Switch Design to Enable Predictive Multiplexed Switching in Multiprocessor Networks

Z. Ding[†], R. Hoare[†], A. Jones[†], D. Li[*], S. Shao[*], S. Tung[†], J. Zheng[*] and R. Melhem[*†]

[†]Dept. of Electrical & Computer Enginnering
University of Pittsburgh
Pittsburgh, PA 15261

[*]Dept. of Computer Science
University of Pittsburgh
Pittsburgh , PA 15261

Abstract

Predictive multiplexed switching is a new approach for building interconnection switches for high performance parallel systems. This approach advocates sacrificing some link bandwidth in return for more efficient network control and simpler connection management. The main idea is to depart from the traditional packet and wormhole switching in favor of row data communication over established communication pipes (connections). The overhead of this circuit switching approach can be justified when established connections are repeatedly used before they are torn down. For this, we use multiplexing to allow multiple connections to share the same resources (links and switches), thus avoiding tearing down connections prematurely. The connection establishment overhead is further reduced by exploring communication locality and predictability in applications that exhibit these properties.

We present the design of an interconnection system which is based on multiplexed switching and which establishes connections either reactively, in response to dynamically generated requests, or proactively, in response to compiler or application directives. A communication prediction component may be supported to reduce the network control overhead in applications that exhibit communication locality and predictability. The design is evaluated using hardware design, synthesis, and cycle-accurate simulation. Comparison with more traditional switching paradigms shows the potential of our predictive multiplexed switching approach.

1. Introduction

Circuit switching and wormhole routing are three dominant switching methods that have been used in parallel computing networks [1]. The Intel iPSC/2 and iPSC/860 use circuit-switched communication; e.g. when a source node and a destination node need to communicate, a dedicated path is established for a message. Wormhole routing has been used in a variety of parallel systems including the Intel Paragon, Cray T3D, IBM Power Parallel SP series, and the Quadrics switch.

Multiprocessor interconnection networks can benefit from both temporal and spatial communication locality just as memory systems exploit locality of references through caches. Temporal locality represents the effect of temporal aggregation of the inter-processor communications [2]. High temporal locality suggests that during any given time period inter-processor communication occurs across a certain number of connections, which can be called a communication working set. This provides the opportunity to reduce communication latency by dynamically grouping and scheduling messages and pre-establishing statically known connections. Spatial locality is determined by the distribution of the connections in the application and determines the size of the working set. It has been shown that each node tends to have a small number of favored destinations for the messages it sends [3, 4]. For example, the NAS parallel benchmark suite exhibits very high spatial locality and therefore contain small working sets [4].

Besides locality, communication patterns in parallel algorithms may exhibit some degree of regularity. This regularity exists due to code artifacts such as loops and implies a certain level of predictability of the communication pattern. By appropriately predicting connection requirements and loading these working set connections into the network before they are actually used, it is possible to reduce contention and increase utilization of the interconnect.

There have been several attempts to take advantage of locality to increase the efficiency of parallel systems. For example, establishing dedicated connections based on application specific communication patterns has been shown to achieve very efficient interconnect utilization in Systems-

on-a-Chip [5] and in clusters [6]. In [7] an analysis framework was developed to exploit locality of communication and in [8] techniques to take advantage of locality for reducing the overhead of communication were examined. Dynamic grouping and pre-scheduling of communication have been shown in [9] to reduce the overall execution time of some parallel programs.

In this paper we present an interconnection system that is intended to enable the exploitation of communication locality and regularity by establishing end-to-end connection pipes on which data is transferred in a DMA manner. This connection oriented paradigm allows proactive circuit establishment when possible to avoid contentions at run time, allows predictive techniques to establish connection before they are needed, and still allow the accommodation of irregular communications (reactively).

Regular and predictable communication patterns are cached by the scheduler. The number of connections that can be simultaneously established in the network is increased, through Time Division Multiplexing (TDM), so that the network can cache the entire communication working set of an application. TDM also allows the flexibility of rapidly changing the size and content of the communication cache in the network to closely track the changes in the working set.

The remainder of this paper is organized as follows: In Section 2, techniques to exploit locality for amortizing the control overhead of connection establishment are discussed. Techniques to utilize connection regularity to predict both statically and dynamically known communication patterns are presented in Section 3. In Section 4, the design of an interconnection system is described. This design allows for the preloading of static communication patterns and the prediction of dynamic communication patterns while efficiently handling irregular and non-local communication patterns. Results from a hardware design, synthesis, and cycle-accurate simulation framework are related in Section 5. Conclusions are discussed in Section 6.

2. Amortizing the control overhead of connection establishment

A dedicated circuit between a source and a destination is very effective for data transmission since it simplifies tremendously the communication protocols. Specifically, no congestion control is needed, no routing or control information has to be included with the data, no intermediate buffering and routing is needed and only end-to-end flow control is required. However, circuit switching is only effective when the cost of establishing the circuits is small compared to the cost of transmitting the data, which is only the case when an established circuit is extensively used before it is torn down.

Circuit switching would be an ideal switching scheme if the capacity of the interconnection network would be large enough to satisfy all the connections requested during execution without any conflict. For instance, if *C* is the set of connections that are used during the execution of a parallel program and the interconnection network can satisfy all the connections in *C* without conflict, then circuit switching will pay the overhead of establishing each connection only once during the execution of the program. Specifically, the overhead will be paid only when a connection is requested for the first time. Moreover, if *C* is known before the program starts execution, then the network can be configured to satisfy the connections in *C* before execution starts, thus removing any run-time overhead to establish the circuits.

Unfortunately, a scalable interconnection network cannot realize all the communication requirements of real applications without conflict. Hence, due to the limited capacity of networks, circuits will have to be repeatedly torn down and re-established during execution. Thus, the resulting large overhead may cause a severe bottleneck at the scheduler which will have to resolve contention among competing requests and establish new circuits at run time.

A possible solution to the problem of limited network capacity is to decompose the set of connections, C, into a number of sets, C_1 ,..., C_k, such that $C = C_1 \cup ... \cup C_k$, and each C_i, $i = 1, ..., k$, can be realized in the network without conflict [10]. Time division multiplexing (TDM) can then be used to realize each set C_i periodically in a separate time slot. In this paper, we call any set of connections that can be realized in the network without conflict, a network configuration set, or simply a configuration. Hence, with TDM, the connections in each configuration set C_i could be realized in the network every *k* time slots.

Although using TDM all the connections in *C* can coexist in the network, each connection gets only *1/k* of the maximum possible connection bandwidth. Hence, it is imperative to keep *k* as small as possible. Exploring communication locality can be very useful in that regard since it implies that only a subset of *C* is being used at any given time during the execution of a program.

Specifically, assume that $W^{(1)}, ..., W^{(p)}$, is a sequence of sets of connections that represents the communication requirements during the execution of the program. That is, the program goes through *p* phases and during the execution of each phase, $j, j = 1, ..., p$, it uses the connections in set $W^{(j)}$. We call each set $W^{(j)}$ a communication working set. Note that the sets $W^{(1)}, ..., W^{(p)}$ are not necessarily disjoint, but $W^{(1)} \cup ... \cup W^{(p)} = C$. During the execution of phase *j*, the multiplexing degree is set to k_j, where the set $W^{(j)}$ can be decomposed into k_j network configurations.

The partitioning of the communication requirements into phases is not unique, but is strongly influenced by the com-

munication locality. For programs with strong spacial communication locality, it should be possible to find a partitioning in which the size of each working set $W^{(j)}$ is small, thus leading to a small multiplexing degree k_j. For programs with strong temporal communication locality, the number of phases, *p*, should be small leading to fewer network reconfigurations during execution. This is a desirable property since network reconfiguration and circuit establishment will be performed at the rate of the change in communication locality, rather than at the rate of communication requests.

In the next section, we will discuss different schemes for identifying communication phases present during program execution. However, it should be clear that there is a tradeoff between the number of phases, *p*, and the size of each working set $W^{(j)}$. At one extreme, we can consider that $p = 1$ and that the entire execution is regarded as a single phase, with $W^{(1)} = C$, and $k_1 = k$. At the other extreme, *p* is considered to be as large as it takes to allow the connections in each working set $W^{(j)}$ to be realizable in the network without conflict. More phases lead to more frequent reconfigurations and thus to larger reconfiguration overhead while larger than necessary multiplexing degree leads to inefficient network utilization. Specifically, if during a phase *j*, the actual communication traffic utilizes only *s* of the k_j multiplexed slots needed to establish the working set, then only s/k_j of the available network bandwidth is utilized.

3. Predictive control of networks

Traditional circuit switching falls naturally into the general framework described in Section 2. Specifically, circuit switching amounts to TDM with a multiplexing degree of one. Hence, each realizable active working set is necessarily a configuration that can be established in the network without conflict. Moreover, the establishment of each new requested circuit represents a change in the active working set. This change may require removing some existing connections even if these connections must be re-established in the near future. As discussed in Section 2, it is crucial for communication efficiency to track and minimize the active working of the running application. TDM allows caching of larger working sets of connections, and provides the ability to change the multiplexing degree as required by the application. In the following, we explore different schemes for identifying, predicting, and tracking communication working sets.

3.1. Compile-time and load-time prediction of working sets

Many parallel applications have regular communication patterns that can be identified either at compile time or at load time after the mapping of the application to processors is determined. In [11], an experimental compiler was developed to determine the communication requirements of programs written in a shared memory language, such as OpenMP. A similar concept was applied in [12] to thread level computations and in [4] to programs that use message passing. In general, it was found that in parallel scientific applications, most inter-processor communications can be determined at compile or load time [13, 14].

Developing parallel programs using message passing gives application developers explicit control over inter-processor communications, while many shared memory parallel languages give the application developer explicit control over the allocation of the address space to memory modules. Moreover, new languages such as StreamIt [15], assume that communication patterns between processes are specified and MPI has facilities called *communicators* for explicitly specifying the communication working set. In order to obtain efficient programs, users usually take advantage of the capability to control communications and memory allocation. Hence, it is reasonable to ask the user to give some directives to the compiler about the active communication working set in different phases of the program, if the user wishes to increase the efficiency of inter-processor communication.

Compiled communication allows the compiler to statically determine and optimize the communication requirements in parallel systems [16]. It has been used in combination with message passing in the iWarp system [17, 18]. In [19], the compiler inserts the commands needed to establish the needed connections in the network before the communication takes place. However, because circuit switching is used, the overhead of establishing the connection turned out to be extremely large. In our work, we use TDM to take advantage of compiled communication for static communication patterns without the significant overhead of circuit switching. A similar TDM approach is proposed in [12] for adaptive System-On-a-Chip.

In this paper we will not elaborate on compiler technology, but we will assume that the compiler can identify the appropriate communication working sets when such an identification is possible [12, 20]. Instead, we will present in Section 4 the design of a communication network which can greatly benefit from compiler identification of the communication patterns.

3.2. Dynamic prediction of the working set

Branch prediction proved to be a very powerful technique for improving the performance of microarchitectures, and many attempts have been made to apply the same concepts to improve communication performance. The idea is

to predict the communication requirement and to establish the corresponding circuits in the network before they are actually needed, thus eliminating circuit establishment overhead. Using the notation of Section 3, the works in [21, 22], for example, attempt to predict the connections in the working set $W^{(j+1)}$ while $W^{(j)}$ is being used. In order for such prediction to be useful, however, the processors should be doing useful computational work while the network is being configured from $W^{(j)}$ to $W^{(j+1)}$.

When TDM is used to increase the size of the set of established connections, the overhead of adding a new connection is incurred only the first time the connection is used. Once established in the network, there is no overhead for reusing the connection. The overhead of establishing a connection when it is used for the first time is similar to the penalty for compulsory misses in caches; if the right cache size is used, then a cache miss occurs only on the first reference to a memory location, while successive references to the same location are all hits. In order to keep the multiplexing degree small, however, a connection which will no longer be used should be removed from the working set. Going back to the cache analogy, trying to keep the multiplexing degrees small is similar to allowing the cache size to decrease by evicting cache lines before they have to be replaced with other cache lines.

Hence, instead of trying to predict when to add a new connection to the working set, the role of dynamic predictions in our network will be to predict when to remove a connection from the working set. The purpose of this paper is not to compare the effectiveness of different predictors but to present a network architecture that will allow such prediction. For this reason, we will use in our experiments a simple "time-out" predictor in which a connection is removed if it is not used for a certain period of time. A different predictor can be implemented by associating a counter with each connection in the working set. This counter is reset to zero every time that connection is used and is incremented every time another connection is used. When the counter reaches a certain threshold, the connection is evicted from the network. In other word, a connection is evicted if it is not used while other connections are being used, but is not evicted if the application is in a computation phase, where no communication takes place.

3.3. Dynamic reconfiguration with compiler assistance

High-level knowledge of the program's structure is useful to dynamic prediction discussed in Section 3.2. This information can either be provided by the user as directives or in many cases discovered by a compiler. For example, consider a compiler that detects different communication patterns between two consecutive loop structures. Even if the compiler cannot detect the patterns themselves, it can insert an instruction in the code that flushes all current connections in the network between the two loops. Thus, when the second loop executes it will not mis-predict the pattern based on the previous loop, but rather build a new working set immediately. This idea has been verified by the work in [20]. Other points that may indicate changes in communication localities include procedure boundaries, "if" statements, and points of remapping tasks to processors for load balancing.

The compiler can assist a dynamic reconfiguration strategy considerably in more subtle ways. The compiler might be able to statically determine a portion of the working set, allowing the dynamic reconfiguration strategy to only work on non-predicted communications. For example, consider the case where a loop contains an embedded "if" statement. The communication pattern for the loop may now depend on the condition of "if" statement. The predictor's knowledge of the conditional can significantly simplify the communication pattern detection. One way this could be used is to store a second level working set that is swapped in only when the conditional is true.

If the compiler can predict only a portion of the communication operations statically, the predicted configurations can be preloaded to the network, while the scheduler can continue to schedule dynamically requested connections that are not preloaded. There are two ways to accomplish this in TDM networks: (1) by increasing the multiplexing degree or (2) by temporarily preempting the preloaded connections that conflict with the dynamic connections, and re-establishing the preloaded connections after the dynamic connections are no longer needed.

4. An example switch design

In this section, we present an example design for an *NxN* interconnection system that allows both compiled communication and dynamic prediction, and yet can be used for applications where communication is neither predictable nor regular. Figure 1 shows the block diagram of such a system. The switching fabric in the system is a passive fabric with no buffering or control capabilities. The fabric can represent a crossbar interconnection, a multistage fabric, a fat tree organization, or any other direct interconnection topology.

The configuration of the fabric is determined by configuration registers. By loading specific values into the registers, specific mappings between the input ports and the output ports are realized. In its simplest forms, a configuration, *C*, may be represented by a Boolean matrix, B, where $B_{u,v}$ is 1 when input *u* is connected to output *v*, and $B_{u,v}$ is 0, otherwise. For the case of a crossbar fabric, the only constraints on B are that there is at most one non-zero entry in

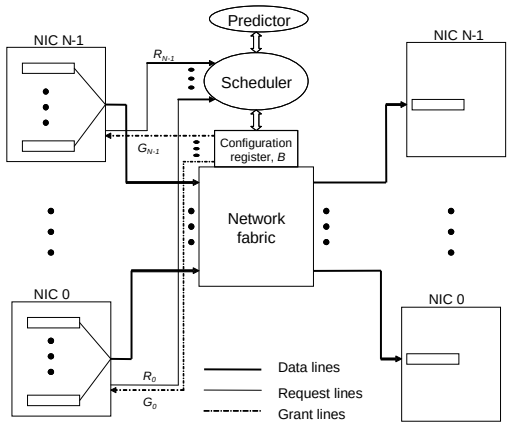


Figure 1. The components of the switching system.

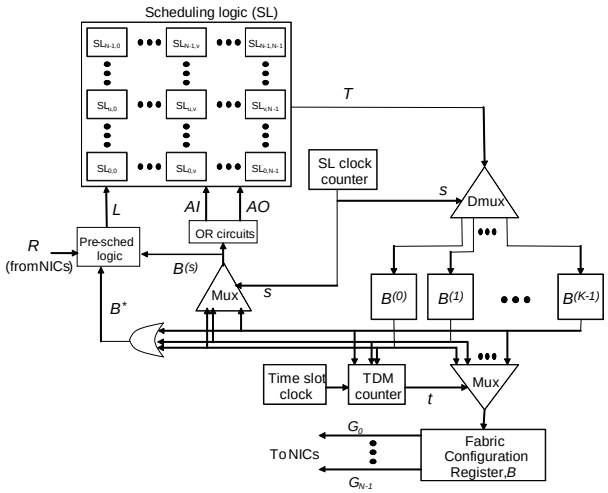


Figure 2. A detailed diagram of the scheduler.

each row and at most one non-zero entry in each column. More complicated constraints may be derived for fabrics that have limited permutation capabilities (e.g. multistage networks) or multi-paths from inputs to outputs (e.g. fat tree fabrics). In the remainder of this paper, we will present a detailed design for a system based on a crossbar fabric.

The network interface card (NIC) for each processor in the system contains an input buffer and an output buffer. The output buffer is used to implement *N* logical queues, one for each destination. For clarity, we show the input and the output side of each NIC separately in Figure 1, and we separate the *N* logical queues of the output buffer.

The scheduler receives a request, $R_u, 0 \leq u \leq N-1$, from each of the *N* NICs indicating which of the logical queues of that NIC is not empty. Hence, each R_u is an *N*-bit signal, $R_{u,0}, ..., R_{u,N-1}$, transmitting to the controller the communication requirements of NIC$_u$. The controller receives requests for connections from all the NICs (a matrix *R*), schedules the connections and communicates an *N*-bit grant signal, G_u , to each NIC$_u$. The scheduler sets the v^{th} signal of G_u, $G_{u,v}$, to 1 whenever a circuit is established between the output port of NIC$_u$ and the input port of NIC$_v$. At most one of $G_{u,v}, v = 0, ..., N-1$ can be non-zero at any given time. Note that the grant signals $G_0, ..., G_{N-1}$, are the rows of the configuration matrix B.

In order to support a multiplexing degree of *K*, the scheduler has to create *K* configuration matrices, $B^{(0)}, ..., B^{(K-1)}$, one for each of the *K* multiplexed time slots. The scheduler satisfies requests from NICs in any of the *K* slots. However, in any given time slot, *t*, only the corresponding matrix $B^{(t)}$ is copied to the fabric configuration register and the corresponding grant signals are sent to the NICs. Figure 2 shows a more detailed diagram of the scheduler, in which a time slot clock controls the copying of the configuration matrix to the switch fabric. The TDM counter shown in the figure is a counter which counts from 0 to *K*-1, but which skips a particular count *t*, if the corresponding matrix $B^{(t)}$ is all zeros. This feature skips over empty configurations and allows the scheduler to reduce the multiplexing degrees by controlling the content of the configuration register.

Note that in the above design, we provide an explicit grant signal from the scheduler to the NICs, thus giving the scheduler the responsibility of controlling the synchronization among the NICs. Specifically, the grant signals indicate to each NIC the period in which it can send data, thus removing the need for the NICs to keep track of the TDM slot boundaries. However, a *guard band* should be enforced between consecutive time slots. During that band, circuits should not be used due to uncertainties in the fabric state. The length of the guard band depends on the variations of the propagation delays of the grant signals and on the time needed to change the setting of the switch fabric. For example, when 1 μs time slots are used, if the time to reconfigure the switch fabric is within 50 ns and the maximum length of a grant line is 50 feet (50 ns propagation delay), then the length of the guard band is 50 ns, which means that 5% of each time slot cannot be used for data transfer. Note that during a 1 μs slot, 125 bytes of data can be transmitted per serial Gb/s link.

The block designated "scheduling logic" in Figure 2 is responsible for generating the schedule for a particular time slot, *s*. The SL counter selects the time slot, *s*, $0 \leq s \leq K$, to which it will try to insert the pending requests. Assuming that the current multiplexing degree is k, $k \leq K$, a simple scheme to select s is to apply a round robin rotation among

$R_{u,v}$	$B^*_{u,v}$	$B^{(s)}_{u,v}$	*Description of the case*	$L_{u,v}$
0	x	0	Connection not requested and not realized in slot *s*	0
0	x	1	Connection not requested and realized in slot *s* (should release)	1
1	1	x	Connection requested and realized in some slot	0
1	0	0	Connection requested and not realized in any slot (should establish)	1

Table 1. The possible inputs to the pre-scheduling logic

$L_{u,v}$	$A_{u,v}$	$D_{u,v}$	*Action*	$T_{u,v}$	$A_{u+1,v}$	$D_{u,v+1}$
0	x	x	No change in connection	0	$A_{u,v}$	$D_{u,v}$
1	1	1	Release the connection in slot *s*	1($B^{(s)}_{u,v}1 -> 0$)	0	0
1	1	0	Need connection but resources not available	0	$A_{u,v}$	$D_{u,v}$
1	0	1	Need connection but resources not available	0	$A_{u,v}$	$D_{u,v}$
1	0	0	Establish connection in slot *s*	1($B^{(s)}_{u,v}1 -> 0$)	1	1

Table 2. The function of a scheduling logic module, SL$_{u,v}$

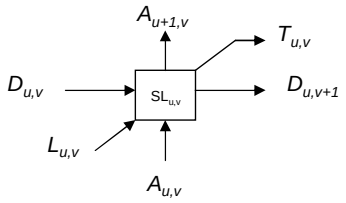


Figure 3. The inputs and outputs to SL$_{u,v}$.

the *k* currently active configurations. The current configuration matrix for slot *s*, $B^{(s)}$, is selected by a multiplexer and fed to a pre-scheduling logic, along with the request matrix R and a matrix B^* which is set to $B^{(0)} + ... + B^{(K-1)}$, where + is the bit-wise *or* operation. The matrix B^* represents all the connections that are currently established in the network (in any of the *K* time slot). Specifically, $B^*_{u,v} = 1$ if and only if the connection from port *u* to port *v* is established during any one of the *K* time slots. By comparing $B^*_{u,v}$, $B^s_{u,v}$ and $R_{u,v}$, the scheduler can figure out whether it needs to make any change to the value of $B^s_{u,v}$ (the state of the connection from *u* to *v* in slot *s*).

In Table 1, we describe the possible cases that the pre-scheduling logic has to deal with. The value of $L_{u,v}$ shown in the last column of the table is generated to be equal to 0 if no change is to be made in the value of $B^s_{u,v}$. The value of $L_{u,v}$ is equal to 1 either if a connection is to be released or if a connection is to be established. In order to release and establish connections, we need to keep track of resource availability. For crossbar switch fabrics, resources are output and input ports. Hence, two vectors AO and AI are used to express the availability of ports in the current schedule of slot *s*. Specifically, AO is obtained by taking the "or" of the columns of $B^{(s)}$. That is, $AO_v = B^{(s)}_{0,v} + ... + B^{(s)}_{N-1,v}$, which is equal to 0 if and only if output port *v* is unscheduled in slot *s* (no input is connected to output *v*). Similarly, the vector AI is obtained by taking the "or" of the rows of $B^{(s)}$. That is, $AI_u = B^{(s)}_{u,0} + ... + B^{(s)}_{u,N-1}$ which is equal to 0 if and only if input port *u* is unscheduled in slot *s* (input u is not connected to any output).

The scheduling logic for a crossbar switch is composed of an *NxN* array of identical modules, SL$_{u,v}$, $u = 0, ..., N-1$, $v = 0, ..., N-1$. Each SL$_{u,v}$ receives the signal $L_{u,v}$ and is responsible for scheduling or releasing the connection from input port *u* to output port *v*. Two sets of port availability signals propagate in the SL array to carry information about the availability of input and output ports in slot *s*. One set of signals, $A_{u,v}$, propagates upwards through rows $0, ..., N-1$ of the array and is initialized such that $A_{0,v} = AO_v$ for $v = 0, ..., N-1$. The other set of signals, $D_{u,v}$, propagates rightwards through columns $0, ..., N-1$ and is initialized such that $D_{u,0} = AI_u$ for $u = 0, ..., N-1$. At any given scheduling module, SL$_{u,v}$, the input $A_{u,v}$ is equal to 0 if and only if output port *v* is available (not occupied) and $D_{u,v}$ is equal to 0 if and only if input port *u* is available (not occupied). Each SL$_{u,v}$ passes $A_{u,v}$ upward (as $A_{u+1,v}$) and $D_{u,v}$ rightward (as $D_{u,v+1}$) unchanged if $L_{u,v} = 0$. However, if $L_{u,v} = 1$, then SL$_{u,v}$ sets $A_{u+1,v} = D_{u,v+1} = 0$ if it is releasing the connection between ports *u* and *v*, or sets $A_{u+1,v} = D_{u,v+1} = 1$ if it is using input ports *u* and output port *v* to establish a connection.

Figure 3 and Table 2 describe the way the availability signals propagate in the scheduling array. Table 2 also specifies the output signal $T_{u,v}$ generated for each input combi-

nation. An output $T_{u,v} = 0$ means that the value of $B^{(s)}_{u,v}$ should not be changed, while an output $T_{u,v} = 1$ means that the value of $B^{(s)}_{u,v}$ should be toggled. Note that by initializing $A_{0,v} = AO_v$, $v = 0, ..., N-1$, and $D_{u,0} = AI_u$, $u = 0, ..., N-1$, we make the unused network ports available to a request $R_{u,v}$ before they are available to another request $R_{a,b}$ if $u < a$ or $v < b$, thus always giving a higher priority to the former. A more fair schedule can be obtained by rotating the priority such that $A_{a,v} = AO_v$, $v = 0, ..., N-1$, and $D_{u,b} = AI_u$, $u = 0, ..., N-1$, where *a* and *b* are selected randomly or through a round robin scheme.

The output of the scheduling logic, *T* ($T_{u,v}$, $u, v = 0, ..., N-1$) is then used to update the configuration matrix $B^{(s)}$ (see Figure 2), thus completing the scheduling process for slot *s* in one SL clock cycle. Note that the period of the SL clock depends on the propagation delay in the scheduling logic and is independent of the period of the time slot clock. In other words, the scheduling for a particular slot, *s*, is performed while the switch fabric is configured according to the configuration for a possibly different time slot *t*.

Because of the time needed for the signals $A_{u,v}$ and $D_{u,v}$ to propagate in the SL array, the scheduling delay should be linearly proportional to the system size, *N*. We have synthesized the scheduler circuit on an Altera Stratix FPGA (EP1S25F1020C-5), and the latency of the resulting circuit is shown in Table 3 for different system sizes. ASIC results tend to be 5 to 10 times better than the FPGA results. In the simulation described in Section 5, we conservatively chose the ASIC performance to be 80 ns for a 128x128 scheduler (about 5x better).

The system described above can be extended to improve the scheduling efficiency and to support the different communication paradigms described in Section 3. For instance,

1. It is possible to use two or more copies of the "scheduling logic" to simultaneously schedule requests on different time slots. The requests can be partitioned among the scheduling logic units or pipelined through them.

2. It is possible to add the capability of inserting a connection in more than one time slot, thus increasing the bandwidth available to that connection.

3. By adding appropriate circuitry to the scheduler, it is possible to keep a connection in the network even if the NIC drops the request signal for that connection. This can be accomplished by adding latches in the paths of the request signals and is useful if it is determined that a connection may be used in the near future. The latches may be explicitly cleared by the NICs or can be cleared to release connections that have not been used for a certain time-out period.

System size	4	8	16	32	64	128
Latency (ns)	34	49	76	120	213	385

Table 3. Latency of the scheduling circuit

4. To support the communication paradigm described in Section 3.3, the request signal from a NIC may be augmented to add the capability of requesting the scheduler to flush all the requests currently established in the network.

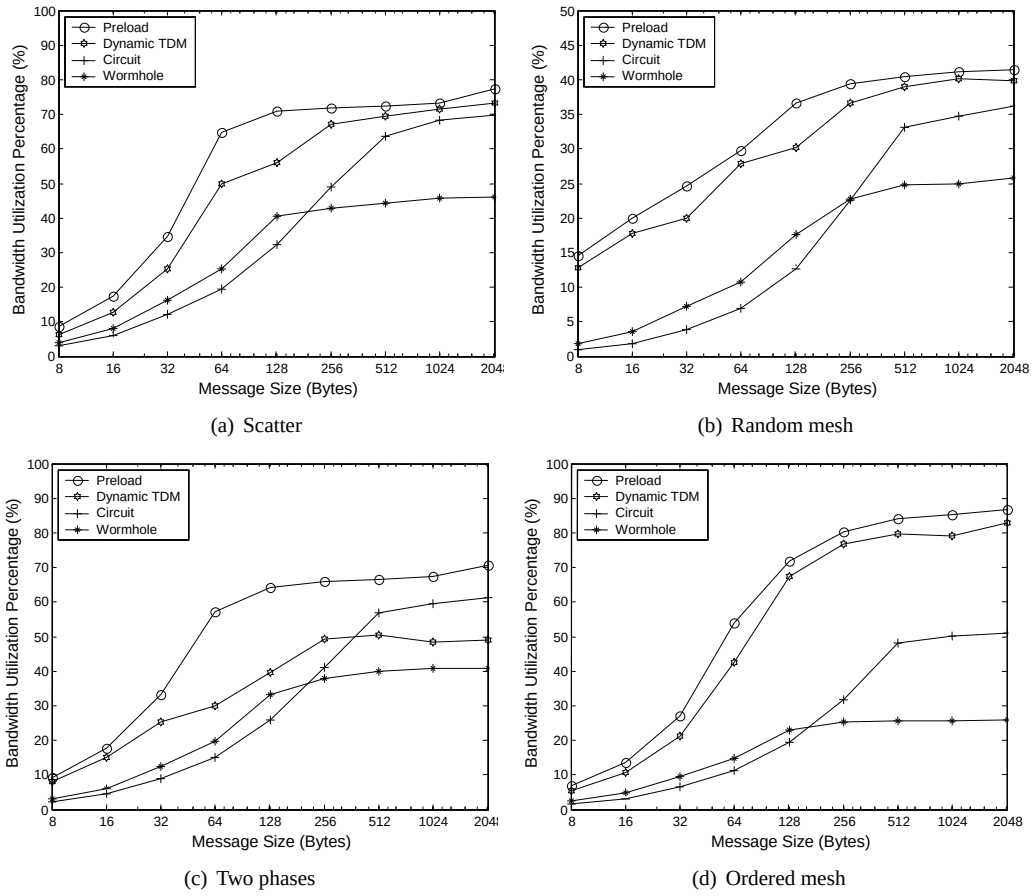
5. To support compiled communications, the request signal may be augmented to transmit to the scheduler specific pre-defined configurations to load onto (or evict from) specific configuration registers. After loading predefined configurations to some registers, the scheduler can still accept dynamic requests for connections from NICs and either satisfy these new requests in other time slots, or temporarily replace some of the connections in the predefined configurations with the new connections, and restoring them after the dynamic connections are no longer used.

As briefly described above, the flushing or establishing of new connections may be requested by the NICs (as a result of commands inserted in the code by the compiler). Alternatively, a separate logic may be added to the scheduler (the component labeled "Predictor" in Figure 1 to observe the status of the request queues in the NICs and the state of the network as determined by the configuration registers. That component may then make decisions about the multiplexing degree to be used, about which connection to keep after the request for that connection is dropped and which connection to evict from the current configuration registers. We are currently designing and evaluating the effectiveness of different types of predictors.

5. System evaluation

In order to validate our theory, we developed a multiprocessor system using hardware design methodologies including synthesis and cycle-accurate simulation. We designed all of the hardware components in VHDL, and synthesized them into FPGA gates to validate our hardware design and to guide our simulations. We then used these quantities to determine the clock frequency that we expected to achieve in an ASIC by conservatively assuming 5x performance improvement over the FPGA.

For our simulations, we created a multi-processor model that contains a single crossbar for communications and a single scheduler for arbitration. Other interconnection fabrics are possible but this represents a baseline topology. We have simulated a 128 processor system that supports wormhole routing, circuit switching, and multiplexing of



(a) Scatter

(b) Random mesh

(c) Two phases

(d) Ordered mesh

Figure 4. Performance results for scatter, random mesh, ordered mesh, and two phases. The Preload and Dynamic TDM utilize a multiplexing degree of four. Ordered and random mesh represents nearest neighbor communications for a 2D mesh.

the communication pattern with dynamic scheduling and preloading a set of communication patterns. Predictive communications utilize the ability to preload the communication pattern into the network. When prediction is not possible, or in cases of misprediction, dynamic scheduling can be employed.

Each of the 128 processors is modeled as a packet generator/receiver and contains a command file that defines the type and sequence of communications that occur. The network interface card/controller (NIC) was designed using synthesizable VHDL and requires a single-cycle delay of 10 ns to send or receive data. This performance is highly optimized but represents real hardware that has been synthesized. The wires in the network assume 10 foot cables using high-speed serial links operating at 6.4 Gb/s. The latency is modeled as a 30 ns delay for parallel-to-serial conversion, 20 ns for propagation delay down a ten foot wire and 30 ns for serial-to-parallel conversion.

For all networks, a 128x128 crossbar fabric is used. For the wormhole routed switch, the crossbar is digital but for the other networks, the crossbar is a Low-Voltage Differential Signal (LVDS) or optical switch. The propagation delay through the digital switch is modeled as 10 ns while the propagation delay through the LVDS/optical switch is neglected as it requires less than 2 ns (equivalent to a 1 foot cable) [23]. Additionally, the overhead of converting between serial and parallel signals at the switch is not required. A 80 ns scheduler is used for all network types, as described in Section 4.

For a wormhole message, the delay through the switch includes the time required to schedule the first flit of the message, which is 80 ns. All subsequent flits in the same worms are routed in 10 ns. In order to ensure fairness within the network, worm sizes are limited and in our simulation we set this limit to 128 bytes. The flit size is 8 bytes. It should be noted that if a message is broken up into two

worms, the cable delay is only seen once as the second worm is buffered within the crossbar switch.

For circuit switching, however, the delay to schedule a message includes the cable delay of 80 ns to send the request, 80 ns to schedule the request, and another 80 ns to send the grant back to the NIC. After that, the point-to-point delay is 30+20+20+30 ns.

We ran four test patterns using message sizes from 8 to 2048 bytes: Scatter, Random Mesh, Ordered Mesh, Two Phase, and Hybrid. These patterns were selected based on a study of the NAS benchmarks that contain many statically known communication operations that do not require run-time prediction. The remaining communication operations in the NAS benchmarks can be easily predicted by simple hardware predictors.

The Scatter test sends a unique message from a single processor to all 128 processors. Random Mesh represents nearest neighbor communications in a 2D mesh but without any predictability while Ordered Mesh represents an ordered nearest neighbor communication pattern. The Two Phase test represents those programs that contain global communication and local communication. In this test, there is one 128-processor all-to-all communication followed by 16 random nearest neighbor communications.

The simulation results are shown in Figure 4. For the Scatter test pattern, there is a notable increase in bandwidth utilization between 32 and 64 bytes. This is due to the fixed duration of each of the communication cycles. Each cycle is fixed at 100 ns or 80 bytes. Messages between 8 and 64 bytes can be transmitted in a single cycle. Messages over 80 bytes are fragmented into multiple cycles and must remain idle when its communication cycle is not active. This is why the efficiency flattens out form 64 to 2048 bytes.

For Preload versus Dynamic TDM, it can be seen that the Scatter performance is very similar. For Random Mesh, both Preload and Dynamic TDM outperform Wormhole and Circuit switching by 10 to 25% but are within 10% of each other. The performance of Circuit switching improves when the message size is large. The Ordered Mesh pattern represents communications that are highly predictable. In our experiments, 4 destinations were used and thus, there was still a relatively high hit-rate for dynamic scheduling of TDM. The Ordered Mesh, as one would expect does very well with Preload. The regularity of the pattern also shows good efficiency for TDM but is not exploited for Wormhole or Circuit switching. For a larger number of destinations, the efficiency of dynamically scheduling TDM is expected to decrease.

For the Two Phased communication test, Preload does better than the rest and the performance of dynamically scheduled TDM drops below Wormhole. This is due to the fairly small set of destinations in the Random Mesh phase and due to the highly structured nature of the All-to-All phase. For the Random Mesh phase, we have shown that both Preload and Dynamic TDM do well but the All-to-All pattern is only exploited by preloading.

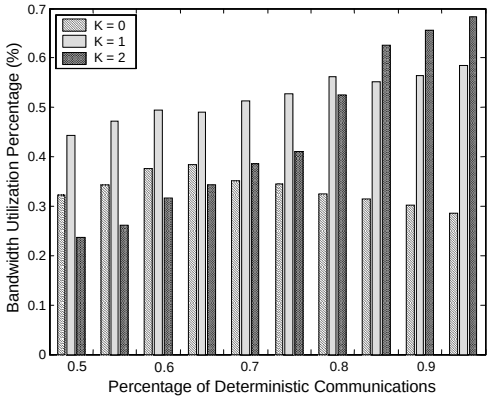


Figure 5. Combining preload of communication patterns with dynamic scheduling. A multiplexing degree of three was used, with *k* slots preloaded. *k* is varied from 0 to 2.

We also simulated the capability of the switch to deal with dynamic communications while preloading the statically known communication patterns. For this experiment, a percentage of the communications are to specific processors and the remaining are randomly sent to any processor. We select a multiplexing degree and we use k slots to preload the static patterns, while the other 3-k slots are use to schedule dynamic communication. We changed k between 0 and 2, and the results are shown in Figure 5. The 1-preload/2-dynamic outperforms the pure dynamic scheme even for low determinism (50%). For 85% or greater determinism, the 2-preload/1-dynamic scheme performed over 10% better than the 1-preload/2-dynamic. This supports the notion of predictive communications as a hit-rate of 85% or better shows dramatic improvement in efficiency.

6. Conclusions

We have presented a design for a flexible multiprocessor switching system that supports dynamic circuit establishment, compiler directed communication and a hybrid mode which allows the compiler or the user to provide useful hints to the connection scheduler. The design relies on an adaptive time division multiplexing scheme to allow the interconnection network to support the communication working set of an application at any given time during execution. The design also allows for different prediction mechanisms for eliminating unused connections from the communication working set and for pre-loading connections into the working set before they are actually needed. Finally, the de-

sign efficiently supports dynamic communications that are not regular and/or cannot be predicted.

We presented the design for a crossbar switching fabric and compared the performance of the connection-based switching design against a wormhole routing system. The advantages of our approach are expected to be amplified when multi-hop networks are considered since it avoids buffering at intermediate switches. This may be particularly efficient if we use LVDS-based switching where signals are not converted from the differential domain to the digital domain at the switches [23]. Moreover, the connection-oriented approach may be the only switching alternative in systems where buffering signals at intermediate switches is not an option, such as in all-optical switching systems [24]

We showed the viability of using multiplexed switching to support both dynamic communication as well as compiled communication. We are currently designing and evaluating predictive schemes and scheduling algorithms to reduce the circuit establishment overhead while keeping the multiplexing degree to a minimum. We are also working on extending the design to switching fabrics other than crossbars. Finally, we are evaluating the tightly coupled collaborations between the compiler, the communication predictor, and the dynamic connection scheduler.

References

[1] J. Duato, S. Yalamanchili, and L. Ni, *Interconnection Networks An Engineering Approach*. Morgan Kaufmann, 2003. Revised Printing.

[2] C. Salisbury and R. Melhem, "A high speed scheduler/controller for unbuffered Banyan networks," *Computer Communications Journal*, vol. 24, no. 9, pp. 1158–1169, 2001.

[3] J. Kim and D. J. Lilja, "Characterization of communication patterns in message-passing parallel scientific application program," in *Proc. of the Second International Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications* (G. Goos, J. Hartmanis, and J. Leeuwen, eds.), pp. 202–216, 1998.

[4] A. Afsahi, *Design and Evaluation of Communication Latency Hiding/Reduction Techniques for Message-Passing Enviroments*. PhD thesis, University of Victoria, Canada, 2000.

[5] W. Ho and T. Pinkston, "A methodology for designing efficient on-chip interconnects on well-behaved communication patterns," in *Proc. of the 9th Int. Symposium on High-Performance Computer Architecture*, pp. 377–388, 2003.

[6] H. G. Dietz and T. Mattox, "Compiler techniques for Flat Neighborhood Networks," in *Proc. of 13th Int. Workshop on Languages and Compilers for Parallel Computing*, 2000.

[7] K. L. Johnson, "The impact of commucation locality on large-scale multiprocessor performance," in *Proc. of the 19th Annual International Symposium on Computer Architecture*, 1992.

[8] L. Roh and W. A. Najjar, "Analysis of communications and overhead reduction in multithreaded execution," in *Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques*, 1995.

[9] J. Fernandez and E. Frachtenberg, "BCS-MPI: a new approach in the system software design for large-scale parallel computers," in *Proc. of the ACM/IEEE Conf. on Supercomputing*, 2003.

[10] R. Melhem, "Time-multiplexing optical interconnection networks; why does it pay off?," in *Proc. of the ICPP Workshop on Challenges for Parallel Processing*, 1995.

[11] X. Yuan, R. Melhem, and R. Gupta, "Algorithms for supporting compiled communication," *IEEE Trans. on Parallel and Distributed Systems*, vol. 14, no. 2, pp. 107–118, 2003.

[12] J. Liang, S. Swaminathan, and R. Tessier, "aSOC: a scalable, single-chip communications architecture," in *Proc. of the IEEE Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 37–46, Oct. 2000.

[13] A. Faraj and X. Yuan, "Communication characteristics in the NAS parallel benchmarks," in *Proc. 14th IASTED Int. Conf. on Parallel and Distributed Computing and Systems, IPDCS 2002*, (Cambridge, MA), pp. 729–734, November 2002.

[14] D. Lahaut and C. Germain, "Static communications in parallel scientific programs," in *Proc. of PARLE*, 1994.

[15] W. Thies, M. Karczmarek, and S. Amarsinghe, "StreamIt: a language for streaming applications," in *Proc. of the Int. Conf. on Compiler Construction*, 2002.

[16] S. Hinrichs, *Compiler directed architecture-dependent communication optimization*. PhD thesis, Carnegie Mellon University, 1995.

[17] T. Gross, "Communication in iWarp systems," in *Proc. of Supercomputing*, 1989.

[18] T. Gross, A. Hasegawa, S. Hinrichs, D. O'Hallaron, and T. Stricker, "Communication styles for parallel systems," *IEEE Computer*, 1994.

[19] F. Cappello and C. Germain, "Toward high communication performance through compiled communications on a circuit switched interconnection network," in *Proc. of the First IEEE Symposium on High-Performance Computer Architecture*, 1995.

[20] G. Viswanathan and J. Larus, "Compiler-directed shared-memory communication for iterative parallel applications," in *Proc. of the 1996 ACM/IEEE Conf. on Supercomputing*, 1996.

[21] M. F. Sakr, S. P. Levitan, D. M. Chiarulli, B. G. Horne, and C. L. Giles, "Predicting multiprocessor memory access patterns with learning models," in *Proc. of 14th Int. Conf. on Machine Learning*, pp. 305–312, 1997.

[22] S. Kaxiras and C. Young, "Coherence communication prediction in shared-memory multiprocessors," in *Proc. of the 16th Int. High Performance Computer Architecture*, 2000.

[23] National Semiconductor Co., "DS90CP04 4x4 low power 2.5 Gb/s LVDS digital cross-point switch." Data Sheet, Jan. 2004.

[24] C. Qiao and R. Melhem, "Dynamic reconfiguration of optically interconnected networks with time division multiplexing," *the Journal of Parallel and Distributed Computing*, vol. 22, no. 2, pp. 268–278, 1994.