A Low-Overhead Asynchronous Interconnection Network for GALS Chip Multiprocessors

Michael N. Horak
ECE Department
Univ. of Maryland
mnhorak@umd.edu

Steven M. Nowick
Dept. of Computer Science
Columbia University
nowick@cs.columbia.edu

Matthew Carlberg
EECS Department
UC Berkeley
carlberg@eecs.berkeley.edu

Uzi Vishkin
ECE Department
Univ. of Maryland
vishkin@umd.edu

Abstract

A new asynchronous interconnection network is introduced for globally-asynchronous locally-synchronous (GALS) chip multiprocessors. The network eliminates the need for global clock distribution, and can interface multiple synchronous timing domains operating at unrelated clock rates. In particular, two new highly-concurrent asynchronous components are introduced which provide simple routing and arbitration/merge functions. Post-layout simulations in identical commercial 90nm technology indicate that comparable recent synchronous router nodes have 5.6-10.7x more energy per packet and 2.8-6.4x greater area than the new asynchronous nodes. Under random traffic, the network provides significantly lower latency and competitive throughput over the entire operating range of the 800 MHz network and through mid-range traffic rates for the 1.36 GHz network, but with degradation at higher traffic rates. Preliminary evaluations are also presented for a mixed-timing (GALS) network in a shared-memory parallel architecture, running both random traffic and parallel benchmark kernels, as well as directions for further improvement.

1 Introduction

A recent NSF-sponsored workshop on networks-on-chip (NoCs) focused on the research challenge of maintaining the scalability of interconnection networks [24]. The consensus is that current techniques, when extrapolated to future technologies, will face significant shortcomings in several key areas. First, power consumption is expected to exceed the budgets for commercial chip multiprocessors (CMPs) by a *factor of 10x* by 2015 following the projected technology roadmap. In addition, latency and throughput are predicted to become significant bottlenecks for system performance. Finally, there are less quantifiable, but significant, issues of increased design time and support for scalability, reliability and ease-of-integration of complex heterogeneous systems. These latter issues are expected to be important requirements for implementating future systems, specifically handling synchronous domains with arbitrary unrelated clock frequencies and allowing dynamic voltage scaling.

The goal of this paper is to address some of these bottlenecks, with the design and evaluation of a low-overhead and flexible asynchronous interconnection network. This work is part of the *C-MAIN* Project (Columbia University/University of Maryland Asynchronous Interconnection Network), an ongoing effort to develop low-cost and flexible NOCs for high-performance shared memory architectures. The target design is a mixed-timing network, shown in Figure 1, consisting of the core asynchronous network surrounded by mixed-timing interfaces. In particular, the network provides fine-grained pipelined integration of synchronous components in a globally-asynchronous locally-synchronous (i.e. *GALS*) style architecture [6, 31, 28]. The synchronous components may be processing cores, function units or memory modules. To support scalability, synchronous components may have arbitrary unrelated clock rates, i.e. the goal is to integrate *heterochronous*

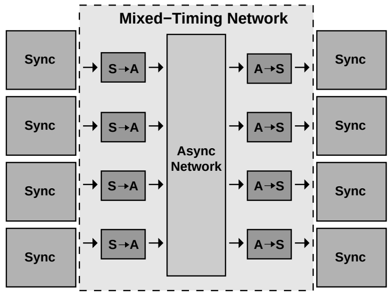

Figure 1. Mixed-timing network

systems [31].[1]

The first contribution is two new highly-concurrent asynchronous network primitives, to support the routing and arbitration functions of the network. Each primitive is carefully designed for high performance and low area and power overheads, using a *transition-signalling*, i.e. two-phase, communication protocol [29], which has only one roundtrip communication per channel per transaction. In principle, transition-signaling is a preferred match for high-performance asynchronous systems, yet it presents major practical design challenges: most existing two-phase asynchronous pipeline components are complex, with large latency, area and power overheads. Mixed-timing interfaces are then designed, based on the approach of [9], with new customized protocol converters. An important overall target of this work is to use standard cell design whereever possible, with static gates and simple one-sided (i.e. "bundled") timing constraints.

The second contribution is the detailed evaluation of the interconnection network at both the circuit and system level. Layouts of the routing and arbitration network primitives are implemented in a commercial 90nm technology following a standard cell methodology, and each primitive is compared in detail with recently-published comparable synchronous primitives implemented in the same technology [2], which use a latency-insensitive style of synchronous communication. The primitives are then assembled into a variant Mesh-of-trees (MoT) topology (see Section 2.1), a network that has proven to be effective in a high-performance, single-chip synchronous parallel processing architecture based on a shared-memory model [2]. This network uses deterministic wormhole routing [4, 23, 5] and extremely simple binary router nodes with low functionality. Reported results on a synchronous shared-memory processor using this topology and node structure have demonstrated its viability for a number of high-performance CMP applications [2], as opposed to the more complex 5-ported routing nodes typical in many NOCs for distributed embedded processors [11, 23, 5]. Hence, a direct comparison was targeted, from asynchronous node implementations to system-level interconnection network.

A detailed evaluation was conducted at many levels of integration. Initial simulations of the new asynchronous routing and arbitration circuits are promising, showing significant ben-

This work was supported in part by NSF awards CCF-0811504 and CCF-0325393, and by an Initiatives in Science and Engineering (ISE) grant from Columbia University. M.N. Horak is currently affiliated with Advanced Simulation Technology, Inc.; M. Carlberg contributed to this work while at Columbia University.

[1]In contrast, some recent solutions have been proposed for specialized systems with narrower bounds on operation, such as *mesochronous* (all components operate at the same clock rate, with stable but unknown phase differences) and *plesiochronous* (all components operate at nominally identical clock rates, but with slight frequency mismatch) systems [31].

efits in power and area, and roughly comparable performance, when compared to synchronous components in identical technology. Detailed simulations were also conducted on an asynchronous network and a mixed-timing version of the network, which were compared to synchronous networks from 400MHz to 1.36 GHz. Finally, the mixed-timing network was embedded and co-simulated with an XMT shared-memory processor [20] on several parallel kernels. The new GALS XMT processor provides comparable performance to the existing synchronous XMT except in the most challenging cases of high clock rate or high traffic rate. Future directions for performance optimization are also outlined.

Related Work.

GALS and Asynchronous NOCs. There has been a surge of interest in recent years in GALS design [8, 31], especially for low- and moderate-performance distributed embedded systems. More recently, several GALS NoC solutions have been proposed to enable structured system design. The CHAIN chip area interconnect [1] provides robust self-timed communication for system-on-chip (SoC) designs, including for a specialized multiprocessor for neural simulations [25]. The Nexus asynchronous crossbar [18] provides system-level communication and has been used in recent Ethernet routing chips. The NoC solution in [4] presents a low-latency service mesh network with an innovative system-level modeling framework. MANGO [6] supports quality-of-service gurantees and adaptive routing. The prototype GALS NoC of [26] supports SoC system debugging. The RasP network [15] uses pulse-based asynchronous pipelines to achieve high performance and small wire-area footprint. Earlier work provided an asynchronous node architecture and implementation for coarse-grain complex-functionality routing nodes [11].

Several of these approaches have been highly effective, especially for low- and moderate-performance distributed embedded systems [1, 4], thus targeting a different point in the design space than the proposed work. Some have lower throughput (e.g., 200 to 250 MHz) [25, 4], while those with moderate throughput (e.g. near 500 MHz) [6, 28]) often have significant overheads in router node latency. Most use high-functionality coarse-grained routing nodes (with 4 routing ports and 1 entrance/exit port, routing tables, crossbar, and extra buffering) [6, 28] based on a standard 5-ported node architecture [11]. Almost all use four-phase return-to-zero protocols, involving two entire roundtrip channel communications channel per transaction (rather than the single roundtrip communication targeted in the proposed work), as well as delay-insensitive data encoding, i.e. dual-rail, 1-of-4, m-of-n (which results in lower coding efficiency than the single-rail bundled encoding used in our work) [1, 6, 11, 25, 18, 28, 4]. Finally, several approaches use specialized circuits with dynamic logic [18] or pulse-mode [15] operation. Closer to our work is a promising recent approach targeting a two-phase protocol using a commercial CAD flow [26]. However, it has overheads due to a delay-insensitive (LEDR) data encoding and flipflop-based registers, and is not currently suitable as a general GALS NOC: it does not provide any routing nodes, only channel multiplexers to support silicon debugging. The GALS neural network system of [25] also includes two-phase channels between chips, with four-phase channels on chip; the former use m-of-n delay-insensitive codes with large encoding and decoding overheads.

Asynchronous Transition-Signaling Pipelines. The proposed NoC is based on Mousetrap pipelines [29], which use a low-overhead latch-based architecture (see Section 2.2). In this paper, these pipeline components are enhanced to support routing and arbitration. Several previous transition-signaling linear asynchronous pipelines have been proposed, but most have significant overheads. Some use complex and expensive latch structures, including specialized capture-pass latches [30] and double-edge-triggered flipflops [7]. Others uses lightweight transparent latches, but require double latch registers per stage (whereas Mousetrap only requires single registers) [27].

Closest to our work are the recent non-linear routing and arbitration nodes by Gill and Singh [13], and extended by Mannakkara and Yoneda [19]. This work makes useful advances in two-phase asynchronous pipeline design, but designs are relatively unoptimized. For the routing primitives, all prior designs stall when one output channel is congested, while the proposed design allows pass-through traffic to the uncongested output channel. In addition, the prior FF-based designs are expected to have higher energy per transaction than the proposed latch-based design. For the arbitration primitives, the prior designs use 2 FF-based data registers vs. only 1 latch-based data register in the proposed design, which should result in significantly worse area, energy, latency and throughput than the proposed design, though ours may have higher glitch power when the latches are transparent. In addition, our design supports wormhole routing, while these do not.

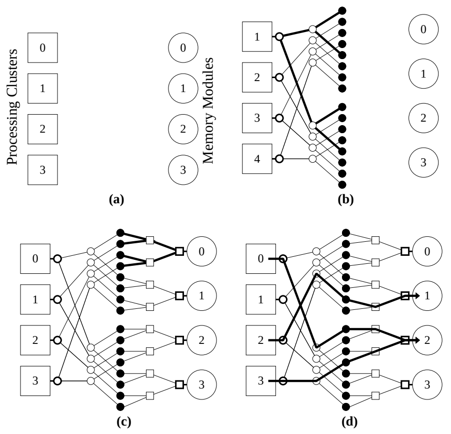

Figure 2. Mesh-of-trees network (N=4)

2 Background

2.1 Mesh-of-trees network

The Mesh-of-trees (MoT) network [2] used in this paper and in recent publications is a variant the traditional mesh-of-trees network [17], designed to provide the needed bandwidth for a high-performance, fine-grained parallel processor using global shared memory. It has been proven effective in recent detailed evaluations on a range of traffic for on-chip parallel processing. Recent extensions have been proposed to reduce area overhead through a hybrid MoT/butterfly topology, which maintains the throughput and latency benefits of MoT with the area advantages of butterfly [3].

The MoT network consists of two main structures: a set of fan-out trees and a set of fan-in trees. Figure 2(b) shows the binary fan-out trees, where each source is a root and connects to two children, and each child has two children of their own. The 16 leaf nodes also represent the leaf nodes in the binary fan-in trees that have destinations as their roots (Figure 2(c)). An MoT network that connects N sources and N destinations has $logN$ levels of fan-out and $logN$ levels of fan-in trees. There is a unique path between each source-destination pair.

A memory request packet travels from the root to one of the leaves of the corresponding fan-out tree. It passes to the leaf of a corresponding fan-in tree, and travels to the root of that fan-in tree to reach its destination (Figure 2(d)). In general, contention can occur when two packets from different sources to different destinations compete for a shared resource. In the MoT network, fan-out trees eliminate competition between packets from different sources, and fan-in trees eliminate competition between packets to different destinations. This separation guarantees that, unless the memory access traffic is extremely unbalanced, packets between different sources and destinations will not interfere. Therefore, the MoT network provides high average throughput that is very close to its peak throughput. There are three switching primitives in a MoT network: (a) routing, (b) arbitration, and (c) linear pipeline primitives (the latter is optional for performance improvement as a microarchitectural "repeater" to divide long wires into multiple short segments).

2.2 Mousetrap pipelines

The new asynchronous network primitives for fan-out and fan-in nodes, introduced in the following section, are based on

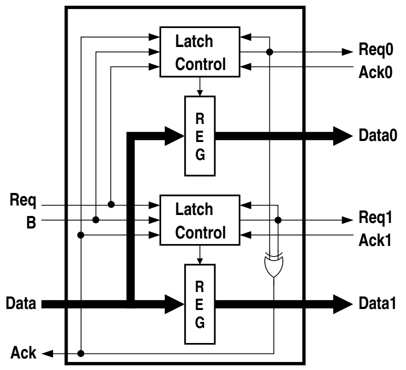

Figure 3. Structure of routing primitive

an existing linear asynchronous pipeline called Mousetrap [29]. Mousetrap is a low-overhead asynchronous pipeline that provides high-throughput operation. Each Mousetrap stage uses a single register based on level-sensitive latches (rather than edge-triggered flipflops) to store data, and simple stage control consisting of only a single combinational gate. These designs use single-rail bundled data encoding, where a synchronous-style data channel is augmented with an extra *req* wire, and a single transition on the req accompanying the data "bundle" indicates the data is valid. The req wire has a simple one-sided timing constraint that its delay is always slightly greater than the data channel. (For further details, see [29].)

3 Asynchronous Primitives

This section introduces the two new asynchronous components: the *routing* and *arbitration* network primitives. These components are the fundamental building blocks of the asynchronous Mesh-of-trees network, and can also be used to construct alternative network topologies. A basic overview is provided, further details can be found in [16, 22].

3.1 Routing primitive

The routing primitive performs a fan-out (i.e. demultiplexing) operation, with one input port and two output ports, shown in Figure 3. During the operation, packets arriving at the input port are directed to exactly one of the two output ports.

Basic operation. Figure 3 shows the structure of the routing primitive. Adjacent primitives communicate using request (*req*) and acknowledgment (*ack*) signals following a transition-signaling protocol. The basic operation, assuming an empty primitive, begins with new data arriving along with a routing signal *B*. An important feature of the routing primitive is that, unlike Mousetrap pipeline stages, the registers are *normally opaque* (i.e. disabled), preventing data from propogating to subsequent stages before the routing decision is made. After the data inputs are stable and valid, a request transition on *Req* occurs at the input. The latch controller selected by the routing signal, *B*, enables its latches (i.e. makes them transparent) and data advances to the selected output channel. The toggle element generates a request transition on *Req0/1* to the following stage. Then, in parallel, the latches are quickly closed, safely storing data, and an acknowledgment transition on *Ack* is sent to the previous stage.

Architecture of the routing primitive. Each primitive consists of two registers and latch controllers, one pair per output port. Each register is a standard level-sensitive D-type transparent latch register that is normally opaque, preventing data from passing through. Each latch controller, shown in Figure 4, is responsible for controlling three signals, which enable data storage and inter-stage communication: (i) the register enable signal (*En*); (ii) the corresponding request output (*Req0/1*) to the next stage; and (iii) the acknowledgment (*Ack*) to the previous stage. The toggle element converts an input *Req* transition to an output *Req0/1* transition on the appropriate port. The toggle output for a specific port will transition once for every data item, when both toggle input (*y0/1*) and enable (*En*) inputs are high. The acknowledgment (*Ack*) signal to the left environment is generated by the XOR gate shown in Figure 3. The XOR gate

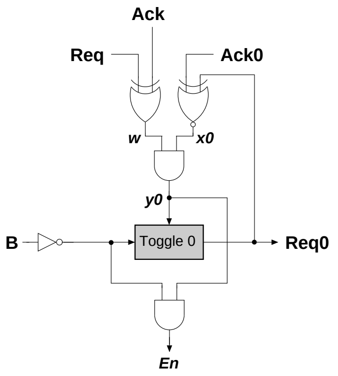

Figure 4. Latch controller of routing primitive

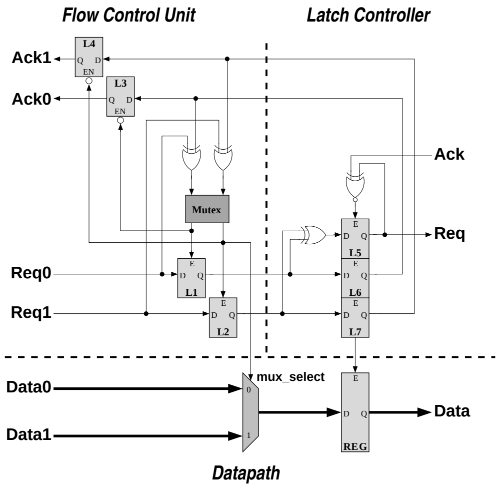

Figure 5. Structure of arbitration primitive

merges two transition signals, *Req0* and *Req1*.[3]

Enhanced concurrency feature. The routing primitive includes a powerful capability to decouple processing between the two output routing channels. In particular, if one of the output channels is stalled, awaiting acknowledgment, the other output channel can successively process *multiple* full transactions. This concurrency feature has the potential for significant system-level performance benefits, since it entirely avoids stalling input packets heading to an unblocked output channel.

3.2 Arbitration primitive

The arbitration primitive accepts data from exactly one of two input ports and forwards it to a single output port, thus providing complementary functionality to the routing primitive.

Basic operation. Figure 5 shows the design of the basic arbitration primitive. An operation begins with new data appearing at the input of an empty primitive followed by a request transition from the previous stage to the flow control unit. The flow control unit will arbitrate the request through a mutex component and perform two actions: setting the correct multiplexer select signal (*mux_select*) and forwarding the winning request to the latch controller by enabling either L1 or L2. The latch controller will then store the new data and concurrently generate a request to the next stage while acknowledging to the flow control unit that data has been safely stored. At this point, the flow control unit will reset the mutex and then acknowledge to the previous stage that it may accept new data on that channel, thereby completing the transaction.

Architecture of the basic arbitration primitive. Figure 5 shows the structure of the arbitration primitive. The arbitration functionality is performed by the mutual exclusion element (mutex), an analog arbiter circuit. The design features seven standard level-sensitive D-type transparent latches (numbered L1 through L7). Latches L3 through L7 are all normally transparent (enabled). Latches L1 and L2 are normally opaque (disabled). XOR gates are used at the inputs of the mutex as

[3]For simplicity, initialization circuitry is omitted, but it is included for all reported experiments.

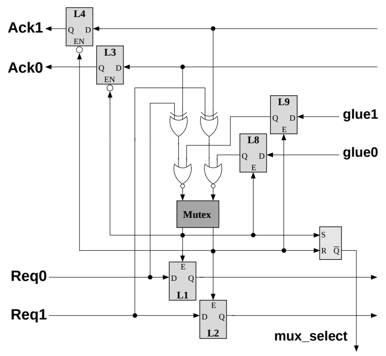

Figure 6. Enhanced flow control unit

"inequality" testers, generating a request transition to the mutex when new data has arrived and then resetting the mutex after that data has been stored in the register. Another XOR gate at the input of latch L5 functions as a "merge" element, joining two transition-signaling signals, *Req0* and *Req1*, into a single signal, *Req*. The XNOR gate is used as a Mousetrap latch enable with feedback path from the output of L5. Finally, there is one multiplexer and register (transparent latch) per data bit.[3]

Power optimization. The basic design of Figure 5 allows unnecessary glitch power consumption to occur on the datapath. The optimization in Figure 6 eliminates this glitching. Specifically, the *mux_select* signal may transition more than once per cycle for transactions on the *Req1* port. The optimization adds an SR latch to store the most recent mutex decision at the end of each transaction. The result of this optimization is that the *mux_select* is limited to at most one transition per transaction. The resulting power savings can be significant, since the majority of the power is consumed in the datapath.

Wormhole routing capability. The final enhancement in Figure 6 is support for wormhole routing of multi-flit packets [4]. A flow-control unit, or flit, is the smallest granularity of message sent through the network. Wide packets are split into multiple flits that travel contiguously through the network. In wormhole routing, once arbitration is won by a packet head flit in an arbitration node, each remaining flit in the packet must be guaranteed unarbitrated access through the node until the last flit of the packet exits. In the design, to bias the selection of the mutex so that the next flit of a multi-flit packet is guaranteed to advance without new arbitration, a "kill your rival" protocol is implemented. When the first flit of a multi-flit packet wins the mutex, the opposing request input to the mutex is forced to zero, or "killed". This operation either prevents future requests at the other mutex input from occurring, or in the case where a request was already pending, kills the opposing request until the entire multi-flit packet has advanced. The kill function is achieved using a NOR gate located at the input of the mutex.

4 Mixed-Timing Interfaces

Mixed-timing interfaces are now introduced to allow for integration into a GALS system with robust communication between synchronous terminals through the asynchronous network, as shown in Figure 1. The mixed-timing interfaces are designed using existing mixed-timing FIFOs [9] and new custom asynchronous protocol converters.

Each mixed-timing FIFO is a token ring of identical storage cells that have data enqueued from one timing domain and dequeued from another. The synchronous portions of the FIFOs have full or empty detectors. Detection circuits generate signals to stall synchronous terminals in order to to prevent overrun or underrun conditions in the FIFO. The asynchronous portions do not require explicit full or empty detection, as they will simply withhold acknowledgment until an operation can be performed. The mixed-timing interfaces provide communication between synchronous terminals and the asynchronous network. Figure 1 shows the interfaces instantiated in the mixed-timing network (marked as "S→A" and "A→S").

Details of the mixed-timing interfaces are shown in Figures 7 and 8. Each interface contains a mixed-timing FIFO and custom protocol converter. The protocol converter translates

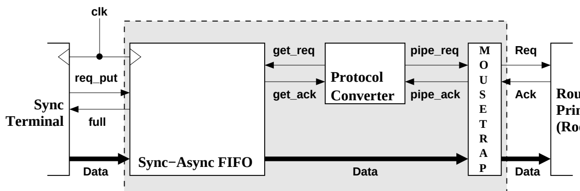

Figure 7. Sync/Async interface block diagram

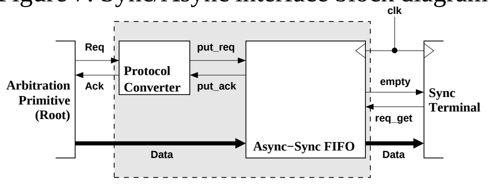

Figure 8. Async/Sync interface block diagram

handshaking signals between the two-phase transition signaling of the asynchronous network and the four-phase return-to-zero signaling of the existing mixed-timing FIFO. Each converter is designed as a low-latency burst-mode asynchronous controller using the MINIMALIST CAD tool [12, 21].

To improve throughput, a Mousetrap pipeline stage [29] is added to the synchronous-asynchronous interface (Figure 7) between the protocol converter and the routing primitive at the root of the fan-out tree. The Mousetrap stage, when empty, will store new data and acknowledge the mixed-timing FIFO faster than the routing primitive.

5 Experimental Evaluation

Detailed evaluations of the new interconnect network are now presented, as well as comparisons to an existing fabricated synchronous version [2], at several distinct levels. These range from detailed post-layout simulation of network primitives to pre-layout system-level evaluation of fully-assembled networks, both asynchronous and mixed-timing (using interconnected post-layout components interconnected with delays extrapolated from a comparable synchronous chip floorplan [2]). Finally, several parallel kernels are run on the mixed-timing network in shared-memory CMP simulation environment.

5.1 Asynchronous primitives

The asynchronous primitives are evaluated using four metrics – area, power, latency and maximum throughput – and are compared in detail with the synchronous primitives recently proposed in [2]. In particular, Balkan *et al.* [2] provide detailed physical layouts in a commercial 90 nm technology library. The asynchronous primitives were designed in the same technology for the purpose of direct comparison. Results are presented for both primitives with 8-bit wide datapath, simulated at 1.2 V and 25°C in the nominal process using Cadence NC-Sim.

These experiments do no currently assess the potential impact of clock gating, since the synchronous chip in [2] did not include this optimization. High-level clock gating (e.g. of an entire tree) is unlikely to provide significant benefits because of the path diversity and rapidly-changing memory access patterns in this topology and architectural domain. While low-level clock gating is certainly possible (e.g. per routing primitive cell), it is expected to add significant area and power overheads, due to the fine granularity of the network nodes. In addition, clock gating still requires global clock distribution, and does not fit well with the heterochronous system goals of this work.

Area and power. As indicated in Table 1, the asynchronous primitives achieve significant power and area savings compared to existing synchronous designs, using 36% and 16% of the cell area respectively and 18% and 9% of the energy per packet of the existing synchronous designs.

Several metrics are used to assess area and power consumption. Area is measured as the total cell area occupied by a single primitive. Energy per packet is reported as the average energy consumed by a primitive during simulation with a sequence of 100 packets containing random data either routed to, or arriving at, random destinations. Leakage power is reported as the subthreshold leakage power consumed by the regular-Vt cells. Idle power is measured when no requests arrive to either primitive at a clock rate of 1 GHz with no clock-gating optimizations. Even with efficient clock-gating schemes, dynamic power is consumed due to the local clock network, while the asynchronous does not consume dynamic power when idle.

Table 1. Network primitives: area and power

Component Type	**Area** (μm^2)	**Energy/ Packet** (*pJ*)	**Leakage Power** (μW)	**Idle Power** (μW)
Routing Async	358.4	0.37	0.56	0.6
Routing Sync	988.6	2.06	1.82	225.6
Arbitration Async	349.3	0.33	0.50	0.5
Arbitration Sync	2240.3	3.53	4.13	388.6

The above results indicate a substantial reduction in area and dynamic and static power for both asynchronous primitives over their synchronous counterparts. The synchronous designs use two registers at input ports of routing and arbitration primitives [2] in order to provide better throughput in congested scenarios, and to support a latency-insensitive style dynamic flow control, for a total of *six datapath registers* between the two primitives. In contrast, the asynchronous primitives have a total of only *three datapath registers* between the two nodes. Furthermore, due to the Mousetrap-based design style, each asynchronous register is a single bank of *transparent D-latches*, while each synchronous register is a single bank of more expensive *edge-triggered flipflops*. Finally, in spite of the single-latch-based asynchronous methodology, each asynchronous latch register, in congested scenarios, can still hold a *distinct data item*, and therefore the network provides 100% storage capacity. As a result, the asynchronous primitives provide significantly lower area and power overheads, as well as a flexible "data-driven" operation (i.e. processing data items only on demand).

Latency and throughput. The asynchronous primitives exhibit competitive performance in terms of latency and maximum throughput, while using less area and power than the synchronous designs. Latency is measured as the delay from a request transition on an input interface to its appearance at an output of an empty primitive. Maximum throughput, given in Giga-flits per second (GFPS), is evaluated under different steady-state traffic patterns. Throughput is measured at the root primitive of a 3-level fan-out or fan-in tree which which captures the interactions between neighboring primitives.

Table 2 shows results of latency and throughput experiments for the routing primitive. For this primitive, throughput is evaluated for three steady-state traffic patterns where consecutive packets are routed to a *single*, *random* or *alternating* destination port. The initial performance results for the routing primitive are encouraging, as the latency and highest throughput approach the results measured from the synchronous design. The latency of the asynchronous primitive is only 6% higher than the synchronous, indicating that in lightly-loaded scenarios, the asynchronous fan-out tree is expected to have similar total latency as a synchronous fan-out tree operating near 2 GHz. With sufficiently distributed routing destinations, the expected performance is the *random* case, which can take advantage of the concurrent operation between ports to achieve 69% of the synchronous maximum throughput. Future work on system-level optimization is expected to increase the maximum throughput in the *random* case. In particular, initial analysis indicates that the addition of linear pipeline primitives (i.e. Mousetrap stages) near the root of the fan-out tree will further improve performance.

Table 2. Routing primitive: performance

Component Type	**Latency** (*ps*)	**Max. Throughput** (GFPS) Single	Random	Alternating
Async	546	1.07	1.34	1.70
Sync	516	1.93	1.93	1.93

Table 3 shows results of latency and throughput experiments for the arbitration primitive. For this primitive, maximum throughput is measured for two steady-state traffic scenarios where successive packets arrive at a *single* port or simultaneously at *all* ports. The latency and highest throughput of the two arbitration primitives are nearly identical. This result indicates the high performance of the arbitration primitive. In a lightly-loaded network, latency is the dominant factor for performance, and the asynchronous primitives should operate comparable to synchronous primitives with clock frequency near 2 GHz. In a heavily-loaded network, the throughput of the root arbitration primitives determine the total output traffic. With well-balanced traffic, the throughput of the asynchronous fan-in tree should approach the *all* scenario.

Table 3. Arbitration primitive: performance

Component Type	**Latency** (*ps*)	**Max. Throughput** (GFPS) Single	All
Async	489	1.08	2.04
Sync	474	2.09	2.09

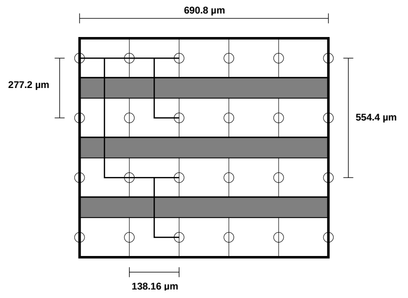

Figure 9. Projected 8-terminal asynchronous network floorplan

Mutual exclusion element. A final component of this evaluation is on the performance of the the mutual exclusion ("mutex") element in the arbitration primitive of Figure 5. Unlike the fixed-latency synchronous arbitration of the previous MoT network [2] and other synchronous NOCs. asynchronous arbitration is implemented by an analog component with variable-time resolution. Theoretically, this element may exhibit arbitrary finite resolution time depending on input scenarios. However, detailed simulations show that this component exhibits nearly fixed delay except for extreme and rare cases: a baseline latency of 150ps when receiving single requests (i.e. no contention), and only noticeable degradation beginning when two competing inputs arrive within the same 2 picosecond interval.

5.2 Mixed-timing interfaces

The performance of the mixed-timing interfaces is now evaluated. Two metrics have been simulated: *latency* and *maximum throughput*. Latency is the delay from a request at the input channel to its appearance at the output channel, beginning with an empty FIFO. Maximum throughput is defined as the highest operating rate where the FIFO does not report full or empty with a highly-active input environment. Table 4 shows simulation results for mixed-timing interfaces with 3-place FIFOs. The latency of the asynchronous-synchronous FIFO varies depending on the exact moment when data items are enqueued during a clock cycle, hence the *Min* and *Max* columns.

Table 4. Mixed-timing interfaces: performance

Interface Type	**Latency** (*ns*) Min	Max	**Max. Throughput** (*MHz*)
Sync-Async	0.97		932.8
Async-Sync	2.95	3.56	843.2

5.3 Asynchronous network

This section presents the preliminary system-level performance evaluation of an 8-terminal pre-layout asynchronous network.

Projected network floorplan. The projected asynchronous floorplan is shown in Figure 9. It is based on the floorplan for the comparable fabricated synchronous MoT network [2].

As in [2], the MoT network is partitioned into four physical slices, each interfacing to two source terminals and two destination terminals, indicated by the white horizontal stripes. The gray horizontal stripes contain inter-partition routing channels. In the synchronous network, the primitives are placed, routed

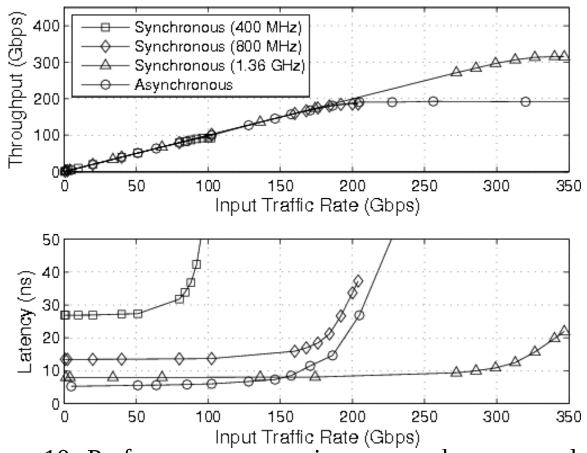

Figure 10. Performance comparison: asynchronous and synchronous networks

and optimized by CAD tools. For the asynchronous network, the tools could not be directly applied while preserving asynchronous timing constraints, hence the routing and arbitration primitives are treated as hard macros, and assigned placement at regular intervals within the partitions (indicated by white circles). Then, derived wire delays based on this placement are assigned to inter-primitive connections for simulation. The path shown in the figure illustrates one fan-out tree of the network.

Experimental setup. Performance is evaluated based on two metrics: *throughput* and *latency*. The evaluation for these metrics is performed following the approach proposed by Dally and Towles [10], adapted to accommodate the analysis of an asynchronous network. In particular, throughput, given in Gigabits per second (Gbps), is the output data rate of the network during the "measurement phase" of the simulation. Latency, given in nanoseconds (ns), is measured as the time from creation of a packet until it reaches its destination. Specifically, packets are first placed into source queues at input ports of the network before they can be inserted. This method ensures that stalling effects at the inputs are captured in latency results.

Three clock rates (400 MHz, 800 MHz and 1.36 GHz) are chosen for a detailed performance comparison with the new asynchronous network. The synchronous network at 800 MHz was selected since it provides fairly high performance, but not peak rate operation, and 400 MHz was selected to show moderate performance. The 1.36 GHz clock rate was selected to show an extreme comparison, where the synchronous design is operating at its maximum possible rate, as determined by static timing analysis (see [2]).

Experiments are conducted under uniformly random traffic with packet source queues installed at network input ports for injecting traffic to accurately measure network performance [10]. In the synchronous case, packets are generated at each port at random clock edges and inserted into queues (implemented in hardware), depending on the desired average rate [2]. For the asynchronous, packets are generated at random intervals, following an exponential distribution with the mean corresponding to the desired input traffic rate, and inserted into queues (implemented in behavioral Verilog).

Note that there is a subtle difference in evaluating the synchronous network versus the asynchronous. The performance of the asynchronous network only depends on input traffic rate; since it operates in a data-driven style, the input traffic drives the simulated behavior. In contrast, the performance of the synchronous interconnection network depends on two distinct parameters: input traffic rate and clock frequency. Here, clock frequency determines the range of valid input traffic rates, and the input traffic rate (relative to that frequency) determines the performance. In addition, the asynchronous network in this simulation framework can accomodate a wide range of input traffic rates (even though saturation will eventually occur), while the maximum input rate of the synchronous network is inherently limited by the clock frequency.

Simulation results. System-level performance evaluation for the asynchronous network is shown in Figure 10, where it is also compared with synchronous results for the three different clock frequencies (400 MHz, 800 MHz and 1.36 GHz). Experiments are conducted for a wide range of input traffic rates up to the maximum input traffic rate of the 1.36 GHz synchronous network. The 1.36 GHz, 800 MHz and 400 MHz networks have maximum input traffic rates of 348.2 Gbps, 204.8 Gbps and 102.4 Gbps, respectively (for 8 terminals and 32-bit datapath), so no further data exists beyond this point on the respective graphs in Figure 10.

For throughput, of the asynchronous network tracks the 1.36 GHz synchronous network up to an input traffic rate near 200 Gbps, where the asynchronous network reaches saturation. The throughputs of the 400 MHz and 800 MHz synchronous networks likewise track up to their individual maximum input traffic rates (102.4 and 204.8 Gbps, respectively), at which point they are cut off. The asynchrononous network, in contrast, is capable of accepting higher traffic rates, since it not inherently limited by a clock rate, hence its graph continues but with no further change in throughput.

Note that one of the driving goals in using the MoT topology is that it sustain high throughput and low latency at the highest traffic rates, to satisfy the high traffic demands of the XMT parallel processor. The figure validates the use of this topology, since the synchronous saturation throughput is 91% of maximum rate [2].

For latency, the asynchronous network always has significantly lower latency than both the 400 MHz and 800 MHz synchronous networks, over their entire operating ranges. It also has lower latency than the 1.36 GHz synchronous network at all input traffic rates up to 150 Gbps, which is at 43.1% of the maximum input rate for this synchronous network; beyond this point, these latency diverge rapidly as the asynchronous network reaches saturation.

Bottleneck Analysis and Directions for Further Improvement. These results on throughput and latency for the asynchronous network are promising, and competitive over a range of scenarios, but also indicate some bottlenecks. In particular, while the asynchronous network dominates both the 400 and 800 MHz synchronous networks, its saturation point and achievable throughput still fall short compared to the 1.36 GHz synchronous network for higher input traffic rates. To identify the bottleneck sources, it is useful to analyze performance results in Tables 2 and 3.

In that section, the synchronous primitive components had throughputs of 1.93 and 2.09 GHz, respectively, without wiring delays [2]. After layout, the synchronous primitives could operate only up to 1.36 GHz. Hence, a synchronous derating factor of approximately 70% occurs on the achievable throughput between pre- and post-layout.

By extrapolation, a similar derating would be expected for the asychronous primitives. Interestingly, in an asynchronous MoT network — unlike synchronous — the performance demand is greatest at two nodes, which are critical bottlenecks: the *root of the routing (i.e. fan-out) network* and the *root of the arbitration (i.e. fan-in) network*. This node streams packets from the source, and receive converging streams at the sink, hence must be activated with highest frequency. Given that in Figure 10, random inputs are simulated, then the relevant entry in Table 2 for the routing primitive root is the "random" throughput result: 1.34 GFPS. The traffic pattern on the arbitration primitive root is unknown, but lies between random ("all", 2.04 GFPS) and single-channel ("single", 1.08 GFPS) in Table 3. Therefore, it is likely that the single bottleneck in the entire asynchronous MoT network is the root of the routing network. The derating factor that appears between Table 2 (1.34 GFPS) and the network simulations of Figure 10 (800 GFPS) is 59.7%. Given the approximations involved, this derating can be regarded as roughly tracking the synchronous factor of 70%.

In summary, an important future direction for performance improvement in the asynchronous network is to significantly optimize the root routing nodes. Three optimizations are

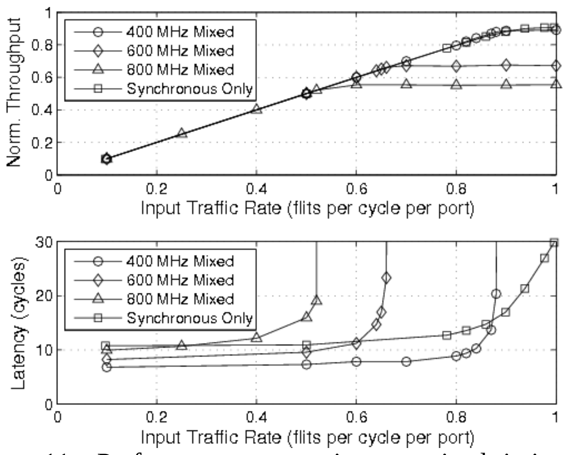

Figure 11. Performance comparison: mixed-timing and synchronous-only networks (normalized to clock cycles)

promising: (i) at the *circuit level*, to apply gate resizing and better repeater insertion at this node, to improve latency and throughput; (ii) at the *micro-architecture level*, the asynchronous design was not optimized for congestion; initial results suggest that insertion of linear Mousetrap FIFO stages at its ports can improve overall system throughput; and (iii) at the *node design level*, to explore higher-throughput variants.

5.4 Mixed-timing network

After evaluating the performance of the asynchronous network, mixed-timing FIFOs are added to form the final mixed-timing network. The mixed-timing network, unlike the synchronous, provides the capability of easily integrating into heterogeneous timing systems with multiple synchronous domains operating at arbitrary or dynamically-variable clock frequencies. Latency and throughput are again evaluated following the methodology from Dally and Towles [10].

In this comparison of networks having synchronous terminals, including the mixed-timing network, the input traffic rate and throughput are given in *flits per cycle per port*, the average rate at which flits enter and exit network ports. Latency is normalized to the number of cycles. Using the normalized values, the synchronous will have *the same performance at all valid clock rates*, relative to clock cycles. The performance of the mixed-timing network, however, depends on the self-timed asynchronous network, and will change with the clock frequency.

Figure 11 shows performance results for the mixed-timing network, a flexible asynchronous network capable of interfacing with multiple synchronous domains operating at arbitrary clock frequencies. Since Table 4 of Section 5.2 indicates that the mixed-timing interfaces have somewhat lower performance than the asynchronous primitives, the mixed-timing network is evaluated at three clock rates that operate below their maximum operating rates: 400, 600 and 800MHz.

The normalized results of Figure 11 indicate a wide range of behavior, from comparable throughput and significantly lower latency for the 400 MHz mixed-timing network, to a mix of advantages and performance degradation in the 600 and 800 MHz mixed-timing networks. At higher clock frequencies, the performance of the mixed-timing network is affected by bottlenecks introduced by mixed-timing interfaces. The 600 MHz and 800 MHz mixed-timing networks achieve maximum throughputs that are 67% and 55% of a synchronous network operating at those frequencies, respectively. Performance bottlenecks occur due to synchronization overheads in the mixed-timing interfaces. In particular, the mixed-timing interfaces must stall when FIFOs approach near-full or near-empty to prevent FIFO errors. As the clock rate increases and approaches the maximum throughput of the interfaces, these penalties occur more often, resulting in lower throughput of the mixed-timing network.

On the positive side, however, it is noteworthy that the 600 MHz and 800 MHz mixed-timing networks provide identical throughput to synchronous at input traffic rates up to 65% and 52%, respectively, and lower latency than synchronous at input traffic rates up to 60% and 29%, respectively.

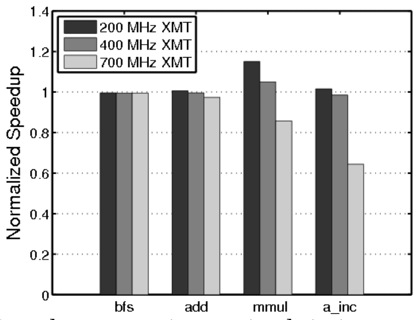

Figure 12. Speedup comparison: mixed-timing vs. synchronous network on 4 parallel XMT kernels (normalized to synchronous XMT performance)

Further design improvements to the mixed-timing interfaces of [9], especially for the async-sync interface (when the sync interface is stalled and empty, waiting for a new item, resulting the large latency overhead listed in Table 4) are expected to have a large impact on further improving system performance.

5.5 Parallel kernels in a GALS CMP architecture

This section presents simulation results on several parallel kernels on an existing shared-memory parallel processor, XMT [20, 14], incorporating the proposed mixed-timing network. These simulations provide realistic traffic loads for the network, as opposed to uniformly random traffic used in previous experiments.

XMT overview. The XMT shared-memory architecture targets fine-grain thread parallelism, and has shown significant benefits in handling complex and dynamically-varying computation (e.g. ray tracing) as well as regular computations (e.g. indexed-tree database searches). Centered around a simple abstraction, the XMT processor also provides a useful framework to support the productivity of parallel programmer [32]. An existing synchronous CMP implementation incorporates a clocked MoT network between multiple cores and multiple memories. For our experiments, the synchronous MoT network is replaced by the new mixed-timing MoT network. The resulting GALS CMP architecture is capable of supporting multiple synchronous cores operating at unrelated clock rates.

Details of XMT parallel interconnect network. The XMT simulation assumes an 8-terminal mixed-timing MoT network, containing 8 source terminals and 8 destination terminals. In particular, for this CMP platform, the mesh serves as a high-speed parallel interconnect between cores and partitioned shared L1 data cache [20, 14]. Each of the 8 cores, or *processing clusters*, itself contains 16 separate processing units, or *thread control units (TCUs)*, for a total of 128 TCUs. In XMT, no data is cached locally in a TCU. The shared L1 cache is partitioned into 8 smaller cache modules, and each TCU may access any cache module. Since the interconnect is designed to support loads and stores, two distinct new mixed-timing networks are modelled: one from processing clusters to caches and one from caches to processing clusters. Each load operation uses a 1-flit packet, and each store operation uses a 2-flit packet [20, 14].

Experimental setup. Three different clock frequencies were selected: 200, 400 and 700MHz. Speedup results for the mixed-timing network are normalized relative to performance of the synchronous network. The synchronous network has speedup of 1.0 in all cases. In addition, four different XMT parallel kernels were simulated: *array summation (add):* each parallel thread computes the sum of a sub array serially, and the resulting sums are added to compute the total for the entire array (size 3M); *matrix multiplication (mmul):* the product of two 64×64 matrices is computed, where each parallel thread computes one row of the result matrix; *breadth-first search (bfs):* a parallel BFS algorithm is executed on a graph with 100K vertices and 1M edges; and *array increment (a_inc):* each parallel thread increments 8 elements of the array (size 32K). The average network ultilization exhibited by these XMT kernels are: *add:* 0.492; *mmul:* 0.395; *bfs:* 0.091; *a_inc:* 0.927.

Overview of results. Simulation results are shown in Figure 12. The *add* kernel provides steady traffic throughout the execution of the program that remains below the saturation throughput of the network with mixed-timing, hence GALS XMT performance remains comparable to an XMT with the synchronous network. The *mmul* kernel has a lower average network utilization than *add*; however, this does not reflect the true pattern of memory access. The traffic appears in bursts, causing very high traffic, followed by periods of low traffic as the processing clusters compute the results. In the 700 MHz case, the burst traffic exceeds the saturation throughput, thereby degrading the performance of the application. The *bfs* kernel has the lowest traffic rate of all benchmarks; since it operates below the saturation throughput of the network, performance is comparable to synchronous for all simulations. The *a_inc* kernel has extremely high traffic. The average input traffic at 700 MHz will be 164.9 Gb/s, which exceeds the saturation throughput of 112 Gb/s. The network latency increases exponentially under that level of traffic, and the performance of the GALS XMT processor therefore decreases.

5.6 Evaluation summary

Overall, the above results are promising, while still indicating areas for further improvement. For the XMT experiments, on both *add* and *bfs* kernels (where the former had an average network utilization of nearly 50%), the GALS XMT had comparable performance to the synchronous XMT at 700MHz. Performance of kernel *mmul* degraded only modestly (by about 14%) at 700MHz. Only kernel *a_inc*, which had high average network utilization (0.927), showed significant degradation (by about 37%). In each case, there was almost no performance degradation, and sometimes an improvement, at the lower clock rates (200MHz, 400MHz). In addition, extrapolating from Figure 11, the mixed-timing network provides lower latency over much of this range.

These results confirm two observations from Section 5.4: the mixed-timing network performs well at (i) relatively high clock rates with low to moderate input traffic, and (ii) low to moderate clock rates with high input traffic. However, further design improvements are still needed, to be explored along the guidelines outlined in Sections 5.3 and 5.4, to make the mixed-timing network more competitive at high clock rates.

6 Conclusions and Future Work

A new low-overhead asynchronous network was introduced for chip multiprocessors, that provides scalable,high-bandwidth on-chip communication, beginning with the detailed design of network primitives and extending to initial system-level performance evaluation. Unlike a synchronous network, it provides support for heterochronous systems consisting of synchronous nodes with unrelated arbitrary clock rates. The synchronous implementations of comparable network primitives use 5.6-10.7x the energy per packet and 2.8-6.4x the area when compared with the new asynchronous designs. Mixed-timing interfaces with new custom protocol converters were then proposed to provide robust communication between synchronous and asynchronous timing domains. Then, network primitives were assembled into a Mesh-of-trees [2] topology for preliminary system-level performance evaluation against a synchronous MoT network, first in isolation and then with accompanying mixed-timing interfaces. Finally, the mixed-timing network is embedded and co-simulated with the XMT processor and the performance is evaluated by running several parallel kernels. The new GALS XMT processor provides comparable performance to the existing synchronous XMT except in the most challenging case of high clock rate and high traffic rate.

As future work, further architectural and circuit-level optimizations discussed in previous sections are expected to improve overall system-level performance, as well as to develop a CAD flow for automated synthesis, possibly building on [26]. In addition, we aim to redesign the asynchronous MoT network as an MoT/butterfly hybrid topology, as recently proposed for a synchronous network [3], to further reduce area overheads.

References

[1] John Bainbridge and Steve Furber. CHAIN: A delay-insensitive chip area interconnect. *IEEE Micro*, 22(5):16–23, 2002.

[2] A.O. Balkan, M.N. Horak, G. Qu, and U. Vishkin. Layout-accurate design and implementation of a high-throughput interconnection network for single-chip parallel processing. In *Hot Interconnects*, August 2007.

[3] A.O. Balkan, G. Qu, and U. Vishkin. An area-efficient high-throughput hybrid interconnection network for single-chip parallel processing. In *Proc. of ACM/IEEE DAC Conf.*, pages 435–440, 2008.

[4] E. Beigné, F. Clermidy, P. Vivet, A. Clouard, and M. Renaudin. An asynchronous NOC architecture providing low latency service and its multi-level design framework. In *IEEE Async Symp.*, pages 54–63, 2005.

[5] L. Benini and G. De Micheli. Networks on chips: A new SoC paradigm. *Computer*, 35(1):70–78, 2002.

[6] T. Bjerregaard and J. Sparsoe. A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip. In *Proc. Design, Automation and Test in Europe (DATE)*, March 2005.

[7] E. Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits*. PhD thesis, Carnegie Mellon University, 1991.

[8] D. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, 1984.

[9] T. Chelcea and S. M. Nowick. Robust interfaces for mixed-timing systems. *IEEE Trans. on VLSI Systems*, 12(8):857–873, August 2004.

[10] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., 2003.

[11] W.J. Dally and C.L. Seitz. The torus routing chip. *Distributed Computing*, 1(3), 1986.

[12] R. M. Fuhrer and S. M. Nowick. *Sequential Optimization of Asynchronous and Synchronous Finite-State Machines: Algorithms and Tools*. Kluwer Academic Publishers, 2001.

[13] Gennette D. Gill. *Analysis and Optimization for Pipelined Asynchronous Systems*. PhD thesis, U. of N. Carolina (Chapel Hill, CS Dept.), 2010.

[14] P. Gu and U. Vishkin. Case study of gate-level logic simulation on an extremely fine-grained chip multiprocessor. *Journal of Embedded Computing*, 2(2):181–190, 2006.

[15] Simon Hollis and S. W. Moore. Rasp: an area-efficient, on-chip network. In *IEEE Intl. Conf. on Comp. Design*, 2006.

[16] M.N. Horak. A high-throughput, low-power asynchronous mesh-of-trees interconnection network for the explicit multi-threading (XMT) parallelarchitecture. Master's thesis, Univ. of Maryland, August 2008. http://hdl.handle.net/1903/8361.

[17] F.T. Leighton. *Introduction to parallel algorithms and architectures: array, trees, hypercubes*. Morgan Kaufmann, 1992.

[18] A. Lines. Asynchronous interconnect for synchronous SoC design. *IEEE Micro Magazine*, 24(1):32–41, Jan.-Feb. 2004.

[19] C. Mannakkara and T. Yoneda. Comparison of standard cell based non-linear asynchronous pipelines. *IEICE Tech. Report. Dependable computing*, 107(337):49–54, 2007.

[20] D. Naishlos, J. Nuzman, C.-W. Tseng, and U. Vishkin. Towards a first vertical prototyping of an extremely fine-grained parallel programming approach. *IEEE TOCS*, pages 521–552, 2003. Special Issue of SPAA2001.

[21] S.M. Nowick and P.A. Beerel. CaSCADE package. http://www.cs.columbia.edu/˜nowick/asynctools/, 2007.

[22] S.M. Nowick, M.N. Horak, and M. Carlberg. Asynchronous digital circuits including arbitration and routing primitives for asynchronous and mixed-timing networks. US Patent App. PCT/US09/50561, 7/14/2009.

[23] U.Y. Ogras, J. Hu, and R. Marculescu. Key research problems in NoC design: a holistic perspective. In *Proc. of CODES*, pages 69–74, 2005.

[24] J.D. Owens, W.J. Dally, R. Ho, D.N. Jayasimha, S.W. Keckler, and L.-S. Peh. Research challenges for on-chip interconnection networks. *IEEE Micro*, 27(5):96–108, 2007.

[25] L.A. Plana, S.B. Furber, S. Temple, M. Khan, Y. Shi, J. Wu, and S. Yang. A GALS infrastructure for a massively parallel multiprocessor. *IEEE Des. and Test of Computers*, 24(5):454–463, 2007.

[26] B. Quinton, M. Greenstreet, and S. Wilton. Practical asynchronous interconnect network design. *IEEE Trans. VLSI*, 16(5):579–588, May 2008.

[27] C.L. Seitz and W.-K. Su. A family of routing and communication chips based on the Mosaic. In *Proceedings of the Symp. on Research on Integrated Systems*, pages 320–337, 1993.

[28] A. Sheibanyrad, A. Greiner, and I. Miro-Panades. Multisynchronous and fully asynchronous NoCs for GALS. *IEEE Design & Test of Computers*, 25(6):572–580, Nov 2008.

[29] M. Singh and S. M. Nowick. MOUSETRAP: high-speed transition-signaling asynchronous pipelines. *IEEE Trans. VLSI*, 15(6):684–697, June 2007.

[30] I. E. Sutherland. Micropipelines. *Comm. ACM*, 32(6), 1989.

[31] P. Teehan, M. Greenstreet, and G. Lemieux. A survey and taxonomy of GALS design styles. *IEEE Design & Test*, 2007.

[32] U. Vishkin. Using simple abstraction to guide the reinvention of computing for parallelism. *Communications of the ACM*, To appear, 2010.