each other. This is expected: increased directory size requires increased memory utilization, but the design is essentially the same. In this case, a single BRAM is large enough even for a 512-entry directory.

4.1.2 Core synthesis

Table 4.2 shows the area overhead for two versions of a single matrix multiplication core, again synthesized for the same Virtex 7 FPGA. The first is a baseline FCUDA core with no NoC support. The second version supports interfacing with the NoC, including the external logic required to respond to incoming network requests. The area overhead to enable network support for a single core is approximately 6% in terms of the number of slices.

Note that the version supporting NoC requests requires significantly more BRAM resources than the baseline design; this is for two reasons. First, in order to allow concurrent access to BRAMs by the core and the NoC, dual-ported BRAMs are used to replace single-ported BRAMs in the original design. Single-ported BRAMs could be used with a slight decrease to performance due to arbitration. Second, the NoC-capable design is required to track the tag of every entry in every scratchpad memory; this essentially doubles the original BRAM requirement of the core. For this kernel on this FPGA, the factor limiting the number of cores will be the number of slices, not the number of BRAMs; thus, no efforts were made to decrease BRAM utilization.

Table 4.2: Area overhead for separating FCUDA cores

Version	**DSP**	**LUT**	**Registers**	**Slices**	**BRAM**
Baseline	17	3474	1774	1320	5
NoC support	17	3873	1949	1399	19
Difference	0	11.48%	9.86%	5.98%	280%

4.2 Simulation Results

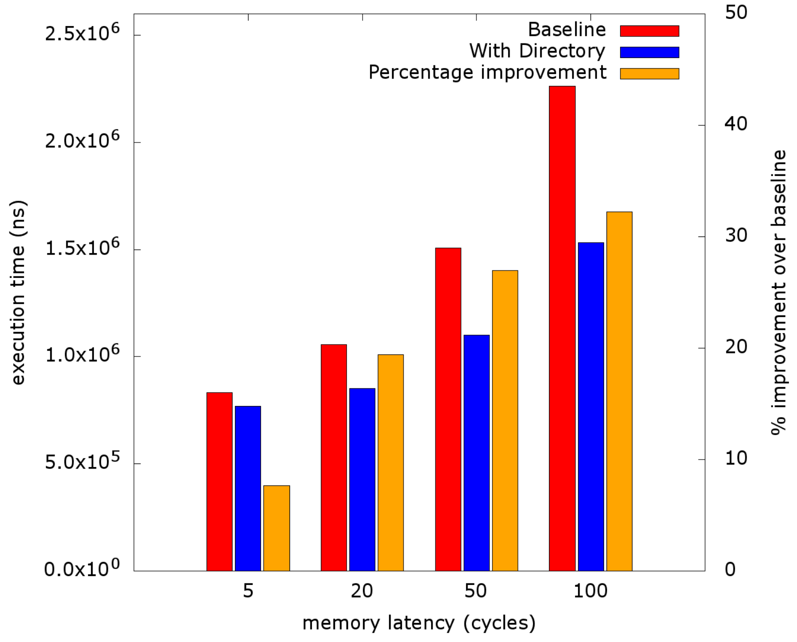
This section presents some cycle-accurate simulation results for some small benchmarks; it aims to show the benefit of the directory system by comparing to a baseline NoC without directories enabled.

4.2.1 Impact of directories

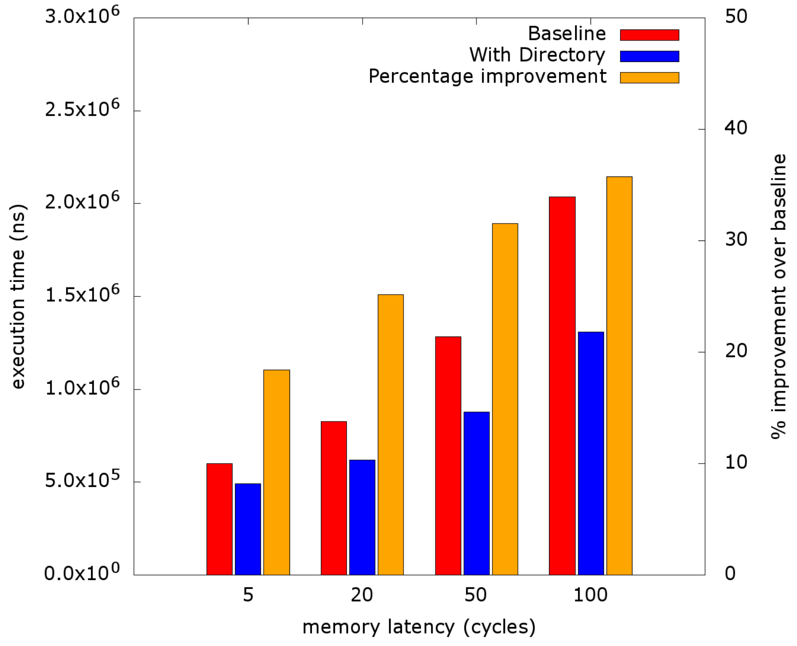
Figures 4.3 and 4.4 show the results of 3 and 9-core simulation runs for two small benchmarks: 1D convolution and matrix multiplication. Each graph shows a baseline system (NoC without directory) compared to a directory-enabled system; results are shown for various off-chip memory latencies. In orange, the percentage improvement of the directory-enabled system over the non-directory system is shown.

For the 1D convolution kernel shown in Figure 4.3, the sharing pattern is quite simple; halo data from a neighboring core is almost guaranteed to be found on-chip. For this kernel, relative performance improves as memory latency increases: the cost of a directory hit stays constant while the cost of a memory access increases.

Figure 4.4a shows the impact of directories on a 3-core matrix multiplication kernel; the improvement is limited (and is even slightly negative for a 5-cycle memory latency). The sharing pattern of matrix multiplication is significantly more complex than that of 1D convolution. Thus, while a 3-core convolution implementation saw significant improvement with directory tracking, 3-core matrix multiplication sees little improvement—with so few cores, there is little locality to be exploited. In contrast, Figure 4.4b shows very significant performance improvements due to directory tracking. The 9 cores act together as a 3x3 "super-tile" and are able to share essentially every request.

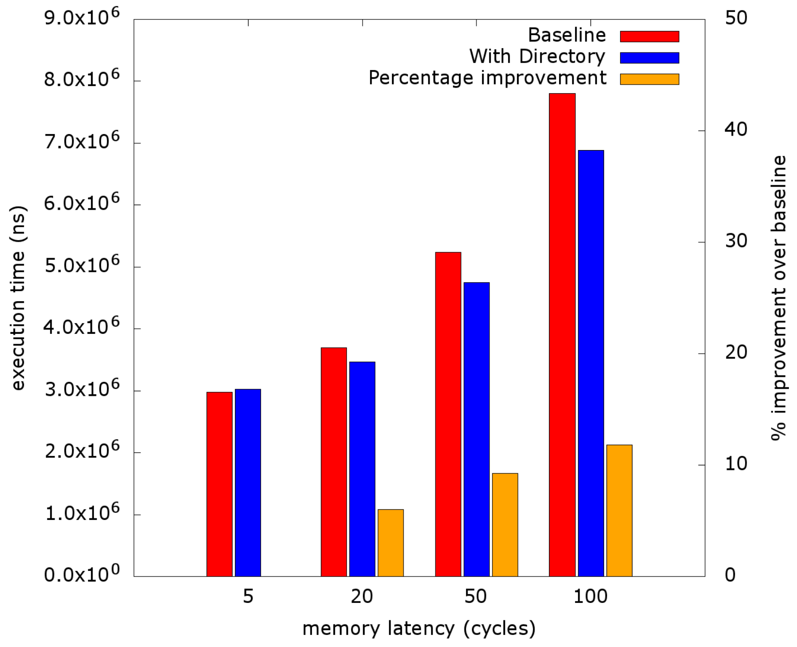


(a) Execution time vs. memory latency: 3 core 1D convolution.

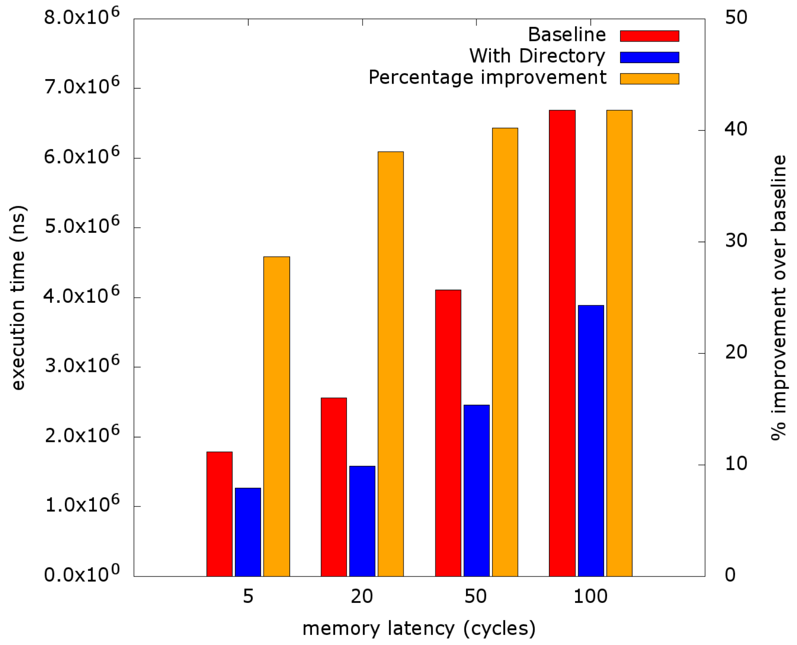


(b) Execution time vs. memory latency: 9 core 1D convolution.

Figure 4.3: Execution time vs. memory latency for 48x48 tiled 1D convolution.



(a) Execution time vs. memory latency: 3 core matrix multiplication. Note that when latency is 5, the directory implementation is slightly slower!



(b) Execution time vs. memory latency: 9 core matrix multiplication.

Figure 4.4: Execution time vs. memory latency for 48x48 tiled matrix multiplication.

Table 4.3: Impact of bypassing and outstanding request tracking on performance

kernel	bypassing	outstanding requests	time
conv1d-3core	no	no	877989
conv1d-3core	yes	no	863544
conv1d-3core	no	yes	877071
conv1d-3core	yes	yes	851454
matmul-3core	no	no	3746578
matmul-3core	yes	no	3743777
matmul-3core	no	yes	3531429
matmul-3core	yes	yes	3469686

Table 4.3 describes the impact that bypassing and handling outstanding requests has on the performance of the directory system. Interestingly, bypassing is not particularly effective. While bypassing does decrease the latency of individual packets, it provides little overall benefit: reaching the memory controller several cycles earlier is not helpful if there are already several enqueued requests!

The memory access pattern significantly affects the impact of outstanding request tracking. For example, in the convolution kernel, requests for identical addresses rarely overlap; thus, tracking outstanding requests provides almost zero performance improvement. In contrast, in matrix multiplication, it is common for requests for the same data to occur at the same time; thus, tracking outstanding requests can be quite beneficial.

In order to examine the impact of tracking outstanding requests more closely, the time a packet may spend waiting for an outstanding request to return was made configurable. The result of sweeping the maximum timeout from 50 to 1000 cycles is shown in Figure 4.5. With 3 cores, there is relatively little benefit to a higher timeout; cores quickly become unsynchronized and outstanding requests cannot be merged. With 9 cores, there is significant latency improvement as the timeout increases: this means that the cores generally stay synchronized and a significant number of outstanding requests are merged in-network.

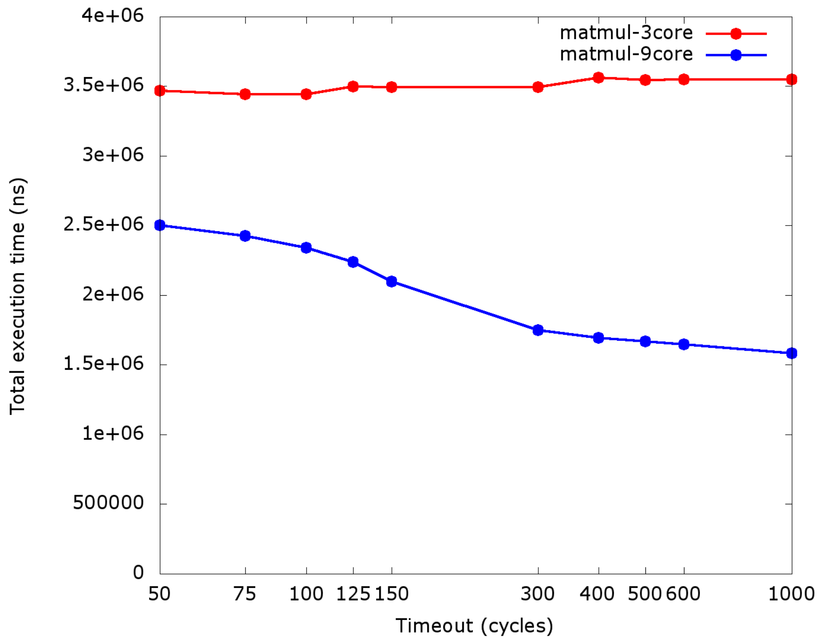


Figure 4.5: Execution time vs. outstanding request timeout. *X axis shown on log scale.*

Finally, Figure 4.6 shows the impact of directory size on the performance of the convolution kernel. For the 3-core implementation, increasing the directory size beyond 32 is not beneficial; the row-wise synchronization of the kernel means that for a 3-core system using 16x16 output tiles, only 48 addresses are relevant in the system at a given time. A 4-router NoC with 32-entry directories can track 128 total addresses, more than meeting the capacity required for optimum performance. For the 9-core system, a 128-entry directory is the optimal size. While the directory is significantly overprovisioned (1152 total entries; only 144 addresses are relevant), smaller directory sizes lead to aliasing and loss of information about on-chip addresses.

Directory size has very little impact on the performance of the matrix multiplication kernel and is not shown here; most requests in this kernel are handled by the outstanding request mechanism.

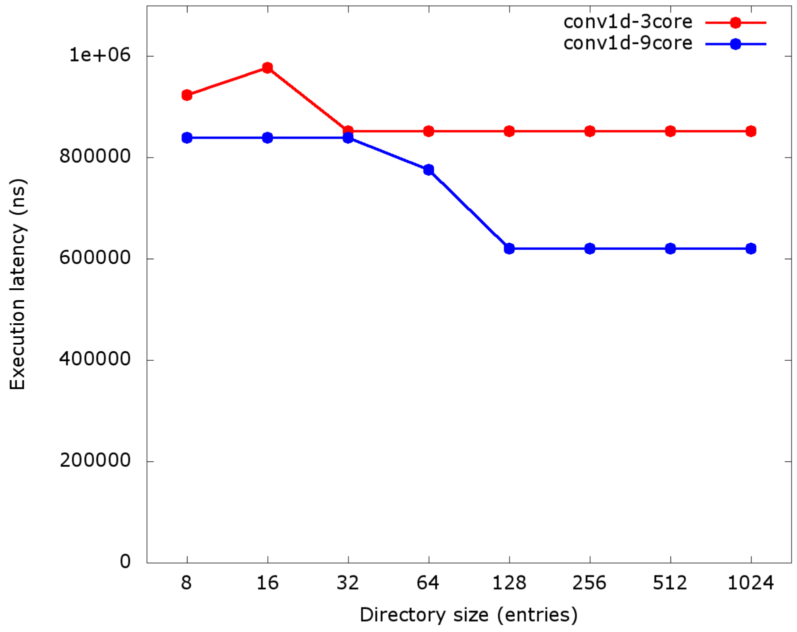


Figure 4.6: Execution latency as a function of directory size for 3 and 9-core versions of the 1D convolution kernel.

4.3 Performance Model

In order to gain a more complete picture of the performance of the NoC-based system, a performance and area model was derived from the previous simulation and synthesis results for the matrix multiplication kernel. This model was used to compare with results achieved by a bus-based model of FCUDA.

A large 2048x2048 matrix multiplication computation is considered on the Virtex 7 690T FPGA. This particular kernel is limited by slices; DSPs and

BRAMs are both plentiful, as shown in Table 4.2. The computation latency for a single tile can be derived from a simulation of a smaller system; with a 16x16 tile, the latency is 765,056 cycles.

Using the synthesis results from above and leaving a small amount of area for a memory controller, 81 baseline FCUDA cores can fit into the 108,000 available slices on the Virtex 7 FPGA.

With a 5-input router size of 1021 slices for the Virtex 7, 55 cores can fit onto the NoC. This assumes an arrangement of 29 routers filling the remaining slices; routers on the edge of the mesh have either 2 or 3 available free ports; utilizing these ports allows more slices to be allocated to computation instead of the network.

Memory latency is modeled slightly differently for the two systems. For the bus-based system, the total latency includes the memory latency, the latency for reading and writing to the bus, and the delay of a memory controller. The NoC-based model includes the memory controller and DRAM latencies as well as the latency required to traverse the network.

The results are shown in Figure 4.7, which shows total execution time as a function of increasing memory latency. For low memory latencies, the bus-based solution is faster than the NoC-based solution. This is mainly due to the overhead required for packet switching of 3 cycles per router; in this 29-node network, the cost of an average directory hit is approximately 50 cycles. For increased latencies, the NoC-based solution shows significantly better performance than the bus-based solution; the crossover point is when directory hits become cheaper than memory accesses.

Three NoC systems are shown. The orange line depicts an ideal NoC which has a 100% directory hit rate - it never incurs the memory delay. For this kernel, the expected directory hit rate is approximately 98% (blue): as with the 9-tile case, the cores essentially act together as a "super-tile" of approximately 7x8 regular tiles. These cores essentially remain in sync, allowing them to achieve a significant amount of temporal locality. Finally, an NoC with a hit rate of only 50% is shown in green.

In a sense, this is intuitive: if memory accesses are free, there is no need to cache any data on chip; it can be found with zero latency off-chip. As memory accesses get more expensive, it is more attractive to keep data on-chip despite the added area cost. This inflection point occurs significantly sooner when the NoC router latency is reduced to 2 cycles, as shown by the purple line in Figure 4.7.

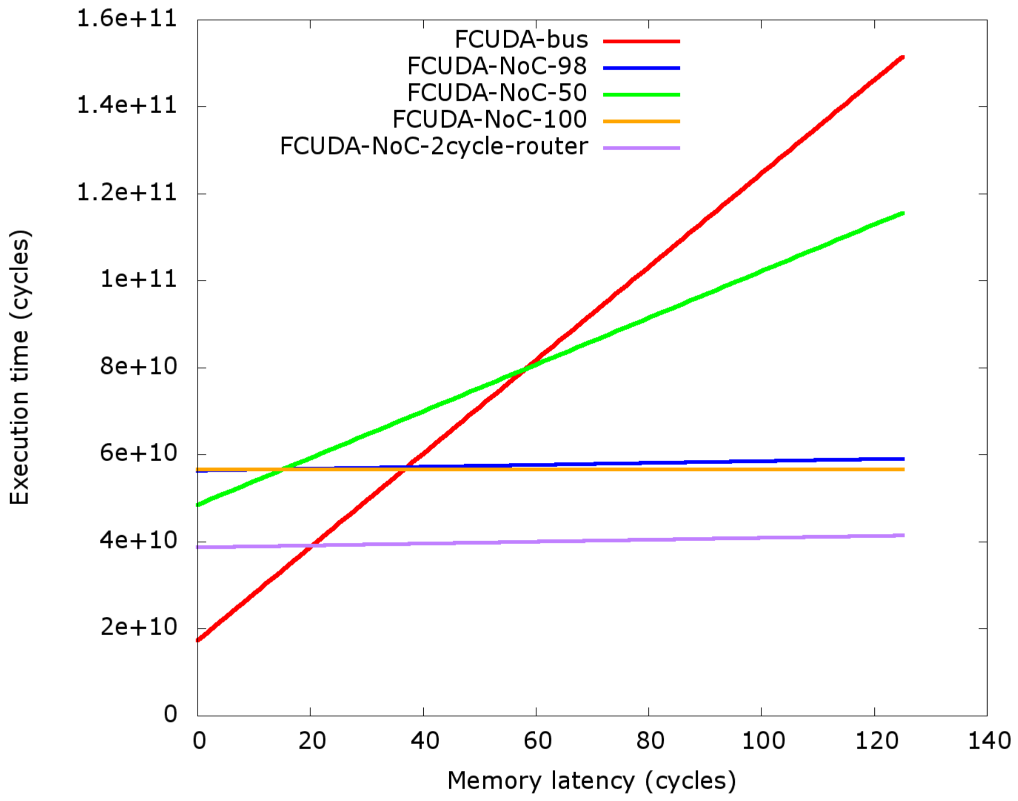


Figure 4.7: Computed comparison with bus-based system.

CHAPTER 5

CONCLUSION

This thesis developed a simplified directory protocol that is readily integrated into an existing open-source Network-on-Chip. The protocol can be used to exploit the spatial and temporal locality implicit in CUDA kernels that exhibit read-only sharing, and can deliver improved overall application performance via interconnected FCUDA cores on an FPGA. The wide range of configurability of the NoC enables users to explore a large design space of directory sizes and enhancements in order to reach the best area-performance trade-off point.

Results show that replacing a traditional bus-based system with directory-enhanced NoC can yield improved system performance when memory latency is high. Network latency is critically important to this conclusion; reducing latency at each router can yield a very significant performance benefit. In addition, the pattern of requests can have a large impact on performance: organizing a group of four cores into a 2x2 square rather than a 4x1 row yields far better locality.

5.1 Future Studies

The Network-on-Chip presents a broad platform for potential future work. There are several interesting avenues that have yet to be explored:

- Clustering of cores within the NoC for improved locality. Similarly, attempt to place directories near to nodes that will access them often; this can reduce round trip time within the network and improve overall performance.

- Shared memory between cores. If two cores access the same data (for example, the same tile in a tiled computation), the data can be stored only a single time in a shared memory. Cores access the shared memory quickly through the network via the directory.

- Additional design space exploration of the directory system. For example, all directories in this work are direct-mapped; what effect would associativity have on performance?

- Explore additional network hierarchies to reduce average path length. For example, a 2D torus topology significantly reduces the cost to cross the network.

- Cache directory information within nodes. For example, on a request, instead of directing the request to the home node first, direct the request to the last node that data was received from.

- Selective directory lookup: in a large network, directory requests may be expensive; is there a benefit to ignoring far-away directory lookups?

REFERENCES

[1] G. Moore, “Cramming more components onto integrated circuits,” *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[2] N. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. Hu, M. Irwin, M. Kandemir, and V. Narayanan, “Leakage current: Moore’s law meets static power,” *Computer*, vol. 36, no. 12, pp. 68–75, 2003.

[3] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc, “Design of ion-implanted MOSFET’s with very small physical dimensions,” *Solid-State Circuits, IEEE Journal of*, vol. 9, no. 5, pp. 256–268, 1974.

[4] D. Geer, “Chip makers turn to multicore processors,” *Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[5] H. Sutter, “The free lunch is over: A fundamental turn toward concurrency in software,” Mar. 2005. [Online]. Available: http://www.gotw.ca/publications/concurrency-ddj.htm

[6] J. Parkhurst, J. Darringer, and B. Grundmann, “From single core to multi-core: preparing for a new exponential,” in *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, ser. ICCAD ’06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1233501.1233516 pp. 67–72.

[7] M. B. Taylor, “Is dark silicon useful?: harnessing the four horsemen of the coming dark silicon apocalypse,” in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC ’12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228567 pp. 1131–1136.

[8] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th annual international symposium on Computer architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000108 pp. 365–376.

[9] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, no. 7, pp. 54–62, Sep. 2005. [Online]. Available: http://doi.acm.org/10.1145/1095408.1095421

[10] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, J. K. Hollingsworth, and M. V. Zelkowitz, "Parallel programmer productivity: A case study of novice parallel programmers," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, ser. SC '05. Washington, DC, USA: IEEE Computer Society, 2005. [Online]. Available: http://dx.doi.org/10.1109/SC.2005.53 p. 35.

[11] NVIDIA, "CUDA C Programming Guide," 2012. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[12] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500

[13] J. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[14] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.-M. Hwu, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, July, pp. 35–42.

[15] P. Zuchowski, C. Reynolds, R. Grupp, S. Davis, B. Cremen, and B. Troxel, "A hybrid ASIC and FPGA architecture," in *Computer Aided Design, 2002. ICCAD 2002. IEEE/ACM International Conference on*, 2002, pp. 187–194.

[16] I. Kuon, R. Tessier, and J. Rose, "FPGA architecture: Survey and challenges," *Found. Trends Electron. Des. Autom.*, vol. 2, no. 2, pp. 135–253, Feb. 2008. [Online]. Available: http://dx.doi.org/10.1561/1000000005

[17] S. Brown and J. Rose, "FPGA and CPLD architectures: A tutorial," *IEEE Des. Test*, vol. 13, no. 2, pp. 42–57, June 1996. [Online]. Available: http://dx.doi.org/10.1109/54.500200

[18] Xilinx Inc., "Virtex-6 FPGA DSP48E1 slice user guide," Feb. 2011. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug369.pdf

[19] Altera Corp., "Enabling high-performance DSP applications with Stratix V variable-precision DSP blocks," May 2011. [Online]. Available: http://www.altera.com/literature/wp/wp-01131-stxv-dsp-architecture.pdf

[20] Xilinx Inc., "7 Series FPGAs memory resources user guide," Oct. 2012. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf

[21] Altera Corp., "Internal memory (RAM and ROM) user guide," Nov. 2012. [Online]. Available: http://www.altera.com/literature/ug/ug_ram_rom.pdf

[22] C. E. LaForest and J. G. Steffan, "Efficient multi-ported memories for FPGAs," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1723112.1723122 pp. 41–50.

[23] M. C. McFarland, A. C. Parker, and R. Camposano, "Tutorial on high-level synthesis," in *Proceedings of the 25th ACM/IEEE Design Automation Conference*, ser. DAC '88. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988. [Online]. Available: http://dl.acm.org/citation.cfm?id=285730.285784 pp. 330–336.

[24] D. D. Gajski and L. Ramachandran, "Introduction to high-level synthesis," *IEEE Des. Test*, vol. 11, no. 4, pp. 44–54, Oct. 1994. [Online]. Available: http://dx.doi.org/10.1109/54.329454

[25] K. Rupnow, Y. Liang, Y. Li, D. Min, M. Do, and D. Chen, "High level synthesis of stereo matching: Productivity, performance, and software constraints," in *Field-Programmable Technology (FPT), 2011 International Conference on*, 2011, pp. 1–8.

[26] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. W. Hwu, "High-performance CUDA kernel execution on FPGAs," in *Proceedings of the 23rd international conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1542275.1542357 pp. 515–516.

[27] Xilinx Inc., "Vivado Design Suite User Guide," July 2012. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug902-vivado-high-level-synthesis.pdf

[28] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, "Autopilot: A platform-based ESL synthesis system," in *High-Level Synthesis*, P. Coussy and A. Morawiec, Eds. Springer Netherlands, 2008, pp. 99–112. [Online]. Available: http://dx.doi.org/10.1007/978-1-4020-8588-8_6

[29] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Öberg, M. Millberg, and D. Lindqvist, "Network on chip: An architecture for billion transistor era," in *Proceeding of the IEEE NorChip Conference*, Nov. 2000. [Online]. Available: http://www.imit.kth.se/\~axel/papers/2000/norchip-noc.pdf

[30] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli, "Addressing the system-on-a-chip interconnect woes through communication-based design," in *Design Automation Conference, 2001. Proceedings*, 2001, pp. 667–672.

[31] L. Benini and G. De Micheli, "Networks on chips: a new SoC paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, 2002.

[32] W. J. Dally and B. Towles, “Route packets, not wires: on-chip inteconnection networks,” in *Proceedings of the 38th annual Design Automation Conference*, ser. DAC ’01. New York, NY, USA: ACM, 2001. [Online]. Available: http://doi.acm.org/10.1145/378239.379048 pp. 684–689.

[33] P. Guerrier and A. Greiner, “A generic architecture for on-chip packet-switched interconnections,” in *Proceedings of the conference on Design, automation and test in Europe*, ser. DATE ’00. New York, NY, USA: ACM, 2000. [Online]. Available: http://doi.acm.org/10.1145/343647.343776 pp. 250–256.

[34] P. Wolkotte, G. J. M. Smit, N. Kavaldjiev, J. Becker, and J. Becker, “Energy model of networks-on-chip and a bus,” in *System-on-Chip, 2005. Proceedings. 2005 International Symposium on*, 2005, pp. 82–85.

[35] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar, “An 80-tile 1.28tflops network-on-chip in 65nm cmos,” in *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, 2007, pp. 98–589.

[36] D. Wentzlaff, “Transferring data in a parallel processing environment,” US Patent US 7 461 236, 12 02, 2008. [Online]. Available: http://www.patentlens.net/patentlens/patent/US_7461236/en/

[37] R. Gindin, I. Cidon, and I. Keidar, “NoC-based FPGA: Architecture and routing,” in *Networks-on-Chip, 2007. NOCS 2007. First International Symposium on*, 2007, pp. 253–264.

[38] M. E. S. Elrabaa and A. Bouhraoua, “A hardwired NoC infrastructure for embedded systems on FPGAs,” *Microprocessors and Microsystems - Embedded Hardware Design*, vol. 35, no. 2, pp. 200–216, 2011.

[39] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins, “Interconnection networks enable fine-grain dynamic multi-tasking on FPGAs,” in *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, ser. FPL ’02. London, UK, UK: Springer-Verlag, 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=647929.740058 pp. 795–805.

[40] B. Sethuraman, P. Bhattacharya, J. Khan, and R. Vemuri, “LiPaR: A light-weight parallel router for FPGA-based networks-on-chip,” in *Proceedings of the 15th ACM Great Lakes symposium on VLSI*, ser. GLSVLSI ’05. New York, NY, USA: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1057661.1057769 pp. 452–457.

[41] A. Ehliar and D. Liu, “An FPGA based open source network-on-chip architecture,” in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, 2007, pp. 800–803.

[42] C. Hilton and B. Nelson, “PNoC: a flexible circuit-switched NoC for FPGA-based systems,” *Computers and Digital Techniques, IEE Proceedings -*, vol. 153, no. 3, pp. 181–188, 2006.

[43] M. S. Papamarcos and J. H. Patel, “A low-overhead coherence solution for multiprocessors with private cache memories,” in *Proceedings of the 11th annual international symposium on Computer architecture*, ser. ISCA ’84. New York, NY, USA: ACM, 1984. [Online]. Available: http://doi.acm.org/10.1145/800015.808204 pp. 348–354.

[44] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon, “Implementing a cache consistency protocol,” in *Proceedings of the 12th annual international symposium on Computer architecture*, ser. ISCA ’85. Los Alamitos, CA, USA: IEEE Computer Society Press, 1985. [Online]. Available: http://dl.acm.org/citation.cfm?id=327010.327237 pp. 276–283.

[45] L. Rudolph and Z. Segall, “Dynamic decentralized cache schemes for MIMD parallel processors,” in *Proceedings of the 11th annual international symposium on Computer architecture*, ser. ISCA ’84. New York, NY, USA: ACM, 1984. [Online]. Available: http://doi.acm.org/10.1145/800015.808203 pp. 340–347.

[46] J. R. Goodman, “Using cache memory to reduce processor-memory traffic,” in *Proceedings of the 10th annual international symposium on Computer architecture*, ser. ISCA ’83. New York, NY, USA: ACM, 1983. [Online]. Available: http://doi.acm.org/10.1145/800046.801647 pp. 124–131.

[47] C. K. Tang, "Cache system design in the tightly coupled multiprocessor system," in *Proceedings of the June 7-10, 1976, national computer conference and exposition*, ser. AFIPS '76. New York, NY, USA: ACM, 1976. [Online]. Available: http://doi.acm.org/10.1145/1499799.1499901 pp. 749–753.

[48] L. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *Computers, IEEE Transactions on*, vol. C-27, no. 12, pp. 1112–1118, 1978.

[49] W. Weber and A. Gupta, "Analysis of cache invalidation patterns in multiprocessors," in *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS III. New York, NY, USA: ACM, 1989. [Online]. Available: http://doi.acm.org/10.1145/70082.68205 pp. 243–256.

[50] S. J. Eggers and R. H. Katz, "A characterization of sharing in parallel programs and its application to coherency protocol evaluation," in *Proceedings of the 15th Annual International Symposium on Computer architecture*, ser. ISCA '88. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988. [Online]. Available: http://dl.acm.org/citation.cfm?id=52400.52442 pp. 373–382.

[51] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," in *Computer Architecture, 1988. Conference Proceedings. 15th Annual International Symposium on*, 1988, pp. 280–289.

[52] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "Limitless directories: A scalable cache coherence scheme," in *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS IV. New York, NY, USA: ACM, 1991. [Online]. Available: http://doi.acm.org/10.1145/106972.106995 pp. 224–234.

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

[54] R. Simoni and M. Horowitz, "Dynamic pointer allocation for scalable cache coherence directories," in *In International Symposium on Shared Memory Multiprocessing.* IPS Press, 1991, pp. 72–81.

[55] K. Alnaes, E. Kristiansen, D. Gustavson, and D. James, "Scalable coherent interface," in *CompEuro '90. Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering*, 1990, pp. 446–453.

[56] M. Thapar and B. Delagi, "Distributed-directory scheme: Stanford distributed-directory protocol," *Computer*, vol. 23, no. 6, pp. 78–80, 1990.

[57] A. Costa, "Boolean functions simplification (logic minimization)," July 2009. [Online]. Available: http://www4.dei.isep.ipp.pt/acc/bfunc/

[58] R. L. Rudell, "Multiple-valued logic minimization for PLA synthesis," EECS Department, University of California, Berkeley, Tech. Rep. UCB/ERL M86/65, 1986. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1986/734.html