Network Synthesis for Database Processing Units

Andrea Lottarini Stephen A. Edwards Kenneth A. Ross
Martha A. Kim
Columbia University, New York, NY
{lottarini,sedwards,kar,martha}@cs.columbia.edu

ABSTRACT

We explore on-chip network topologies for the Q100, an analytic query accelerator for relational databases. In such data-centric accelerators, interconnects play a critical role by moving large volumes of data. In this paper we show that various interconnect topologies can trade a factor of 2.5× in performance for 3.3× area. Moreover, standard topologies (e.g., ring or mesh) are not optimal.

Significant prior work on network topology specialization augments generic topologies with additional dedicated links. In this paper, we present a network specialization algorithm that builds a specialized network first then introduces a generic network as a fallback. We find our algorithm produces networks that are 1.24× slower than the highest-performance generic topology considered (a fat tree), and 18% smaller than the least expensive (a double ring). Moreover, our method produces topologies that outperform those produced by others by 1.21× while being 25% smaller.

1. INTRODUCTION

Analytic query processing is a mature and critical business application. Given the ubiquity of these workloads and the exponentially growing data sets on which they operate [16], it is crucial that their performance and efficiency be optimized. There have been multiple academic proposals to augment Database Management Systems (DBMS) with specialized hardware [27, 28, 4, 12], as well as recent industrial designs that implement such solutions [20, 18].

This paper explores how to design on-chip networks ("NoCs") for such systems. We consider the Q100 database processor [27, 28] whose design principles were recently embraced by Baidu [20]. The Q100 contains a heterogeneous set of fixed-function processing elements; *tiles* in Q100 terminology. Each tile implements a relational operator such as a join or filter. These processing elements operate on streams of data corresponding to columns of the database. The Q100 architecture can readily exploit both pipeline parallelism: performing different operations on the elements of a column, and data parallelism: performing the same operation on multiple columns. This allows the Q100 to process queries faster, using far less power than a general purpose system running a software DBMS.

DAC '17, June 18-22, 2017, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/3061639.3062289

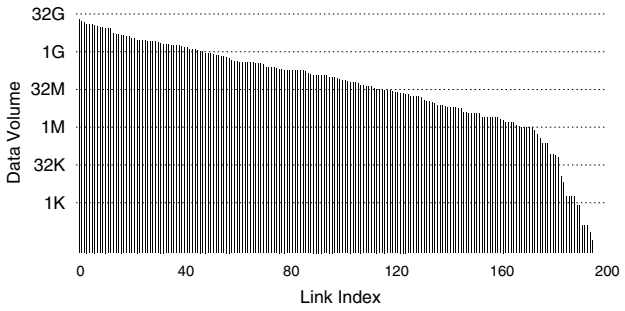

Figure 1: Motivation for limited specialization: the volume of link traffic falls off exponentially from the busiest links, suggesting only a handful of specialized links are necessary. (TPC-H on the Q100)

The tiles are highly specialized and have limited configurability; complex query plans are executed by routing data between tiles. Therefore, the NoC is central to the Q100, greatly influencing both its area footprint and performance. For example, a fat tree NoC would require 40% of the area of the computation tiles where as a simple double ring would consume only 12%. However, the fat tree allows the Q100 to complete the TPC-H benchmark 2.5 times faster than the ring. TPC-H is the most widely used benchmark for query processing analytic workloads [3], both in academia and industry. We will use it in the remainder of the paper in order to evaluate each interconnect's impact on query processing performance.

Ideally, we would want the system to perform as if a full crossbar were present while devoting minimal area to the NoC. The fat tree and the double ring are good examples of high performance and economical NoC topologies, respectively. Moreover, under a fixed area budget, a small, performant NoC creates room for more processing tiles and potentially increased performance.

Relatively few edges carry the vast majority of data in our application, which motivates our approach to custom network design. Figure 1—our results from profiling the TPC-H query plans execution on the Q100—illustrates this. These patterns arise from the workload, as many tiles communicate preferentially with other tiles to implement the computation expressed by query plans. In particular, data is only ever sent over 196 of the 896 possible edges (i.e., 28 output ports times 32 inputs). Hoping existing algorithms could exploit such patterns, we first tried applying two NoC synthesis algorithms found in the literature [15, 19] to the Q100 system. We found that since these methods are strictly additive to the base NoC topology; their solutions will be strictly larger than the base topology, and thus also slower due to the increased degree of the routers in the resulting NoC.

Our network synthesis algorithm remedies this. As others do, we expose a parameter than can be tuned—effectively the degree of specialization—that allows a designer to produce networks that fall between general-purpose and fully specialized. However, unlike the additive state of the art, we introduce dedicated, low-contention links for the most important communication pairs and connect *the remainder* of the ports through a generic topology. In Section 3 we present our algorithm and prove it generates deadlock-free networks. Our experiments show we produce networks that are faster and smaller than competing algorithms. We put more of the high-volume communication on dedicated links while reducing the size of the standard interconnect that serves the less-important traffic.

In Section 5, we use our algorithm to find an interconnect that executes the TPC-H benchmark only 1.24× slower than a fat tree topology while consuming only 82% of the area of a double ring. Compared with NoCs produced by competing algorithms [19, 15], our algorithm reduces NoC area by 25% while increasing system performance by 1.21×.

2. BACKGROUND AND RELATED WORK

2.1 Query Processing Accelerators

As analytic query processing is a stable workload that is widely used in enterprise computing and that operates on highly structured data, it is a good candidate for acceleration. As such, many researchers have proposed acceleration of key operators including hash-joins [13], nested loop joins [24], partitioning [26], sorting [14], filtering and aggregation [9], as well as full analytical queries [27, 4, 20]. Other notable approaches to analytic acceleration include modifications to the storage subsystem to increase the DBMS performance [12] as well as near storage compute [25].

2.2 Background on the Q100

The Q100 [27, 28], whose NoC we consider here, is a spatial architecture for the acceleration of relational analytic queries. The Q100 is composed of a mix of processing elements (called *tiles*), which are specialized units that perform relational algebra operations[1]. An instance of a Q100 accelerator can have replicated tiles in order to exploit parallelism in the query plan. The tiles stream data to each other over an on-chip interconnection network, which is the subject of the experiments that follow.

A SQL query is compiled to a Q100 query plan, which is simply a DAG in which the nodes are tile operations and the edges represent producer/consumer dependencies. The tiles pass data to each other, according to these producer-consumer relations, using an on-chip interconnection network, which is the subject of the experiments that follow. Because the query plan may call for more resources than the target accelerator has (e.g., a plan that requires three sort operations running on a Q100 instance with only one *Sorter* tile) a software scheduler partitions the work into sequential steps, none of which exceeds the accelerator tile resources.

The Q100 belongs to a class of emerging architectures for accelerators that are composed of multiple, possibly redundant processing elements capable of performing key kernel functions. The benefits of such organization is that the accelerator can more easily *adapt* to changing workloads than a monolithic accelerator design [5].

[1]Those types are *Reader*, *Writer*, *Boolgen*, *Colfilter*, *Case*, *ALU*, *Joiner*, *Aggregator*, *Sorter*, and *Merger*.

2.3 Interconnection Synthesis

Networks-on-Chip, or NoCs, are a *de facto* standard for System-on-Chip integration [8, 1]. Application-specific network synthesis techniques have been widely investigated to improve and automate the design of networks for SoCs. The typical approach considers a fixed communication graph with clearly defined endpoints.

In this paper we compare against the two recent network specialization techniques of Koibuchi et al. [15] and Ogras et al. [19]. Both approaches start with a standard NoC topology then augment it with additional links to bypass congestion. The criteria for when a new link is introduced differ between the two methods. Koibuchi et al. produce a new NoC by adding random links in different ways then picking the resulting NoC that shows minimum diameter. This approach is meant to reduce congestion by reducing the amount of hops for all paths regardless of the amount of traffic they carry. In contrast, Ogras et al. use traffic traces to decide which link to add.

By contrast, we begin with an empty network, build links for a user-specified number of high-traffic paths, and finally connect all remaining paths with a network of standard topology. In Section 5 we compare the quality of the network our algorithm produces with both Koibuchi and Ogras's method, and find that our approach exploits specialization in a more resource-conscious way.

Notice that neither our method, nor the other two methods we compare against [15, 19] consider communication timing [11].

A fully generative approach to interconnection synthesis never uses a standard topology. With such techniques, since the space of possible solutions is much larger than for additive techniques, additional constraints are necessary for these methods to find a solution in a reasonable time. Murali et al. [17] provide an algorithm that hierarchically partitions the communication graph. Pinto et al. [22] propose an algorithm relying on weighted unate covering solvers to generate a NoC. Srinivasan et al. introduce an approach that specifies the synthesis problem as an ILP formulation [23]. All these methods use floorplanning and frequency constraints to reduce the space of admissible solutions. In our case we operate at a higher level of abstraction, with no restrictions on the placement of the NoC endpoints. Therefore, these generative methods can not be readily applied.

Lastly, there are techniques to map computation into a standard topology such as a mesh or a torus (e.g. NETCHIP [2]). These solutions are orthogonal to our work and could be applied to further improve performance of the standard fallback networks we utilize.

3. NETWORK SYNTHESIS ALGORITHM

Our algorithm operates in two phases: it begins by building a partial, specialized network by considering a user-specified number of high-traffic edges and building custom, point-to-point links and routers to carry their load. After doing this, it connects all remaining edges through a generic "fallback" network. We describe this below; Figure 4 shows pseudocode.

3.1 Specialization

The algorithm starts with an empty network consisting only of ports that must be connected and, like prior NoC synthesis algorithms, a communication graph. Each node in the graph represents a type of port (e.g., the output of a filter tile or the input to the sorter tile).[2] Each directed, weighted edge indicates the relative amount

[2]Because we target systems that allow multiple instances of each type of tile, a node in our communication graph represents a *set of interchangeable physical ports*, i.e., the same kind of port on identical tiles. This represents a slight extension over prior network synthesis algorithms that assume edges in the communication graph to have a one to one correspondence to NoC ports.

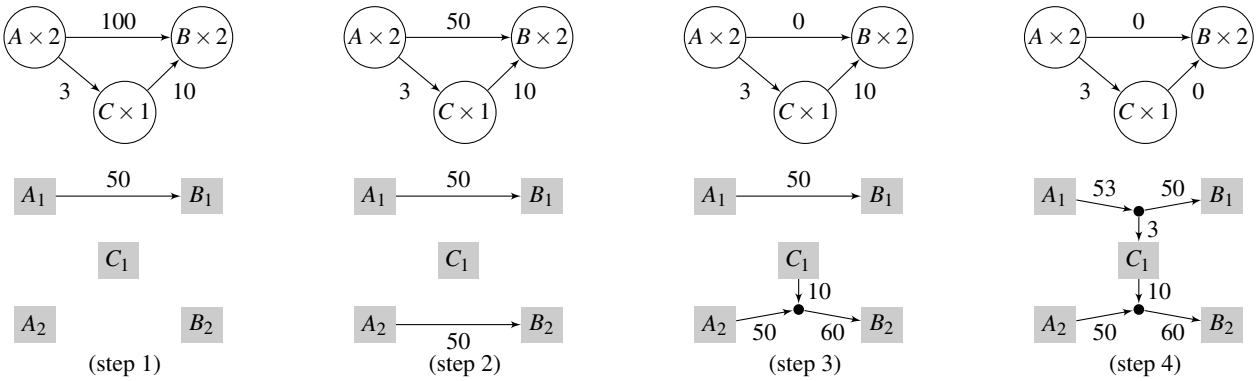

Figure 2: Our algorithm performing four specialization steps on a system with two A processing elements, two B's, and one C. The top graphs depict the communication patterns as observed in simulation; the bottom graphs depict the structure of the synthesized network. In step 1, our algorithm selects the highest-weight edge ($A \to B$) and adds the point-to-point connection $A_1 \to B_1$ with load 50 because there are two A's and two B's sharing the load of 100. In step 2, $A \to B$ has been reduced to 50 but remains the highest so $A_2 \to B_2$ is added. $C \to B$ is highest in step 3, but B_2 already has a connection so a router is added with a link from C_1. Finally, $A \to C$ is selected and another router is added connecting A_1 to C_1, although would also have been possible to add another link from the existing router to C_1.

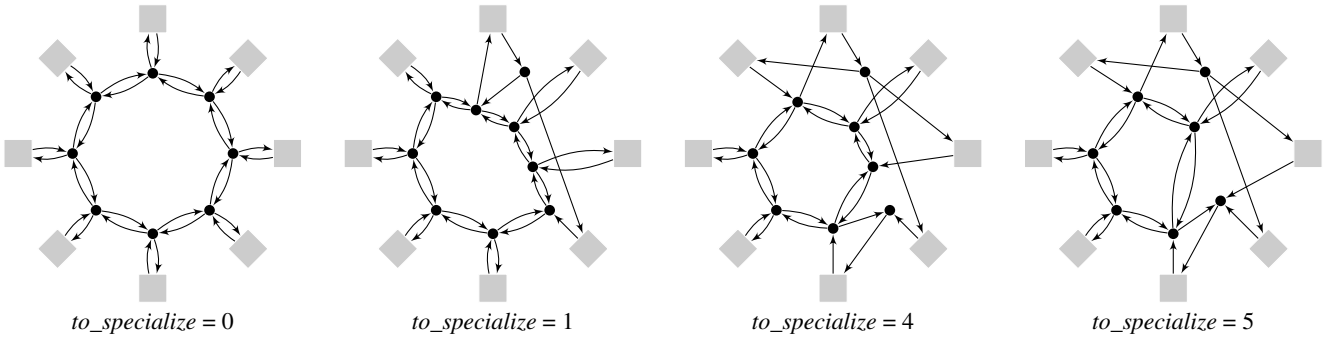

Figure 3: Networks produced by our algorithm employing a ring fallback for a certain system with 8 ports. More specialization steps produces a more customized network and a smaller ring.

or importance of the communication between the source and destination nodes. We will use the term *edge* to denote an abstract connection in the communication graph while a *link* will indicate a physical connection in a NoC.

During specialization (i.e., while *to_specialize* > 0 in Figure 4), we select the highest-traffic edge from the communication graph and introduces a link in the physical network to serve it. Since a Q100 device may have multiple physical replicas of a tile to exploit parallelism between operations on different columns, we select the least heavily loaded instances (as determined by *lightest* in Figure 4) of both the source and destination port. It then annotates the new physical link with an expected load, which is the total load on the communication edge, divided by the minimum number of instances of producer or consumer ports. If the least-loaded source or destination already has an outgoing or incoming link (respectively), this step may require introducing a router to share the port. Once a physical link is introduced, the algorithm deducts this expected load on the newly added physical link from the communication graph's edge.

Figure 2 illustrates this process for a simple example. The top row depicts the communication graph in which each node is labeled with a port type and count. In this example, there are two type-A ports, two type-B's, and a single C. The edges indicate data flowing from one type of port to another with a weight indicating the edge's importance. In this example, communication between A-type and B-type ports is the most important, followed by that from A to C and C to B. The bottom row of Figure 2 depicts the physical network under construction. Here, each physical port is represented explicitly (e.g., A_1 and A_2 are the two type-A ports), routers are introduced (the black circles), and each edge represents a physical point-to-point link with an associated expected load.

Each step from left to right in Figure 2 illustrates a single specialization step. In step 1, the physical link between A_1 and B_1 is marked with a load of 50 because the edge from A to B has a weight (load) of 100 and there are two instances of A and B in the target architecture. Had there been three instances of both port B and A, the physical link would have an expected load of 33.

In step 2 of Figure 2 we see the algorithm choosing the least loaded port when deciding which physical instance of a port should serve an edge. There the algorithm connects A_2 and B_2 because they are unused; A_1 and B_1 already have an expected load of 50. If either the source or destination port already has an incident link, we introduce a router to serve both logical communication streams, as shown in step 3 of Figure 2.

The algorithm proceeds greedily for the desired number of specialization steps, at each one selects and removes the highest-weight

```
# Synthesize a partially specialized network
# endpoints        List of ports in the NoC, each with a type
# edges            Priority queue of communication graph edges
# to_specialize    Number of edges to specialize
# fallback         Ring, mesh, torus, or fat_tree
specialize(endpoints, edges, to_specialize, fallback):
  noc = NoC(endpoints) # Create a network of just endpoints
  r   = noc.create_router() # Add fallback router
  while edges is not empty:
    src_type, dest_type, load = edges.pop() # Get busiest edge
    # Locate the least loaded endpoints
    src  = lightest(endpoints, outgoing, src_type)
    dest = lightest(endpoints, incoming, dest_type)
    # Count how many tiles exist for both source and destination
    src_count = |{e : e ∈ endpoints, e.type = src_type}|
    dest_count = |{e : e ∈ endpoints, e.type = dest_type}|
    if to_specialize > 0: # Create specialized link
      min_count = min(src_count, dest_count)
      noc.add_link(src, dest, load/min_count) # May add router
      # Put the edge back after adjusting its load
      edges.push(src_type, dest_type, load − load/min_count)
      to_specialize = to_specialize − 1
    else: # Create unspecialized link to the fallback router
      noc.add_link(src, r, load)
      noc.add_link(r, dest, load)
  # Replace the fallback router with a fallback network
  r.transform_to_network(fallback)
  return noc

# Return the lightest loaded endpoint of a particular type
# endpoints  List of ports in the NoC, each with a type
# direction  Incoming or outgoing
# type       Endpoint type to consider
lightest(endpoints, direction, type):
  # Consider only endpoints of a certain type
  weights = {e.direction.weight : e ∈ endpoints, e.type = type}
  return the endpoint ∈ weights with minimum weight
```

Figure 4: Our algorithm for NoC synthesis.

edge from the graph. While a larger number of specialization steps produces a more customized network, more specialization is not always better because it can produce more irregular networks that are either too large, too slow, or both. By exposing the degree of specialization (the *to_specialize* parameter in Figure 4) we can quickly generate many networks with differing degrees of specialization, allowing the designer to evaluate and select the best option.

3.2 Generalization

After specialization, the remaining unconnected ports are connected to each other and the network by a standard "fallback" interconnect. In Figure 4, our algorithm first introduces the *fallback* router, which serves as a placeholder for the generic network and to which all edges that were not specialized during the specialization phase are connected. Once every edge has been added to the network, the *fallback* router is replaced with a standard network: we currently support ring, mesh, torus, and fat tree topologies, although others would be possible.

Figure 3 depicts the output of our algorithm using a ring fallback after 0, 1, 4, and 5 specialization steps. As more specialized links are introduced, the *smaller* the fallback network becomes.

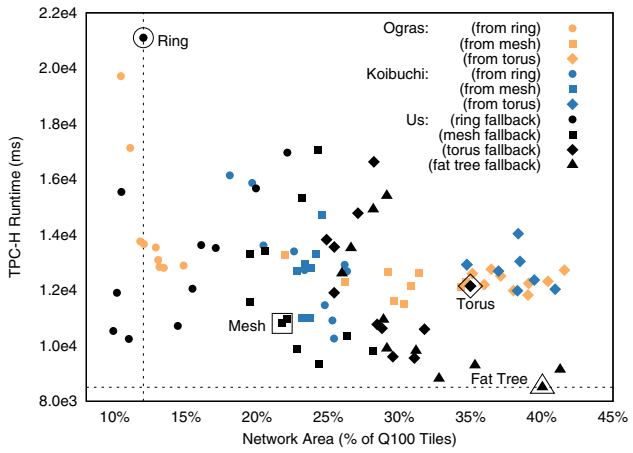

Figure 5: Design space exploration of semi-specialized NoC topologies. We compare our algorithm with Ogras et al. [19], Koibuchi et al. [15], and four generic topologies. We find that our approach to network specialization is the most effective, producing designs that approach the performance of a fat tree and have a smaller resource-cost than a simple double ring, which are indicated by the dashed lines in the plot.

3.3 Proof of Deadlock Freedom

Our algorithm generates networks that are free from deadlock or livelock. We show this by relying on the argument of Dally and Seitz [7]: there is an ordering of channels in the generated network such that every path will traverse channels in descending order.

We rely on the fallback network already having this property. Known deadlock-free routing algorithms exist for each type of fallback network we currently support. For example, a mesh can be made free of deadlocks by using dimension ordered routing or the turn model [10]. As a result, we can safely treat traffic that crosses our fallback network as going through a deadlock-free black box; we can treat it as a single edge in the ordering argument.

The generated part of the network use minimal destination based routing with no adaptivity. Specifically, data follows exactly two kinds of routes. For paths considered during the specialization phase of our algorithm, our network sends data through at most two routers: one connected to the source tile and one connected to the destination tile. For all other paths, data enters at most one router, then traverses the fallback network, then traverses at most one router to reach its destination. Once data emerges from the fallback network, it never reenters.

Our algorithm enforces this property by construction. During the specialization phase, each time a new link is added, the source router's routing table is updated to steer data sent to the destination through the newly added link. For every other path, the relevant routers are instructed to send data to the fallback network instead. Furthermore, these paths do not interfere with each other and the routing tables remain fixed throughout the system's execution.

Thus, every path traverses each of the following links, in order, no more than once:

source → specialized router → fallback router →
path within the fallback network → fallback router →
specialized router → destination

This is the total order on link types that Dally and Seitz's argument demands; our generated networks are deadlock-free.

4. EXPERIMENTAL METHODOLOGY

We evaluate interconnect topologies using a Q100 with 18 tiles and 16 input and output ports to memory.[3] Because a tile may have multiple inputs or outputs, the network will have 66 input ports and 76 output ports. The slight skew is due to the fact that the tiles in the Q100 design tend to have more inputs (network outputs) than outputs (network inputs). Each link is 32 bits wide.

To evaluate a network, we consider its performance relative to its size. We use CONNECT [21] to produce FPGA-optimized, synthesizable RTL from a network description. We then synthesize the RTL using Quartus (targeting a mid-range Altera Stratix 5SGXEA7N1F45 FPGA) to obtain the network size and maximum clock frequency. In order to minimize the NoC area we fix the buffer count to the minimum amount allowed by the tool (4 entries), use Input Queued routers, and turn off all pipelining options. To calculate the overall performance of a network, we simulate TPC-H on our cycle-level Q100 simulator using the network in question. This produces a total cycle count for the workload which, multiplied by the clock period from Quartus, produces the total runtime. We then limit the frequency of the overall design to the frequency of the slowest tile in our implementation: the *merger* tile which operates at a frequency of 145MHz.

In all fallback networks we use minimal destination-based routing. The only exceptions are the fat trees which require dynamically changing routing tables to ensure non-blocking communication [6]. CONNECT does not support dynamic routing, so while the dynamic policy is accounted for in our simulation, the area and frequencies derived from CONNECT correspond to simpler static routers and thus should be considered lower bounds on fat tree area and upper bounds on fat tree frequency. This does not impact our final conclusions as the data show that even with these allowances, the fat trees are the largest interconnects.

To gather the communication graph, we simulate 19 (out of 22) TPC-H queries and register the amount of data flowing across between each combination of of tiles' ports; the three queries that are left out contain operators that are not supported by our current compiler infrastructure. TPC-H is the standard benchmark for analytic query processing workloads [3]. We run the queries on a database with a scaling factor of 1, meaning the whole database is 1GB.

For scheduling query plans to the finite resources of the Q100 device we employ a greedy "longest job first" heuristic that schedules the longer latency operations first on the processing element which would require the smallest number of network's hops for all its input operands. We found that, on average, this simple greedy heuristic produces schedules that are only 5% slower than the best schedule out of 10000 random valid schedules.

5. EXPERIMENTAL RESULTS

We compare with two other network specialization algorithms developed by Ogras et al. [19] and Koibuchi et al. [15]. The idea behind both methods is to start with a standard network topology to which dedicated links are added for important connections. Ogras, at each step, exhaustively consider all possible pairs of non-adjacent nodes and greedily select the one which reduces a cost function the most. The cost function they use is the free packet delay of each communication – a metric proportional to the hop count – weighted by the amount of traffic it carries. Koibuchi et al. instead produce a fixed number or randomly augmented graphs – they found 100 to be a reasonable number – from which they select the one with the smallest diameter. Although the original

[3]Three *Aggregator*, two *Boolgen*, one *Sorter*, five *Colfilter*, two *ALU*, two *Joiner*, one *Merger*, two *Case*

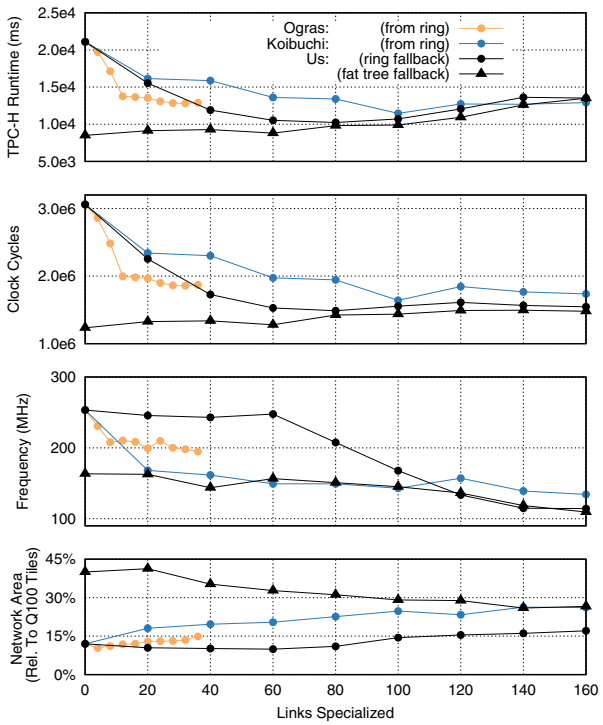

Figure 6: The best NoC configuration we found came from specializing 60 edges and employing a double ring fallback. This network is superior to those from other methods because our NoCs are more resource conscious.

paper by Ogras et al. only considered a mesh base topology, we apply their method to specialize a ring and torus as well. Like us, both these strategies seek to balance the benefits of specialized and generic network topologies. However, whereas their specialization is strictly additive to the generic network starting point, we start with no network, introduce dedicated links, and lastly fill in with a generic network.

For all algorithms, we sweep the degree of specialization from no specialization (i.e., the base network topology) to fully specialized. Figure 5 plots the performance and area trade-off of the resulting networks for different degrees of specialization. It is clear that the most desirable networks are bidirectional rings that are slightly specialized. This is consistent with the observed patterns of communication for the Q100 accelerator (Figure 1) where a small set of edges carry most of the traffic. Koibuchi et al. also observe how ring topologies provide the most benefits with their approach. However, the networks produced by our synthesis algorithm outperform both standard topologies and the specialized topologies produced by the other algorithms. Figure 5 shows how all points in the Pareto frontier, other than the fat tree, are NoCs produced by our algorithm.

To develop our intuition as to why our method outperforms the others, in Figure 6 we analyze the impact of growing specialization on all figures of merit: completion time, clock cycles, frequency, and area. In these plots x links specialized for Ogras and Koibuchi's method means x dedicated links added, and for us x specialization steps. Note that Ogras does not specialize beyond 40 links due to the method's constraint that each router can have no more than one

long-distance link.[4] We plot the data for the strongest base topology, the ring, as well as the fat tree which was the least congested at all times[5].

Starting with no specialization, Figure 6 shows that a ring topology will have the smallest area, but performs poorly. As we increase specialization, the number of clock cycles tends to drop, but the frequency does too as the specialization breaks the regularity of the network. For the same reason the area of the ring will increase. Therefore, performance will not necessarily increase with specialization. This is exactly what limits the other techniques we are comparing against. Networks produced by Koibuchi or Ogras' method might outperform ours when considering raw clock cycle count and a given specialization target. However, they are much larger and can only operate at slower frequencies with respect to our more resource conscious NoCs. It is also clear that using information about the traffic patterns helps ours and Ogras' method produce networks that are better than what the random approach can achieve. Finally, notice how a dense fallback network like a fat tree "slims down" as we remove incoming and outgoing connections. However, because the specialized network is now less versatile, congestion slowly increases as well. For all methods frequency drops rapidly after 60 links are specialized, emphasizing the importance of regularity in the NoC structure.

6. SUMMARY AND CONCLUSIONS

Database Processing Units are a promising class of accelerators that target analytical query processing, a ubiquitous application. To the best of our knowledge this is the first paper that specifically targets the design and optimization of NoCs for these types of systems. The algorithm we presented allows rapid network design space exploration by sweeping the number of specialized links. It finds networks that perform better per unit area than standard topologies and custom networks obtained by other algorithms for NoC synthesis. In particular, our algorithm found networks that improve application performance by 1.21× and reduce area by 25% relative to state-of-the-art network synthesis techniques.

7. ACKNOWLEDGMENTS

This work is supported by the National Science Foundation under CCF-1065338 and by C-FAR, one of the six SRC STARnet Centers sponsored by MARCO and DARPA. We also wish to thank Luca Carloni and the anonymous reviewers for their contributions.

8. REFERENCES

[1] L. Benini and G. De Micheli. Networks on chips: A new SoC paradigm. *IEEE Trans. on Computers*, Jan. 2002.

[2] D. Bertozzi et al. NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. *IEEE Trans. PDS*, 2005.

[3] P. Boncz, T. Neumann, and O. Erling. *TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark*. Springer International Publishing, 2014.

[4] E. S. Chung, J. D. Davis, and J. Lee. Linqits: Big data on little clients. In *ISCA*, 2013.

[4]We experimented with relaxing this constraint to create a fairer comparison to our method. However, this did not improve performance and since this was not part of the original algorithm, we do not report results for it.

[5]We do not apply Ogras or Koibuchi's method starting from a fat tree since it is already non-blocking and therefore will not benefit from extra links.

[5] J. Cong et al. Composable accelerator-rich microprocessor enhanced for adaptivity and longevity. In *ISLPED*, 2013.

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

[7] W. J. Dally and C. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE ToC*, (5), May 1987.

[8] W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In *DAC*, 2001.

[9] C. Dennl, D. Ziener, and J. Teich. Acceleration of SQL restrictions and aggregations through FPGA-based dynamic partial reconfiguration. In *FCCM*, 2013.

[10] C. J. Glass and L. M. Ni. The turn model for adaptive routing. In *ISCA*, 1992.

[11] W. H. Ho and T. M. Pinkston. A methodology for designing efficient on-chip interconnects on well-behaved communication patterns. In *HPCA*, 2003.

[12] S.-W. Jun et al. Bluedbm: An appliance for big data analytics. In *ISCA*, 2015.

[13] O. Kocberber et al. Meet the walkers: Accelerating index traversals for in-memory databases. In *MICRO*, 2013.

[14] D. Koch and J. Torresen. FPGASort: A high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. In *FPGA*, 2011.

[15] M. Koibuchi, H. Matsutani, H. Amano, D. F. Hsu, and H. Casanova. A case for random shortcut topologies for HPC interconnects. In *ISCA*, 2012.

[16] A. McAfee and E. Brynjolfsson. Big Data: The management revolution. *Harvard Business Review*, October 2012.

[17] S. Murali et al. Designing application-specific networks on chips with floorplan information. In *ICCAD*, 2006.

[18] F. Nagel et al. Code generation for efficient query processing in managed runtimes. *VLDB*, Aug. 2014.

[19] U. Ogras and R. Marculescu. It's a small world after all: NoC performance optimization via long-range link insertion. *IEEE Trans. VLSI*, July 2006.

[20] J. Ouyang et al. SDA: Software-defined accelerator for general-purpose distributed big data analysis system. In *HotChips*, 2016.

[21] M. K. Papamichael and J. C. Hoe. CONNECT: Re-examining conventional wisdom for designing NoCs in the context of FPGAs. In *FPGA*, 2012.

[22] A. Pinto, L. P. Carloni, and A. L. Sangiovanni-Vincentelli. Constraint-driven communication synthesis. In *DAC*, 2002.

[23] K. Srinivasan et al. Linear-programming-based techniques for synthesis of network-on-chip architectures. *IEEE Trans. VLSI*, Apr. 2006.

[24] J. Teubner and R. Mueller. How soccer players would do stream joins. In *SIGMOD*, 2011.

[25] L. Woods, Z. István, and G. Alonso. Ibex: An intelligent storage engine with support for advanced SQL offloading. *VLDB*, 2014.

[26] L. Wu et al. Navigating big data with high-throughput, energy-efficient data partitioning. In *ISCA*, 2013.

[27] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross. Q100: The architecture and design of a database processing unit. In *ASPLOS*, 2014.

[28] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross. The Q100 database processing unit. *IEEE Micro*, May 2015.