Determining Optimal NoC Topology to Maximize Performance

Charles Drews

csd305@nyu.edu (N11539474)

Abstract

Network-on-a-Chip (NoC) has emerged as the most popular type of interconnection network in multicore and many-core processors. Recent research suggests it is not unreasonable to anticipate that in the future these networks will be wholly or partially reconfigurable, allowing the NoC topology to be customized to meet the communication needs of a particular application. If this comes to pass, there will need to be some means of selecting which topology to use for a given application. This paper proposes a framework for determining the optimal NoC topology for a target application in order to maximize application performance, and presents an implementation of this framework as a tool which takes a target application and certain constraints as input and provides a suggested, novel NoC topology as output. Experimental results are described which compare the performance of various target applications using the framework's suggested topology with performance using several standard, general purpose topologies.

1 Introduction

NoC has emerged as the most common type of interconnection network to provide for communication between components in multicore and many-core processors, surpassing previous types such as bus interconnects and crossbars. This is due to the reduced communication latency of NoC compared to bus interconnects, and the reduced cost of NoC compared to crossbars in terms of both area consumed and energy dissipated. However, the communication latency, area consumed, and energy dissipated by an NoC depend significantly on the physical layout, or topology, of the network.

This topology represents the amount and positioning of links between components in the network. If each tile, consisting of a processing core and its associated private cache, of a chip multi-processor (CMP) is connected directly with one link to each other tile, then the topology is said to be fully connected. Such a topology would provide the lowest possible communication latency, as the path length for any communication would be just one link. However, this topology would also maximize the physical area on the chip consumed by network links, since all possible links would exist, and the presence of all possible links would also lead to increased energy dissipation, as some energy is inherently lost as data travels across each link of the network. The combination of the area consumed and the energy dissipated by the NoC can be thought of as the cost of the NoC topology.

This situation calls for a compromise between latency and cost. Numerous standard topologies exist which seek to strike an appropriate balance of that compromise for general use, including rings, meshes, and torus-based configurations. However, recent research suggests that reconfigurable NoC topologies may be possible in the future, which would allow for novel topologies to be used for each application executed, rather than a single, general purpose topology for all applications.

If this comes to pass, there must be some manner of selecting what topology to use for a given application. A successful framework for selecting a topology could be used in many different ways. It could be incorporated into an operating system, which might use the framework to determine the optimal NoC topology

for an application upon the first execution of that application, then re-use that optimal topology for each subsequent execution of that application. Alternatively, the framework could conceivably be incorporated directly into the hardware so that after running an application once (or a given, small number of times), the hardware could automatically determine the optimal topology for that application and use it on subsequent executions. Finally, the framework could also be implemented as a tool for use by programmers during the development stage of an application. If reconfigurable NoC topologies come to be, then it is possible future parallel programming languages and APIs may give programmers the ability to explicitly control what topology is used for their application. In that scenario a tool could be used to test an application under development, determine the optimal topology, and build it right into the code of the program.

Regardless of how an topology selection framework is implemented, it is clear that without one it would not be possible to take full advantage of a reconfigurable NoC. This paper proposes one such framework, implemented as a tool which can analyze a target application and select an optimal NoC topology subject to constraints on the number of links each component of the NoC can support, and the total number of links allowable in the network. The concepts behind this framework may support the development of the possible framework implementations described above.

2 Literature Survey

The most common type of interconnection network between tiles in a chip multi-processor (CMP) is a Network-on-a-Chip (NoC) which is comprised of routers generally located at each tile, and links of relatively short length between routers[1]. With this architecture, communication between cores travels as data packets through routers associated with each core, or node, and links between routers. The latency of such communication depends on the number of link a packet must traverse. Therefore, the physical layout, or topology, of those links is an important factor in the overall communication latency experienced by the system, as that topology determines how many steps are required to travel between any two cores. Currently, a 2D mesh is the most common topology choice, as it provides a relatively low number of steps on average between cores (as opposed to a ring topology), scales well to greater numbers of cores (as opposed to a crossbar topology), and is relatively simple to implement (as opposed to both crossbars and novel complex topologies)[1, 2].

Many novel NoC topologies have been proposed, such as the nine different configurations tested in [5]. Interestingly, the topologies that ranked near the top in one measure tended to rank near the bottom in another measure, with the overall "winner" being a topology that was middling in most of the measures. The key takeaway here is that there is no obvious best NoC configuration for all uses. Each application has unique communication needs between cores, and will achieve best performance with a particular NoC topology that may not serve a different application well at all. In light of that, one may wish to vary the NoC architecture for each application.

Fortunately, there has recently been much research into reconfigurable NoCs featuring components that can be dynamically altered to change what paths are available between cores. One option for implementing a reconfigurable NoC is to add a series of switches to the links between routers which can be turned on or off to direct the path of a packet around one or more nodes along the way to its destination, thereby avoiding the latency inherent in passing through the routers associated with those intermediate nodes[3]. A second option for implementing a reconfigurable NoC is to utilize optical interconnection technology, which can use varying light wavelengths to effectively bypass intermediate nodes and create direct links to target nodes[4]. An example of this second approach can be found in [6], which describes an traditional electronic NoC augmented by optical "shortcut" paths which can be reconfigured to provide extra bandwidth as needed. Both approaches suggest that future architectures may allow for novel reconfigurable NoC topologies where any pair of nodes can possibly be linked, though perhaps with some constraint on the number of links active at any given

time, as neither approach proposes to lift the constraint of power dissipation as the number of active links increases.

To summarize, router-based NoCs have become the leading choice for facilitating communication between cores in a CMP, and while 2D mesh is the most common configuration for now, the varying communication needs of different applications, coupled with the emergence of reconfigurable NoCs suggest a future where NoC topology can be dynamically customized to best suit a given application. In order to determine how to best handle a given application, it will therefore be necessary to have some understanding of that application's communication needs.

Various researchers have investigated approaches for tracking an application's inter-thread communication. [7] introduces a tool named CETA which automatically produces communication graphs featuring an application's threads as graph nodes, and communication activity between threads as edges labeled with the total volume of data transmitted between the two threads connected by that edge. The volume information may facilitate the decomposition of the communication graph into sub-blocks that can simplify the task of determining which pairs of threads, and therefore which pairs of cores that "host" those threads, require the most communication and would most benefit from being directly connected by the NoC (as opposed to being connected via multiple steps/routers). [8] presents an approach which focuses on applications written in pthreads and produces comparable thread communication graphs. [9] presents a somewhat different approach to tracking inter-thread communication which relies on the hardware and operation system to use the Translation Lookaside Buffer (TLB) to build a communication matrix representing communication between all pairs of threads. The takeaway here is that there are numerous options for gathering the inter-thread communication profile of a target application, so it is not unreasonable to set a goal of using this information to determine the ideal NoC configuration.

The next step then, is to consider whether any research has combined these elements and proposed a means for determining the ideal NoC configuration for a target application. [10] does indeed combine these elements, but with some limitations. This paper presents an analysis of the communication profiles of eight different multithreaded applications, and describes an approach for designing an interconnection network configuration which both maximizes application performance and minimizes the number of links needed. The limitation of this effort is its focus on HPC; the programs featured are strictly scientific in nature and the analysis considers only parallel code written in MPI. This is very useful for scientific applications being run on clusters or supercomputers, but does not specifically address NoCs within a CMP on a general purpose machine for a consumer application.

Another paper that ties together inter-thread communication data and interconnection network topology is [11]. The proposed approach does not discuss the gathering of communication profile data, but rather assumes such data as an input, then creates a layout of theoretical processing elements to maximize throughput for the target application, then finally creates an interconnection scheme designed to maximize throughput subject to constraints on the total number of links in the network and the number of links connecting to each reconfigurable switch. This paper is very robust, but one limitation is that the proposed approach focuses on cluster environments and interconnection between processors, rather than a CMP and interconnection between cores on one processor. Also, this approach was only tested on graphs representing stream computing applications, and the graphs were all artificially generated, rather than representing actual application execution. I believe there is room for additional research into a tool that applies similar methodologies to selecting an NoC topology and is tested on the PARSEC suite of real benchmark applications.

3 Proposed Idea

When a multithreaded application is executed on a CMP, threads are assigned to physical cores in a manner possibly specified by the programmer, depending on the parallel language used in application development, or possibly as determined by the operating system. Either way, once the assignment of threads

to cores has been made, there will be an inherent need for threads to send and receive data to and from threads residing on different physical cores. This need is driven by synchronization between threads, cache concurrency protocols, etc. The data passed between threads makes up the communication traffic carried on the NoC.

Intuitively, varying NoC topologies will result in different flows of communication traffic, with different path lengths the traffic must traverse on average. The more network links and routers a particular communication must pass through on its route from source to destination, the more latency is introduced to that communication. The obvious goal is to reduce the average path length, and therefore average latency, while conforming to physical constraints such as the number of links each component can support and the total number of links present. These physical constraints ensure the balance between latency and cost do not tip too far toward low latency at the expense of excessive cost.

An NoC topology can be described as a graph of nodes connected by edges. The nodes represent the components of the CMP connected to the NoC, such as the tiles which each contain a physical core and that core's private L1 cache, and the L2 cache which may be segmented and co-located on tiles with the cores and L1 caches, or may be positioned separately from the cores. The edges represent physical links connecting nodes.

The weight of each edge is defined as the volume of data passed over that particular link during the execution of a target application. The degree of a node is the number of links connecting to that particular node. It is realistic to assume a constraint on the degree of each node, because the nodes are real, physical objects, and an unlimited degree would mean an unlimited number of physical connections to that node which would exhaust the area available and leave little room for the core or private cache that need to reside there. Also, it is realistic to assume a constraint on the total number of links allowable for the NoC, as an unlimited amount would not only take up all the area of the processor chip, it would also dissipate significant energy, making an NoC with unlimited links undesirable.

The first step of the proposed framework is to profile the target application by simulating its execution over a fully connected NoC topology, ignoring constraints on edge count and node degree. This step is not intended to represent a realistic scenario for executing the application, but instead is intended to represent the ideal scenario in terms of throughput performance, providing a benchmark against which the throughput performance of realistic topologies suggested by the framework can be compared. During this step and all subsequent steps, the framework assumes a static number of cores and a static configuration of cache and main memory. The output of this profiling step is a graph of the target application's NoC communication, specifying the volume of data passed along each link as weights on the graph edges.

Next, the framework modifies the fully connected topology by removing links until both the node degree and edge count constraints are satisfied. The manner in which edges are selected for removal is key to the framework, and proceeds as follows, with significant inspiration from [11]. Links present in the NoC are considered in order by increasing weight, starting with the lowest weight link. If removal of the link under consideration results in a graph that is still connected (i.e. a path still exists between any two nodes), then that link is removed. Then the next link will be considered in the same manner, and so on, until all links have been considered or until the total number of links does not exceed the edge count constraint.

Each time a link is removed, the traffic that had been flowing over that link must be rerouted. Since each link connects two nodes, when a link is removed, all possible alternate paths between those two nodes are considered and ordered by increasing weight. If a path consists of multiple links, the weight of the path is the sum of the weight of its component links. The traffic from the removed link is then rerouted along the lowest weight alternate path, and the weight of the links in that path are updated accordingly.

If all links are considered and the total count still exceeds the constraint, then the links are reconsidered, again starting with the lowest weight remaining link and proceeding in increasing order by weight. The link under consideration is removed, causing the

graph to become disconnected. In order to reconnect the graph, the lowest weight link in each disconnected graph component is identified. These two links are replaced with links connecting the two graph components. For example, if the lowest weight link in one graph component connects nodes a and b and is called {a,b} and the lowest weight link in the other graph component connects nodes c and d and is called {c,d}, then these two links are removed and replaced with two new links: {a,c} and {b,d}. The new links are initially be given a weight of zero, then the traffic from the removed links is rerouted, possibly along the new links. This process reconnects the graph without increasing the total link count. If removal of a particular link will result in a disconnected graph where one component consists of a single node with no links, then such a disconnection could not be remedied by this process, and that link is not removed. This process proceeds by increasing link weight until the edge count constraint is met, and traffic is rerouted as described above upon each link removal.

Next, the degree of each node is considered. It is possible that even if the edge count constraint has been satisfied, some nodes may still be connected to more links than allowed by the node degree constraint. For each node that exceeds the degree constraint, its edges are categorized into two groups: "heavy" links that connect to another node that also exceeds the degree constraint, and "light" links that connect to a node that satisfies the degree constraint. Heavy links are removed first, starting with the lowest weight and proceeding in order of increasing weight. Each removal only takes place after checking that the graph will remain connected, as described above, and after each removal traffic is rerouted, also as described above.

If all heavy links are removed and the given node still exceeds the degree constraint, then light links are removed, subject to a connectedness check and followed by traffic rerouting. If a node cannot be brought into compliance with the degree constraint without disconnecting the graph, then the lowest weight link connecting to that node is removed, and the then disconnected graph is reconnected according to the procedure described above.

During this second phase where the node degree constraint is considered, each time a link is removed, the overall link count becomes less than the total link constraint. There is no reason to reduce the overall link count below that constraint, so after removing a link for node degree constraint purposes, a new link is added between two nodes that can each support one more link without exceeding the node degree constraint. Each time a link insertion is required, each possible pairing of nodes that can support an additional link will be considered along with weight of the edge that originally linked that pair after the initial fully-connected profiling step. The pairing with the greatest original edge weight is selected for the new link. The new link is initialized with a weight of zero, then the traffic from the removed link is re-routed, possibly over the new link.

The result of this framework is a novel NoC topology that connects the given number of cores while complying with the node degree and edge count constraints. This topology is the output of the framework and represents the optimal topology under the given constraints.

One obvious question is whether subsequent executions of the target application will be similar enough to the profiling execution that the novel topology will be useful. I assert that they will. Presumably operating systems make assignments of program threads to cores in a consistent manner on each execution (excepting the rare cases of core failure), so the communication between program threads should manifest itself as similar NoC traffic on each execution.

Furthermore, even if the target application is run again with different input data, I assert the novel topology with still be useful. NoC communication between threads is primarily driven by synchronization and cache concurrency protocols. The global or shared variables that must be synchronized between threads will not change if the input data changes. There will still be the same number of shared variables that require synchronization when written. Similarly, the writes to shared variables stored on one core's L1, thereby requiring invalidation of other cores' copies of that block to maintain coherence, would not change significantly with input data, as the number of shared variables would remain the same. Therefore, I believe the pattern of NoC

communication will be similar when the input data changes, and in the case of significantly larger input data (requiring more loop iterations for example) the communication pattern may repeat more times, but should be a similar pattern.

4 Experimental Setup

The experimental setup consists of an implementation of the above framework in the form of a tool written in Python (files: `tool.py`, `comparisons.py`, `writeconfigs.py`). This tool relies upon a pre-existing simulator at two stages, the initial profiling of the target application, and then again in testing the suggested novel topology against commonly used topologies. The simulator the tool uses is Multi2Sim, which can simulate a wide variety of hardware configurations, including novel NoC topologies[12].

Step one is to run the `tool.py` script specifying a target application. This calls Multi2Sim to run a simulation of the target application over a fully connected topology. The fully connected NoC is specified using Multi2Sim's `--net-config` option which allows the input of a configuration file describing the NoC and its topology. The tool translates the output from Multi2Sim into a graph describing the inter-core communication profile of the application. The tool relies upon a pre-existing third-party Python library called NetworkX to provide an API for graph manipulation[13]. The tool modifies the graph to perform the link removal procedures described above and produces its output in the form of a recommended topology. The tool translates this output into the format of Multi2Sim's network configuration text file by calling the `writeconfigs.py` module to write output files.

Step two is to run the `comparisons.py` script, specifying the same target application and target application arguments used in step one, and a specified topology from the following list: novel (i.e. the topology just created in step one), ring, mesh, or torus. This phase of the tool again relies on Multi2Sim to simulate the target application over the specified topology. This step is repeated as necessary for each topology option in order to provide data points for comparison.

To test the tool, target applications were drawn from the PARSEC benchmark suite[14]. Specifically, tests were performed using the pre-compiled versions of the PARSEC 2.1 programs distributed by Muilti2Sim from www.multi2sim.org/benchmarks/parsec-2.1.html. These executables are compiled specifically for compatibility with the Multi2Sim simulator, and correspond to the medium data set offered by PARSEC.

The tool, both in the profiling phase and in the comparison testing phase, must instruct Multi2Sim on the specifics of the hardware to be simulated. The tool assumes the following hardware specifications for all simulations:

- Nine cores:
  - Homogeneous with x86 architecture
  - One hardware thread per core
  - Core 0 is the primary/main/lead core (this is implicit in Multi2Sim)
- Private L1 cache (for both instructions and data) co-located with each core:
  - Each with 128 sets of 2-way associative cache with block size 256 bytes, latency of 2 cycles, and a "least recently used" block replacement policy
  - The specific values in the previous bullet are consistent with the examples provided in Multi2Sim's documentation, understood to be default values
- Two banks of shared L2 cache:
  - Each bank handles half of the virtual address space (which is 32-bit, per Multi2Sim's capabilities)
  - Each L2 bank is connected to two NoC switches
  - L2 bank 0 connects to the switches corresponding to cores 0 and 1

  - L2 bank 1 connects to the switches corresponding to cores 1 and 2
  - Single link path between the primary Core 0 and L2 bank 0 (i.e. One might have positioned Core 0 one link away from L2 in the novel topology and many links away from L2 in the standard topologies in order to give the novel topology an unfair advantage. This was not done; the distance of the connection between Core 0 and L2 does not vary between topologies.)
  - Each L2 bank has 512 sets of 4-way associative cache with block size 256 bytes, latency of 20 cycles, and a "least recently used" block replacement policy
  - The specific values in the previous bullet are consistent with the examples provided in Multi2Sim's documentation, understood to be default values

- One bank of shared main memory:
  - Connected only to the two L2 banks via a switch
  - Not connected to the NoC that links the cores with each other and L2 (Multi2Sim requires separate networks for cores/L1/L2 and L2/main memory - this tool focuses on the NoC for cores/L1/L2 only.)
  - Block size of 256 bytes and latency of 100 cycles
  - The specific values in the previous bullet are consistent with the examples provided in Multi2Sim's documentation, understood to be default values

- NoC characteristics:
  - Bandwidth of 256 bytes per cycle and input/output buffers of size 1024 packets for all switches/nodes in the NoC
  - The specific values in the previous bullet are consistent with the examples provided in Multi2Sim's documentation, understood to be default values
  - The links between Cores 0-2 and the banks of L2 do not vary between simulations, only the links between the nine cores

The design is intended to represent a generic multicore system, and the selection of nine cores was intended to facilitate the simulation of mesh and torus topologies, which can be easily applied to cores arranged in a square, in this case 3 x 3, layout. The topologies simulated by the tool are as follows:

- Fully connected: all nine cores are connected by all possible links (total of 36 links)
- Novel: as determined by the framework, subject to link count and node degree constraints
- Ring: NoC links from core 0 to core 1, 1-2, 2-3, etc until 8-0 completes the circle (total of 9 links; max node degree of 2)
- Mesh: 2D grid with cores 0-2 in row one, cores 3-5 in row two, and cores 6-8 in row three (total of 12 links; max node degree of 4)
- Torus: similar to mesh with additional links between the end nodes of each row/column (e.g. between cores 0 and 2) (total of 18 links; max node degree of 4)

5 Experiments and Discussion

If the proposed framework and tool is successful, it will produce the following results:

- It will identify a novel topology that is consistent with the constraints provided; the tool will produce a viable output for any valid constraint values
- The performance of the suggested novel topology will exceed the performance of common, standard topologies in one or both of the following parameters:
  - Average latency: the average delay in cycles of each message transferred across the NoC (Latency)

  - Millions of instructions committed by the application per second (MIPS)

- The performance of the suggested novel topology is not expected to match that of the ideal, fully connected topology used in the profiling stage, but should lie between the fully connected topology and other comparator topologies when graphed

The specific experiments run are described by the following:

- Blackscholes, Bodytrack, Canneal, and Dedup from the PARSEC suite were each run through the tool as a target application
- Each target application was simulated on fully-connected, novel (as determined by the tool), ring, mesh, and torus topologies
- The constraints placed on the novel topology were 15 total links (intended to be a middle-ground between mesh's 12 links and torus's 18 links) and maximum node degree of 5 (to allow slightly more flexibility than the maximum degree of 4 seen in mesh and torus)
- Each target application was simulated with nine software threads, matching the nine hardware threads offered by the simulated system
- All simulations performed by Multi2Sim, both in the initial profiling stage of the tool and the comparison stage of the tool, were halted after 100 million committed instructions. This was done to reduce the real-world time of simulation to a feasible length in order to facilitate testing with more than just one target application.
- All simulations assumed the same simulated hardware profile (with the exception of the NoC topology which of course varies between simulations) as described above

Specific arguments provided to each PARSEC program, which in some cases determine data size, are as follows:

- Blacksholes: 9 path/in_4K.txt path/prices.txt
- Bodytrack: path/sequenceB_2 4 2 2000 5 0 9
- Streamcluster: 10 20 64 8192 8192 1000 none path/output.txt 9

The most promising results were the latency statistics (see Figure 1), which show an small advantage to the novel topologies compared with the standard topologies (ring, mesh, torus). The latency statistics represent the average latency (in cycles) a packet experiences while being transferred across the NoC.

One of the outcomes of the framework is to make sure there are direct one-link paths between the pairs of nodes that communicate most. This is in contrast with standard topologies where two nodes that communicate heavily may end up on opposite sides of the ring or grid. Therefore, the novel topologies created by the framework have the effect of reducing average path length, which in turn reduces latency, since latency is introduced each time a packet passes through a node's router on the way to its destination.

The reduction in latency compared to standard topologies is quite small though. For Blackscholes, the novel topology had an average latency just 0.01% better than that of the torus. The novel topology for Bodytrack fared slightly better at 1.77% better than the torus. Streamcluster fared the best with a 5.73% improvment for novel vs. torus. I suspect generally small amount of improvement is related to my decision to limit each simulation to 100 million instructions. Each of the multi-threaded target applications has at least some portion that runs in serial, and I believe it is likely that the serial portion is often skewed towards the beginning of the execution while data is checked and algorithms are set up. Therefore, by artificially ending each simulation early I may have ended up with a higher percentage of serial execution than parallel execution during my simulations.

Why is that significant? When serial execution is occurring, the work is being done by core 0 (due to the way Multi2Sim schedules threads) and the only NoC traffic is between core 0 and the L2 nodes. Therefore, during serial execution the rest of the NoC does not impact the performance. Only during parallel execution does the rest of the NoC play a role. If

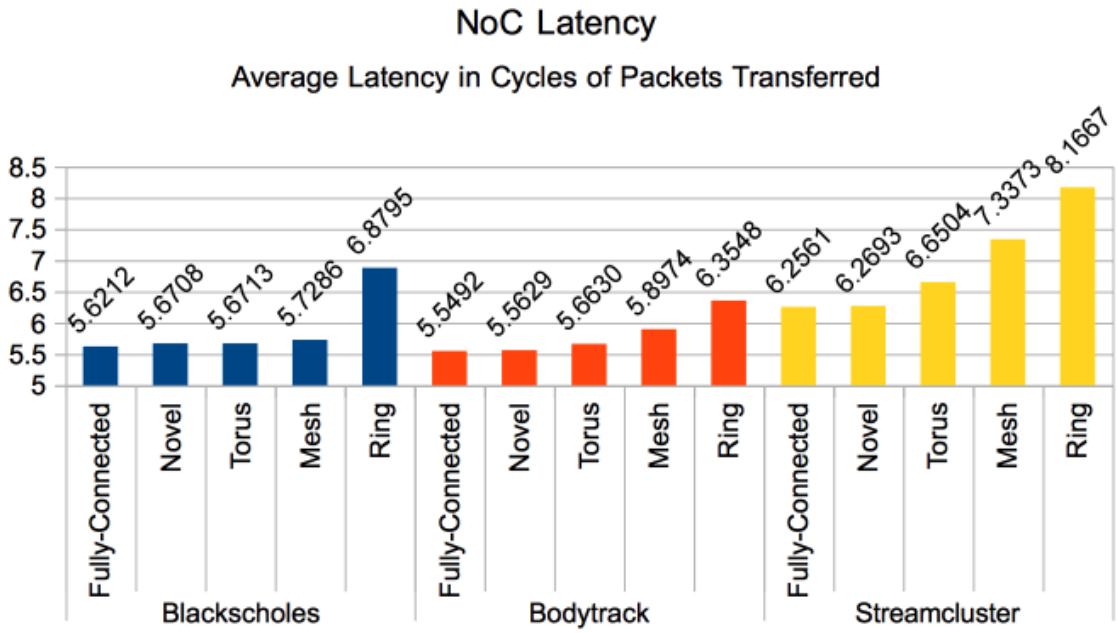

Figure 1: Latency of selected PARSEC applications over various NoC topologies

my simulations were indeed skewed toward serial execution, that could significantly blunt any effect my novel topologies have on performance. While reviewing the detailed simulation output, it was obvious that in all simulations there was vastly more traffic to/from core 0 than to/from any other node in the NoC. This supports my theory that my simulations were skewed toward serial execution.

It is possible that if time permitted me to run all simulations to their natural completion, the ratio of parallel to serial execution would be much higher and the benefits of the novel topologies would be more significant. Such further testing would be an important part of any future work on this project.

It is also important to note that in all experiments I selected 15 as the maximum link count constraing on the novel topology. This is between a 3x3 mesh's 12 links and a 3x3 torus's 18 links. If I had allowed the novel topology to have as many links as the torus, it stands to reason that average path length would be slightly lower and latency also slightly lower. The fact that the novel topologies performed similarly or slightly better than the torus with 3 fewer links does speak to the effectiveness of the novel topologies.

The overall performance of each program, as measured in millions of instructions per second (MIPS) shows virtually no difference between topologies (see Figure 2). This is good in the sense that at the novel topologies do at the very least function as well as standard topologies. However, this result is not desireable in that it shows no benefit from the novel topologies vs. the standard topologies.

It is important to note, however, that the fully-connected NoC used in the profiling stage also does not appear to have produced any better MIPS than the standard topologies, despite being an unrealistically ideal configuration. This suggests that the same issue discussed above is likely having a significant impact: ending the simulations early may be skewing the simulations toward serial execution and minimizing any impact the NoC topology might have on MIPS. I believe it is not unreasonable to suggest that simulations run to completion might possibly show at least a small benefit in MIPS from the novel topologies.

Regarding my choice of PARSEC programs to test, I attempted to test several more but encountered an issue where some of the PARSEC programs effectively ran 100% in serial, with no NoC traffic to/from other cores besides core 0. In those cases the topology of the NoC was entirely irrelevant and did not impact the performance. As described above, I believe this phenomenon is caused by stopping the simulations at 100 million instructions, which for some of the PARSEC programs must be before the parallel portion begins. In early testing with an even lower instruction cap the same problem occured with Blackscholes and Bodytrack, but raising the cap to 100 million allowed at least some parallel execution from those programs. The other programs appear to require more instructions in order to have parallel execution, but unfortunately time did not allow for longer simulations (100 million instructions required ~ 15 minutes; in early testing allowing Blackscholes to complete normally took ~ 2 hours).

6 Conclusion

The tool was successful in that it functioned as expected: it applied the framework described above, and created a network configuration file that accurately described the output of that framework in the proper format for use by Multi2Sim. From a mechanical perspective, the tool accomplished what it set out to do.

The deeper question though, is whether the framework itself was successful. Did the framework create a novel topology that maximized performance? I must honestly conclude no, it did not. The performance of the target applications was, on average, no worse on the novel topologies than it was on the standard topologies, and in some cases it was indeed better. However, the performance improvements that did exist were small enough to be considered negligible. In other words, the novel topologies performed about as well as the standard topologies (specifically mesh and torus, and not ring which as expected performed the worst).

There are some positive takeaways though: the slight improvements seen in latency under the novel

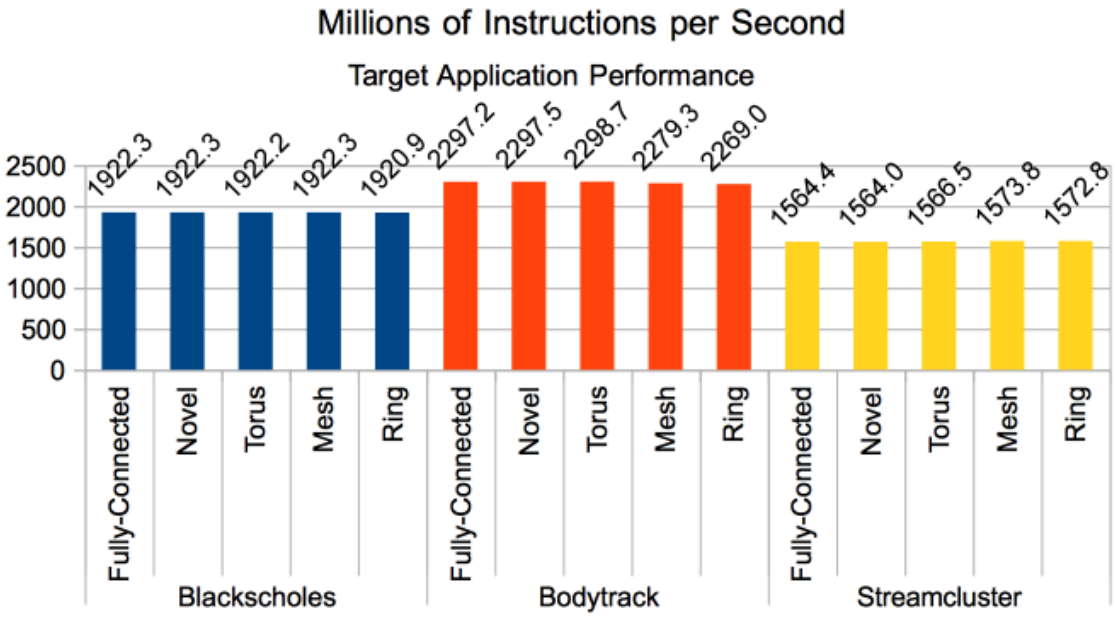

Figure 2: MIPS of selected PARSEC applications over various NoC topologies

topologies have potential to grow under lengthier simulations; and the novel topologies performed no worse than a torus configuration, despite having three fewer links.

7 Future Work

Future work may include significant (and significantly time consuming) further testing of the tool, including:

- Using more or all of the PARSEC applications as target applications in the tool
- Allowing each target application to run to completion rather than halting at 100 million committed instructions
- Increasing the number of software threads in the target application beyond the number of hardware threads offered by the simulated system
- Simulating each target application multiple times with varying data sizes
- Creating novel topologies under a variety of total link count and maximum node degree constraint values

In addition to further testing, the tool itself could be enhanced to accept an arbitrary number of cores as command line input, and possibly the number of hardware threads per core as well. This would add significant flexibility to the tool and allow it to simulate cutting edge manycore systems.

References

[1] W. Dally and B. Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. Proceedings of the 38th Annual Design Automation Conference, 2001.

[2] J. Camacho Villanueva, et al. A Performance Evaluation of 2D-Mesh, Ring, and Crossbar Interconnects for Chip Multi-Processors. Proceedings of the 2nd International Workshop on Network on Chip Architectures, 2009.

[3] M. Modarressi, H. Sarbazi-Azad. Reconfigurable Cluster-based Networks-on-Chip for Application-specific MPSoCs. Proceedings of the 2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors.

[4] L. Bai, et al. A Cluster-based Reconfigurable Optical Network on Chip Design. 2012 Symposium on Photonics and Optoelectronics.

[5] F. Sibai. Which On-Chip Interconnection Network for 16-core MPSoCs? 2010 International Conference on Complex, Intelligent and Software Intensive Systems.

[6] C. Debaes, et al. Architectural Study of Reconfigurable Photonic Networks-on-Chip for Multi-Core Processors. LEOS Annual Meeting Conference Proceedings, 2009.

[7] A. Liu and R. Dick. Automatic Run-Time Extraction of Communication Graphs From Multithreaded Applications. Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis, 2006.

[8] S. Wen, et al. Measuring and Visualizing Thread Communications for Pthread Applications. 13th International Conference on Parallel and Distributed Computing, Applications and Technologies, 2012.

[9] E. Cruz, et al. Using the Translation Lookaside Buffer to Map Threads in Parallel Applications Based on Shared Memory. IEEE 26th International Parallel and Distributed Processing Symposium, 2012.

[10] S. Kamil, et al. Communication Requirements and Interconnect Optimization for High-End Scientific Applications. IEEE Transactions on Parallel and Distributed Systems, Vol. 21, No. 2, February 2010.

[11] D. Ajwani, et al. Graph Partitioning for Reconfigurable Topology. IEEE 26th International Parallel and Distributed Processing Symposium, 2012.

[12] R. Urbal, et al. Multi2Sim: A Simulation Framework for CPU-GPU Computing. Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, 2012.

[13] A. Hagberg, et al. Exploring network structure, dynamics, and function using NetworkX. Proceedings of the 7th Python in Science Conference, August 2008

[14] C. Bienia and K. Li. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. Proc. of the 5th Annual Workshop on Modeling, Benchmarking and Simulation, June 2009.