Available online at www.sciencedirect.com
SCIENCE DIRECT®
J. Parallel Distrib. Comput. 65 (2005) 1237–1252
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

A framework for the design, synthesis and cycle-accurate simulation of multiprocessor networks☆

Raymond R. Hoare[a,*], Zhu Ding[a], Shenchih Tung[a], Rami Melhem[b], Alex K. Jones[a]

[a]*Department of Electrical and Computer Engineering, University of Pittsburgh, 3700 O'Hara Street, 348 Benedum Hall, Pittsburgh, PA 15261, USA*
[b]*Department of Computer Science, University of Pittsburgh, 6137 Sennott Square, Pittsburgh, PA 15261, USA*

Received 1 April 2005; accepted 1 April 2005
Available online 8 August 2005

Abstract

This paper introduces a framework for the design, synthesis and cycle-accurate simulation for parallel computing networks of 128+ processors. In order to accurately characterize the network, we present a bottom-up design methodology in which each of the components are designed using a hardware description language and synthesized to an FPGA for performance estimation of the final ASIC implementation. The components are then integrated to form a parallel computing network and simulated using a cycle-accurate simulator with network traffic described by command files. This enabled us to simulate various switching techniques, three of which are presented in this paper: wormhole switching, circuit switching and a newly introduced technique called predictive circuit switching. In our experiments, four different representational traffics are generated for our simulation and, to show the flexibility of this model, we vary the cable lengths and thus their latency for all four test cases. Our results show that this hardware design, synthesis and cycle-accurate simulation methodology provides a useful method for evaluating design tradeoffs in parallel networks. A non-blocking queue, with up to 128 internal queues, and a real-time bandwidth scheduler, for up to 128 ports, were designed in hardware with hardware synthesis results presented. From our network simulation results, we conclude that predictive circuit switching exceeds the performance of packet switching for highly predictable traffic, like collective communications, and for heavily loaded unpredictable traffic with small packet sizes. As expected, predictive circuit switching significantly underperforms both packet and circuit switching for unpredictable traffic.

Keywords: Network; Simulation; Hardware; Predictive circuit switching; Circuit switching; Wormhole routing

1. Introduction

By definition, parallel processing solves a single problem by tightly coordinating the efforts of multiple processors in order to perform a particular computation faster than is possible on a single processor. It is the performance of the network that determines the achievable speedup. In fact, there is a point when adding an additional processor to a computational problem actually increases the total execution time; a result of the additional communication and coordination overhead. This overhead is a function of the network's performance.

A network's design is not only dependent on the topology and routing algorithms but is also dependent on the design of each of the different components. For clusters, a network includes the network interface cards, the cabling, the switches and the overall topology. The performance of the entire system is also dependent on the interactions of these components when they are being used by different traffic patterns. In order to gain insight into both of these

☆ This work has been supported in part by the DARPA High Productivity Computing Systems program and by International Business Machines.

* Corresponding author. This paper is based upon work done in the context of the PERCS project at IBM, which is supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCH3039004. Los Alamos National Laboratory is operated by the University of California for the U.S. Department of Energy under contract W-7405-ENG-36. We are grateful to Mootaz Elnozahy for his leadership, support and encouragement as well as his many good suggestions throughout this work and for the PERCS project in general.

E-mail addresses: hoare@pitt.edu (R.R. Hoare), zhd2@pitt.edu (Z. Ding), shtst8@pitt.edu (S. Tung), melhem@cs.pitt.edu (R. Melhem), akjones@ece.pitt.edu (A.K. Jones).

0743-7315/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2005.04.022

areas we introduce a modular approach that decomposes the network into its individual hardware components for accurate characterization and we introduce two methods of interconnecting these components to simulate the dynamic behavior of large parallel systems.

In this paper, we present a unified framework and bottom-up methodology for the hardware design, synthesis, and cycle-accurate simulation of parallel computing networks. A major objective of this effort is to build a modular design and simulation framework in which components can easily be assembled and modified to build different systems.

For evaluation, we utilized a cycle-accurate hardware simulator, available for ASIC and FPGA hardware design, providing the ability to inspect different signals down to the nanosecond level of detail. The modular simulator simple scalar [4], which has been built for computer architecture research, and network [5], simulator for network simulation, are similar examples. Using this methodology in our simulation, each of the components is designed using a hardware description language and synthesized to an FPGA for performance estimation of the final ASIC implementation. These components are then integrated to form an entire network of N processors capable of sending and receiving data as specified in command files. This uniform design and simulation framework enables direct comparison of various switching techniques for parallel computing networks.

The rest of this paper is organized as follows. In Section 2, we provide background information about the different types of networks and introduce a new network type called *predictive circuit switching* network. A description of our design, synthesis and simulation methodology is described in Section 3, along with a description of each of the components in our design and simulation framework. In Section 4, we show how these different components can be assembled to form four different types of networks, and how they can be simulated with cycle-level accuracy. The results of our simulations are described in Section 5, and conclusions are offered in Section 6.

2. Background

In switching networks all processors are connected to each other through one or more switches. *Circuit switching*, *packet switching*, *wormhole switching* and *time-division multiplexed switching* are among the dominant switching methods that have been used in parallel computing networks. These networks are reviewed here to highlight the different types of designs and thus, the different components that must be designed, modeled and simulated to analyze their performance under a given traffic pattern. This is followed by a review of network simulator research.

Packet switching networks send limited sized data payloads through the network by adding routing information to the front of the payload, thereby creating a data packet. Each data packet is independent of others [8,20,21]. When a large amount of data needs to be communicated, multiple packets are created and sent through the network. The Intel iPSC1 [9] was packet switched, with message packets stored in their entirety and retransmitted at each intermediate node in a hypercube network. For packet switched networks, the size and quantity of the buffers, the arbiter and the application's communication contention impact performance.

Wormhole switching improves on packet switching by establishing a path through the network as it is routed. The packet is broken up into flits and, at any given time, only one flit needs to be buffered within a single switch. Therefore, the problem of large buffer sizes in packet switching is removed. The head of the worm establishes the route though the network and all subsequent flits take the same path. In cut-through routing, worms can cross paths without blocking each other, as the switch that contains the contention simply time multiplexes from one worm to the other. Wormhole switching has a lower connection setup latency than circuit switching and does not block other traffic after a connection is established (if cut-though is used.) Wormhole routing was introduced in the Torus Routing Chip [26], and has been used in a variety of parallel systems including the Intel Paragon [11], Cray T3D [28], IBM Power Parallel SP series [1], and Quadrics [30]. The size of the flit, the speed of the switch and the types of buffers impact performance. For all systems, the topology and wire length also impact performance.

Circuit switching, unlike packet and worm hole switching, establishes an entire source-to-destination route before any data are sent [35]. Establishing this route incurs a high latency cost and once the circuit is established, it may block other circuits from forming. The benefit is in the very simple switching elements, as they do not need to contain any data buffering and only need enough logic to determine their current configuration. In fact, the switch fabric does not need to be digital as the data being sent does not need to be inspected after the circuit is established. Optical, low-voltage differential signal (LVDS), and similar switches have small propagation delays that are equivalent to a few feet of wire with very high throughputs. An external switch controller is needed because data cannot be examined within the switch. Optical switches currently suffer from micro second switching times, whereas LVDS switching elements are in the ten microsecond range [24,25]. The Intel iPSC/2 [27] and iPSC/860 [36] use circuit-switched communicate; when a source node and a destination node need to communicate, a dedicated path is established for a message. The performance of this method is impacted by the latency to establish a circuit and by contention within the communication pattern.

Time division multiplexing (*TDM*) *switching* [12,31,39] is an extension of circuit switching in which the switch alternates between k configurations, where each configuration establishes circuits between the inputs and the outputs of the switch. Hence, a particular connection, circuit, is established every k time slots and thus receives $1/k$ of the band-

width, where k is the multiplexing degree. In other words, scheduling a connection on a TDM switch means scheduling it repeatedly, on any one of the k multiplexed slots.

Predictive circuit switching, also referred to as predictive switching, is a TDM switching variation in which the k settings of the switch are predetermined by predicting the communication requirements of the application. Predictive circuit switching is motivated by the observation that a portion of communications within parallel computing has regularity and is highly predictable and thus, can be deterministically scheduled. There are communications whose destinations are dependent on the data being calculated and appear random. However, many applications use a virtual topology, like a two or three-dimensional mesh to perform their calculation because their applications map to the physical world. In such applications, we observe that a particular processor only needs to communicate with a small subset of processors. This improves the probability of predicting the destination of any communication. Also, by examining the source code, many communications are dependent on the loop variable and are deterministic. There are also collective communications in which the entire communication pattern can be determined for a sequence of messages across all processors. There are many methods of predicting data traffic patterns that range from run-time profiling to explicit definition within the parallel programming language [2,16,10,34,40]. However, given the scope of this paper, we do not discuss prediction methods further. Performance issues with predictive circuit switching, aside from the prediction accuracy, involve the design and interaction of the network interfaces controllers and the switch elements. The topology and wire length can also impact performance.

In order to facilitate new designs, a variety of simulators have been created. As early as 1976, CEGRELL built a simulation model to study a full-duplex message switched computer network [7]. A lot of research has been performed on building specific simulation models to evaluate network performance. As indicated by Mars [19], four general approaches are normally used to simulate a communication network: using a general purpose simulation language (e.g. SIMSCRIPT [33]), using a communication oriented simulation language (e.g. OPNET [29]), using a communication oriented simulator (e.g. BONES [13]), and using a general purpose language (e.g. C/C++). Rexford and his colleagues [32] presented an object-oriented discrete-event simulation for evaluating network designs. Liu and Dickey [18] studied buffered and un-buffered switch networks by changing the configuration of the buffers in their simulation. Gorton, Kerirdge and Jervis built a simulator, called Occam, to simulate microprocessor system at component level [14].

3. Design and simulation methodology

The objective of our methodology is to provide a rigorous design flow for high-performance parallel processing networks that scale to hundreds or even thousands of nodes.

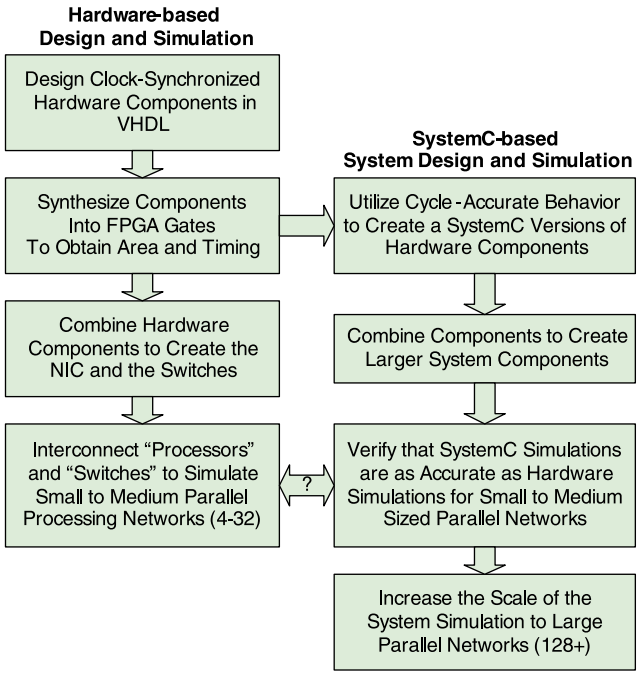

Fig. 1. Design flow methodology to create cycle-accurate simulations for large system sizes using VHDL and SystemC.

This presents design and simulation problems as simulators are typically software-based while ASICs are utilize hardware description languages (e.g. VHDL, Verilog) that requires a complex set of design and simulation tools. By combining these design and simulation methodologies, we ensure that the simulation components have the exact behavior as their ASICs counterpart. Fig. 1 shows the combined hardware and software design flows. The left column shows a traditional hardware design and simulation flow for ASICs and FPGAs while the right column shows the transformation of the results from the hardware design flow into software components. Integration and interconnection of these software components can form larger components and/or systems. Verification between the hardware and software systems is possible for small to medium-scaled networks.

For accuracy, we have designed and implemented our components using the VHDL hardware description language to prove that our components represent real hardware. To gain nanosecond level performance data, we have synthesized our VHDL into FPGA gates using Mentor Graphics Precision Synthesis software [37]. This enables the extraction of both latency (i.e. cycles of delay for the first result) and bandwidth (i.e. number of results per cycle under steady state conditions). The synthesis tool performs a detailed timing analysis and reports a maximum clock frequency. From our functional simulations, we can determine the number of cycles required for any given operation. By multiplying this cycle count by the nanosecond duration of the cycle, which is one over the maximum frequency, the component latency can be determined. The end design is expected to

target ASIC technology, but FPGA timing results can be more easily obtained and compared with published results. We conservatively estimate that the ASIC performance will be five times faster than the FPGA results. We could incorporate ASIC synthesis tools into this flow to improve accuracy to the sub-nanosecond level but this fine-tuning is not necessary for even moderately different networks as long as all systems utilize the same hardware estimations.

The hardware behavior and performance are then used to create an identical module in SystemC, a C++ variant that enables cycle-accurate simulations. These SystemC modules can be interconnected and compiled to produce an executable that simulates the behavior of the entire network. Thus, this framework provides a methodology for designing entire parallel networks that are as accurate as hardware simulations but enable large systems to simulate in reasonable amounts of time on a single workstation.

SystemC is C++ based hardware design language that was developed to promote system-level simulation and to enable hardware–software cosimulation. [17,6]. Fundamentally, SystemC is a set of parameterized template classes built in C++ that allow the creation of hardware structures available in other languages such as bit-vectors, processes and ports. Like other hardware languages, such as VHDL and Verilog, it is possible to describe a SystemC design behaviorally, at the register-transfer level, and structurally. The advantage of SystemC is most highly visible in the fact that it essentially C++ code. As a result, SystemC designs, along with their corresponding test benches, may be compiled directly into a software binary that becomes a custom simulator for that particular hardware design. Similarly, for system-level simulations designed in SystemC, combining software and hardware portions becomes much easier as they can be combined and built using a single program. For traditional hardware simulation techniques such as using ModelSim [23] for VHDL or Verilog, a foreign language interface is required to communicate between hardware and software components. The most important advantage of SystemC for our simulation environment is its increase in capacity over more traditional hardware simulation methods. Because ModelSim must be able to simulate every VHDL construct, even those rarely used, it incurs significant overhead. For the equivalent SystemC simulation, a custom simulator is built and only the components required for the application are incorporated into the simulator. From our experience, this results in accelerated performance by a factor of three and results in a factor of five for memory utilization. In fact, we found that our VHDL simulations using ModelSim only scaled to 32 processors while our SystemC simulations scaled to over 128.

In order to accurately build and simulate a high-performance multi-processor network, the network interface controller (NIC) hardware and the switch element(s) must all be designed in hardware for maximum performance. To simulate the entire system, the processing elements, the wires and the topology must be accurately modeled but do not need to be designed using a hardware description language. However, to validate our designs, we built a 32 processor system entirely in VHDL and then built an equivalent system in SystemC using equivalent components. In the next section, we show the system-level results but in this section, we focus on the fundamental components that we created and reused throughout the different systems that we built. The processing element component reads data transmission commands from a file and sends data into the NIC. The processing element component also receives data from the NIC and records it to a different file with a timestamp. The NIC, however, was designed in hardware using three different components: a single-wide data queue component, a N-wide data queue component and custom control logic. The two different types of data queues are described in more detail in this section. The wire component emulated the behavior of a high-speed network cable. The switch is comprised of a scheduler component and a switch fabric component. The scheduler determines the configuration of the switch and as a result, its performance and design is central to the network's performance. Thus, the schedule component was designed in hardware. The switch fabric can be an analog, digital or optical device and, as such, only its behavior is described. All buffers within the switch were modeled using the data queue components. The reminder of this section describes each of the individual components while the next section describes different systems that we constructed from these components.

3.1. The process element component

A significant portion of a communication's delay is in software overhead and in moving data from the processor, or from memory, to the network interface card. The literature supports the benefit of innovative approaches in this area. However, this paper focuses on the performance of the network and does not consider the delays associated with the processor/memory to NIC interface. We are doing this for two reasons. First, the only modifiable components within a processing node in a cluster are the network interface cards and the software executing on the processor, with a fixed processor-to-NIC interface. Second, the network design and the processor interface are not tightly coupled. Improvement on the processor interface will help all networks and improvements on the network will benefit all types of processor interfaces. Thus, we virtualize the processor as an outgoing queue that contains data to be sent out onto the network, and as an incoming queue that receives packets from the network.

Each processor has its own input file that contains a number of predefined commands, shown in Fig. 2. The command send tells a processor element (PE) to generate data with a specific message size and destination. The command wait emulates a period in which the PE is performing computation and thus, no traffic is generated. In addition to these

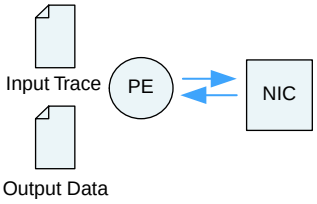

Fig. 2. Processing element components.

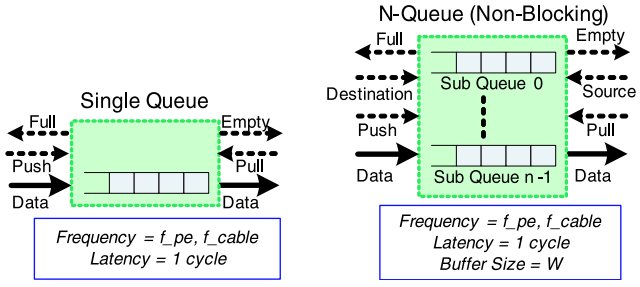

Fig. 3. The single queue and the N-queue components.

basic commands, advanced commands can be grouped to perform more complex MPI functions, like broadcast, blocking send, blocking receive, and barrier, among others. The amount of data that are sent is described in the input file, but the actual data that is sent are not important to the network operation, as the network does not inspect the data payload of the packets. For debugging purposes, however, the payload of the packet is used to send the source and a timestamp the packet was created. At the destination PE, this information along with its arrival time is stored in the output file for postprocessing and performance summary. This processor model allows us to test a variety of traffic models by simply creating a set of input files. Section 5 describes a number of communication patterns used for our experiments.

3.2. Data queues

One of the fundamental components in a network is the data queue. Anywhere data are being sent or received, there is a need to buffer data. Functionally, a queue receives a stream of data and outputs the stream in the same order. We have created two different data queues, a single queue and an non-blocking N-queue, shown in Fig. 3. The single queue is simply a first-in, first-out queue, while the N-queue is a single component that represents N different queues grouped together with a single write port and a single read port.

The *Full* and *Empty* status lines indicate the availability of the queue for writing and reading, respectively. For the single queue, these status lines are a single bit wide and for the N-queue they are N-bits wide to indicate the status of each of the N internal queues. Writing data into the single queue simply requires holding the *Push* signal high for a single cycle, but for the N-queue the *Destination* queue must also be specified. Likewise, holding the *Pull* signal high for a single cycle will read data from the single queue, but with the N-queue the *Source* must also be specified.

The performance of queue can also vary. If the *Pull* signal is active (i.e. a '1'), the queue outputs a data value every cycle. The frequency of data movement into or out of the queue is one word per cycle. This frequency along with the width of the queue determines its bandwidth. The latency of the queue is the amount of time between placing a data value in an empty queue and the time it can be removed.

The single queue was already designed within the Mentor Graphics HDL Designer system as a component in their ModuleWare library [22]. Similarly, both major FPGA manufactures, Xilinx [38] and Altera [3], have wizards for configuring FIFOs that automatically generate synthesizable hardware components. The simple queue has a single clock cycle of latency through the queue, with the cycle frequency of 108 MHz for the Altera FPGA, EP1S25F1020C-5. Thus, it's throughput is 108 million words per second where the width of the queue is the word size that can be arbitrarily configured. The latency is one cycle or $1/108\,\text{MHz} = 9.2\,\text{ns}$.

The N-queue can be designed in numerous ways, depending on the objective sought. The simplest implementation is to replicate the single queue N times and multiplex the *Head* and *Tail* of the queues using the source and destination as select lines, respectively. While this is appealing from a rapid design perspective, it suffers from inefficiency, as N dual-ported RAMs and N comparators are needed. We observe that during any given instance at most one queue will have data placed into it and at most one queue will have data retrieved from it. The same queue can have both read and write access simultaneously but this means that only a one dual-ported memory is needed to buffer the packet data. There will need to be N head pointers and N tail pointers to addresses in data RAM. The problem is keeping track of the head and tail pointers for each of the N internal queues, as well as updating the *Full* and *Empty* status lines.

For our design, we implemented the *Head* and *Tail* pointers using two small register files that have three address ports. For a *Pull* operation, the *Head* pointer is used to specify the address in the data RAM for reading, and for a *Push* operation, the *Tail* pointer is used to specify the write address in the RAM. The second port is used to write back the incremented pointer after the read, or write, is performed. The third port is used to update the *Full* and *Empty* flags. On a *Pull* operation, the head-of-queue is incremented, if the *Head* and the *Tail* pointers are the same, then the queue is *Full*. On a Push operation, if the incremented *Tail* and *Head* pointers are the same, then the queue is *Empty*. Thus, each register file must have two read ports and one write port. As shown in Table 1, we obtained hardware area and performance results by synthesizing our VHDL to an FPGA. There is negligible decrease in performance as it

Table 1
N-queue hardware synthesis and performance results $N = 4$–128, width = 64 bits, FPGA target: Altera EP1S25F1020C-5

	N					
	4	8	16	32	64	128
Logic cells	361	480	1439	1988	3939	8010
	(1.4%)	(1.9%)	(5.6%)	(7.8%)	(15.4%)	(31.2%)
Memory (bits)	16,386	32,768	65,536	131,072	262,144	524,288
	(0.8%)	(1.7%)	(3.4%)	(6.7%)	(13.5%)	(27.0%)
Clock constraint (MHz)	78	88	69	67	63	59
Throughput (Gbps)	4.9	5.6	4.4	4.3	4.0	3.8
Latency (ns)	13	11	14	15	16	17

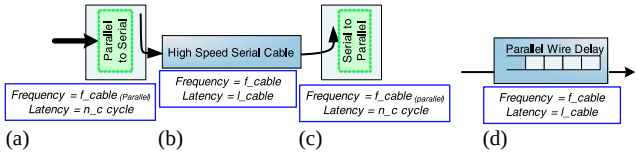

Fig. 4. Four wire delay models (a) parallel to serial, (b) high-speed serial, (c) serial to parallel and (d) parallel wires.

scales to 128 queues but a proportional increase in circuit size (i.e. logic cells) which is due to the 3-ported register file. ASIC results are expected to be five to ten times faster.

3.3. Wires

The physical layer interconnection of a system can have a drastic and dynamic impact on the entire system [15]. Early in the system design, the characteristics of the communication channels are specified in general terms. This may include the functionality, latency, bandwidth and bit-error rate of each network link. In complex systems, these characteristics alone can significantly alter a system's performance and guide the underlying system design. Optical switches may combine these performance characteristics with unique topologies such as multicasting and time/wavelength division multiplexing. In order to design a system effectively, these communication characteristics must be simulated along with the entire system.

At a high level of abstraction, physical interconnections can be modeled as parallel wires that contain a specific delay. This is easily achieved using the VHDL command `A<=B after 5 ns`, which assigns the value of *B* to the output *A* after a delay of 5 ns. This command can be used for busses as well as wires, but causes a latency delay and a bandwidth of the same value. For example, changing *B* from low to high and then back to low within a 5 ns period will not cause a corresponding change in *A*. However, using the VHDL statement: `A <= transport B after 5 ns` will enable changes smaller than 5 ns to be seen on *A*. When the source of the signal is generated by a clock edge, or is regulated by some other portion of the circuit, then the simple delay is sufficient for modeling and more complex mechanisms are not required.

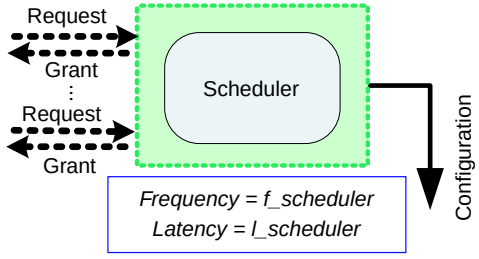

Fig. 5. The scheduler.

For our simulations, we have defined four components as shown in Fig. 4. The first component is a parallel-to-serial converter, the second is a high-speed serial cable, the third is a serial-to-parallel converter, and the last is a parallel wire. For the parallel-to-serial component, there will be a one-clock cycle delay associated with this component at the clock cycle frequency of the sender. The high-speed serial cable has a latency that is proportional to the length of the cable being simulated and can be conservatively estimated as 1–2 ns per foot. The bandwidth for this cable must also be specified, as this will determine its throughput. The serial-to-parallel converter must wait until all of its bits have been serially shifted into the register, thus, there is a latency associated with this component. This latency is its width serial frequency, which must be equal to its parallel frequency. Lastly, the parallel wire component is used to define delays that are within a chip. It should be noted that the parallel wire component can also be used to simulate a sequence of parallel to serial, high-speed serial and serial to parallel components.

3.4. Network scheduler

One of the critical components of a network is its arbitration logic. If multiple processors send data to a single destination, there will be a conflict within the network that needs to be resolved. Within a packet switched network, this would be seen as multiple packets in different input ports that have the same output port. For circuit switching, multiple NICs would request a circuit to the same destination. Irrespective of the network type, there must be some arbitration logic that determines which processor, or port, gets priority.

Table 2
Scheduler hardware synthesis and performance results for four to 128 ports

	N					
	4	8	16	32	64	128
Latency (ns)	34	49	76	120	213	385
Frequency (MHz)	28.6	20.6	13.1	8.3	4.7	2.6
Millions of scheduled ports/s	114	164	209	265	300	332

To handle arbitration, we created a component called a scheduler that receives up to N requests for N destinations as shown in Fig. 5. Each *Request* input is an N-bit bit-vector, which specifies the destinations to which it needs to send the data. For PE_j, if *Request[i]* = '1' then PE_j has data that it needs to send to PE_i. The output of the scheduler is a *Grant* signal for each of the N *Requests*, in which each PE is granted at most one destination that was requested. A *Configuration* is output from the scheduler to the switch fabric indicating its configuration.

To give a baseline for this component, we have created multiple schedulers for crossbar with 4–128 inputs and outputs. Other schedulers can be created for different switch fabrics. Our designs use N levels of logic to determine the schedule in a single cycle for all N processors. Each level of logic has two inputs, the available resources, *Available*, and the requested destinations, *Request*, along with two outputs, the granted destination, *Grant*, and the remaining destinations that have not been scheduled, *Available*. All four signals are N-bit bit-vectors, in which the bit position indicates a destination value. By bit-wise AND'ing *Available* with *Request*, a bit-vector of available destinations is decoded. One of these destinations is selected, and that bit position in the *Grant* bit-vector is set to '1', with all other bits set to '0'. The same bit position in *Available* is set to '0', and assigned to *Available*, which is sent to the next level of logic.

To avoid a biased schedule, we have implemented a round-robin priority scheme that determines the order in which requests are granted. With no priority schemes, PE_0 would always get the highest priority, while PE_{N-1} could starve. Using a round-robin priority schedule, the available resources are first offered to PE_j and then to $PE_{(j+1)\,MOD\,N}$, and so on. This essentially shifts the priorities by one every scheduler cycle, and ensures fairness.

This hardware algorithm appears to be a sequential program, but given the gate-level simplicity of each level, all N levels can be calculated in a single cycle. Performance results are shown in Table 2 and, while the maximum clock frequency decreases as N increases, the overall number of destinations that can be scheduled increases as N increases.

3.5. Switch fabric

Switching fabrics have been around since the early days of parallel computing with a mature research field in multistage interconnection networks. In this paper, we have separated the control portion of the switch, which we refer to as the scheduler, from the data path portion of the switch. The switch fabric can be a single crossbar, a multistage interconnect switch, or any kind of topology as shown in Fig. 6. The results in Sections 4 and 5 are for a crossbar.

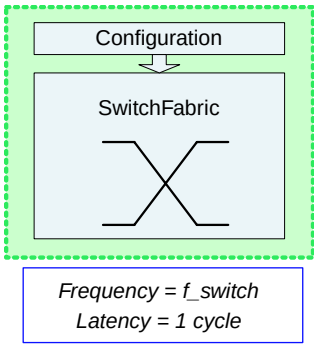

Fig. 6. Switch fabric.

Table 3
The switch fabric component

Performance	Digital	Electrical (LVDS)	Optical
Latency	> 10 ns	10–50 ns	1000+ns
Throughput	100–500 Gbps	2.5 Gbps	1–10 Gbps
Delay	20–200 ns	1 ns	< 1 ns

Depending on the type of technology used to create the switch, the configuration time and the propagation time, i.e. the configuration and data latency, respectively, can change by orders of magnitude. Digital switches have faster switching latency, but have a longer propagation delay and a lower per-pin throughput than low-voltage differential signal (LVDS) switches and optical switches. Table 3 gives three sets of parameters that are typical of digital switches, LVDS switches and optical switches. These values will also have a higher propagation delay. Electrical and optical switches have higher switch latency, but a much larger propagation delay. In fact, their propagation delay approximates the delay in one foot of cable. The throughput of all devices is similar, as digital devices utilize more pins or, more commonly, will combine LVDS transceivers within their chip packaging.

4. System simulation

This section illustrates the design methodology for large systems using the proposed design and simulation framework. In the prior section, we described the individual components that were created using a hardware description language and characterized by their FPGA performance. We examine the characteristics of three networks and show how different networks, with different characteristics, can

be compared using the common set of components and simulation framework. The three different types of networks we created were introduced in Section 2 and are packet switching, circuit switching and predictive circuit switching. In this section, we show the various parameters that can be set for the different components.

The behavior of each module is predefined but the performance can be modified. Each module is given a *frequency* and a *latency* parameter. Recall that the inverse of the frequency of a hardware device is duration of one clock cycle and that this duration is determined by the target technology. As the density of transistors increases, the clock frequency also increases. This parameter is therefore technology dependent. The latency of a particular component is specified as the number of cycles required to achieve the result. This parameter is design-dependent and can be derived from the architecture of the component. The internal storage of a component, if applicable, is also design-dependent. Thus, by specifying the frequency, the latency and the buffer size of each component we can characterize an entire system. Many components utilize the same frequency as will be shown.

4.1. Wormhole switching

Wormhole switching networks decompose all communication into small point-to-point messages that are routed through the network independently. At the destination, the original message is reassembled from the individual packets. To simulate a wormhole switching network we represent the processor as a data-sender and as a data-receiver and do not consider the overhead associated with the processor interface or the creation of network packets. As such, we expect that there will be an additional latency, for a 'real' network, that is not considered here. However, our goal was to be able to compare and contrast different networks and, as such, the processor and bus interface circuitry would be the same.

The network interface card/controller is shown in Fig. 7 as two simple queues, one for output traffic and one for inbound traffic. For clarity, only one NIC for sending and receiving data is shown. In our simulation, data can be sent and received by NICs simultaneously. A small amount of control logic was added to the NIC to handle backpressure, not shown in the figure. For the switch, we implement an input buffered switch and a single scheduler to perform the routing/arbitration. We simulate this using an N-queue component for inbound traffic, a scheduler component for routing/arbitration and a switch fabric component for the crossbar. Data coming from the switch into the NIC are buffered into a simple queue and written to a file with a time stamp by the processing element. Each worm was created with using 64-bit words, a one word header, a one word flit, a 10 flit payload and a one word tail.

By using input buffering, we enabled the scheduler to improve the switch utilization, since it has knowledge of all destinations for each of the N ports. As described earlier, we have implemented these N-queues in hardware and have

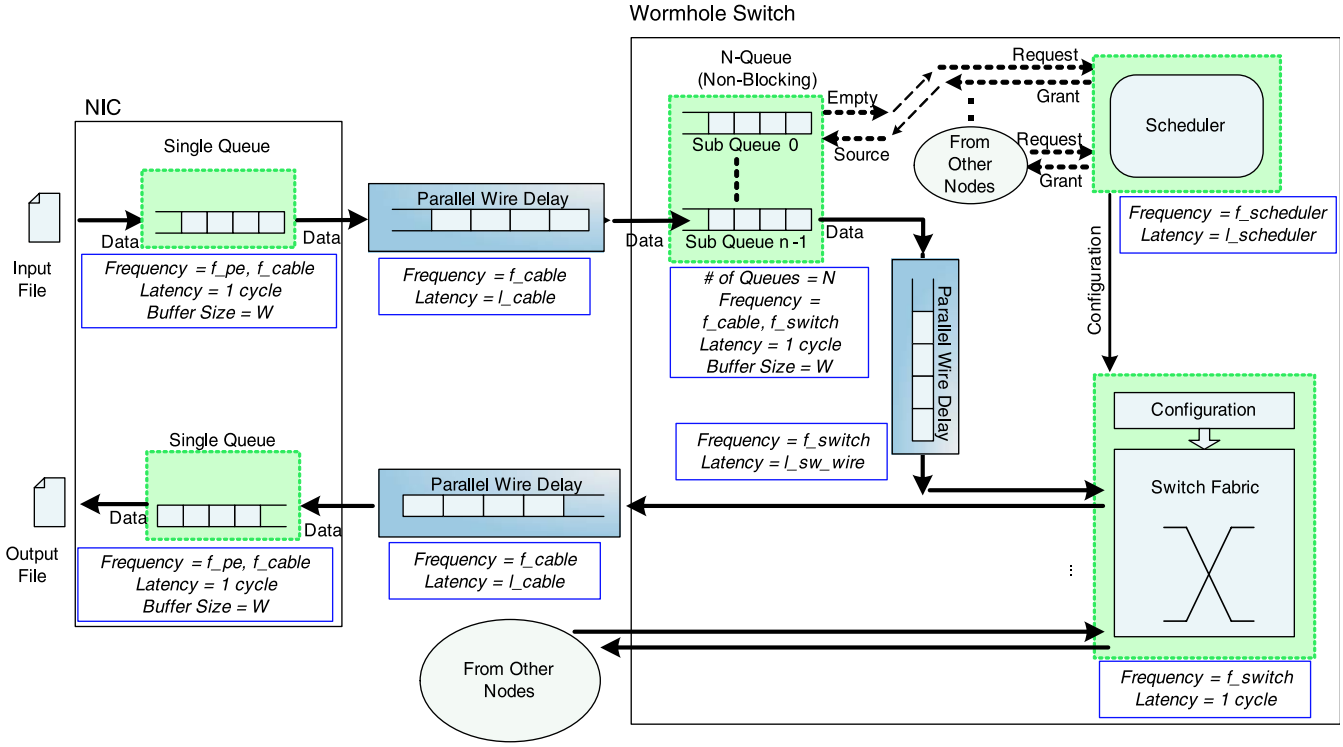

Fig. 7. Wormhole switching network.

shown that head-of-line blocking can be avoided even for a large N by using hardware, as long as there is only one word written and one word read per cycle. This assumption is realistic for a switch, as each port has a single cable receiving inputs and a single switch fabric interface. This, however, is not true for all designs and must be taken into consideration. The scheduler component receives an N-bit vector from each N-queue specifying which of its internal queues have data. It then allocates the bandwidth using the round-robin priority scheme described in Section 3 but schedules all destination ports within a cycle. This enables a single-cycle allocation of bandwidth and enables out-of-order routing from each port, both of which increase network utilization. Other schemes can be implemented by simply changing the design of the scheduler and re-running the traffic traces.

4.2. Circuit switching

Circuit switching utilizes the same components as wormhole routing even though these are very different networks. Surprisingly, the major difference is the location of the buffers and the corresponding distance between these buffers and the schedule, shown in Fig. 8. For wormhole routing, the N-queue was next to the switch fabric, but in circuit switching, this buffer is located within the NIC. Thus, each time a packet needs to be sent, a request must first be sent to the scheduler, the scheduler must determine if the request can be granted, the circuit is established and an acknowledgement is sent to the NIC. The circuit is maintained until the NIC's buffer is empty for that particular source-destination connection.

Some switch fabrics, like all optical switches, cannot buffer data and thus, circuit switching is required. By having a central scheduler that has knowledge of all of the data that needs to be sent in any given cycle, there is a greater chance to improve network utilization. For large networks, a wormhole routing switch only sees data that are in its queues while a circuit switch with centralized scheduler has complete knowledge of all pending traffic. For networks with multiple stages, the wormhole routed switch will only

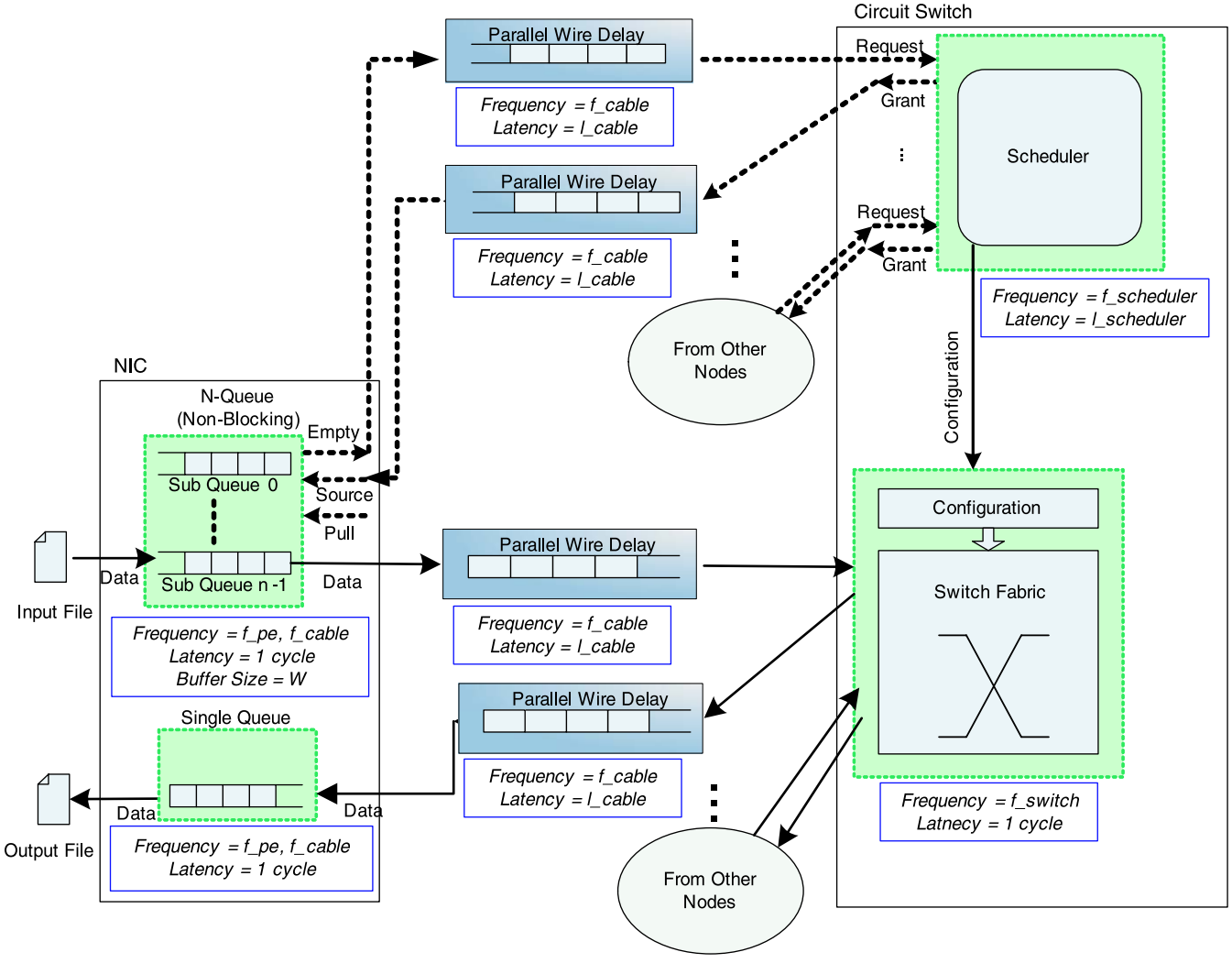

Fig. 8. Circuit switching network.

have local knowledge and not global knowledge of the pending data traffic.

4.3. Predictive circuit switching

In order to avoid the overhead of circuit switching, we introduce predictive circuit switching, shown in Fig. 9. In this type of network, the circuits are setup before they are requested. This concept is analogous to cache prediction, but rather than fetching data before it is requested, predictive circuit switching configures the network before it is needed. If the network predicts accurately, a "hit", then there is no setup latency and the network appears as if each source is directly connected to its destination. Recall that the latency of a message is its wires, buffers, scheduler and switch fabric. Circuit switching removes the buffers from the switch and reduces its latency of a switch to that of a small segment of cable because data can stay in the optical or analog domain. By predicting the next connection, the scheduler delay is hidden because it is precomputed. The switch configuration still exists, but is minimal for LVDS switch elements and other similar technologies.

However, when the switch predicts incorrectly, there is a miss penalty that can be substantial. If the network supports preemption, the penalty can be little more than that of circuit switching, but if the predictive circuit switch has a complete, predefined set of configurations, then an unpredicted communication may have to wait for a few communication cycles before it can send its data. This paper introduces the concept of predictive circuit switching, affirms its benefits during predictable traffic, and examines its drawbacks during unpredictable traffic. For this initial discussion and simulation, we use a round-robin prediction method that cycles a fixed set of destinations. We have demonstrated through simulation that this scheme is better than packet and circuit

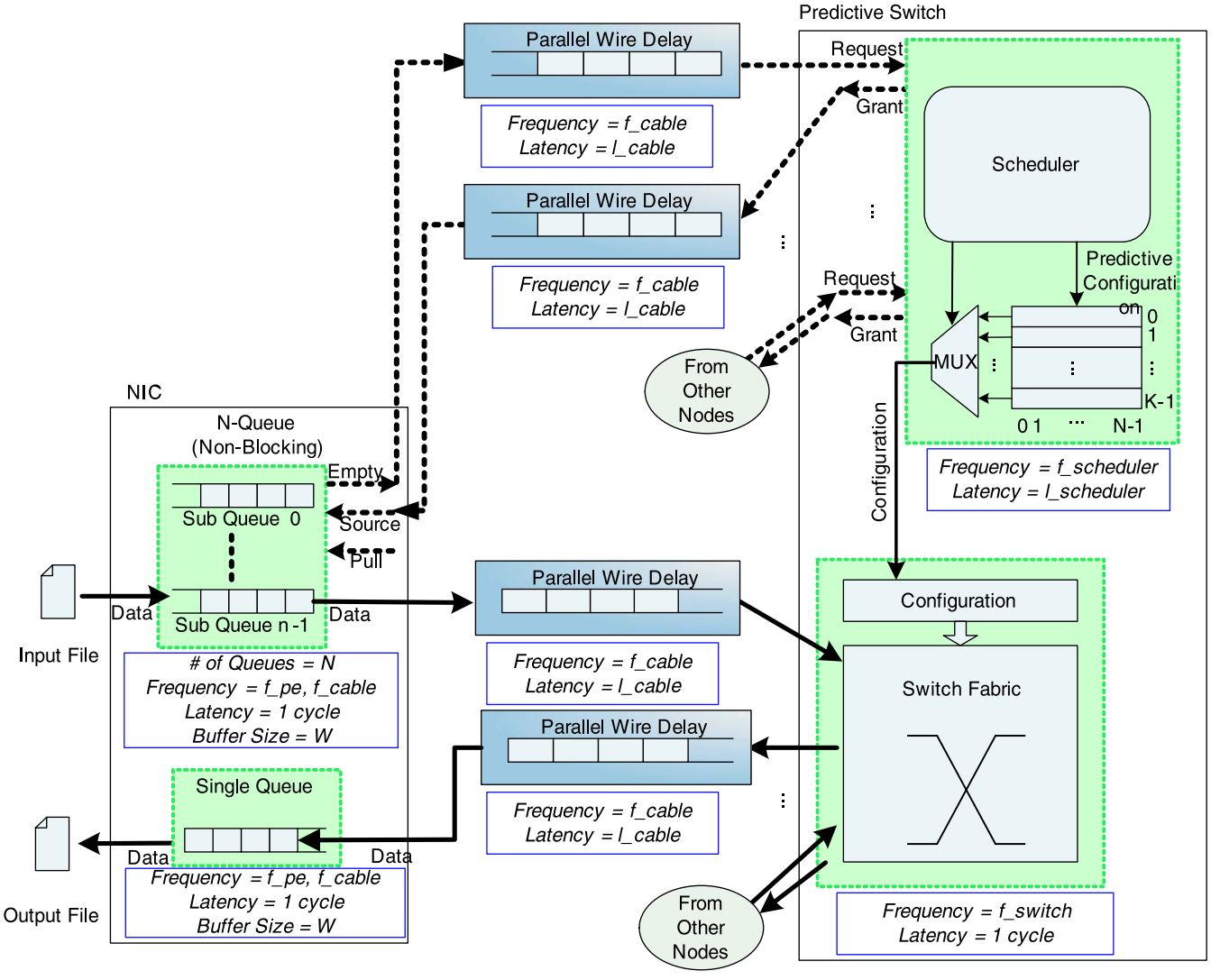

Fig. 9. Predictive switching network.

switching when there is a high degree of predictability, and that this scheme has a high cost for missed predictions.

5. Performance results

Our experiments are based on the three kinds of switching networks described in the previous section: wormhole switching network, circuit switching network and predictive switching network. All parameters in our simulation are configurable and in the experiments performed, we set these parameters to values that match our hardware synthesis results. Our initial experiments utilized ModelSim to execute our VHDL simulations for 32 processors. We found that scaling beyond this level exceeded the 1.5 GB memory capacity but did not fully utilized the processor. By converting to our SystemC modules and executing the software version, we validate that our two models are nearly identical and that our SystemC framework scales to 128 processors. Using SystemC, the bottleneck was processing and not memory capacity.

The frequency of the processor, *f_pe*, determines the data generation speed, and was set to 500 MHz to approximate a fast processor/NIC interface. The buffer size for the N-queue component is small at 128 bytes, which is 16 64-bit words. In our 'large buffer' simulations, this is increased to 240 KB or 30 K words but this is only shown for SystemC as this was not possible for our VHDL simulations. The clock frequency for cable *f_cable* was set to 100 MHz to emulate a 6.4 Gb/s throughput as our wire simulator actually passes 64-bit words. All tests were simulated twice with short cables and with long cables. For systems using short cables, the cable latency, *l_cable*, was set to 10 ns, while for systems using moderately long cables, the cable latency was set to 100 ns. These latencies approximate 10 foot and 100 foot cables. The scheduler's working frequency, *f_scheduler*, was set to 100 MHz, with a latency, *l_scheduler*, of 2 cycles. The clock frequency of the switch fabric, *f_switch*, was also set to 100 MHz with a latency, *l_switch*, of 1 cycle as this fabric does not contain any buffers and could be in the analog domain.

There are four traffic sets that represent the corner cases for contention and predictability for a heavily loaded network. This shows the impact of latency on performance and motivates the need to consider the physical layout of large clusters. In order to incorporate the collective impact of latency, peak bandwidth and contention, we show our results in terms of the effective bandwidth which we calculate by dividing the total number of data bits sent by the total time requires for a set of messages. We normalize this value by dividing it by peak bandwidth of the cable.

5.1. Low contention with low predictability

Our first test case sends random traffic to all destinations. This represents low predictability as the traffic is random and low contention as the traffic is evenly distributed. The performance of five different networks is shown in Fig. 10. The *ideal* circuit switch and the *ideal* wormhole switch are shown as dotted lines and are calculated, not simulated, using the latency and bandwidth characteristics from their components. As such, contention is not considered. The ideal numbers represent the upper bounds of performance for each switching technique. The dashed line is the predictive circuit switching, while the solid lines show wormhole and circuit switching. Since the traffic has low predictability and is random, the scheduler for predictive switching has to cycle through all destinations for each processor.

We find that for small messages, predictive circuit switing and wormhole routing are close in performace. However, for long cables, the performance of circuit switching drops because the distance of the control path between the NIC

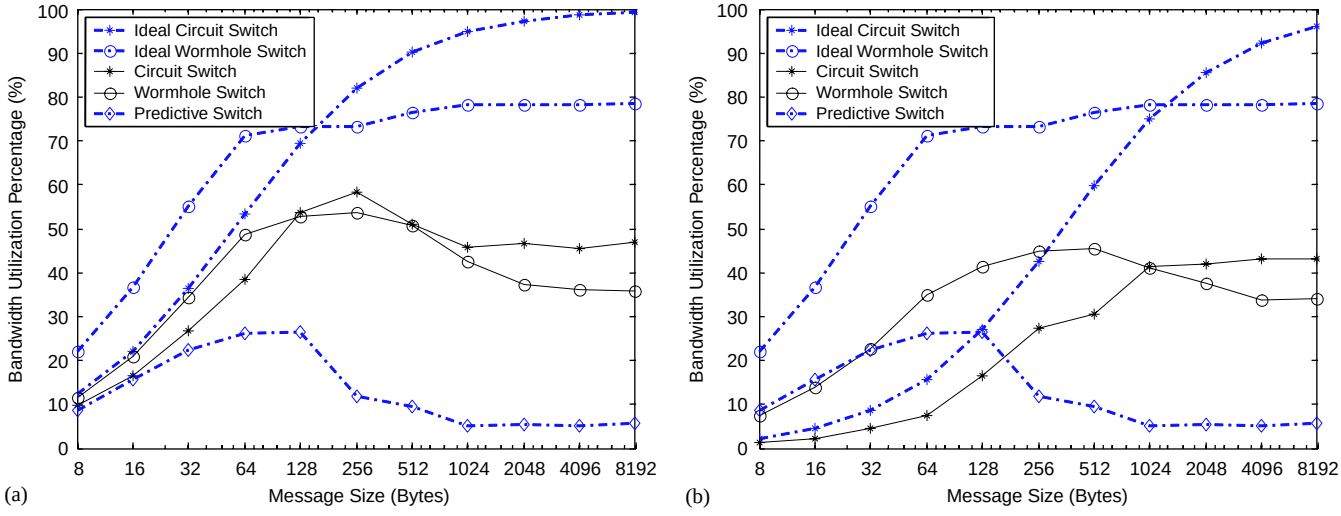

Fig. 10. Random-to-all communication pattern: (a) 10 foot cable; (b) 100 foot cable.

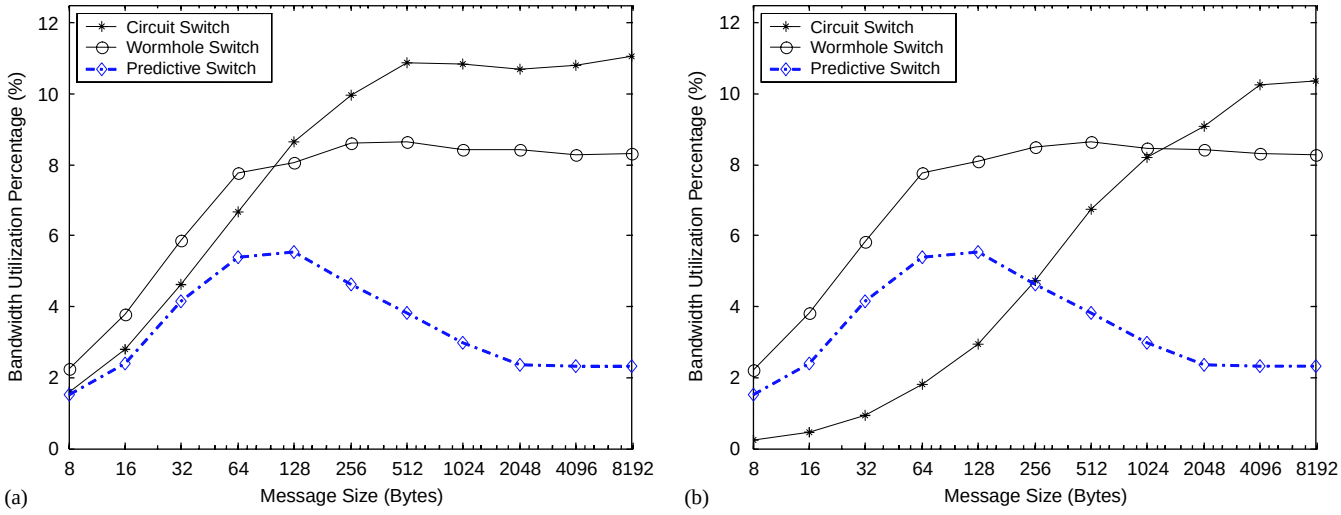

Fig. 11. Random-to-four communication pattern: (a) 10 foot cable; (b) 100 foot cable.

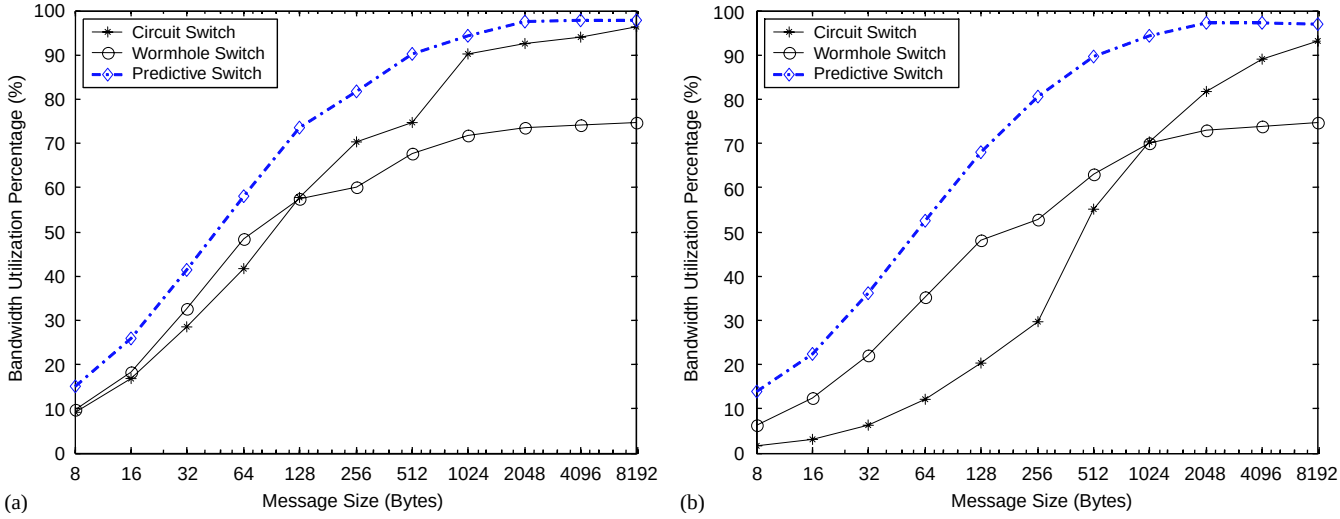

Fig. 12. All-to-all communication pattern: (a) 10 foot cable; (b) 100 foot cable.

and the switching node increases. Therefore, it takes more time to establish circuit connections. Because the random traffic has very low predicability, in the predictive switching technique all 32-destination configurations are preloaded for each processor. All 32-destination configurations are rotated sequentially in round-robin manner. If there is one message to be sent, it must wait for the average of 16 communciation cycles, which drastically increases the message latency. Each communication cycle can send 80 bytes, so message that requires more than one cycle, it must wait for 31 cycles before it can continue to send its data. Hence, the hit ratio is very low. This describes the worst-case scenario for predictive circuit switching. The overall performance of the predictive switching, circuit switching and wormhole routing drops considerably when the message size over 128 bytes. This is due to the limited buffer size. After the buffer is full, the scheduler within the switching node will have fewer options to route packets. The simulation for large buffer will be described later.

5.2. *High contention with moderate predictability*

In our next test case, we continue with random traffic but restrict the number of destinations to a set of four processors to model 2D mesh communications. Performance is shown in Fig. 11. The reduction in destinations for a given processor increases the contention within the network, and also decreases the penalty of a miss prediction. For our predictive switch, we held to a strictly round-robin schedule without preemption. Thus, if there is only one packet in the buffer, it must wait for an average of two communication cycles, and for larger data messages, these cycles increase in duration. For this trace, as with the previous trace, the commu-

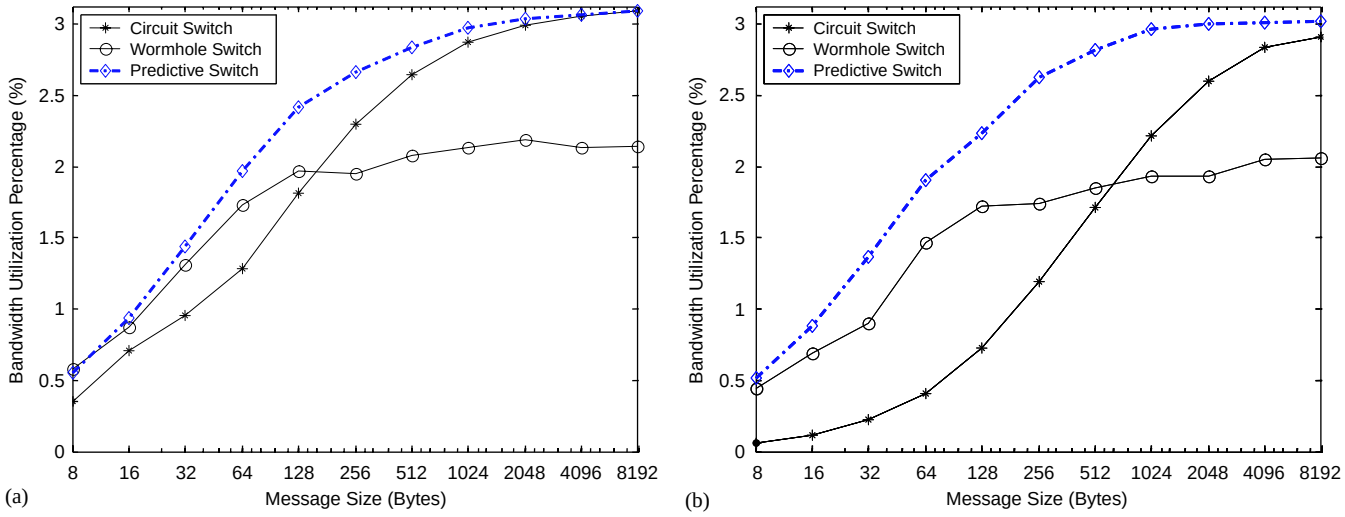

Fig. 13. Gather-to-one communication pattern: (a) 10 foot cable; (b) 100 foot cable.

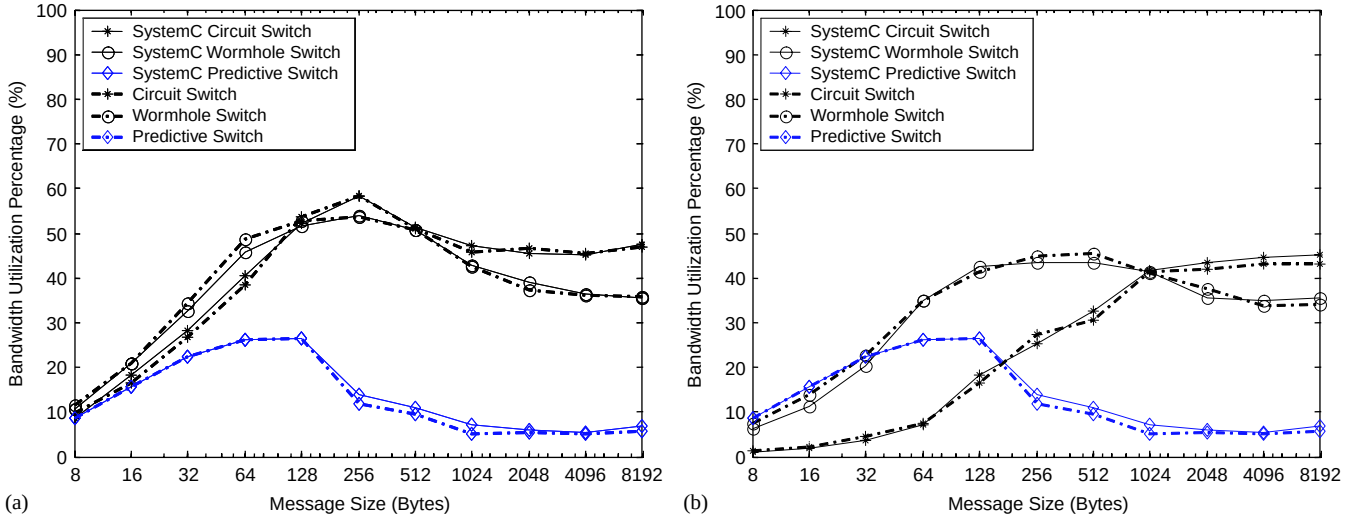

Fig. 14. SystemC simulation vs. VHDL simulation (random-to-all communication pattern): (a) 10 foot cable; (b) 100 foot cable.

nication cycle sends 80 bytes of data, such that all messages that are larger than 80 bytes must wait for their next cycle in the round-robin. The idle time for this trace is only 3 cycles making the effective bandwidth larger than the previous trace, which had the potential to idle for 31 cycles.

Wormhole switching proved to be superior to predictive switching and to be superior to circuit switching for small and medium-sized messages, but was less efficient for large messages. The crossover point changes based on the cable length, as a longer cable impacts the latency factor significantly for circuit switching. For large messages, circuit switching surpasses even *ideal* wormhole switching.

5.3. Moderate to high contention with high predictability

In this test case, we exclusively test personalized *all-to-all* communications for one benchmark, shown in Fig. 12, and *gather* for another benchmark, shown in Fig. 13. In this experiment, we assume that either the compiler or the user has specified this traffic pattern and that the schedule is preloaded within an appropriate round-robin scheme. Given this assumption, we have adjusted the communication cycle duration for predictive circuit switching to be that of the message size.

As one might expect, the predictive circuit is the most efficient for all buffer sizes. For small messages, there is still an overhead for switching the circuit between different messages and as a result all three techniques have a low effective bandwidth. For very short cables, predictive circuit switching is very similar to circuit switching, but for longer cables, predictive circuit switching outperforms circuit switching because of its lower per-message latency. Similarly, predictive circuit switching outperforms wormhole switching because the wormhole switch still needs to arbitrate which source gets to send to a particular destination.

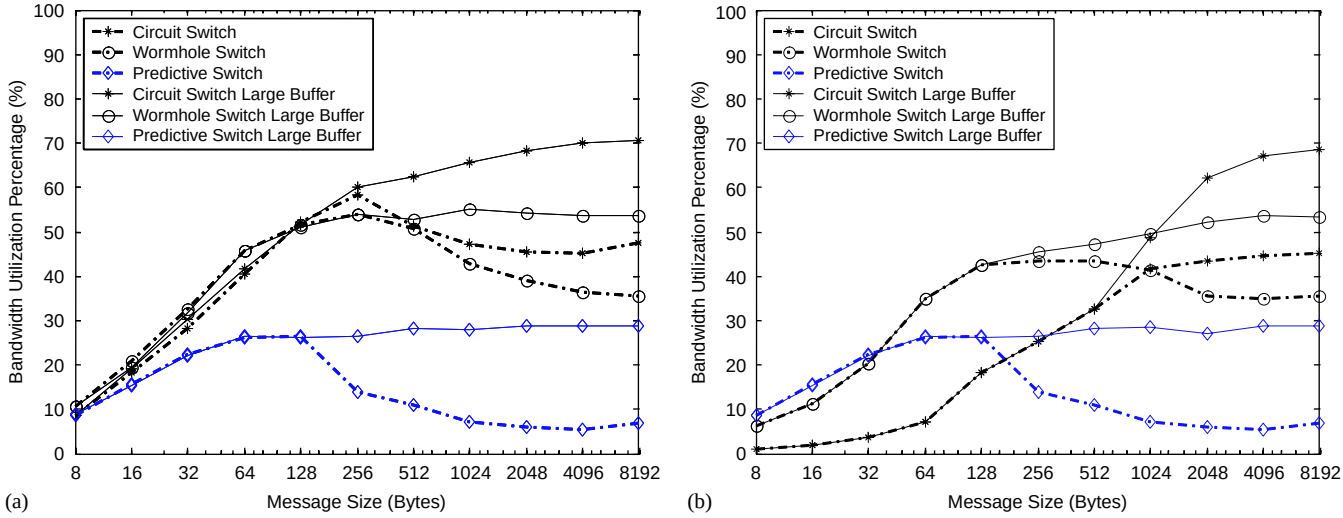

Fig. 15. Simulation of buffer size vs. bandwidth (random-to-all communication pattern): (a) 10 foot cable; (b) 100 foot cable.

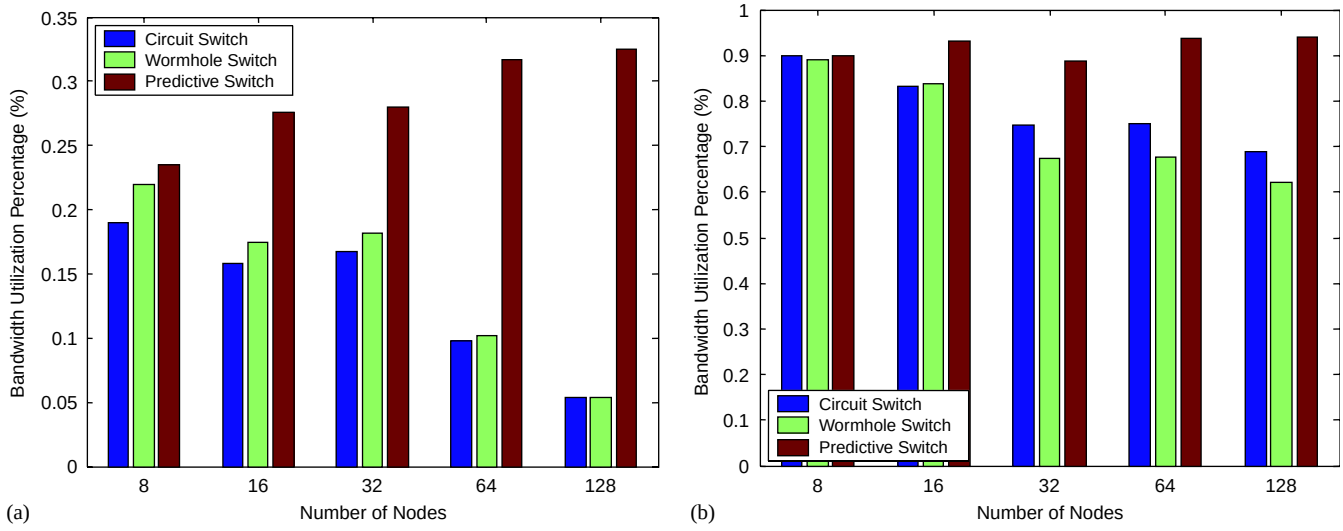

Fig. 16. SystemC system simulation for up to 128 processors (all-to-all communication pattern, 10 foot cable): (a) 16 byte message; (b) 512 byte message.

For the *gather* communication operation, the effective bandwidth peaks at 3% because only 1 of the 32 processors is receiving data. Again, this is a highly predictable communication pattern and thus, having an appropriate schedule within a predictive scheduler is beneficial.

5.4. Scaling from 32 to 128 processors using SystemC

In our design and simulation framework, we propose that the modules be designed using a hardware design flow to extract performance characteristics. To avoid the scaling problem that is inherent to VHDL simulations, we utilize the performance characteristics of the VHDL module design to create equivalent SystemC components that are then integrated into a larger system. To verify that this methodology is accurate, we have simulated the random-to-all traffic pattern in both VHDL and in SystemC. As can be seen from the performance curves in Fig. 14, the SystemC and VHDL simulations are nearly identical.

One of the benefits of using SystemC is more efficient use of memory. We noticed from our graphs that performance seemed to drop when the buffers became full. We tried to increase the buffer size of the N-queue in our VHDL simulations but we ran out of physical memory. We expanded the buffer size for each destination to 240 kbytes in SystemC simulation and noticed the expected performance improvement. Fig. 15 shows the bandwidth utilization for both small (128 byte) and large (240 kbyte) buffers.

To illustrate the benefit of the scaling our simulations to 128 processors, we simulated the *all-to-all* communications pattern for all three networks using 10 foot cabling.

When the number of nodes is small, predictive switching, circuit switching and wormhole switching have similar performance. However, when the number of nodes increases to 128 for small messages (16 bytes), predictive switching significantly outperforms the others, shown in Fig. 16(a). For medium-sized messages, the benefits of predictive switching become more pronounced as the system size increases, shown in Fig. 16(b).

6. Conclusions and future directions

This paper has presented a common framework for designing, synthesizing and simulating parallel computing networks. By using a hardware design flow, each component can be designed separately and characterized in terms of latency and bandwidth. By using FPGAs as the target technology, we are able to present performance results that can be compared against, and give insight into, ASIC performance. The hardware synthesis tools provide a maximum frequency of the device, and from simulations we can determine the latency in terms of clock cycles. By multiplying the cycle latency and the device frequency, we can accurately determine the latency down to the nanosecond (10^{-9}) level of accuracy.

By making our framework modular, we are able to create different networks using components. The input and output files provide the network traffic. By using the VHDL hardware description language with a hardware simulator, we are able to simulate the entire network to cycle accuracy using communication traces. The network performs the actual routing and contention arbitration necessary to route data through a large parallel computing network. This level of system simulation enables us to examine the true behavior of the network with a specific set of parameters, and with specific switching techniques.

This paper also introduced predictive circuit switching, and compared it with wormhole routing and with circuit switching using trace files with different levels of contention and predictability across various message sizes. We observe that predictive circuit switching is the most efficient of the three techniques when the traffic is highly regular, e.g. collective communications, and when both the packets are relatively small with multiple packets in different outgoing destination queues.

Future directions of this work include the simulation and creation of other network switching techniques, as well as, the establishment of parallel communication benchmarks that will fit into the presented framework. Network traffic for future simulations will have a mixture of predictable and unpredictable communications with various sized messages. One such benchmark will be constructed to test a variety of collective communications, while others will focus more on point-to-point messages.

We plan on expanding the number of switching levels and simulating a large FAT tree built from a variety of switch element sizes. There is also room within the scheduler design to enable it have a variety of priority schemes. The N-queue will be expanded to enable multiple readers so that a buffered crossbar can be created as the switch element. In summary, we have presented a framework for the design, hardware synthesis and cycle-accurate simulation of multiprocessor networks that will enable exploration of various designs with a common analysis metric.

References

[1] G.A. Abandah, E.S. Davidson, Modeling the communication performance of the ibm sp2, in: Proceedings of the 10th International Parallel Processing Symposium, 1996, pp. 249–257.
[2] A. Afsahi, N.J. Dimopoulos, Hiding communication latency in reconfigurable message-passing environments, Ph.D. Thesis, University of Victoria, 1999.
[3] Altera, Stratix II device handbook, Tech. rep., Altera Inc., 2004.
[4] T. Austin, E. Larson, D. Ernst, Simplescalar: an infrstructure of computer system modeling, IEEE Comput. 35 (2) (2002) 59–67.
[5] L. Breslau, D. Estrin, K.F. adn Sally Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, H. Yu, Advances in network simulation, IEEE Comput. 33 (5) (2000) 59–67.
[6] N. Calazans, E. Moreno, F. Hessel, V. Rosa, F. Moraes, E. Carara, From VHDL register transfer level to SystemC transaction level modeling: a comparative case study, in: Proceedings of the 16th International Symposium on Intergrated Circuits and Systems Design, 2003.
[7] T. Cegrell, A simulation model of the TIDAS computer network, IEEE Trans. Commun. 24 (3) (1976) 355–358.
[8] F.M. Chiussi, A. Francini, Scalable electronic packet switches, IEEE J. Selected Areas Commun. 21 (4) (2003) 486–500.
[9] J. Duato, S. Yalamanchili, L. Ni, Interconnection Networks: an Engineering Approach, Morgan Kaufmann, Los Altos, CA, 2003.
[10] A. Faraj, X. Yuan, Communication characteristics in the NAS parallel benchmarks, in: Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002), 2002, pp. 729–734.
[11] R. Foschia, T. Rauber, G. Runger, Modeling the communication behavior of the intel paragon, in: Proceedings Fifth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 1997.
[12] A. Ganz, Y. Gao, Tdma communication for ss/tdma satellites with optical inter-satellite links, in: IEEE International Conference on Commutations, vol. 3, 1990, pp. 1081–1085.
[13] A.D. George, R.B. Fogarty, J.S. Markwell, M.D. Miars, An integrated simulation environment for parallel and distributed system prototyping, Simulation 72 (5) (1999) 283–294.
[14] I. Gorton, J. Kerridge, B. Jervis, Simulating microprocessor systems using occam and a network of transporter, in: IEE Proceedings on Computers and Digital Techniques, vol. 136, 1989, pp. 22–28.
[15] R. Hoare, S. Tung, B. Farren, Z. Ding, Incorporation of physical layer characteristics into system level modeling of large digital systems, in: Proceedings of the International Conference on Applied Modeling and Simulation, IASTED2002, 2002.
[16] S. Karlsson, M. Brorsson, A comparative characterization of communication patterns in applications using MPI and shared memory on an IBM SP2, in: Proceedings of the Second International Workshop on Network-Based Parallel Computing, 1998, pp. 189–201.
[17] A. Ki, B.-I. Park, J.-G. Lee, C.-M. Kyung, Transaction level modeling of SoC with SystemC 2.0, in: SOC Design Conference, 2003.

[18] Y. Liu, S. Dickey, Simulation and analysis of enhanced switch architecture for interconnection networks in massively parallel shared memory machines, in: Proceedings of Second Symposium on the Frontiers of Massively Parallel Computation, 1988, pp. 487–490.
[19] P. Mars, Some aspects of simulation in telecommunication networks, in: Twelfth UK Tele-traffic Symposium, Performance Engineering in Telecommunications Networks (Digest No. 1995/054), IEE, 1995, pp. 1/1–1/4.
[20] N. McKeown, Scheduling algorithms for input-queued cell switches, Ph.D. Thesis, University of California at Berkeley, 1995.
[21] N. McKeown, M. Izzard, A. Mekkittikul, W. Ellersick, M. Horowitz, Tiny tera: a packet switch core, IEEE Micro 17 (1) (1997) 26–33.
[22] Mentor Graphics, Design exploration tutorial, Tech. rep., Mentor Graphics, 2001.
[23] Mentor Graphics, ModelSim, Tech. rep., Mentor Graphics, 2000.
[24] National Semiconductor Corporation, SCAN90CP02 1.5 Gbps 2 × 2 LVDS crosspoint switch with pre-emphasis sand IEEE 1149.6, Datasheet, February 2004.
[25] National Semiconductor Corporation, SCAN50C400 1.25/2.5/5.0 Gbps quad multi-rate backplane transceiver, Datasheet, January 2004.
[26] L. Natvig, High-level architectural simulation of the torus routing chip, in: Proceedings of the Verilog HDL Conference, 1997.
[27] S.F. Nugent, The iPSC/2 direct-connect communications technology, in: Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications: Architecture, Software, Computer Systems, and General Issues, vol. 1, ACM, New York, 1988, pp. 51–60.
[28] W. Oed, The cray research massively parallel processing system: Cray t3d, Tech. rep., Cray Research, 1993.
[29] OPNET Technologies, Inc., Modeler: Acelerating Network R&D, Whitepaper, 2004.
[30] F. Petrini, W.C. Feng, A. Hoisie, S. Chaudhury, The quadrics network: high performance clustering technology, IEEE Micro 22 (1) (2001) 46–57.
[31] C. Qiao, R. Melhem, Dynamic reconfiguration of optically interconnected networks with time division multiplexing, J. Parallel Distributed Comput. 22 (2) (1994) 268–278.
[32] J. Rexford, W. Feng, J. Dolter, K.G. Shin, PP-MESS-SIM: a flexible and extensible simulator for evaluating multicomputer networks, IEEE Trans. Parallel Distributed Systems 8(1) (1997).
[33] E.C. Russell, Building Simulation Models with Simscript II.5, CACI Products Company, 1999.
[34] M.F. Sakr, S.P. Levitan, D.M. Chiarulli, B.G. Horne, C.L. Giles, Predicting multiprocessor memory access patterns with learning models, in: Proceedings of the Fourteenth International Conference on Machine Learning, 1997, pp. 305–312.
[35] C. Salisbury, R. Melhem, A high speed scheduler/controller for unbuffered banyan networks, in: Proceedings of the IEEE International Conference on Communications, vol. 1, 1998, pp. 645–650.
[36] J.C. Wang, S. Ranka, Scheduling of unstructured communication on the intel iPSC/860, in: Proceedings of the 1994 ACM/IEEE conference on Supercomputing, 1994, pp. 360–369.
[37] K. Wilson, J. Glodoveza, Mentor gaphics unveil powerful synthesis tool to meet requirements of next-generation programmable logic design, Tech. rep., Mentor Graphics, 2002.
[38] Xilinx, Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete data sheet, Tech. rep., Xilinx Inc., 2004.
[39] K.L. Yeung, Efficient time slot assignment algorithms for tdm hierarchical and nonhierarchical switch systems, IEEE Trans. Comput. 49(2) (2001).
[40] X. Yuan, R. Melhem, R. Gupta, Algorithms for supporting compiled communication, IEEE Trans. Parallel and Distributed Systems 14 (2) (2003) 107–118.

Dr. Hoare is an Assistant Professor of Electrical Engineering at the University of Pittsburgh. He received his Bachelor of Engineer degree from the Steven's Institute of Technology in 1991. He obtained the Master's Degree from the University of Maryland and his Ph.D. from the Purdue University, in 1994 and 1999, respectively. Dr. Hoare teaches Hardware Design Methodologies at the graduate level, Computer Organization and Software Engineering.
His research focus is on high-performance parallel architectures. For large parallel systems, his focus is on communication and coordination networks. For systems on a chip, he is focused on parallel processing architectures and design automation for application-specific computing. Dr. Hoare is one of the founders, and is the General Chair for the IEEE Workshop on Massively Parallel Processing.

Zhu Ding is a Ph.D candidate in the Department of Electrical and Computer Engineering, University of Pittsburgh. She received her B.S. and M.S. degree in Electrical Engineering from the Southeast University, Nanjing, China, in 1997 and 2000, respectively. Her research interests include parallel computer architecture, interconnection network, high-speed switching and FPGA design. She is a member of IEEE and IEEE Computer Society.

Shenchih Tung is a Ph.D candidate in the Department of Electrical and Computer Engineering, University of Pittsburgh. He received his B.S. degree in Electrical Engineering from the National Taiwan Ocean University, Taiwan, in 1997. He received his M.S. degree in Telecommunication from the Department of Information Science from the University of Pittsburgh in 2000. His research interests include parallel computing architecture, multiprocessor systems on a chip, networks-on-chip, parallel and distributed computer simulation and FPGA design. He is a member of IEEE and IEEE Computer Society.

Rami Melhem received a B.E. in Electrical Engineering from the Cairo University in 1976, an M.A. degree in Mathematics and an M.S. degree in Computer Science from the University of Pittsburgh in 1981, and a Ph.D. degree in Computer Science from the University of Pittsburgh in 1983. He was an Assistant Professor at the Purdue University prior to joining the faculty of the University of Pittsburgh in 1986, where he is currently a Professor of Computer Science and Electrical Engineering and the Chair of the Computer Science Department.
His research interest include real-time and fault-tolerant systems, optical networks, high-performance computing and parallel computer architectures. Dr. Melhem served on program committees of numerous conferences and workshops. He was on the editorial board of the IEEE Transactions on Computers and the IEEE Transactions on Parallel and Distributed systems. He is serving on the advisory boards of the IEEE technical committees on Computer Architecture. He is the Editor for the Kluwer/Plenum Book Series in Computer Science and is on the editorial board of the Computer Architecture Letters, The International Journal of Embedded Systems and the Journal of Parallel and Distributed Computing. Dr. Melhem is a fellow of IEEE and a member of the ACM.

Alex K. Jones received his B.S. in 1998 in Physics from the College of William and Mary in Williamsburg, Virginia. He received his M.S. and Ph.D. degrees in 2000 and 2002, respectively, in Electrical and Computer Engineering at the Northwestern University. He is currently an Assistant Professor at the University of Pittsburgh in Pittsburgh, Pennsylvania. He was formerly a Research Associate in the Center for Parallel and Distributed Computing and Instructor of Electrical and Computer Engineering at Northwestern University. He is a Walter P. Murphy Fellow of Northwestern University, a distinction he was awarded with twice.
Dr. Jones' research interests include compilation techniques for behavioral synthesis, low-power synthesis, embedded systems, and high-performance computing. He is the author of over 30 publications related to high-performance computing and power-aware design automation including a book chapter in Power Aware Computing (Boston, MA: Kluwer, 2002). He is currently an Associate Editor of the International Journal of Computers and Applications. He is also on the Program Committee of the Parallel and Distributed Computing and Systems Conference and the Microelectronic System Engineering Conference.