15-418 Final Report

Yu-Ching Wu (yuchingw) and Dane Engman (dengman)

May 5, 2024

Contents

1 Summary

Our project aims to simulate a more accurate cache to cache transfer mechanism for cache coherency. To do this, we augmented the CADSS with a system verilog based interconnect simulator. The two simulators communicate over sockets. Our network simulator incorporates a clock divider to create two separate clock domains which reflects modern heterogeneous chip design, and implements 4 separate synthesizable networks, a bus, ring, crossbar, and mesh.

2 Background

An important component to parallel performance is the mechanisms by which cache coherence is maintained. Although limiting true and false sharing of cache blocks is desirable, it is the fastest mechanism for sending data between cores and a performant multicore CPU needs to be able to effectively handle these messages.

With core counts of server class CPUs rapidly rising the nature of the network that communicates updated cache blocks among CPUs becomes increasingly important. A common network used is a ring, which for a 4 core CPU guarantees low levels of area dedicated to the interconnect as well as fairly low latency due to the low number of nodes between any pair of cores. However, this is much more problematic for high core count CPUs since 2 very distant cores could be communicating. With a network like a ring this has an $O(N)$ latency, where a network such as a mesh would have an $O(log_2(N))$ latency.

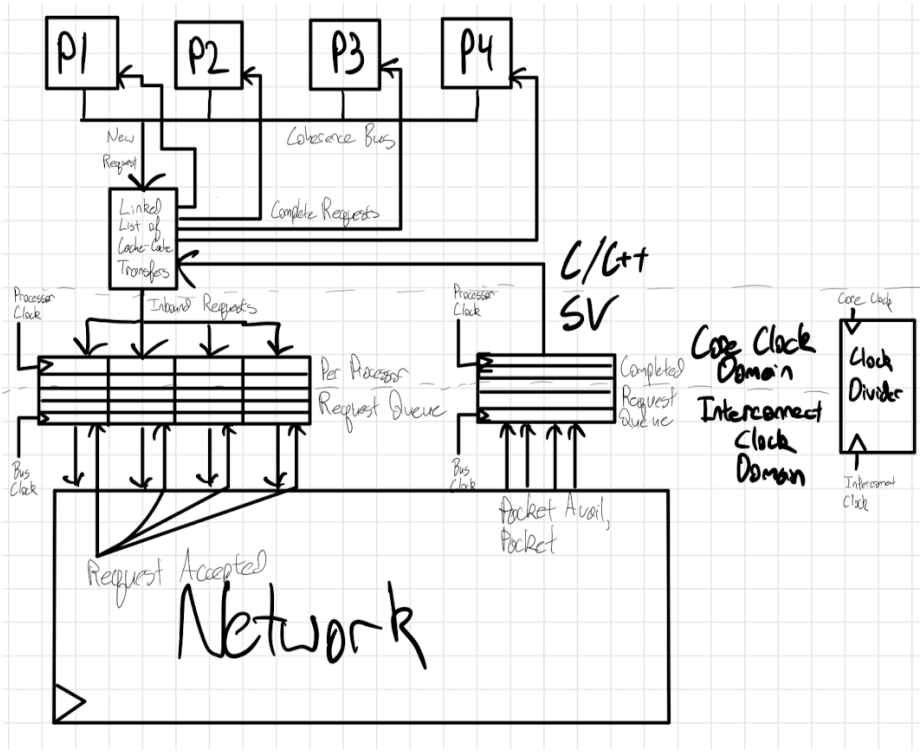
To explore this, we are starting CADSS (Computer Architecture Design Simulator for Students). This gives us a mechanism for running traces on multiple concurrent processors with an MI cache coherence protocol and a basic cache. The current cache coherence network is effectively a singular bus with no support for concurrent cache to cache transfers and other messages.

Our project hopes to explore the impact of different networks their parameters on different workloads. We are using SystemVerilog, a hardware description language used for developing and verifying RTL (Register Transfer Logic). This allows us to create cycle-accurate synthesizable networks. Although most SystemVerilog simulators allow for interconnection with C code, since we are working with a preexisting C-based simulator, it was difficult to directly interconnect the two. To overcome this, we developed a socket-based server in SystemVerilog that can communicate with CADSS using short messages.

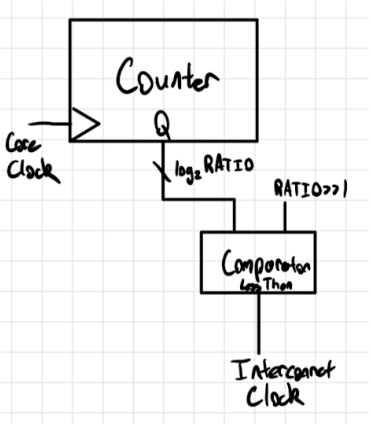
We implemented a bus as a baseline for comparison against. However, this bus implements a request queue for the transfer bus separate from the main coherence communication bus. This allows other processors to communicate and enqueue requests while other requests are outstanding. There is still a single serialized connection to DRAM.

3 Approach

3.1 Simulator Interconnect



3.1.1 Clock Divider



The simulator in its original state can handle a single outstanding memory request at a time, emulating a bus. There is a prescribed time for checking for residency in cache followed by either other cores responding with the data or

memory responding with the data. This does not make for a very interesting network simulator since there will never be any contention on the network with a single outstanding request. To solve this, we keep the cache searching part exclusive but if a request can be serviced by a cache-cache transfer, the request is moved to a separate new request queue, implemented as a linked list.

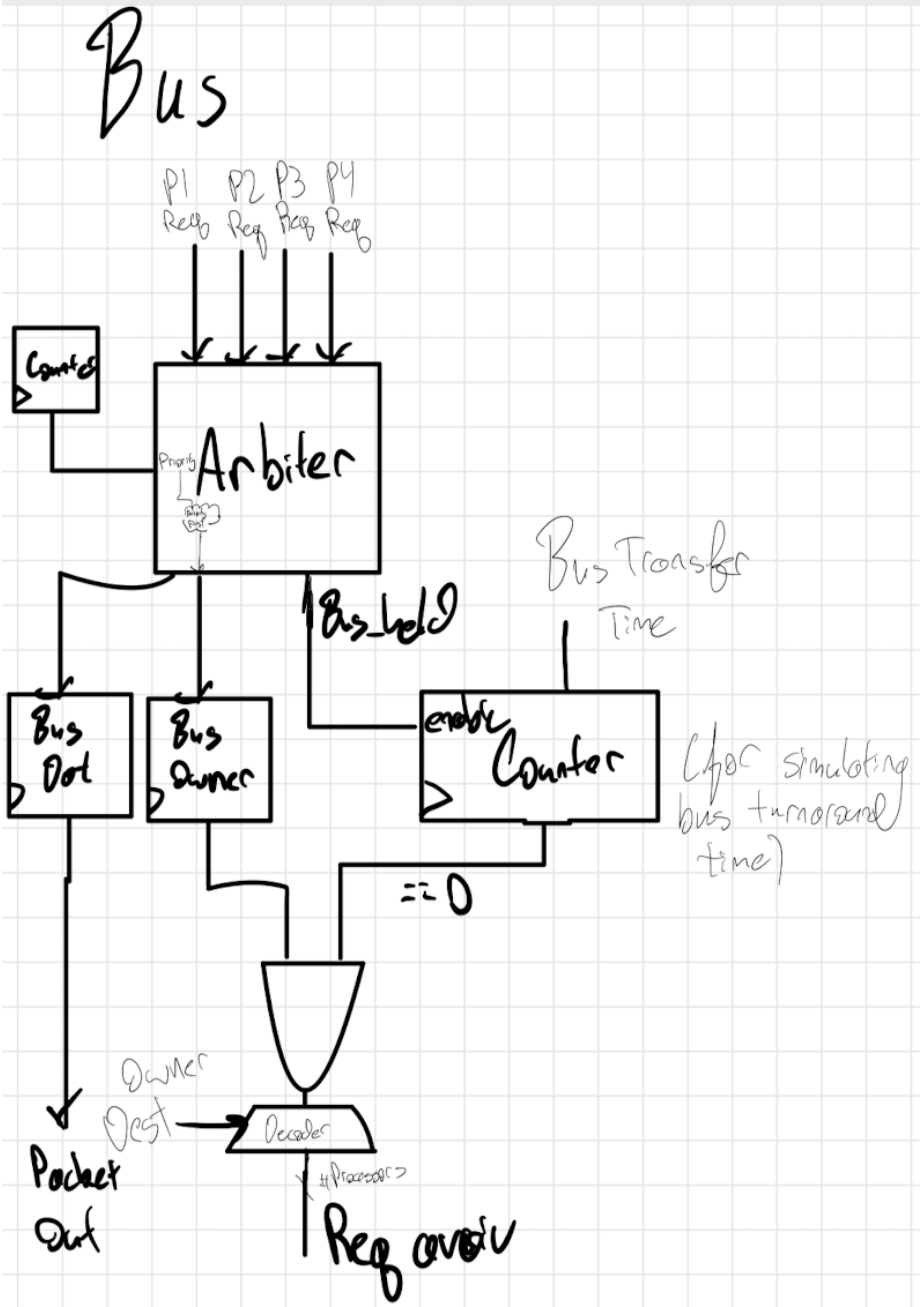
When a request is added to the linked list, it is also sent over the sockets connection to the SystemVerilog interconnect, which is running the server through DPI calls to C code. The simulator also sends messages whenever a clock cycle has elapsed, which generates a corresponding clock tick in the network simulator.

Upon receipt of the transaction, the SystemVerilog adds it to a software queue. This allows for the transfer across clock domains, since the input and output can be clocked at different rates. The request remains in the queue until the interconnect handshake signal asserts that it has received the packet, at which point it is popped.

We then wait for it to be received. We also track the number of in-flight requests, which could be used for further data analysis. Upon receipt, it is placed into an outbound queue corresponding to the processor to receive. This handles traversing back into the core clock domain. On the next clock tick all received packets are communicated back to the core.

Upon receipt in the main simulator, the linked list is searched for the corresponding entry. An error is raised if it is not found, otherwise the corresponding processor's request is marked as filled.

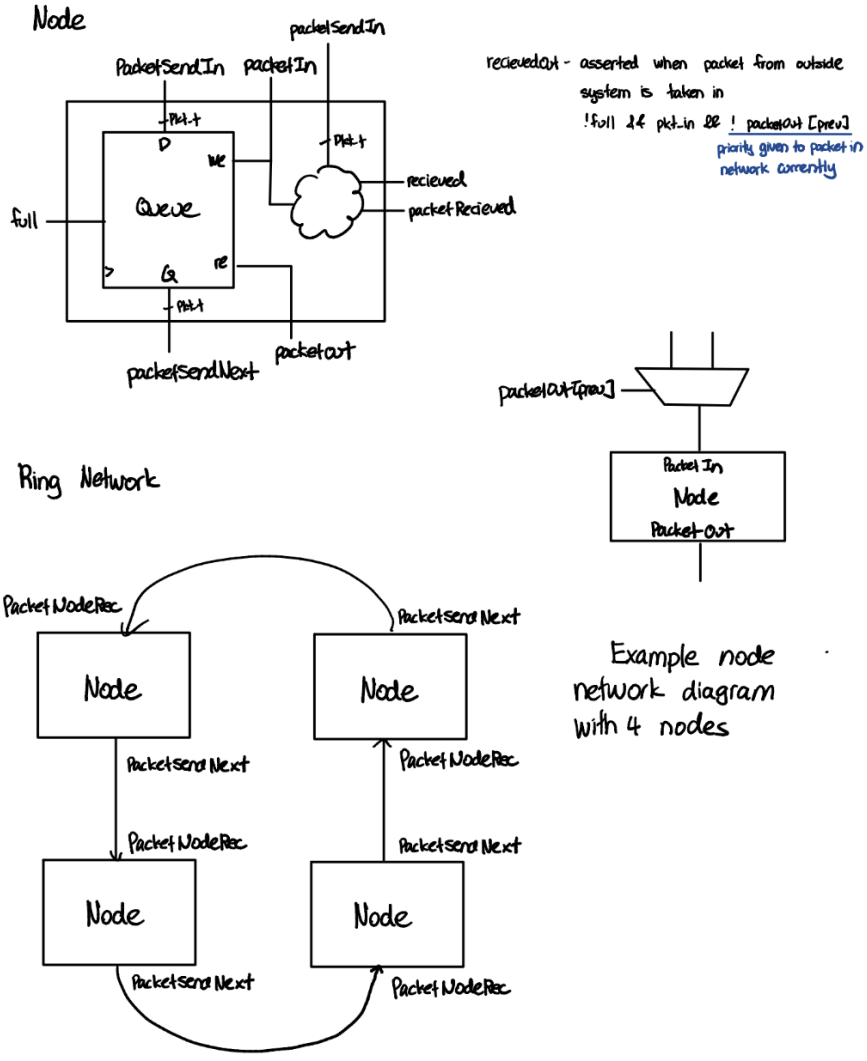
3.2 Bus



The bus is the simplest of network topologies, but requires some mechanisms to ensure fairness and to accurately simulate its behavior. Since a true bus would require time for signals to propagate, especially given the high distances

and powers required, we use a counter to ensure there is enough time for this. An arbiter ensures only one processor can send its request at a time. This is accomplished with a counter that increments with every transaction. The arbiter gives first priority to the prioritized process and otherwise chooses the first available process.

3.3 Ring



The ring is a direct network topology where the nodes are arranged in the form of a ring. It has the benefit of low cost and being easy to implement, but it also has O(n) latency (n/2 average latency), and the bisection bandwidth remains constant as the number of nodes increases, making it hard to scale in

size.

The implementation of the ring using rtl is as follows. In each node of the ring there consists of two section, the logic to determine if the packet is received by the ring, and the queue to store the packets in the case of contention in the ring. Outside the node, we select what packet to pass in. It prioritizes the packet that is being sent from the other nodes first to reduce congestion.

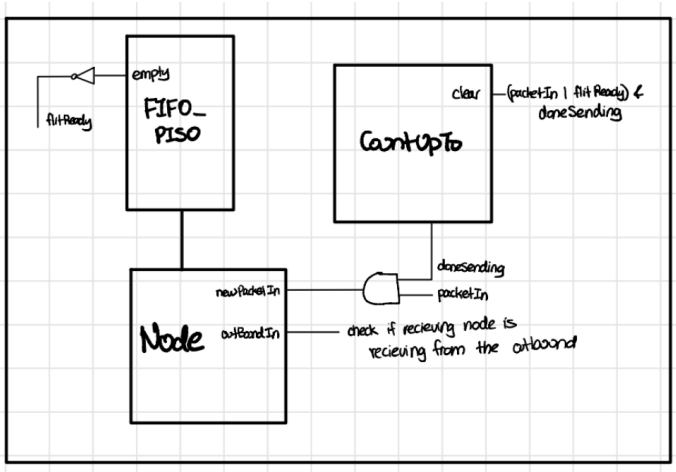
Difficulties

It was difficult managing the input from the core and the input from the other nodes. When there is a packet from another core sending in and a packet from the core sending in, we had to stall the packet sending in from the core to avoid deadlock in the ring.

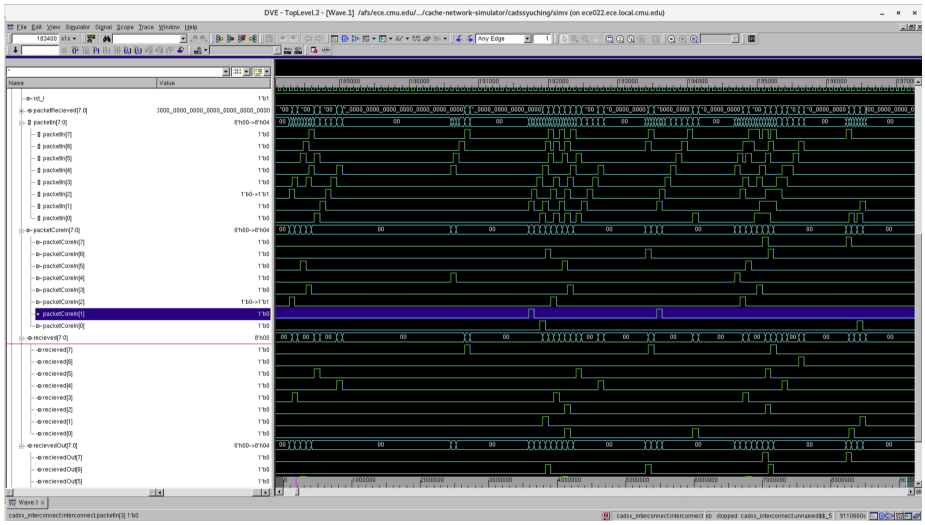
Implementing Wormhole

We also spent a lot of time trying to implement wormhole with the ring. Unfortunately we ran out of time in the end and wasn't able to get wormhole working successfully.

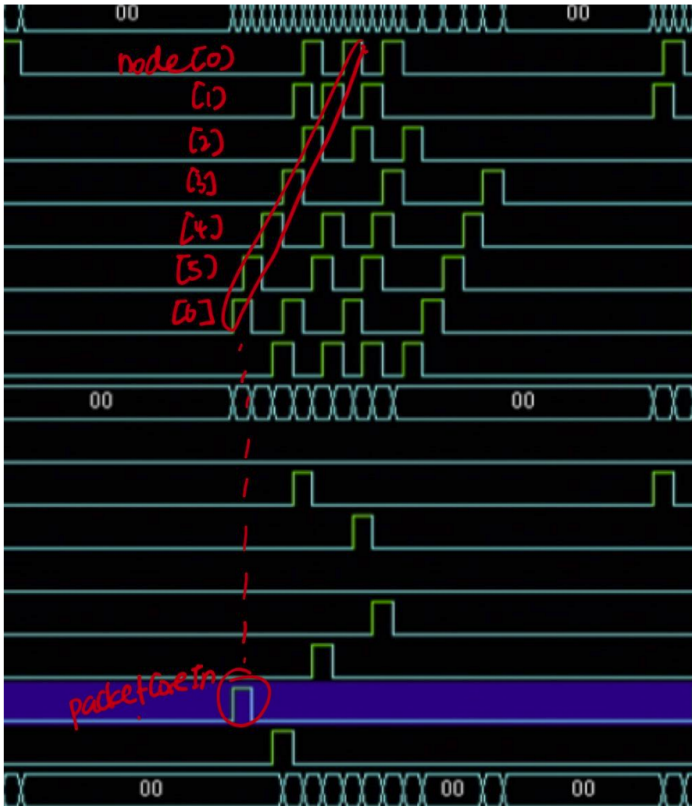
In each node, we had a counter to time when the head of the packet is received. If the head of the packet is received, then we will route based on the head of the packet. Otherwise we route the packet to where the last packet was sent. If the head of the packet's destination is the current node that we are at, through a serial in parallel out register we will continuously take packets in and combine the packet together. Otherwise we will simply continue to send the packet to the next node in the router.



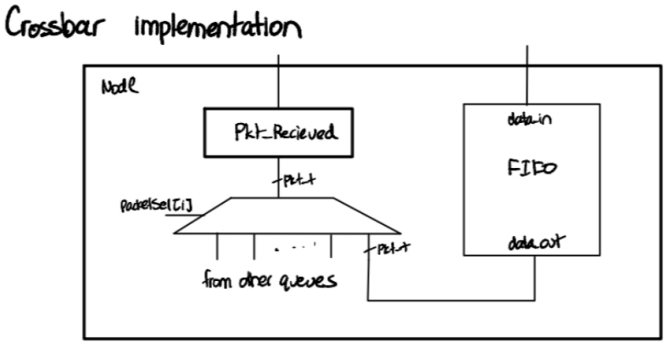
The following is an image of waveform indicating packets being sent through the network



The image below indicated a packet flowing through the ring topology. The packet was sent from node 6 and has a destination of node 0. You can see the packet sent in signal flowing from node 6 to node 7. In addition you can also see the signal of packetCoreIn below it.



3.4 Crossbar



The crossbar is a direct network topology where the nodes are all connected. It has the benefit of O(1) latency and the bisection bandwidth increases with as the number of nodes increases, being able to receive a maximum of n node per clock cycles. However the cost of building the crossbar is O(n^2), making it difficult to scale and layout.

The implementation of the crossbar using rtl is as follows. For each of the node, there is a mux selecting what input to select from all of the other nodes of the network. There is no round robin, we always try to select the node that has the higher node id to send to the destination address.

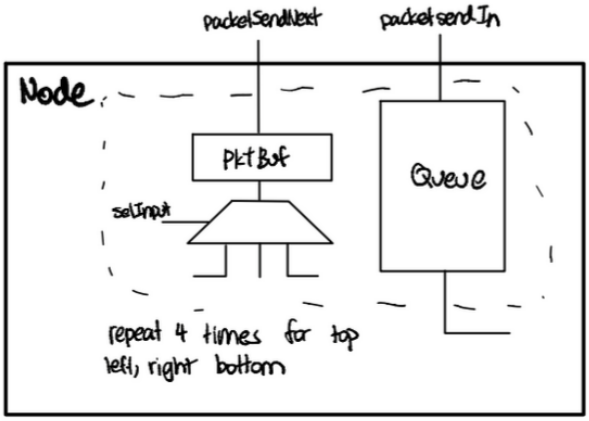
Difficulties
It was difficult syncing where the packet is being read from when where the packet is being sent from. As the packet is originally being placed inside a queue, we would have to remove the packet from the queue after the node is being selected to read from. To resolve this issue, I created a matrix containing all the nodes that are trying to send a packet on the y index and their destination on the x index. Thus by checking if there are any packets that have a lower node id and is trying to send to the same node, we know whether the node is free to receive packets and where it is recieving packets from.

3.5 Mesh

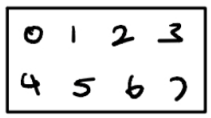
The mesh is a direct network topology where the nodes are connected to form a grid. It has the benefit of O($\sqrt{n}$) latency at O(n) area cost and its bandwith increases as the number of cores increases. Thus it is the most commonly used network today for cache coherence due to its benefits.

We have a working mesh that is able to send and receive packets successfully, however due to limited time constraints, we were unable to gather the data of the number of clock cycles the mesh takes for each of the traces.

Our implementation of the mesh is as follows.
In each node in the mesh it contains four inputs, one input from each direction top, down, left, right. We had set top = index 0, right = index 1, bottom = index 2, left = index 3.



We have a 2 by 4 mesh network, organizing in the following way. The routing algorithm that we chose to use was the y x routing, where we first route in the y direction. After the packet is in the correct y level, we will then route the packet in the x direction. For example if we want to route a packet from node zero to node seven, we will first send the packet to njode 4, then to node 4, 5,



node mesh organization

6, finally 7.

Difficulty

The routing algorithm is the most difficult part of the mesh. We eventually decided to route the packets in the y direction first and then in the x direction to prevent deadlock. We also have to take care of packets being sent from core, for the 2 by 4 network that we have implemented, the packet from the core is mapped to the unused port of each of the four nodes. If a larger network were to be implemented, we would have to create a dedicated port to take care of the packet coming from the core.

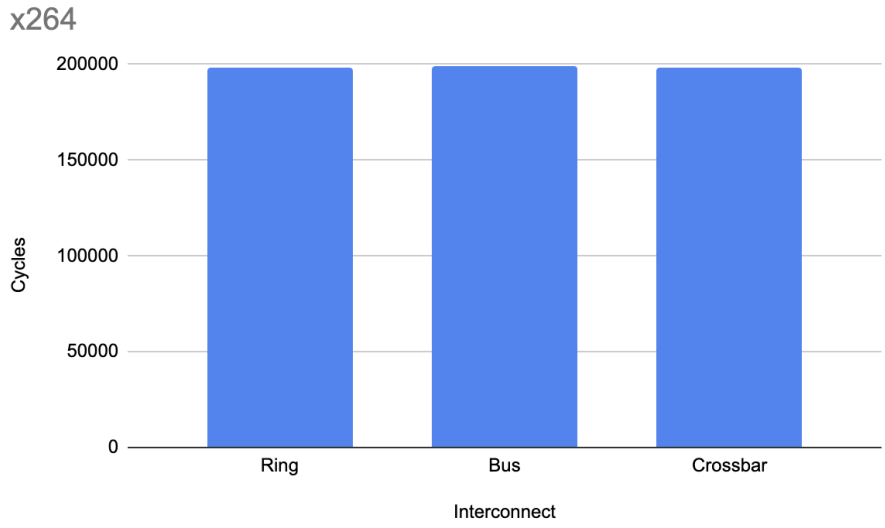
4 Results

To collect results, we measured the time for all processors to complete 3000 instructions. We would have liked to have completed longer traces but this was not possible due to the latency required for the sockets system we used.

We analyzed 3 different benchmarks from the PARSEC benchmark, a benchmark focusing on chip-multiprocessors.

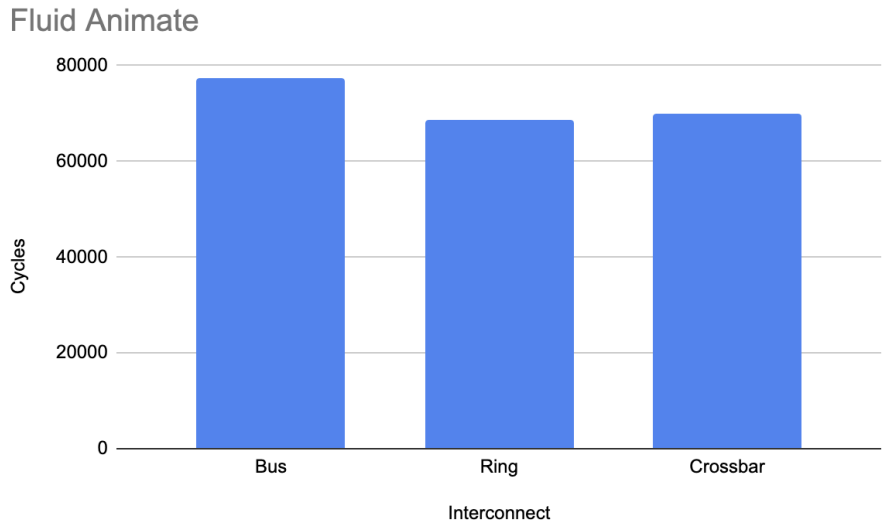
Note that results were limited by our ability to load the network for contention. In a real CPU like the Skylake generation, cache to cache transfer takes about 10 us or around 30 cycles. Our network does not quite reflect this. More importantly, our transfer times are not necessarily reflective of the physics or true timing that would be required to implement them. Such an exploration would be more interesting but beyond the scope of our abilities.

4.1 x264

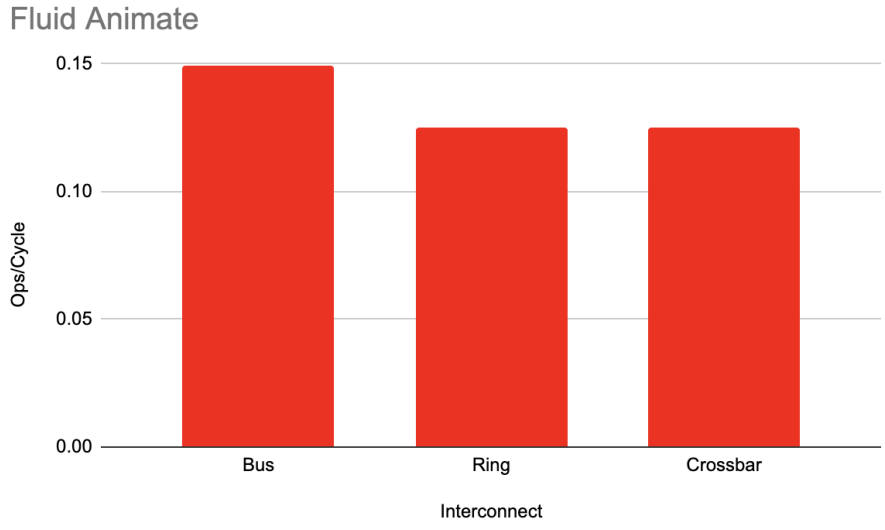


As we can see, no matter which interconnect we use results do not differ greatly. However, Careful examination of results shows that the crossbar requires the fewest cycles, with the ring in the middle and the bus being the slowest interconnect. This aligns with natural expectations of how these networks would perform. The crossbar, with the shortest path, has the fewest delays due to data, the ring is able to support some concurrency but requires longer paths between some processors, and the bus with its slow turnaround time and lack of parallel transmission is the slowest. This behavior may be due to the trace's characteristics. x264 refers to an open source encoding software that is capable of compression and decompression. The data has high sharing and exchange, and although granularity is larger with our MI concurrency system it is still possible false sharing is especially sever.

4.2 Fluid Animate



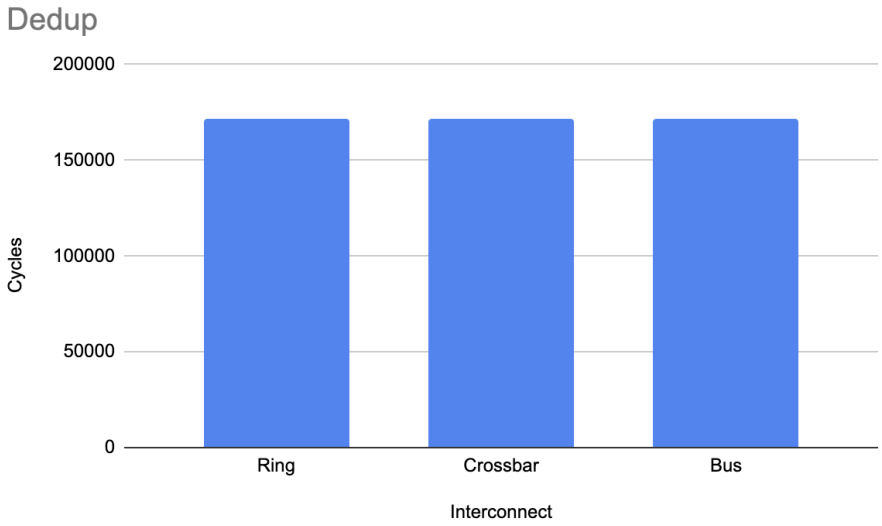
We see a very surprising result in Fluid Animate. The bus is the slowest performing, by a significant margin. The crossbar is faster, and the ring is the fastest. However, this is also a result of arbitration and how our networks interact. The second graph provides more insight into the results, and some deficiencies in arbitration.



We can see that the bus implementation, although the slowest, also allowed another of the threads to advance farther. Specifically, with both the ring and crossbar cores 2 3 and 4 only completed around 3000 instructions. However,

with the bus core 4 was able to complete 10000 instructions, contributing to its higher ops/cycle. This is likely because of differences in arbitration. The ring does not have specific arbitration, but an existing packet is always given priority to prevent a traffic jam. The crossbar always gives priority to the lowest node, which may be why the 1st thread makes so much progress. The bus, with its round robin arbitration, seems to give a somewhat wider distribution. Differences in workload may be part of the result for the slower execution for some cores. It is noted to have low sharing and medium exchange. Further, it runs 9 different kernels. This may lead to different cores executing different portions of the workload, with some sharing much more data than others. This is an interesting real world example of how sharing can lead to work load imbalance and differences in execution.

4.3 Dedup



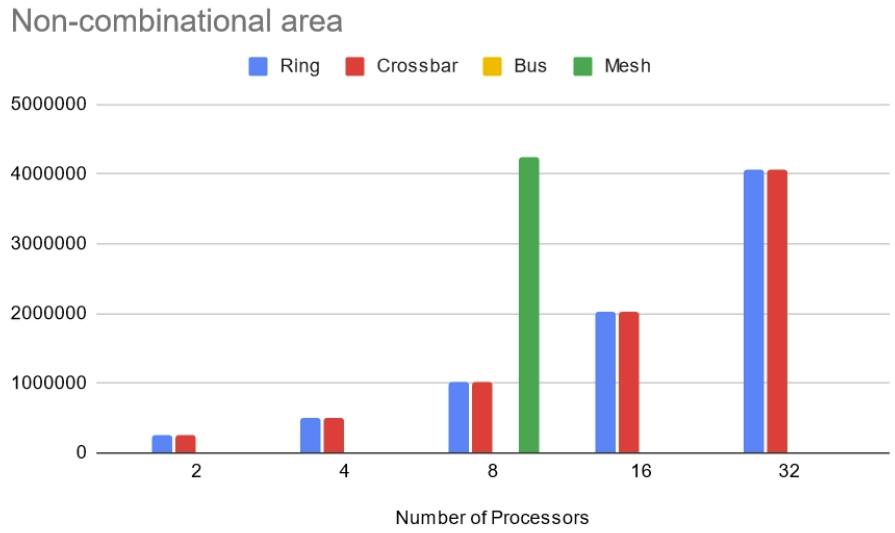
We can see that dedup has fairly similar performance characteristics to x264. Dedup is an enterprise storage program with a similar tipe of parallelism and similar working set characteristics as x264, leading to similar performance trends. Crossbar is again slightly faster than ring, which is slightly faster than the bus. One interesting trend is that it is consumes far fewer cycles in execution. This may be because although there is still high sharing, it may be lower in this region of the program or slightly lower overall.

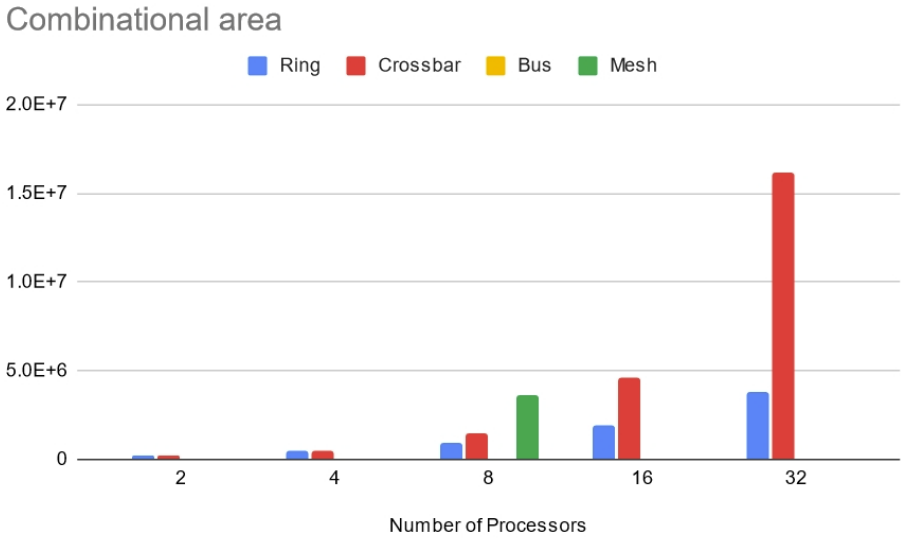
4.4 Area results

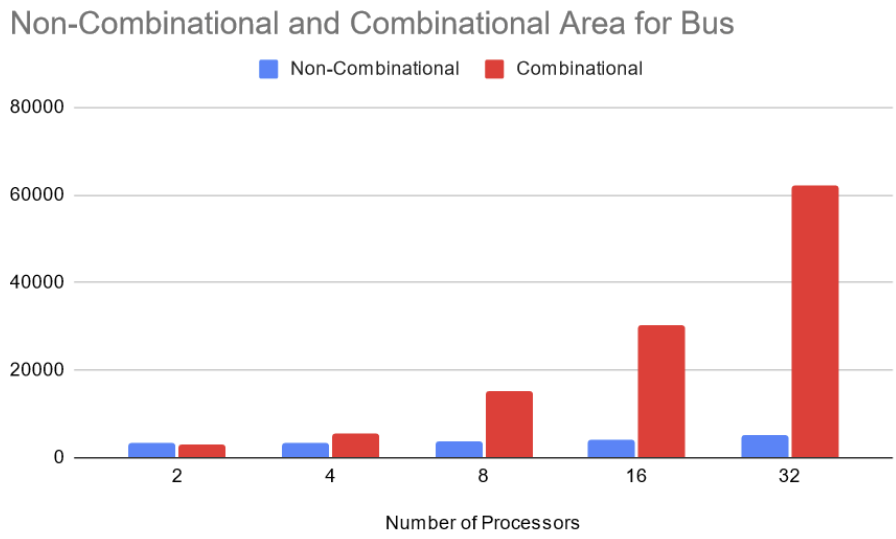
There are two types of area, combinational area and non-combination area. Non-combination area usually means area for registers/latches, where the data is held temporally for a clock period and the data in the register only changes

on the clock edge when enabled, thus is useful to be used to create queues and other data structures in the networks. Combinational area refers to structures like gates that are not clocked and are used to produce intermediate results. Area is important to be considered for both power analysis and the area in and of itself. Power if often a limiting factor for chipe nowadays as it is difficult to disappate the heat from the power. And power if often correlated to area, especially non-combination area. In addition, increase of area also increases the cost of the chip.

Note that we only have data for 8 cores for a single mesh configuration.







4.4.1 Analysis of Combinational Area

We see in the combinational area the major disadvantage to a crossbar. The highly connected graph needs a large amount of logic and area dedicated to arbitration between all the different connections and the connections themselves. This scales exponentially, and would easily become untenable for large core counts. The ring and bus however scale linearly, meaning their area required will be manageable even for high core counts. The mesh network is also expected to scale in this way since it arbitrates between the same number of nodes no matter the network size.

4.4.2 Analysis of Non-Combinational Area

As we can see, crossbars and rings have similar non-combinational area which scales proportionally to size. This makes sense since they have approximately the same buffering capacity per node. Note that in a real network, this might need to be adjusted to allow for greater bandwidth, virtual channels, or other routing schemes. The bus, with its fixed single buffer entry, does not require any mroe area or by extension power. The mesh requires a fairly large amount, although scaling should be linear with the number of cores.

4.4.3 critical path

On a sidenote, another factor that is important in hardware is the length of the critical path, which determines how fast we can drive the clock, i.e. how many operations we would be able to do within a certain time frame. Normally this would be important for NOC, as if the distance between the connections between the nodes varies, the critical path follows the longest wire, making a

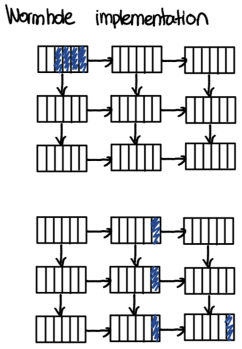
torus mesh perform poorly in this regard. In order to measure the critical path accurately, would would have to lay out the locations of the processors and take into account the distance between the processors, which is not possible due to limited time frame.

4.4.4 Limitations on Bus Analysis

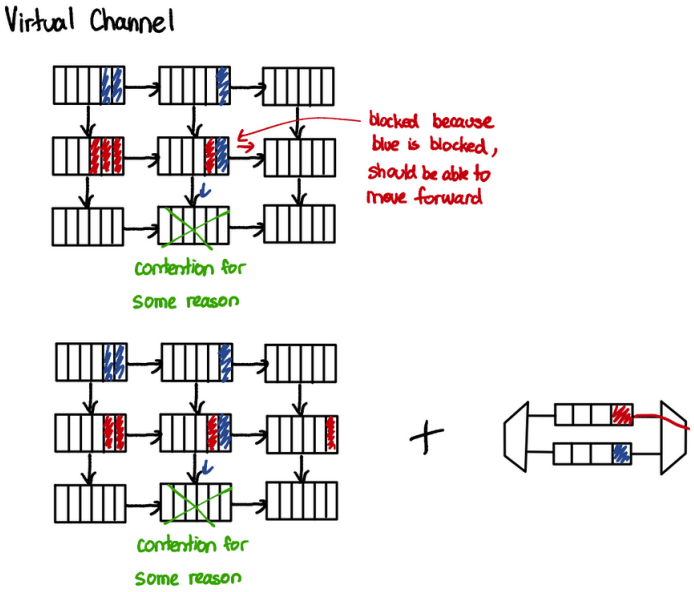
On another note, these graphs do not capture the full difficulty of implementing a bus. Since we did not synthesize with full cores, DC did not have to actually create such a long routing path, or more importantly include drivers strong enough to traverse a long distance. In a real CPU this would be significant, and the parasitic capacitance would also be large. Transmitting at high speed would not be possible, nor would turning the bus around to a different driver quickly.

5 Further Explorations

We tried to implement wormhole routing in the ring topology, but it became too difficult. Wormhole is a packet routing method where we only send a flit a section of a packet each time. The benefits of wormhole implementation is lower latency of packet routing and reduced buffer size required. However, wormhole implementation also results in lots of congestion and therefore increases the risk of deadlock.



Another routing method that we were trying to implement was virtual channel. Virtual channel allows later packets to be routed when the current packet at the top of the queue is being blocked by congestion.



We hoped to avoid some issues in cores stalling out by implementing load and store buffers. We would have used linked lists to track which instructions are dependant on a pending load and store and continued in execution until

the reorder buffer is filled. However, the simulator in its current state does not have information about memory instruction source or destination registers, so we were unable to track this without significant retooling of the simulator.

We experimented in creating our own traces to have more control over analyses by using Contech and Pin. However, we had some difficulty in how to create accurate multicore traces. We had some success with Pin, but due to address space randomization causing inconsistencies we elected not to include these traces.

6 Workload Distribution

Work was split in a 50-50 manner.

6.1 Feature Distribution

1. CADSS Interconnection: Dane
2. Bus: Dane
3. Ring: Yu-Ching
4. Mesh: Yu-Ching
5. Crossbar: Yu-Ching
6. Contech and Pin Trace Development: Dane
7. Load and Store Buffer Analysis: Dane

7 Sources

The PARSEC Benchmark Suite: Characterization and Architectural Implications

CADSS

CADSS Support

Some code was derived from 18-341 router project.

https://chipsandcheese.com/2022/10/14/skylake-intels-longest-serving-architecture/

Network-on-Chip, by Santanu Kundu and Santanu Chattopadhyay