# Link Pipelining for an Energy-Efficient Asynchronous Network-on-Chip

Daniel Gebhardt
School of Computing
University of Utah
gebhardt@cs.utah.edu

Junbok You  Kenneth S. Stevens
Dept. of Electrical and Computer Engineering
University of Utah
{jyou , kstevens}@ece.utah.edu

***Abstract*—Wire latency across the links of a NoC potentially limit available bandwidth, especially in deep submicron technology. Pipeline buffers may be placed on long links to increase throughput and flit buffering. In asynchronous (clockless) NoCs, this can be done to only those links that will yield benefits, and is especially useful in heterogeneous embedded SoCs with irregular layouts and traffic. We evaluate two strategies that determine where link pipeline buffers should be placed in the topology. The first compares available link bandwidth, based on physical wirelength, to the throughput needed by each source-to-destination path, for each link. The second adds buffers to a link such that its bandwidth at least matches the throughput of a core's network adapter. These strategies were integrated into our network optimization tool for an application-specific SoC. Simulations used its expected traffic patterns, floorplan-derived wirelengths, and self-similar traffic generation for more realistic behavior. Results show improved large-message network latency and network adapter output buffer delay. There was a slight power increase with the addition of pipeline buffers, but our proposal is a *complexity-effective* improvement by the power-latency product metric. The strategy of pipelining certain links based on their expected traffic provides a more efficient performance increase compared to solely a traffic-oblivious addition of buffers.**

## I. Introduction

### A. Link Pipelining and Asynchronous NoCs

A network-on-chip (NoC) is an on-chip communication concept to help reduce the difficulty of integrating many intellectual-property (IP) blocks (cores) into a system-on-chip (SoC). Data transferred between cores is formed into packets and sent through shared links and switching circuits similar to macro-scale computer networks.

A link of a NoC can be *pipelined* by adding a latch or register-based buffer along its length when its wire delay limits throughput. This often occurs with long links, small process technology, and relatively fast clock speeds. Another benefit is improved throughput from the additional buffering space it provides, assuming a compatible link-level flow-control.

This work focuses on a class of SoCs termed *application-specific* in which the NoC is designed and optimized for performance and energy with knowledge of the specific tasks done by the SoC. These fixed-function SoCs can use many specialized cores, and thus are less flexible than general-purpose chip-multiprocessors (CMPs) or even platform-SoCs. However, knowledge of the traffic, such as bandwidth between particular cores, provides an opportunity for optimization. In this paper, we explore the system-level effects of link pipelining for an asynchronous NoC that has already been optimized for link-specific bandwidth requirements using an existing tool [1].

Many globally-asynchronous locally-synchronous (GALS) interconnect solutions rely on a clock, either with standard synchronous clock distribution, or a mesochronous method. However, an asynchronous (also called clockless) network has a number of potential advantages over a clocked network in a GALS environment. Standard arguments for asynchronous (async) circuit design include robustness to process/voltage/temperature variation, and average-case instead of worst-case performance. However, there are also NoC-specific considerations. In a synchronous NoC, the clock tree for all routers and pipeline buffers can consume significant power as shown in a heterogeneous network [2], and in a large CMP, 33% of router power [3]. Many SoC designs have quite bursty and "reactive" traffic. In this case, async methods are beneficial in that they consume little dynamic power during periods of low traffic without relying on clock gating techniques.

Available bandwidth on each async link can vary depending on wirelength between sender and receiver. This is in contrast to clocked networks that commonly use a single frequency for all routers, which is wasteful on links not requiring high bandwidth. Bandwidth can be modified in an async NoC by changing the distance between routers, or by inserting link pipeline buffers on only those links that will benefit. We explore this concept here, with a proposal of novel insertion strategies for heterogeneous SoC designs.

Figure 1 shows the functional components of an async NoC, and how it interfaces with the cores of an SoC. A core has an interface using a standard protocol such as OCP or AMBA. The network adapter converts this protocol to the one used in the particular NoC design. It can also synchronize between two timing domains; in our work, between the core's clock domain and the asynchronous network domain. This also enables each core to operate at its own frequency, as is done in GALS design. An example network adapter implementation of this concept is presented in [4].

An asynchronous channel, which transfers data from a sender to a receiver, is shown in Figure 2. Instead of a synchronous clock signal determining when data should be

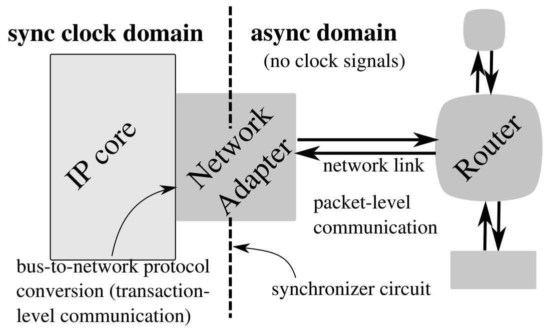

Figure 1: Component diagram of an asynchronous NoC.

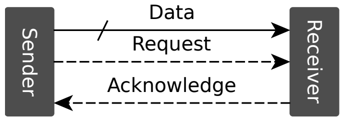

Figure 2: Signals of an asynchronous channel handshake.

read, a handshaking protocol performs the function. Typically a *request* and *acknowledge* signal are used to accomplish this. The sender generates the request signal to indicate new data is available. The receiver responds with the acknowledge signal, indicating data is stored.

### B. Related Work

There have been a number of NoC proposals for incorporating storage and/or control logic within inter-router links. The work of *iDEAL* showed that the performance penalty of reduced-complexity routers with few input/output buffers can be improved by using links containing storage elements [5]. It demonstrated that for a traditional synchronous NoC and mesh topology, moving storage from routers to the links significantly reduced network power at a very slight performance reduction. Link pipelining is described for the *XPipes* network components [6]. These are placed primarily to meet clock timing requirements. Error detecting link pipeline circuits were designed to achieve greater NoC robustness while maintaining high throughput [7]. Elastic buffers, similar to the asynchronous buffers used here, were used to reduce router complexity by using the link as a distributed FIFO buffer [8]. Throughput per energy was improved compared to the baseline architecture. Elastic and asynchronous link pipelining was explored in [9], but with an ad hoc approach to determine where and when a buffer should be inserted on a link. It also did not evaluate effects on large-message latency. A number of energy-efficient proposals, including pipelined links shared between multiple sources, is given by [10]. It uses a standard mesh topology and homogeneous SoC for evaluation and does not address the optimization problem of determining the the number of buffers on each link. Link pipelining for a delay-insensitive asynchronous NoC are described in [11], where multiple virtual channels can overlap packet transmissions at the flit level to maintain high link utilization. The paper did not describe the conditions or depth of the pipeline, nor was a system-level evaluation of the proposal given.

## II. Link Pipelining

### A. NoC Component Overview

The goal of pipelining a link is to reduce the cycle-time created by wire delay by inserting a buffer between sender and receiver. Throughput along a link is improved, at the expense of single-flit latency and a slight power increase over only wire repeaters. This organization is illustrated in Figure 3, where a buffer is placed between a router and network adapter. Our asynchronous pipeline buffer is composed of a bank of latches (rather than flip flops) and a handshake controller. This arrangement is called a *linear controller*. The use of latches saves almost half the area and power compared to a traditional synchronous flip-flop design. Note, the pipeline buffer does not negate the need for wire repeaters (large inverters). In this work, we assume the buffers for the separate input and output channels are located in the same proximity, but this is not required. From this point forward, when we use the term *buffer* in the context of describing the network configuration we refer to the collection of latches and controllers for both channel directions.

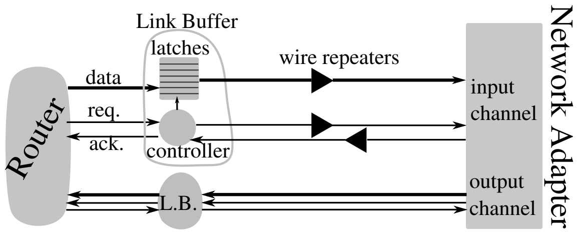

Figure 3: Organization of NoC link components showing detail for a router's output channel, and the equivalent link buffer (L.B.) for its input channel.

Our asynchronous router is designed for efficiency and simplicity. Each switch directs a flit to one of two output ports. With bi-directional channels, this results in a three-ported router. Possible topologies include a ring, tree, or irregular, but this work considers trees, as they have the minimal number of routers. The packet format consists of a single flit containing source-routing bits in parallel, on separate wires, with the data bits. The packet is switched through a demultiplexer controlled by the most-significant routing bit. The bits are simply rotated for the output packet. The number of required routing bits is determined by the maximum hop count of a network. This format has the overhead of requiring routing bits with every flit, but does not require an extra header-flit carrying routing information common to other packet formats. Its energy overhead is further reduced in that series of packets following the same source-to-destination path do not cause switching in the routing bits. More details are shown in [1], and we use the same 32-bit data + 8-bit route format for this work.

This design uses an Artisan cell library on the IBM 65 nm *10sf* process. We used post-layout back-annotation to evaluate the router/linear controller's latency, HSPICE simulation for energy measurement, and leakage power from SOC Encounter.

Latency and energy results are shown in Table I, and assume 25% of bits switch each flit. Forward latency is the time between when an input channel receives a request signal to the time the router/buffer asserts the outgoing channel's request signal.

TABLE I: Circuit-level properties of router and linear-controller circuits.

| | Energy/flit (pJ) | Leakage ($\mu$W) | Forward Latency (ps) |
|---|---|---|---|
| Router | 1.03 | 9.76 | 460 |
| Lin. Cntrl. | 0.45 | 1.21 | 130 |

### B. Insertion Strategies

We propose two strategies to determine under what conditions a link pipeline buffer should be inserted. The first, *path-specific* buffer insertion, pipelines links that require a throughput greater than a fraction, $k$, of the link's available bandwidth (ABW). This strategy is termed *path-specific* because the required throughput ($\mathcal{T}_{req}$) is derived from the source-to-destination paths that utilize the link; it is the sum of average throughput of each of these paths ($\mathcal{T}_p$). The number of buffers, $\aleph$, to insert on link $\ell$ is

$$\aleph_\ell = \left\lfloor \frac{\mathcal{T}_{req}}{(ABW_\ell \times k)} \right\rfloor \text{ where } \mathcal{T}_{req} = \sum_{path\ p\ using\ \ell} \mathcal{T}_p$$

The value of $k$ is a user-parameter, and is explored in Section IV. In other words, this is an insertion threshold based on the percentage of link utilization.

For the application-specific SoC we consider in this work, a common representation of core-to-core traffic requirements is a *core communication graph* (CCG) as shown in Figure 4. The expected average throughput required between cores is shown with an annotated edge. For this particular CCG, traffic is assumed to be equal in both directions, but this is often not the case.

Given a topology and CCG, each link in the topology is attributed a required throughput. Available link bandwidth is based on link wirelength, as shorter links have faster cycle-time. The intuition behind this is that high-traffic links will benefit from additional buffering to reduce contention in the preceding routers, and also to decrease latency from a receiving router's *ack* assertion (indicating the next flit may be sent) to a sending buffer *req* assertion (indicating the next flit is ready).

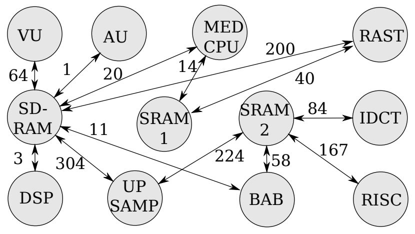

Figure 4: MPEG4 decoder *core communication graph*, with edges in MBytes/second.

The second strategy adds pipeline buffers to links with an available bandwidth less than the throughput of the network adapters. We call these *core-throughput matching buffers* (CTMBs). This is analogous to wire pipelining for clocked networks when a link fails to meet the timing requirement derived from the clock period. For async systems, however, this is optional; links can be slower than the sending or receiving component and don't have to meet a particular clock period. The intuitive advantage behind this strategy is to make sure cores are never slowed down by a wire delay in sending or receiving a flit. This advantage may or may not be worth the additional power overhead of the buffers, depending on system requirements and communication properties.

We integrated these two strategies within our async network topology and placement tool, *ANetGen* [1]. This tool searches for a network solution with the best power and latency characteristics, and generates a topology, floorplan, and SystemC simulator for the network.

## III. Methodology

We evaluate our link pipelining proposals using an abstraction of an MPEG4 decoder SoC, originally described by [12] but with throughput changed to that shown in Figure 4. This benchmark has been used in several other NoC research projects [1], [9], [13], [14].

*ANetGen* was configured to produce a topology and placement of the routers and pipeline buffers for various values of the path-specific $k$ parameter. This was done for two sets of link pipeline configurations; with path-specific buffers only, and with both path-specific and CTMBs. Simulations were run for each of the sets described above, where the $k$-parameter was varied to change the threshold of where path-specific buffers were inserted. $k$ values of 1.0, 0.05, 0.03, and 0.02 resulted in the total number of path-specific buffers of 0, 1, 3, and 7, respectively. Due to the details of this SoC's specific traffic requirement, and the floorplan, there was only one link where path-specific buffers were inserted – the link connecting the *sram2* IP core. When CTMBs are added, seven more buffers are inserted on the longest links.

Our simulator used Orion 2.0 wire models [15] to estimate dynamic wire energy, but we changed the wire leakage parameters to match the IBM process in our router and buffer evaluation. The Orion NVT library estimated this power to be between 5x and 7x greater than the IBM process. We also changed the fraction of data bits that are assumed to switch in successive flits to 0.25 from 0.5 in our previous work.

The simulator uses a self-similar traffic generator, rather than the more common Poisson-distribution. This provides bursty traffic at the message level that more closely models a real application [16].

This study uses *message latency*, rather than packet latency, because we believe this is a better metric for representing movement of a useful piece of data, especially in media processing. The traffic generator outputs chunks of data, or *messages*, that simulate the requests from software to the transaction layer of SoC communication. The self-similar behavior is seen in the times of generation of these messages. A message is broken down into contiguous packets to be sent over the network as fast as it will allow. Message generation cannot be stalled by the network, but message latency through the network to a destination IP core is delayed by the bandwidth of the network and transient effects of packet contention with other paths. As such, message latency is defined as the time the first packet of the message leaves the sending core's output buffer and enters the network to the time the tail packet leaves the network and enters the destination core. We set the traffic model's bias $b$-value to be 0.7 on each CCG path, the simulated time of 67 ms, a 256-byte message size, and a 2 Gflit/sec network adapter sending rate. More details on the traffic generator, simulator, and models are described in [1].

Another performance metric we measured and present in Section IV is the packet delay in a network-adapter's output buffer. This is defined as from the time the traffic generator places a packet in the output buffer to the time the packet enters the network. The entry time is set for each packet by the traffic generator when it pushes an entire message to the buffer at once. Therefore, the last packet of a 64-packet message would have a minimum delay of 64 sender-cycles. The traffic generator operates detached from the network flow control so an infinite buffer is needed to accept its traffic at any time. The network then empties that buffer as quickly as possible.

The floorplan with 3 path-specific buffers on the *sram2* link and CTMBs is shown in Figure 5. Cores are labeled with their names, routers are dark circles, and pipeline buffers are gray squares. Logical connectivity between components is shown with black lines. Note that this does not show actual wire routing. Network adapters are assumed to be where a link is attached to a core. Notice that buffers are equally spaced across a link. The longest, from *au* to router *4* has three pipeline buffers: pb0, pb1 and pb2. The path-specific buffers connecting *sram2* are pb7, 8 and 9.

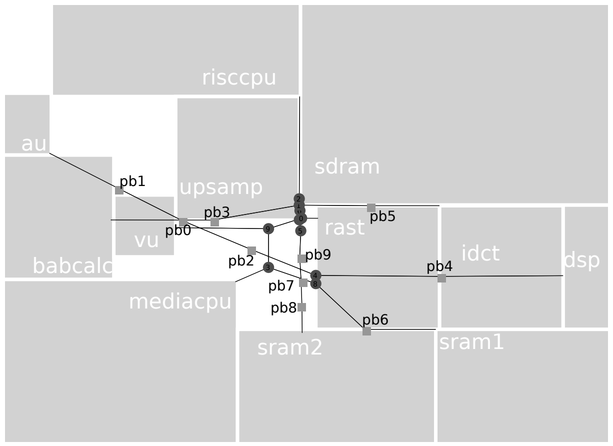

Figure 5: SoC floorplan showing cores, routers, buffers, and topology, cropped and edited for legibility.

## IV. Results

### A. Power and Latency

A succinct metric to determine the benefit of adding link pipeline buffers is *power-latency product* (PLP) of the network. This is similar to the *energy-delay product* commonly used in CPU architecture or VLSI comparisons. The *power* term is the sum of dynamic and leakage power of the routers, wires, and wire pipeline buffers. The *delay* term is the mean packet latency through a network adapter's output buffer, added to the mean message (256 bytes) latency through the network. Delays were normalized to give equal weight to network and output buffer latencies. Figure 6 shows this metric on the Y-axis, with the X-axis showing the total number of path-specific buffers inserted on all links. Data is given in two series, with and without additional core-throughput matching buffers.

The addition of one path-specific buffer greatly lowers (improves) PLP. A slight additional improvement is seen with three buffers. PLP gets worse after this when seven buffers are used. This chart is also useful in showing the benefit of the CTMBs. With just CTMBs, the PLP is improved, but only very slightly from the initial solution. With three to seven path-specific buffers, the combination of both has the best PLP, with the optimal point approximately three path-specific buffers.

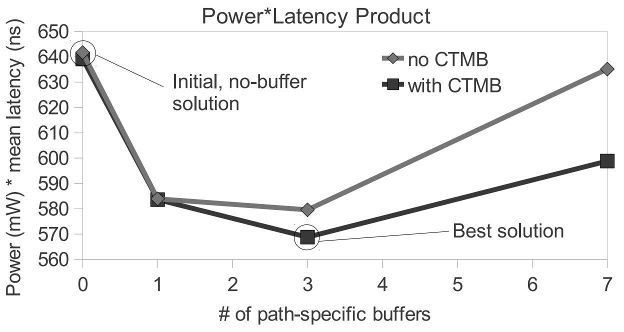

Figure 6: Power-latency product for various numbers of buffers inserted based on link utilization percentage. Results shown with and without core-throughput matching buffers.

Power consumption of various network configurations is shown in Table II. Results are organized first with only the path-specific buffers, and second with both path-specific and CTM buffers. All power values are a sum of dynamic and leakage power for each component. The wire power is due to the drivers/repeaters (large inverters) along its length. For these experiments the topology and router placement are left constant, with only the number of buffers varying. Therefore, the power of the routers and wires is constant because the same traffic is sent in each trial and the total wirelengths in each case is nearly identical.

Mean latencies are shown in Table III for packets in the network adapter's output buffer and messages in the network. Both of these metrics generally improve as more path-specific

TABLE II: Power (dynamic+leakage) of buffer configurations.

| | Routers | Wires | Buffers | Total (mW) |
|---|---|---|---|---|
| Without core-throughput matching buffers | | | | |
| 0 path bufs | 1.64 | 4.12 | 0.00 | 5.76 |
| 1 path bufs | 1.64 | 4.12 | 0.12 | 5.88 |
| 3 path bufs | 1.64 | 4.12 | 0.36 | 6.13 |
| 7 path bufs | 1.64 | 4.12 | 0.85 | 6.61 |
| With core-throughput matching buffers | | | | |
| 0 path bufs | 1.64 | 4.12 | 0.06 | 5.82 |
| 1 path bufs | 1.64 | 4.12 | 0.18 | 5.94 |
| 3 path bufs | 1.64 | 4.12 | 0.42 | 6.18 |
| 7 path bufs | 1.64 | 4.12 | 0.90 | 6.66 |

TABLE III: Mean latency (ns) varying path-specific buffers.

| | # of path-specific buffers | | | |
|---|---|---|---|---|
| Measurement Location | 0 | 1 | 3 | 7 |
| Without CTMB | | | | |
| core's output buffer | 15281 | 12257 | 11430 | 11644 |
| network | 55.74 | 54.58 | 52.92 | 53.58 |
| With CTMB | | | | |
| core's output buffer | 15332 | 12400 | 11248 | 10693 |
| network | 54.92 | 53.78 | 51.72 | 51.59 |

buffers are used, however with 7, there is a slight rise in latency in the networks without CTMBs.

## B. Message Latency per Path

Besides overall mean message latency, another picture of performance is seen by looking at the message latencies for each source→destination path in the SoC. For example, the path from source core *sram2* to destination core *risccpu* has a median latency of 65 ns with no link buffering, as seen in Figure 7 with the path-ID `21`. These measurements are within the network, after a message has begun its exit of a core's output buffer, until it completely enters a receiving core.

The addition of link buffering improves median latency significantly on some paths, notably `20,21,22` which carry high-traffic from *sram2*. Latency rises slightly with the addition of more buffers. Buffers were added to the link connecting the *sram2* core to a router, which explains the benefit on those paths. Other paths do not benefit from these added buffers. The effects on maximum message latency is not conclusive, although a few paths seem to benefit slightly, such as `0` (*au→sdram*) and `18` (*sram1→rast*). This is from reduced contention, a side-effect of the improved connection to *sram2*.

The effects on per-path message latency by adding CTMBs is shown in Figure 8. All values on the chart are normalized to the configuration with only the noted number of path-specific buffers and no CTMBs (as shown in Figure 7). The total link buffers in the network is shown in the series label, and paths that showed little difference were removed from the figure. Median latency is improved on many paths, and is an indication of increased throughput when the network is uncongested. Note that the paths improved with CTMBs are different than those improved with path-specific buffers; paths `20,21,22` did not show much change.

Maximum latency improvements were mixed, with some drastically worse paths, and many slightly improved ones. The paths showing worse maximum latency are the topologically longest, and thus have the highest probability for contention and delay. The addition of CTMBs increases the rate messages can enter the network, but not necessarily provide beneficial throughput increases "downstream." The effect is, in the worst case, longer waiting times within the network rather than in the core's output buffer. The benefit of path-specific buffers to maximum latency seems to apply less broadly than median delay benefit. However, some paths do benefit from an increasing number of buffers such as `path 21` in 65 nm, connecting *sram2→risccpu*.

These results show that the addition of link pipeline buffers can improve NoC latency of a large chunk of data, but the optimal insertion parameters for a particular design should be explored. CTMBs offer decreased median and maximum message latency for many paths, at the expense of increased maximum latency on some, and a slight power penalty. Path-specific buffer insertion improves message latency for some paths, but not all. It does not seem to worsen maximum latency for any path, and lowers overall mean packet latency when using CTMBs, or without. In summary, the optimal configuration for this SoC, considering energy-efficiency and overall performance, is three path-specific buffers with CTMBs.

## V. Conclusion

In this work we have introduced two strategies for determining which links of an asynchronous NoC should be pipelined. We incorporated these strategies into our NoC optimization tool, *ANetGen*. One of the advantages of an asynchronous network is that this type of optimization can be targeted at specific links in the network, guided by expected traffic patterns. This is in contrast to a typical synchronous NoC that must change frequency on the whole network, or use more synchronizers between clock domains of different frequencies.

For a benchmark SoC, our tools varied the number of link pipeline buffers under the two strategies. Analysis of the simulation results showed the greatest benefit to power*latency product was using three path-specific buffers with CTMBs The path-specific buffer insertion strategy is *complexity-effective* relating to energy, in that it increases performance more than it costs in energy, with and without the use of CTMBs.

We will further explore the benefits of link-specific pipelining in our network on a variety of other SoC traffic models and under varying conditions, such as burstiness. We especially want to explore how this is beneficial at smaller process technology nodes, at 32 nm and beyond. where wire delay continues to worsen relative to transistors.

## Acknowledgments

This research is supported by the National Science Foundation under grant CCF-0702539 and Semiconductor Research Corporation under task 1817.001. We would like to thank ARM and IBM for providing the 65 nm library cells and process technology.

## References

[1] D. Gebhardt, J. You, and K. S. Stevens, "Comparing Energy and Latency of Asynchronous and Synchronous NoCs for Embedded SoCs," in *Proc. Int'l Symp. on Networks-on-Chips*, May 2010, pp. 115–122.

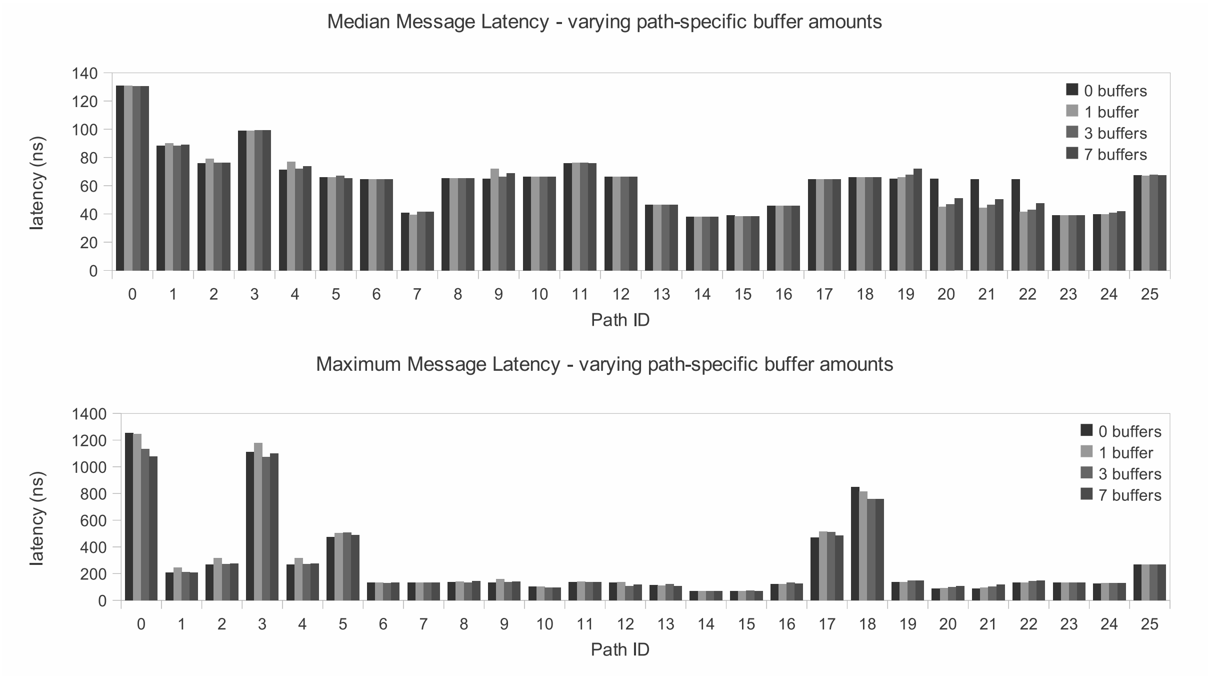

Figure 7: Median and maximum message latencies through the network for each source-to-destination path. Data series represent various numbers of path-specific buffers, and no CTMBs.

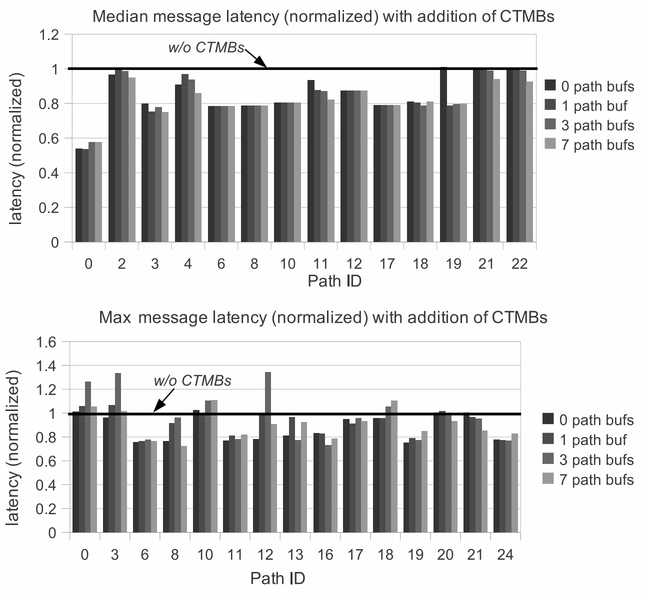

Figure 8: Change in message latency through the network with the addition of CTMBs. Values are normalized to the network configuration with path-specific buffers only. Only paths exhibiting a relevant change are shown.

[2] I. M. Panades, F. Clermidy, P. Vivet, and A. Greiner, "Physical implementation of the dspin network-on-chip in the faust architecture," in *Proc. Int'l Symp. on Networks-on-Chips*, 2008, pp. 139–148.

[3] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar, "A 5-ghz mesh interconnect for a teraflops processor," *IEEE Micro*, vol. 27, no. 5, pp. 51–61, 2007.

[4] T. Bjerregaard, S. Mahadevan, R. G. Olsen, and J. Sparsø, "An OCP compliant network adapter for GALS-based SoC design using the MANGO network-on-chip." in *Proc. of Int'l Symp. on System-on-Chip*, 2005, pp. 171–174.

[5] A. K. Kodi, A. Sarathy, and A. Louri, "ideal: Inter-router dual-function energy and area-efficient links for network-on-chip (noc) architectures," in *Proc. Int'l Symp. on Computer Architecture*, 2008, pp. 241–250.

[6] D. Atienza, F. Angiolini, S. Murali, A. Pullini, L. Benini, and G. De Micheli, "Network-On-Chip Design and Synthesis Outlook," *Integration-The VLSI Journal*, vol. 41, no. 3, pp. 340–359, 2008.

[7] R. R. Tamhankar, S. Murali, and G. De Micheli, "Performance driven reliable link design for networks on chips," in *Proc. Asia and South Pacific Design Automation Conference*, 2005, pp. 749–754.

[8] G. Michelogiannakis, J. Balfour, and W. Dally, "Elastic-buffer flow control for on-chip networks," in *Proc. Int'l Symp. on High Performance Computer Architecture*, 2009, pp. 151–162.

[9] J. You, D. Gebhardt, and K. S. Stevens, "Bandwidth Optimization in Asynchronous NoCs by Customizing Link Wire Length," in *International Conference on Computer Design*, 2010, pp. 455–461.

[10] J. Kim, "Low-cost router microarchitecture for on-chip networks," in *Proc. Int'l Symp. on Microarchitecture*, 2009, pp. 255–266.

[11] T. Bjerregaard and J. Sparso, "Implementation of guaranteed services in the mango clockless network-on-chip," *Computers and Digital Techniques*, vol. 153, no. 4, pp. 217 – 229, 2006.

[12] E. B. V. D. Tol and E. G. T. Jaspers, "Mapping of mpeg-4 decoding on a flexible architecture platform," in *Media Processors*, 2002, pp. 1–13.

[13] S. Murali, P. Meloni, F. Angiolini, D. Atienza, S. Carta, L. Benini, G. D. Micheli, and L. Raffo, "Designing application-specific networks on chips with floorplan information," in *Proc. Int'l Conf. on Computer-Aided Design*, 2006, pp. 355–362.

[14] A. Pinto, L. P. Carloni, and A. L. S. Vincentelli, "A methodology for constraint-driven synthesis of on-chip communications," *IEEE Transactions on Computer Aided Design*, vol. 28, no. 3, pp. 364–377, 2009.

[15] A. Kahng, B. Li, L.-S. Peh, and K. Samadi, "Orion 2.0: A fast and accurate noc power and area model for early-stage design space exploration," in *DATE*, April 2009, pp. 423–428.

[16] M. Wang, T. Madhyastha, N. H. Chan, S. Papadimitriou, and C. Faloutsos, "Data mining meets performance evaluation: fast algorithms for modeling bursty traffic," in *Proc. International Conference on Data Engineering*, 2002.