Compiler Techniques for Efficient Communications in Circuit Switched Networks for Multiprocessor Systems

Shuyi Shao, *Student Member, IEEE,* Alex K. Jones, *Member, IEEE,* Rami Melhem, *Fellow, IEEE*

Abstract— **In this paper we explore compiler techniques for achieving efficient communications on circuit switching interconnection networks. We propose a compilation framework for identifying communication patterns and compiling these patterns as network configuration directives. This has the potential of providing significant performance benefits when connections can be established in the network prior to the actual communications. The framework includes a flexible and powerful communication pattern representation scheme that captures the property of communication patterns and allows manipulation of these patterns. In this way, communication phases can be identified within the application. Additionally, we extend the classification of static and dynamic communications to include persistent communications. Persistent communications are a subclass of dynamic communications that remain unchanged for large segments of the application execution. An experimental compiler has been developed to implement the framework. This compiler is capable of detecting both static and persistent communications within an application. We show that for the NAS Parallel Benchmarks, 100% of the point-to-point communications can be classified as either static or persistent and 100% of the collectives are either static or persistent with the exception of IS. Simulation-based performance analysis demonstrates the benefit of using our compiler techniques for achieving efficient communications in multiprocessor systems.**

Index Terms— **Compiled Communication, High Performance Computing, Multiprocessor Systems, MPI, Circuit-switching networks, Communication Patterns.**

I. INTRODUCTION

MANY high performance computing systems use packet-switched networks to interconnect system processors. As systems get larger, a scalable interconnect can assume a disproportionately high portion of the system cost when striving to meet the demands of low-latency and high-bandwidth. Although the quest for cheap, low-latency, high-bandwidth packet-switched interconnections for large scale systems is worthwhile, circuit switching networks [1], [2] have the potential of achieving higher efficiency than packet and wormhole networks with relatively lower cost. However, the overhead of circuit establishment can be relatively large, and the benefits of circuit switching can only outweigh its drawbacks when communication exhibits locality and when this locality is appropriately explored. Thus, using compiler techniques to explore the communication patterns of parallel applications–referred to as compiled communication in [3]–[5]–becomes a promising approach for achieving efficient communications in the high performance computing domain. There are other techniques to analyze communication locality such as trace analysis and to leverage communication locality such as runtime scheduling. However, trace analysis can be mislead with specific data sets and runtime analysis is subject to cold start misses and potentially allows thrashing which can lead to poor performance.

In contrast, our approach discovers the communication patterns between logical nodes based on the structure of the application code and through system calls during execution, provides them to a run-time system that manages circuit switched interconnections. We propose a framework that integrates traditional and new compiler techniques to realize this approach.

The motivation of this work partially stems from the proposal to include an optical circuit switching (OCS) network in the design of next generation high performance computing systems [6]. In that proposal long-lived bulk data transfers are routed through all optical switches, which are characterized by high data rates with high overhead for circuit establishment. An OCS is less expensive than its electronic counterpart as it uses fewer optical transceivers. This interconnection technique is more effective if connections are pre-established and the relatively long switch times are amortized over the lifetime of the connections. For example, Shalf et. al. proposes another interconnection network that use both passive (circuit switching) and active (packet switching) switches to deliver performance equivalent to that of a fully-interconnected network [7]. In their approach, the switches must be reconfigured to emulate a suitable interconnection topology to achieve the best performance for an application. The effectiveness of this topology optimization process heavily depends on the ability to identify the communication pattern of the application. The research in this direction in the high performance computing domain mandates the exploration of new compiler techniques and the development of a system-

Manuscript was initially received December, 2006 first revised November, 2007, and revised again March 2008.

This work is partially supported by NSF award number 0702452, the PERCS project at IBM, Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCH3039004.

atic compiler-based approach to infer the communication topology at compile-time.

Based on the temporal and spatial localities of communications and the capability of the compiler to identify the temporal properties and the topology of the communications, we classify communications during a phase of an application's execution into three categories: *static*, *persistent* and *dynamic*. In this context, the topology of communication is the specification of the source and destination of the messages exchanged.

Static – Communication is static if it can be completely determined through compile-time analysis. That is the compiler can identify both the temporal locality and the exact topology of the communication.

Persistent – Communication is persistent if, though the compiler cannot determine the exact topology of communication, it can determine that the topology does not change during the phase. That is, its temporal locality can be identified by the compiler, but its spatial properties remains unknown until run-time.

Dynamic – Communication is dynamic if it it neither static nor persistent.

Given that communications of applications may change during different phases of execution, an important part of the analysis of communication patterns is to identify and segregate different communication phases. We observe that a main source of communication temporal locality originates from loop structures of parallel programs. Hence, it is natural to consider loops that contain communications as the building blocks of phases.

In Figure 1, we illustrate the definition of static, persistent and dynamic communications when a phase is defined as a loop. Specifically, the communication is static if the topology can be completely resolved at compile-time, as in Figure 1(a). In Figure 1(b), the topology can not be determined until run-time. However, once defined, the topology is repeatedly used within the loop. In this case we call the communications *persistent*. In Figure 1(c), the communication is *dynamic* because during each iteration of the loop, the topology is re-calculated.

The above classification implies different possibilities for reducing communication overhead. For example, considering circuit switching networks with preloading capability such as an OCS network, the earliest opportunity for determining the network configuration for a static communication is at compile-time. Thus, network configuration instructions may be statically inserted into the code by the compiler at phase boundaries to establish connections between logical nodes in the system. These logical nodes are translated into physical nodes by the runtime system. For persistent communication, the topology of the communication is not known at compile-time. However, it is possible to insert at compile-time network configuration instructions containing symbolic expressions specifying the topology that will be resolved at runtime. By placing these network configuration instructions at the earliest point where the expression can be resolved, the network reconfiguration may still be able to take place prior to the actual communication within a phase amortizing as much as possible the network reconfiguration time. Additionally, if process migration between physical processors occurs (e.g. for fault-tolerance), even within a phase, our compilation approach is unaffected as it deals with logical nodes. For a circuit switching target, the runtime system can easily migrate the connections along with the process to the new physical processor.

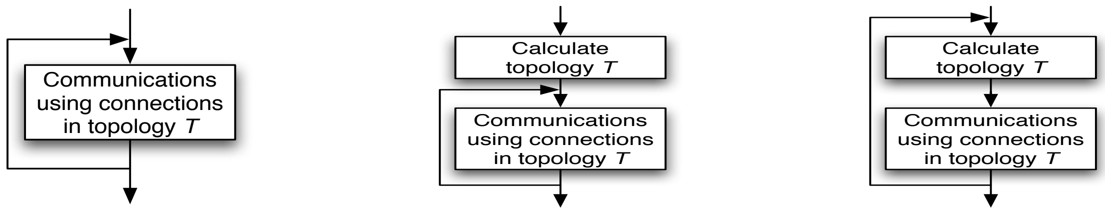
In this paper, we present a compilation framework for identifying and compiling static and persistent communication patterns. Many previous efforts to analyze parallel applications' communications characteristics are based solely on trace analysis [7]–[9]. However, the traces can provide the communication information for only a single execution instance of an application on a particular platform. Our approach is to reveal the underlying communication patterns and make it available at compile-time. We implement the compilation framework by developing an experimental compiler which identifies communication patterns from the source code. It compiles the communication pattern of an application and enhances it with network configuration directives. Another capability of the compiler is to augment the code with trace generation instructions which can produce information about the communication pattern correlated to the structure of the source (e.g. loops, conditionals, etc.).

We use a powerful scheme to represent communication patterns that includes both collective and point-to-point communications in terms of communication vectors and matrices. The vectors and matrices contain exact values if the communication pattern contains only static communications. Otherwise, they may contain symbolic expressions for later resolution. This scheme allows the manipulation of communication patterns through a set of convenient operations. It is also flexible and can be easily tailored to other types of communication analysis.

The remainder of this paper is organized as follows. Section II reviews major related work. We describe our framework in Section III and we describe an experimental compiler which implements the framework in Section IV. A set of results from applying the experimental compiler are presented in Section V. Conclusions are given in Section VI.

II. Background and Context

High performance computing systems are being built out of ever-increasing numbers of processors [10], [11]. These large systems, however, typically use packet or wormhole routing for interprocessor communication with only a few systems built around statically configured circuit switching networks. The most well known example of a circuit switching based network is the NEC Earth Simulator [12], which uses a huge electronic crossbar, with 640x640 ports. The ICN (Interconnection Cache


Communications using connections in topology T
Calculate topology T
Communications using connections in topology T
Calculate topology T
Communications using connections in topology T

(a) Static: T is resolvable at compile-time. (b) Persistent: T is calculated before a phase. (c) Dynamic: T changes inside a phase.

Fig. 1. Static, persistent and dynamic communications when a phase is a loop.

Network) [13], [14] is similar to the Earth Simulator in its one-to-one relationship between each processing node and one channel or circuit that it handles. Circuit switching can also be achieved in parallel systems using passive optical components through time-division multiplexing (TDM) [15], [16], wavelength division multiplexing (WDM) [17] or a combination of the two [18]–[20], which does demonstrate an interest in optical circuit switching in the high performance computing domain.

Circuit switching hardware continues to improve due to improvements in technology. Newer technologies such as optical networking continues to be an alternative to electronic circuit switching that provides several advantages such as capabilities to handle long wire lengths and achieve high bandwidths. The biggest argument for use of optical circuit switching (OCS) is a savings in cost over fast electronic networks as OCS removes the need for expensive transceivers to convert signals between optics and electronics [6]. However, the reconfiguration time of optical switching may be relatively long (ms vs μs) [6], [21], [22]. Hence, implementing circuit switching with relatively long switching latency mandates a technique to amortize connection establishment overhead and to explore the locality of communications in parallel applications. This inspires us to combine compiled communication and circuit switching techniques.

Our previous work [23] shows that much of the circuit switching overhead can be amortized by pre-establishing connections and re-using connections as much as possible. In fact, the network configuration pre-loading scheme has been shown to perform better than traditional wormhole and non-predictive circuit switching techniques in many instances. However, this solution requires that the communication pattern must be known early enough, ideally at compile-time. In cases where the communication operations require more bandwidth than the network can provide, it is necessary to detect the communication pattern and pre-schedule the communication operations in the different phases for circuit switching to be effective [24].

There have been several attempts [8], [9], [25], [26] to understand the communication characteristics of parallel applications. Shires, et. al. presented an algorithm for building a program flow graph representation of an MPI program [25]. They provided an interesting basis for important program analyses useful in software testing, debugging and code optimization. In [9], Vetter and Mueller examined the explicit communication characteristics of several sophisticated scientific applications, while focusing on the Message Passing Interface (MPI) [27] and by using hardware counters on microprocessors. Faraj and Yuan [8] investigated the communication characteristics of MPI implementations of the NAS parallel benchmarks [28]. Ho and Lin studied the static analysis of communication structures in programs written in a channel-based message passing language called communication compiling component [26]. Although these attempts analyzed the communication characteristics of parallel applications such as the ratio of different kinds of communications and implicitly discuss persistence of communication and phases, they do not provide any systematic way to identify accurate communication patterns with respect to connections or specify a quantitative measure of persistence.

Many interesting research projects in the high performance computing domain can take advantage of or rely on information about communication patterns. For example, Cappello and Germain proposed an approach to associate compiled communications and a circuit switched interconnection network [3]. Yuan et. al. explored using compiled communication for HPF-like parallel applications as an alternative to dynamic network control [4]. The compiled communication technique requires that a large portion of static communications be identified at compile-time. Dietz and Mattox studied the Flat Neighborhood Network (FNN) which uses the communication patterns to determine the design of the network [29]. Liang et. al. described a compiler, which supports compile-time scheduled communication for their adaptive System-On-a-Chip (aSOC) communication architecture [30]. As previously described, our previous work introduces a switch design which can use our compilation technique to pre-program a TDM network switch [23]. Each of these techniques can take advantage of compile-time knowledge of the communication pattern to reduce the overhead in the interconnection network.

In contrast to these efforts we provide a systematic technique to address compiled communication in MPI applications and to leverage this information for a circuit switching network.

III. COMPILATION FRAMEWORK

In this section, we present our compilation framework in the context of MPI programs. We also present it in the context of the abstract communication model described next.

A. Model of the Target Systems

We assume a high performance computing system in which computing nodes are interconnected through a circuit switched network. The circuit switching network can support k simultaneous connections to each processing node, loading pre-computed network configurations, and reconfiguration at the boundaries of communication phases. We also assume the system is able to efficiently perform dynamic and collective communications. Based on these assumptions, an abstract model of the target high performance computing incorporates two separate communication networks: a circuit switching network used for static and persistent communications, and a packet switching network to accommodate collectives and short-lived dynamic communications. This interconnect model provides a significant cost advantage over a very, highly-buffered fast packet switch [6]. An overview of the model is shown in Figure 2. Separating the communication classes in this manner enables us to target each class with the most appropriate network technology and operating mechanism. Static and persistent communications will be compiled and dispatched to the circuit switching interconnect network.

B. Compilation Paradigm

Current compiler techniques, such as control and data flow graph (CDFG) analysis, inter-procedural analysis, and array analysis can be used to infer information from the code that is essential for analyzing the communication behavior of an entire program with explicit message passing. Our compiler identifies the MPI functions in the code, and uses these analysis tools to construct a communication pattern for the application. As MPI allows messages to be received from any source, our approach is to profile the message send operations to determine the communication pattern. The representation for a communication pattern is detailed in Section III-C.

The communication behavior of the application is partitioned into *communication phases* with the knowledge of the target network specification. By mapping the communication patterns into a sequences of phases it is possible to create more efficient *communication working sets* or groups of communications that occur in relatively close proximity. The granularity of the communication phases depends on the capacities of the communication network. Thus, the communication pattern identified during analysis according to particular interconnection network specification can be leveraged through network configuration instructions inserted into the application.

For static communication patterns, the compiler generates a network configuration file that can be used by

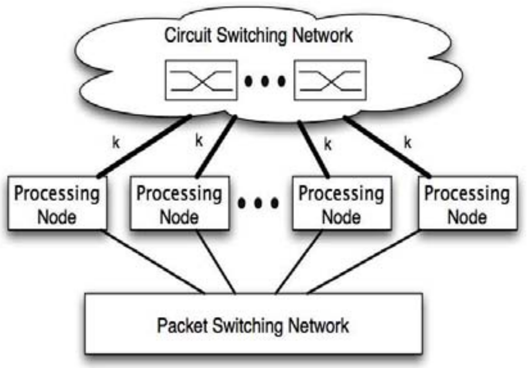


Fig. 2. High performance computing with circuit switching.

the loader and inserts respective network configuration setup instructions in the program. For persistent communication patterns, the compiler inserts symbolic network configuration instructions that aim at pre-establishing the needed connections at run-time prior to the actual communications.

C. Phases and Pattern Representation

Several research groups have observed that the communication operations in many applications exhibit regular patterns [6], [26], [30], [31]. Additionally, it has been shown that these regular communication patterns can often be discovered through analysis of the source code [3], [8], [32]. Our compilation framework is motivated by these two discoveries.

Often, a parallel program is written to solve a particular scientific problem. These applications are often organized in phases and although different processors may take different paths, they tend to work in the same computational phase at approximately the same time but primarily on their local data. Given that these parallel applications have computational phases, we can expect their communication operations to behave similarly. For example, the communication topologies of adaptive applications evolve during their execution time. Even for parallel applications that have static communication patterns, their active communication working set may change as the phases change. The result is one or more *communication phases*. Communication phases are not identical to computational phases, but are strongly associated with them. For example, some computational phases contain no communication and thus can be ignored when identifying communication patterns. Several computational phases may just yield a single communication phase. The number of phases is an artifact of the analysis used to partition the *communications* into *phases*.

To effectively perform compile-time communication analysis it is necessary to represent the communication patterns identified from the code in a form that is both flexible and accurate. In the following sections we describe a communication matrix/vector pair that are used to represent the communication pattern in an application. Additionally, we describe how the communication pattern representations can be manipulated to closely correspond to the underlying communication network.

The fact that the communication pattern of an application contains phases is important and must be described while representing the pattern. However, the traditional technique to represent the communication pattern of an application is to describe its rough logical topology, (e.g. 2-D mesh, hypercube, etc.), or to provide a communication matrix. Such representations are too coarse and can not describe the communication topologies accurately. They also fail to disclose temporal information. Our communication pattern representation scheme is designed specifically to avoid this limitation and to effectively represent the temporal and spatial properties of the pattern.

We define all the communication operations of an application as a *communication pattern*. When the execution is partitioned into phases, the communications within each phase need to be specified. There are two types of communications in MPI applications: collective communications and point-to-point communications. In order to represent the collective communications in a pattern, we define a *c-enumeration* to describe the set of collective communication functions invoked in a parallel application.

Definition A ***c-enumeration*** is a list of all the collective communications that appear in a parallel application. Each collective communication is represented by a pair, the function name and optionally the corresponding communicator. For example, for the function names, we use `AA`, `AV`, `AR`, and `RD` to represent MPI_Alltoall, MPI_Alltoallv, MPI_Allreduce, and MPI_Reduce respectively. The communicator is omitted if it is the default MPI communicator. For each related MPI communicator, the same collective MPI functions have exactly one instance in the *c-enumeration*. Each communication pattern retains a unique *c-enumeration*.

Example 1: `CEE` = {`AA, AR , (AR, commu1), RD`} indicates that there are three different types of collective communications in the application. The first two and the last operations are performed in the default MPI communicator while the third operation, MPI_Allreduce, is performed in a user-defined communicator `commu1`. The communication detection component of the framework is responsible for building *c-enumerations*.

Formally we use the grammar described in Figure 3 to represent the communication pattern that describes the communications in different phases of a program. In the expression for *c-enumeration*, *col*$_i$ represents any collective MPI function, and $comm_i$ represents the corresponding MPI communicator. In this figure, we assume that m is the number of elements in *c-enumeration*.

A communication pattern consists of a sequence of phases. A basic *communication phase* is described by a $c-vector$ and a $p-matrix$ that represent all the collective and point-to-point communications, respectively, in that phase. A phase may also be a loop of a sequence of phases that repeat in any execution instance of an application, represented by square brackets in the grammar.

$$communication\ pattern \rightarrow c-enumeration, phases$$
$$c-enumeration \rightarrow \{(col_0, comm_0), ..., (col_{m-1}, comm_{m-1})\}$$
$$phases \rightarrow \epsilon | phase\ phases | [phases]\ phases$$
$$phase \rightarrow <c-vector, p-matrix>$$
$$c-vector \rightarrow \epsilon | <w_0, w_1, ..., w_{m-1}>$$
$$p-matrix \rightarrow \epsilon | deterministic\ p-matrix | symbolic\ p-matrix$$

Fig. 3. The grammar for communication pattern representations.

Definition A $c-vector$ corresponds to a *c-enumeration*. Each element of the vector represents the weight of the corresponding collective communication in the *c-enumeration*.

Definition A $p-matrix$ is a $N \times N$ matrix that describes a set of point-to-point communications. The entry in position i, j describes the weight of communication from processor i to j.

Definition δ represents any unknown values, variable, vector, or matrix.

In the above definitions of $p-matrices$ and $c-vectors$ we do not enforce a specific meaning for the communication weight. However, some options include (1) a single bit value to indicate if point-to-point communications from the source processor to the destination processor exist (2) the message volume or (3) message count. In the case that the compiler cannot construct even a symbolic expression for a point-to-point communication the δ symbol is used in the symbolic expression for that matrix entry.

A $p-matrix$ is **deterministic** if the total number of processors N is known and each entry of the $p-matrix$ is a constant. A deterministic $p-matrix$ is used to represent static communications. When the size N of a $p-matrix$ is a symbolic constant and/or any entry can only be described by a symbolic expression instead of a constant, it is a **symbolic** $p-matrix$. A symbolic $p-matrix$ can always be described by a formula list. Persistent communications can usually be described by symbolic $p-matrices$.

Example 2: As shown in Figure 4, PM_A and PM_B are a formula list and a deterministic $p-matrix$, respectively. PM_A describes a communication pattern in which each processor $rank$ sends to $rank + x$ and $rank - x$ if $rank - x > 0$ where x is determined at run-time and N is the total number of processors. In the case $x = 1$ and $N = 4$, a deterministic $p-matrix$ PM_B is inferred from PM_A.

Example 3: The communication pattern of IS (integer sorting) program from NAS parallel benchmark suite [28] is shown in Table I and the deterministic $p-matrix$ in phase 2 is shown in Figure 5.

The communication pattern representation scheme can be used to represent both compile-time identified com-

$$\begin{pmatrix} (rank+x) \mod N \\ rank>x : (rank-x) \mod N \end{pmatrix}$$
(a) PM_A

$$\begin{pmatrix} & 1 & & \\ & & 1 & \\ & 1 & & 1 \\ 1 & & 1 & \end{pmatrix}$$
(b) PM_B

Fig. 4. A $p{-}matrix$ PM_A described by a formula list and the corresponding deterministic $p{-}matrix$ PM_B.

TABLE I
THE COMMUNICATION PATTERN OF IS.

$c{-}enumeration$	{AR,AA,AV,RD}	
Phases	$c{-}vector$	$p{-}matrix$
phase 0	$<1,1,1,0>$	NULL
phase 1	$<1,1,1,0>$	NULL
phase 2	$<0,0,0,1>$	PM_IS

munication patterns and the communication patterns identified from execution traces. The main advantage of our pattern representation scheme is that it captures the time evolving, or *phased*, property of communication patterns. The concept of phases suggests the need to manipulate the communication patterns from different phases and to schedule the communications in the pattern at different granularities according to the parameters of the target system.

D. Manipulating Communication Patterns

If the interconnection network capacity is insufficient to establish all the circuits required by an application, then it is necessary to divide the program execution into phases. A communication phase can more precisely represent an active connection *working set*. The interconnection network will be reconfigured at the beginning of each phase to establish the circuits most frequently used in that phase. Clearly, two conflicting criteria guide phase identification. Namely, phases should be small enough to allow the interconnection network to accommodate the communication requirements within the phase, but should be large enough to avoid the overhead of reconfiguring the network. In general, the goal of the phase formation should be to determine the largest communication working set that can fit within the capacity of the circuit switching network and that requires the least reconfiguration during execution.

One strategy to determine the communication phases in the program is to assume that each loop in the application is a phase and to manipulate these initial phase decisions to group the communications into new phases that are best suited to the capacity of the network. For instance, we may want to combine two adjacent phases if the network capacity is large enough to realize both of them; we may want to remove some infrequently used connections if the newly combined phase is slightly beyond the capacity of the network. Communications over these removed connections will be delivered through the packet switching network.

Given a specific communication pattern, the determination of whether or not that pattern fits in a given interconnection network depends on the network itself.

$$\begin{pmatrix} & 1 & & & & & & \\ & & 1 & & & & & \\ & & & 1 & & & & \\ & & & & 1 & & & \\ & & & & & 1 & & \\ & & & & & & 1 & \\ & & & & & & & 1 \\ & & & & & & & \end{pmatrix}$$

Fig. 5. $p{-}matrix$ PM_IS (with 8 processors).

For example, a non-blocking network, such as a crossbar, can realize any pattern that is a permutation, while network-specific algorithms have to be applied to determine if a pattern fits into a blocking network [33]. In our system, we assumed that the interconnection network is composed of a number, k, of parallel crossbars, and thus can realize k permutations simultaneously. In general, if the communication pattern is represented by a deterministic $p-matrix$, then determining if the pattern fits in the network is straight forward. However, for patterns determined by symbolic expressions, that determination depends on the complexity of the expression, and in some cases may not be possible unless the expressions are evaluated.

We define three core operations required to deal with different communication phases. **Merge** combines the $p{-}matrices$ and $c-vectors$ of two adjacent phases of a communication pattern into a new phase. If binary communication weights are used, this is equivalent to an OR operation. **Filter** removes connections below a threshold from a phase of a communication pattern. **Unwrap** is the equivalent to completely unrolling the loop and merging all of the resulting phases into a single phase. This is particularly useful to deal with nested loops or adjacent loops that logically should be in the same phase.

Definition $Merge(phase_i, phase_j)$:
Parameters: $phase_i = <c{-}vector_i, p{-}matrix_i>$, $phase_j = <c{-}vector_j, p{-}matrix_j>$.
$phase_i$ and $phase_j$ are adjacent phases.
Semantics: $phase_i$ and $phase_j$ are merged into a new phase $phase_{new} = <c{-}vector_{new}, p{-}matrix_{new}>$ where $c{-}vector_{new} = c{-}vector_i + c{-}vector_j$ and $p{-}matrix_{new} = p{-}matrix_i + p{-}matrix_j$. The $+$ operation depends on the definition of the weights in the $c{-}vector$ and $p{-}matrix$. For example, if the weights are binary bits in a deterministic $p{-}matrix$, the $+$ operation is equivalent to an OR operation.

Definition $Filter(phase, T)$:
Parameters: $phase = <c{-}vector, p{-}matrix>$, T is a threshold.
Semantics: This operation replaces $phase$ in the communication pattern by a new phase in which any entry with value less than T in $c{-}vector$ or $p{-}matrix$ will be set to 0.

Definition $Unwrap(phase)$:
Parameters: $phase = [<c{-}vector, p{-}matrix>]$.
Assume the loop body of $phase$ has been merged into a single phase "$<c{-}vector, p{-}matrix>$" as above.

Semantics: This operation unwraps the loop indicators of $phase$. If the communication weights are defined as single bit values, this operation uses one "$<c{-}vector, p{-}matrix>$" to replace a loop of "$<c{-}vector, p{-}matrix>$". If the weights are defined as message volume or message counts, the weights in the resulting phases are multiplied by the loop iteration number compared to the weights in "$<c{-}vector, p{-}matrix>$"

E. Phase Partitioning Algorithm

The communication phases within an application typically become apparent at run-time. However, the structure of the source code can often provide enough direction to determine reasonable phase lengths and boundaries. Loops play a key role in the determination of phases in the application [34] and work similarly for communication phases. Thus, we use loops and the code blocks between loops as the starting point to build phases. Adjacent phases whose joined communication pattern do not violate the capability of the network can be merged together to form larger phases.

The control structures arising from branch structures create difficulties in merging adjacent phases. We break down branches into two categories, *rank-dependent* and *data-dependent*. Rank-dependent branches are the most difficult structures to handle as some processors execute each branch, concurrently. Thus, our solution in this case is to merge all phases contained into a single phase and to filter out lowest-bandwidth connections until the resulting pattern can fit into the network. However, in some cases it may be possible to determine optimized patterns for each branch depending on static knowledge of the condition. This is discussed in Section IV.

In data-dependent branches, all processors will take either one direction or the other. This condition holds because only branches containing communication operations are considered. If data dependent conditions allowed different processors to follow different branches containing communication operations, it would be problematic to have matching sends and receives. Thus, for these branches, communication patterns can be merged within branches and individual branches, but need not necessarily be merged across branches as is necessary with rank-dependent conditionals.

For the algorithm we describe several different basic operations used in addition the core operations from Section III-D:

Adjacent(P$_j$, P$_k$) returns true if P$_j$ and P$_k$ are two directly adjacent phases and they are not separated by loop or conditional boundaries and returns false otherwise.

FilterUntilFit(P,NET) removes the lowest weight connections from the communication pattern of phase P until P fits in network NET.

CanBeMerged(P$_j$, P$_k$, T, NET) returns true if the two phases, P$_j$ and P$_k$ can be merged and fit into the network NET without violating the user specified parameter T. T is a threshold on the maximum weight of a connection that can be filtered.

The algorithm is presented in Figure 6. The code starting with line 1 constructs the initial phases of the application by merging the adjacent basic-phases that fit into the network. Starting in line 7, all phases in the rank-dependent branch structures are merged and the communication pattern is filtered until it fits into the network, regardless of the values of T or TM. Starting in line 13 and continuing through the end of the pseudocode, the algorithm revisits merging phases within loops. First, loops that contain a single phase are unwrapped at line 14. Starting in line 15, when possible, data-dependent branch structures are flattened into single phases. Finally, at line 18 adjacent phases are re-examined in case newly unrolled loops or merged branch structures have created phases that can be merged. This continues until the phases reach a steady state creating the final phase partition of the application. Communication instructions can now be placed into the code prior to the execution of each phase.

```
1  Create a basic phase for each MPI communication function call

2  do
3    foreach phase P do
4      foreach phase Q such that Adjacent(P, Q)==1 do
5        if CanBeMerged(P, Q, T, NET) then Merge(P, Q)
6  while(phases can be merged)
7  foreach rank-dependent branch structure BS do
8    foreach branch B_i of branch structure BS do
9      foreach phase P in B_i do
10         foreach phase Q in B_i such that Adjacent(P, Q)==1 then
   Merge(P, Q) into P_i
11     for any two branches B_i, B_j in BS containing phases P_i, P_j,
   respectively, Merge(P_i, P_j) into P_BS
12   FilterUntilFit(P_BS, NET)
13   do
14     foreach loop L if L contains a single phase P_L then
   Unwrap(P_L)
15     foreach data-dependent branch structure BS do
16       for any two branches B_i, B_j in BS such that each contains
   a single phase, P_i, P_j, respectively do
17         if CanBeMerged(P_i, P_j, T, NET) then Merge(P_i, P_j)
18     foreach phase P do
19       foreach phase Q such that Adjacent(P, Q)==1 do
20         if CanBeMerged(P, Q, T, NET) then Merge(P, Q)
21   while (phases can be merged)
```

Fig. 6. Pseudocode for the phase partitioning algorithm.

IV. Implementation of an Experimental Compiler

An experimental compiler has been developed to implement the compilation framework described in Section III for MPI applications written in C or Fortran 77.

A. Compiler Techniques

Certain traditional and new compiler techniques are needed for building a compiler that supports compiled communication. These techniques need to accomplish the following main tasks.

Determination of static communication patterns: The compiler techniques needed to determine the communication pattern are different from traditional compiler analysis and optimization techniques in many aspects. For instance, optimizing compilers employ constant propagation and folding in an effort to reduce and remove unnecessary instructions. In the context of compiled communication, constant propagation and folding is important for detecting the actual value of source, destination, and message volume variables in communication operations. Thus, expressions that contain variables related to the processor id and system size (e.g. number of processors), while not constant at compile-time, are considered constant because they are resolved prior to program execution. For example, the two send operations in the code from Figure 7(a) contain the expressions `(myrank+k)%nprocs`, `(myrank+2*k)%nprocs`, `(myrank-k)%nprocs`, and `(myrank-2*k)%nprocs)`. Because `myrank` and `nprocs` are actually the processor id and number of processors in the system, respectively, we consider these operations to be static if `k` is a constant. We call this symbolic expression propagation. These communication patterns can be resolved into actual numeric values at load-time when the number of processors are known.

Addressing SPMD style: MPI programs are written in SPMD style. Each processor independently executes the same program on its private data. Nevertheless, often different processors take different execution paths as occurs with *rank-dependent* branch structures. Hence the compiler may use control and data flow analysis to segregate communication in the same communication matrix with *decision points* (DPs) that correlate to these branch structures. For static communication, DPs can be used to reduce the size of communication matrices. For persistent communication, DPs are used as predicate conditions for communication represented by symbolic expressions. For example, a naive approach to building a communication matrix based on the code from Figure 7(a), where `k=1` is shown in Figure 7(b). This is built from including the send operation for the *then* and *else* part of the code for each possible processor id. However, the variable guarding the branching statement may be static. For example, in Figure 7(a) the condition `myrank < nprocs/2` is entirely based on constants or variables related to the processor id and system size. Thus, this expression is static. As a result, only two send operations are required to be added to the matrix for each processor id resulting in the matrix shown in Figure 7(c). Hence, careful analysis of the branching statement is required in order to accurately identify the communication patterns. As all phases in *rank-dependent* branches are merged in line 7 of Figure 6, this can help keep useful connections from being filtered out of the network.

Inter-procedural Analysis: An additional consideration of our compiler is the inclusion of inter-procedural analysis. Such a cross-boundary analysis requires sophisticated control flow techniques and again is different

```
void p(int k) {
int i;
reconfigNetwork(config1);
for(i=0; i<1000; i++) {
  if (myrank < nprocs/2) {
    MPI_Send(buf,1, MPI_INT_TYPE,
      (myrank+k)%nprocs,1000,COMM);
    MPI_Send(buf2, 1, MPI_INT_TYPE,
      (myrank+2*k)%nprocs,1000,COMM);
  } else {
    MPI_Send(buf,1,MPI_INT_TYPE,
      (myrank-k)%nprocs,1000,COMM);
    MPI_Send(buf2, 1, MPI_INT_TYPE,
      (myrank-2*k)%nprocs,1000,COMM);
} } }
```

(a) Decision point (DP) code.

$$\begin{pmatrix} & 1 & 1 & & & & 1 & 1 \\ 1 & & 1 & 1 & & & & 1 \\ 1 & 1 & & 1 & 1 & & & \\ & 1 & 1 & & 1 & 1 & & \\ & & 1 & 1 & & 1 & 1 & \\ & & & 1 & 1 & & 1 & 1 \\ 1 & & & & 1 & 1 & & 1 \\ 1 & 1 & & & & 1 & 1 & \end{pmatrix}$$

(b) Conservative matrix.

$$\begin{pmatrix} & 1 & 1 & & & & & \\ & & 1 & 1 & & & & \\ & & & 1 & 1 & & & \\ & & & & 1 & 1 & & \\ & & 1 & 1 & & & & \\ & & & 1 & 1 & & & \\ & & & & 1 & 1 & & \\ & & & & & 1 & 1 & \end{pmatrix}$$

(c) Matrix with DP analysis.

Fig. 7. Example of a communication matrix.

from the traditional inter-procedural passes as its goal is to examine variables at the point of communication. For example, consider the procedure `p(k)` from Figure 7(a), which contains an instruction to send a message to `myrank+k` or `myrank-k`, and `p` is called from two different locations, L1 and L2, with parameters A and B, respectively. If A is known at L1, while B is not known at L2, then the communication within p is static when p is invoked from L1, while it is not if p is invoked from L2.

Network configuration instructions: The example shown in Figure 8 demonstrates a sample of network configurations for the code in Figure 7(a). Each configuration file is a list of connections, each of which is scheduled to a particular circuit switching network prior to execution. Each connection description in the file contains 3 fields: `net_id source destination`, referring to the switch number, source, and destination processor. Figure 8(a) shows the network configuration for a single circuit switch and Figure 8(b) extends the configuration for a second circuit switch. The switch configuration is passed to the network using the `reconfigNetwork(config1)` function highlighted in Figure 7(a).

In this example, when only a single switch is available the connections specified in Figure 8(b) would actually be filtered out and satisfied in the packet switch as described in Figure 6. The configurations assume the matrix from Figure 7(c). If the conservative matrix was constructed from the compiler (Figure 7(b)), the system would require four switch planes, or the compiler would filter out additional connections, possibly retaining unused connections.

B. Compiler Implementation

We have developed an experimental compiler which implements the framework shown in Figure 9. It is

```
config1:
0 0 1
0 1 2
0 2 3
0 3 4
0 4 3
0 5 4
0 6 5
0 7 6
```

(a) One switch.

Same as (a) appending

```
1 0 2
1 1 3
1 2 4
1 3 5
1 4 2
1 5 3
1 6 4
1 7 5
```

(b) Two switches.

Fig. 8. Example of network configurations for Figure 7(a).

based on the SUIF compiler infrastructure [35] for detecting communication patterns from source code of certain applications and enhancing them with compiled communications. The SUIF compiler infrastructure is an open source, source to source compiler research toolkit supporting both the C and Fortran 77 languages. The front end of the SUIF compiler compiles parallel applications to SUIF intermediate format. A tool, porky, from the SUIF compiler system is used to perform basic transformations, such as copy propagation and constant propagation in order to discover more details about the characteristics of different communication(e.g. static, dynamic, or persistent). The compiler contains four key compilation passes described in the next several sections:

1) Communication Detection Pass: In this pass, all the MPI communication calls and parameters are located in the SUIF intermediate code. Based on MPI standards, we can explicitly enumerate all possible MPI operations. With this list, we can easily identify, within the compiler, all the MPI operations that occur in the code. This pass collects information about these communication operations' parameters. For instance, this pass can identify that `me` is the parameter variable holding the rank value and `nprocs` is the parameter variable holding the total number of nodes from the following source code.

```
  call mpi_comm_rank(mpi_comm_world, me, ierr)
  call mpi_comm_size(mpi_comm_world, nprocs,
ierr)
```

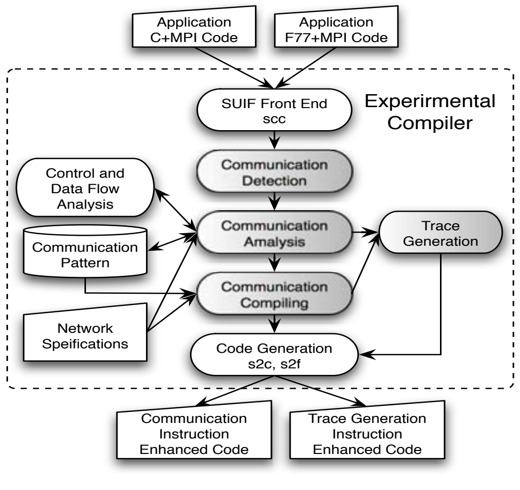


Fig. 9. The experimental compiler.

2) Communication Analysis Pass: During communication analysis, the MPI operations and parameters used in the MPI functions identified during communication detection are used to identify and represent communication patterns. Control and data flow analysis and interprocedural analysis are used to determine the location at which communication operations are resolved in the code. If the source, destination, and message volume of a communication operation can be resolved as constants, the communication is static. If communication operations are discovered to be persistent as described in Figure 1(b), they are further examined as to whether they are static. We extend inter and intra-procedural constant propagation into *symbolic expression propagation* to determine whether communications are static. For example, it is necessary to recognize and consider the processor ID, number of processors, problem size, and various other parameters as static to ensure that the topology can be determined statically. As previously described, we use loops as basic phase delimiters, and then merge contiguous phases to form larger phases when it is possible. To obtain proper communication phases within the code, we use the information discovered in CDFG analysis and communication detection pass. The proper determination of phases is based on factors such as phase length, amount of static and/or persistent communication present, and communication to computation ratio. We use the algorithm from Figure 6 to determine the communication phases.

3) Communication Compiling Pass: The communication compiling pass further optimizes the communication pattern identified during analysis, compiles the pattern and inserts network configuration directives into the application to assist in configuring the interconnections. The communication compiling pass exposes static and persistent communications to circuit switching interconnection networks with the goal of reducing the communication overhead. For example, this compiler can insert instructions to preload network configurations for the switches proposed in [23]. Currently, two types of network configuration instructions are considered in our experimental compiler.

Network configuration setup instructions are used to pre-establish network connections. These instructions can reduce the setup overhead of connections and overlap network control with computation.

Network configuration flush instructions are used to flush the current network configuration or a subset of circuits from the current configuration. Such instructions can be used to remove the expired circuits. It also provides potential to speedup the parallel applications as it reduces contention for the circuits.

For static communication patterns, the content of the communication pattern derived during analysis is inserted into the code using the `reconfigNetwork()` system call as shown in Figure 7(a) designed to preconfigure the network at the beginning of each communication phase. For persistent communication patterns, the communication analysis component provides communication pattern information as a function of variables that will not be known until run-time. Values for calculating these symbolic expressions are passed to the `reconfigNetwork()` function as parameters. The scope and value availabilities of variables in the symbolic expressions, in addition to the location of the communication operations, put constraints on the locations where these network configuration instructions may be inserted.

4) Trace Generation Pass: The experimental compiler also has the capability to automatically add trace generation instructions (e.g. print statements) within MPI applications. This pass relies on the information discovered in the communication detection pass and the communication analysis because it generates traces for both MPI functions and related code structures such as *if*s, *loop*s and *procedure boundaries*. The resulting annotated code can be compiled and executed on the parallel architecture normally just like the original code. The only difference is that the newly included print instructions will record traces. These traces are primarily used to study communication patterns of an execution instance and/or verify that the communication patterns detected by the compiler are accurate. This pass provides a useful tool for automating trace generation for MPI applications and relieve researchers from the trouble of generating traces manually.

V. Results

Our compiled communication techniques can provide certain information about MPI parallel applications beyond the current approaches in the literature. In Section V-A we examine the impact of considering persistent communications in addition to static and dynamic communications. This knowledge can be leveraged in the compiler to determine the communication patterns for these benchmark applications, as shown in Section V-B. Simulation-based performance analysis is presented in Section V-C to demonstrate the benefits of using our compiler techniques for achieving efficient communications in multiprocessor systems.

A. Classification of communications of NAS Parallel Benchmarks

We used our experimental compiler to profile the communication statistics of the NAS Parallel Benchmarks v2.4.1. The percentage of static, persistent, and dynamic communications are shown for point-to-point operations in Table II and for collective operations in Table III. In all cases the compiler accurately detected the classification compared to an ideal static analysis.

For the point-to-point operations, Integer Sort (IS), Conjugate Gradient (CG) and Lower-Upper Symmetric Gauss-Seidel (LU) contain only static communications. Multigrid (MG), Block-Tridiagonal (BT), and Scalar-Pentadiagonal (SP) contain only persistent communications. For BT and SP, the destination set for each node is calculated prior to all point-to-point communications and are used through application completion. For MG, there are two communication stages. In each stage, the destination set for each node is calculated prior to the communications and is retained until each stage completes. These two stages contains multiple outermost loops.

TABLE II
THE PERCENTAGES OF DIFFERENT POINT-TO-POINT COMMUNICATIONS IN NAS BENCHMARKS.

Benchmark	Static	Persistent	Dynamic
IS	100%	0%	0%
CG	100%	0%	0%
MG	0%	100%	0%
BT	0%	100%	0%
SP	0%	100%	0%
LU	100%	0%	0%

TABLE III
THE PERCENTAGES OF DIFFERENT COLLECTIVE COMMUNICATIONS IN NAS BENCHMARKS.

Benchmark	Static	Persistent	Dynamic
IS	0.4%	0%	99.6%
CG	100%	0%	0%
MG	100%	0%	0%
EP	100%	0%	0%
FT	0.1%	99.9%	0%
BT	100%	0%	0%
SP	100%	0%	0%
LU	100%	0%	0%

The data in Table III were acquired through compile-time analysis with the exception of IS and FT for which the percentage of dynamic communications were obtained from a class B execution trace on 128 processors. The reason is that IS and FT consist of more than one class of collective operations and this leads to different results with different execution configurations—problem size and number of processors. For FT, while the number of total nodes increases, persistent all-to-all communications dominate the message volume and the percentage of static communications is extremely low.

B. Identifying Communication Patterns

To show the capability of our compiler to identify communication patterns, we consider the IS, LU, MG and CG benchmark programs from the NAS parallel benchmark suite and one application LBMHD (Lattice Boltzmann model of magneto-hydrodynamics [36], [37]). For all of these benchmarks as well as the COMOPS and synthetic applications described in Section V-C.2, the compiler was able to discover all of the static and persistent communication operations. This was verified through a comparison with manual analysis of the application. While compiling the NAS parallel benchmarks, the total number of processors, referred to as N, has to be set as a build parameter so that it is known at compile-time.

IS: The compiler identified communication pattern for IS has been shown in Table I and Figure 5. The weights in $c{-}vectors$ and $p{-}matrices$ are binary values. In phases 0 and 1, collective operations `AR`, `AA`, and `AV` are executed with no point-to-point communication. Phase 2 combines the `RD` collective operation with the point-to-point matrix shown in Figure 5. By using Merge and Unwrap operations, phase 0 and 1 can be combined.

LU: The LU benchmark is demonstrated as another example. When identifying the communication patterns for LU, we set the threshold such that the $Filter$ operation removed all the collective communications from all the phases. Initially the compiler identified 11 phases. Ten of them contain only a small number of connections. Using the $Merge$ operation the $p{-}matrix$ shown in Figure 10 is obtained. From this matrix we can see that the number of destinations varies from 2 to 4 yielding a reasonably sized *working set*.

CG: The compiler initially identified two communication phases from the source code. It turns out that each of these phases are identical and can be merged. The $p{-}matrix$ for the compiler predicted communication pattern is shown in Figure 11.

MG: When analyzing MG, the compiler discovered that the communication destinations depend on run-time input data. However, after the topologies are determined, they are used for an extended period of time. Hence the communication pattern is persistent. Therefore, it is only possible to construct symbolic $p{-}matrices$ or formula lists from the source code. These $p{-}matrices$ are resolved at runtime to populate the network.

We show the resolved 12 $p{-}matrices$ for the default input file supplied with the benchmark shown in Figure 12. $p{-}matrices$ 0, 1, 6-10 correspond to Figure 12(a), $p{-}matrices$ 2 and 11 correspond to Figure 12(b), $p{-}matrix$ 3 corresponds to Figure 12(c), and $p{-}matrix$ 4 corresponds to Figure 12(d). $p{-}matrix$ 5 is empty because the branches containing zero point-to-point communications were taken in the decision points for the parameters specified in this case.

Lattice Boltzmann method to model magneto-hydrodynamics (LBMHD): The communication pattern of application LBMHD is shown in Figure 13 and Figure 14. Our compiler identified a single communication phase. Because the number of processors N and the processor rank $rank$ are determined at run-time and not known at compile-time, we generate a formula list in Figure 13 to describe the $p{-}matrix$. Each processor has a set of four other processors with which it communicates. Since N and $rank$ are the only symbols in the expression, this matrix is considered statically known as it may be entirely resolved at load-time. Thus it is possible to entirely configure the network for LBMHD based on compiler analysis prior to execution. Figure 14 shows the pattern for a run on 64 processors.

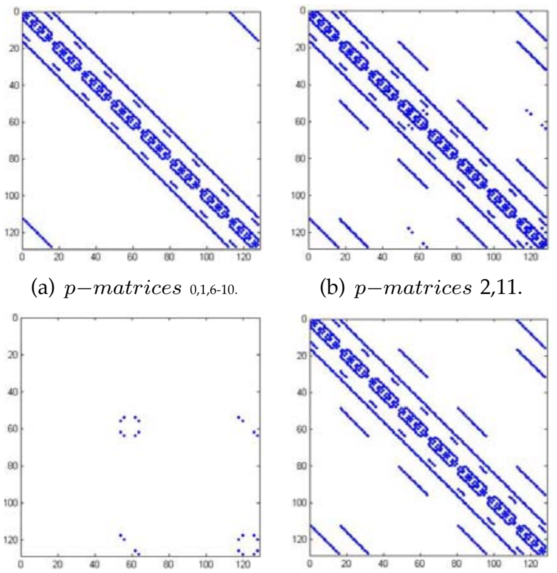

(a) $p{-}matrices$ 0,1,6-10. (b) $p{-}matrices$ 2,11.
(c) $p{-}matrix$ 3. (d) $p{-}matrix$ 4.

Fig. 12. The $p{-}matrices$ of MG ($N = 128$).

$$\begin{pmatrix} ((\lfloor rank/N_y \rfloor + 1) \mod N_x + (rank - \lfloor rank/N_y \rfloor * N_y) \\ ((\lfloor rank/N_y \rfloor - 1) \mod N_x + (rank - \lfloor rank/N_y \rfloor * N_y) \\ \lfloor rank/N_y \rfloor * N_y + ((rank - \lfloor rank/N_y \rfloor * N_y) + 1) \mod N_y \\ \lfloor rank/N_y \rfloor * N_y + ((rank - \lfloor rank/N_y \rfloor * N_y) - 1) \mod N_y \end{pmatrix}$$

Fig. 13. LBMHD $p{-}matrix$ described by a formula list where $N = N_x * N_y$.

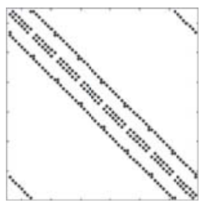


Fig. 14. LBMHD $p{-}matrix$ ($N = 64$).

The above compiler-predicted communication patterns have been verified by comparison to their counterparts extracted from traces. We run the applications to which trace generation instructions have been inserted by our compiler to collect traces. The communication patterns identified from the corresponding traces are identical to what the compiler has identified from the source code.

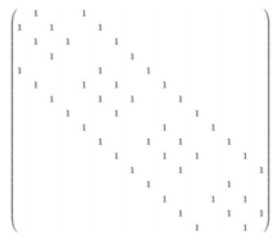


Fig. 10. Single bit $p{-}matrix$ PM_LU ($N = 16$).

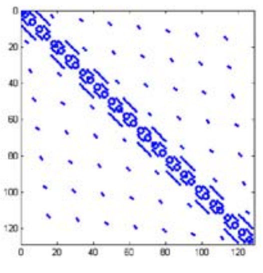


Fig. 11. The $p{-}matrix$ of CG ($N = 128$).

C. Performance Analysis

To analyze the efficiency of applying compiled communication techniques to static and persistent communications in MPI applications, it is necessary to examine performance data. Thus, we describe our simulation testbed used to run performance experiments.

1) Multiprocessor System Simulator: We developed an event-driven multiprocessor system simulator using C++ with CSIM. The simulated system containing N-processors implements the communication model from Figure 2. The diagram of the simulated system is shown in Figure 15. In the system each processor reads communication instructions from an input trace file, which is either a real trace from an application execution or a synthetic stream of simulation instructions, and emulates the corresponding communication operations. In the trace files communication instructions can be separated by computation instructions "`COMP t`", each of which emulates a serial of computation over time `t`.

Each processor has a circuit switching NIC connected to a number of circuit switching networks and a packet switching NIC connected to a number of packet switching networks. Hence, the simulated system may have multiple circuit switches and multiple packet switches. Each packet switch is a 2-stage FAT tree network built using fast buffered crossbar switches [38], [39]. Each circuit switch is a $N \times N$ crossbar and can be configured to realize any permutation at any time. A centralized scheduler is responsible for setting up and tearing down connections in all the circuit switches. To avoid the long connection establishment delay, a processor sends a message to the circuit switching NIC only when there is a connection available in any of the circuit switches. Otherwise, the message is dispatched to the packet switches. Hence there must be at least one packet switching network in the system.

When dealing with the phase boundaries consisting of patterns consisting of point to point communication operations, several processors may have proceeded to the next communication phase while others remain in the original phase. In our simulator, when a processor leaves a communication phase, it releases the circuits from that phase and requests the circuits from the new phase. In some cases these circuits might not be immediately available due to resources held by processors in the previous phase. In that case messages proceed through the packet switch until the circuits are released. A similar process would be possible for collective communications

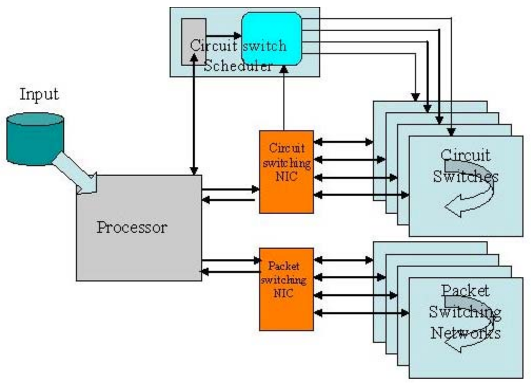


Fig. 15. Simulation system overview..

that use a relaxed blocking concept as described in [40].

When simulating many packet switches, we assume that a NIC has independent buffers for each packet switching network. The messages in each buffer are handled sequentially, however, the NIC can handle different buffers simultaneously. Each incoming message from the processor to the NIC is assigned a buffer randomly. We assume that all packets of that message may only be delivered through the corresponding packet switching network.

The simulator allows the specification of many system parameters. In our experiments we used 256 processors at 10GHz with 5000 cycles of overhead for MPI send and receive. We also used link bandwidth of 4Gb/s for the FP network and the circuit switch while 400Mb/s is used for the supplemental slower packet switching network. We use a circuit establishment delay of 3ms for the OCS and 20 μs for the fast circuit switch. We assume that the packet switch is a two-level FAT tree with link propagation delay varying between 130 and 400 ns.

The simulator was extensively tested to verify its correctness. For example, we explicitly tested single source single destination non-blocking message sending tests with and without saturation of the network to ensure back-pressure occurred when the throughput was exceeded. We ran similar tests with multiple source and destination artificial traffic for both an under-loaded and saturated network achieving expected results according to theory. We ran similar experiments for blocking sends. We also examined contention at the destination from single source and multiple sources with the same amount of traffic to see the improvement in latency while the completion time remains constant.

2) Simulations: Packet switching networks with buffering at cross-points [38], [39] can achieve very high performance. Unfortunately, the cost of this type of switch is very high making it impractical. However, it can be considered a best case scenario for comparison of what a fast packet switch can achieve. The simulation results show that we can achieve the same level, or even better level of performance than buffered crosspoint switches using much less expensive circuit switches when the communication patterns are known. We simulated two different types of systems to demonstrate this: 1) systems with only fast packet switching networks, referred to as "Fast Packet switching" or *FP* and 2) systems with multiple circuit switches to which as many connections as possible are preloaded and pined during execution, referred to as "Preload pined" or *PP*.

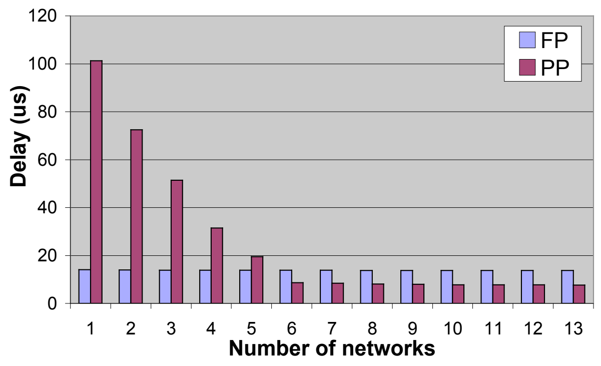
For each system, we run the simulation with different number of networks. All links in both the fast packet switching networks and the circuit switches have the same bandwidth of 4G bps. We model the circuit switches as optical micro-electro-mechanical system (MEMS)-based switches whose connection setup overhead is set to 3ms [6], [21], [22]. For each *PP* system, a relatively slow packet switching network with bandwidth of 400M bps is included. In what follows, we report the average message delay as the performance metric. Simulation results of 256-node traces of 4 NAS benchmarks are shown in Figure 16.

The communication degrees of MG256, CG256, SP256 and BT256 are 13, 5, 6 and 6, respectively. The communication is typically evenly distributed across each node pairing with the exception MG256 whose dominant communication is evenly distributed among only 6 of the 13 pairs. Hence, as we increase the number of circuit switching networks, we see a near linear reduction in the average message delays for the 4 NAS benchmarks. For instance, we can reduce the message delay of SP256 with 6 circuit switches to as low as 37.2% of what *FP* can achieve. This trend is maintained for all the applications we have investigated. Simulation results in Figure 16 show that we can reduce message delay using 6 circuit switches (by 37.2% to 62.8%) compared to *FP* for all the investigated applications.

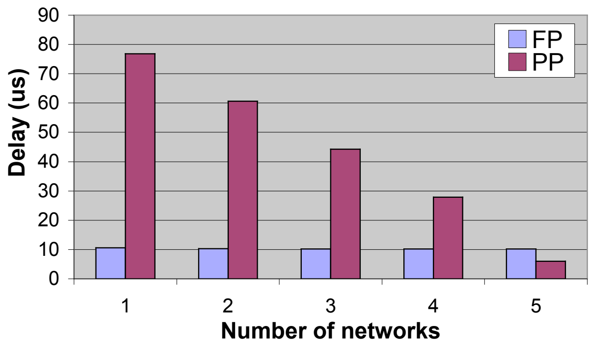
Our technique typically works well for highly static and persistent communication. Table II shows that all the 4 NAS benchmarks demonstrated here contain either completely static or persistent communication patterns leading to this linear delay reduction per switch plane. Interestingly, adding additional fast packet switches did not improve the communication delay for two main reasons: (1) the NAS benchmarks produce messages infrequently enough that the majority of messages have left the buffer in the outgoing NIC prior to the arrival of the next message and (2) we assume that individual messages cannot be broken up and sent across separate network planes due to problems like out of order arrival of packets. Supporting out of order arrival of messages could potentially improve the performance of multiple packet switch configurations [41] but leads to significant additional complexity in the system and potentially degrades performance of in order messages [42]. Given that many parallel application are dominated by static and persistent communication and it is difficult to achieve a significant benefit with multiple packet switch networks and that these fast packet switching networks are of such high complexity and cost, our approach using circuit switching is very promising.

All the applications discussed above either have a single phase or have a dominant phase in which communication volume is much larger than other phases. In order to demonstrate the benefit of reconfiguring the network at phase boundaries, we simulate systems with multiple circuit switches to which connections are loaded at the beginning of each phase and pined during the phase, referred to as Phased Preload Pined—*PPP*. We also consider a circuit switch with a faster circuit establishment delay for the PPP case called PPP-Fast.

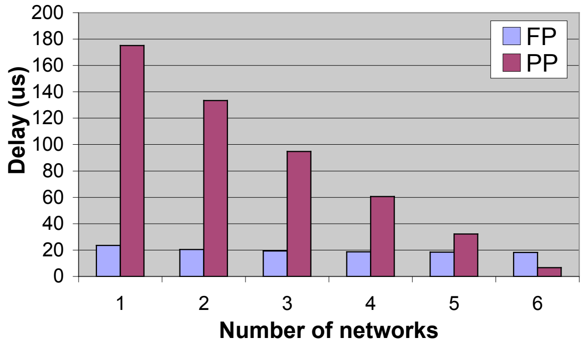
PPP can achieve higher performance than *PP* in certain circumstances. For example, given a communication pattern, in which the i^{th} phase has a communication degree n_i and the global communication pattern has a degree m, it is reasonable to expect that $m > max\{n_i\}$. It is obvious that *PPP* can obtain the same performance using only



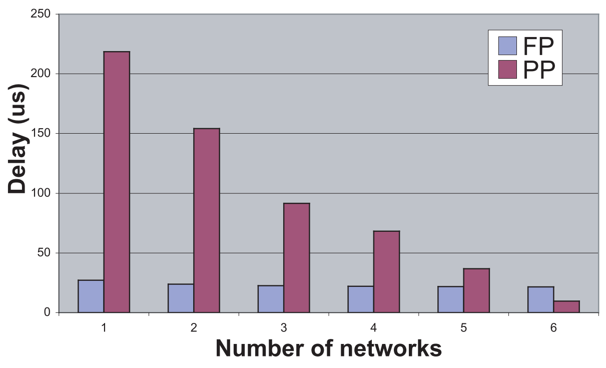
(a) Message delay of MG256.



(b) Message delay of CG256.



(c) Message delay of SP256.



(d) Message delay of BT256.

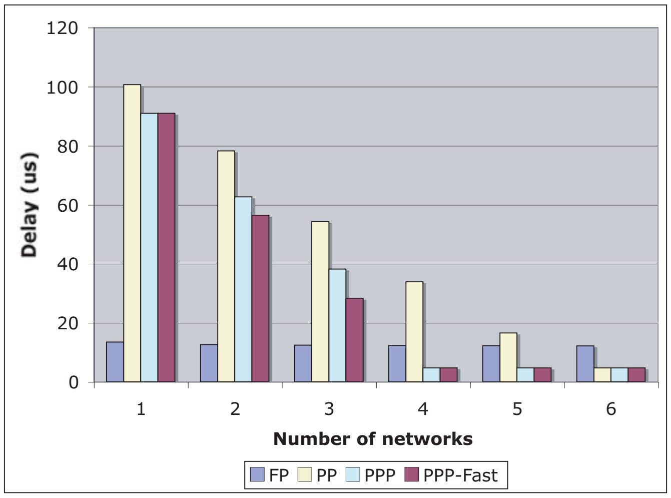
Fig. 16. Message delay of MG256, CG256, SP256, and BT256.

$max\{n_i\}$ circuit switches as what PP can obtain with m circuit switches if the network reconfiguration delay is small or can be overlapped with computation. We use a synthetic program SYN256 (Figure 17(a)) and the ASCI COMOPS benchmark [43] (Figure 17(b)) to demonstrate the impact of changing the network configuration between phases.

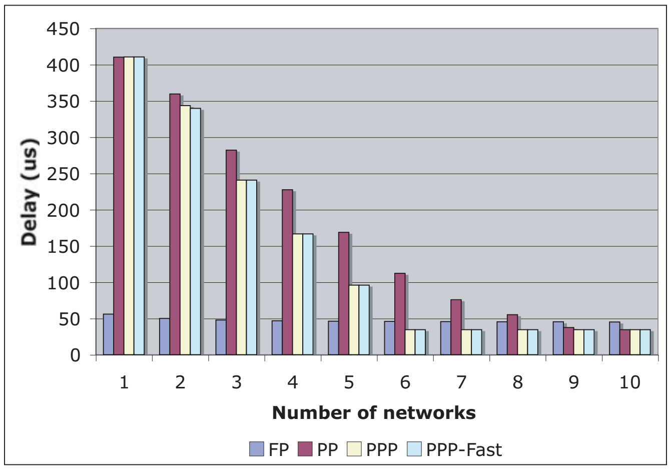
SYN256 has 3 phases, each phase performs 2-D mesh communications along 2 dimensions on a 3-D mesh and has a communication degree 4 while the communication degree of the entire program is 6. For the COMOPS256 benchmark the point-to-point communication consist of three phases: ping-pong, 2-D ghost cell update, and 3-D ghost cell update. Here the topology of the 2-D communication is not embedded in the topology of the 3-D communication. Simulations show that for both programs the message delay of PPP is lower than PP when using the same number of circuit switching networks and it decreases much faster than PP when we increase the number of circuit switching networks. PPP-Fast does provide an advantage over PPP in some cases as expected, however, in most cases the connection establishment can be sufficiently amortized to make PPP and PPP-Fast equivalent.

The communication degrees of parallel applications are typically small [6], [44]. Although the maximum communication degree of some parallel applications can be very large, the dominant communication degree may still be small and the low bandwidth connections can be filtered out with minimal performance loss using a circuit switch approach [6]. Therefore, we can expect that 10 or fewer circuit switches are sufficient to serve a parallel application in most cases.

As previously mentioned, we have primarily reported message delay as this demonstrates the advantage of our approach even when the benchmark is not communication bound, as is the case with many of the NAS benchmarks. Figure 18 shows the application completion time with various network configurations for COMOPS on 256 processors (Figure 18(a)), which is communication bound and an example from NAS, SP on 256 processors (Figure 18(b)), which is computation bound. For the communication bound application, COMOPS, the improvement in execution time mirrors the improvement in message delay. For the computation bound application, SP, the completion time improves slightly to reflect the improvement of the communication, however, the overall improvement is relatively small. We do not report PPP for SP as it contains only a single dominant communication phase.



(a) Message delay of SYN256.



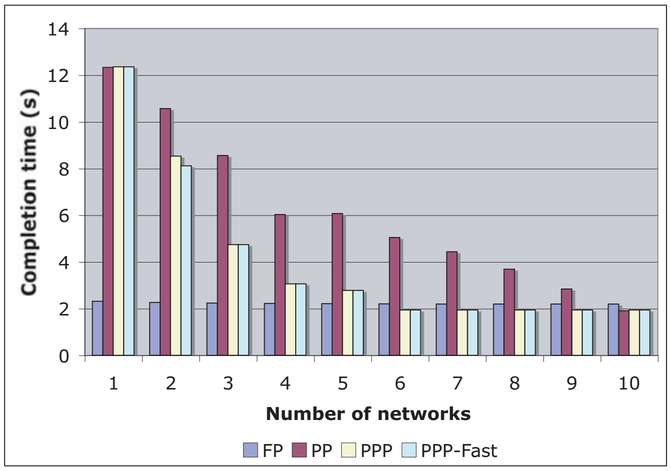
(b) Message delay of COMOPS256.

Fig. 17. Message delay of COMOPS256, SYN256.

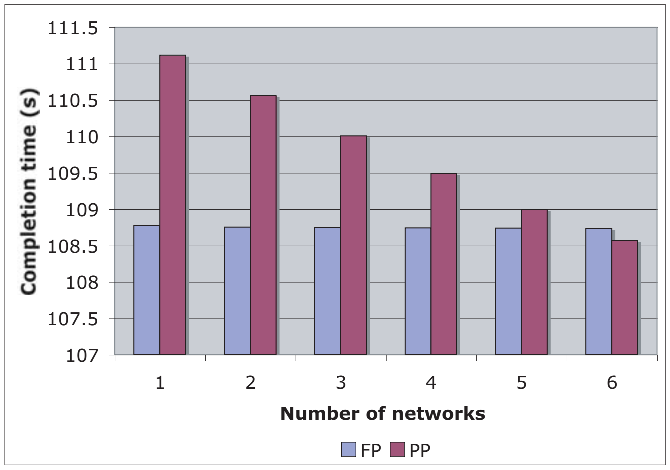
VI. Conclusions

In this paper we explore compiler techniques to obtain efficient communications for MPI applications on circuit switching networks. A compilation framework integrating fairly sophisticated analysis is described. In the framework, communication patterns are identified and compiled into network configurations. This allows significant performance benefits when connections can be established in the network prior to the actual communication operation with relatively few circuit switches. The framework includes a flexible and powerful communication pattern representation scheme that can capture the property of communication patterns and allow manipulation of these patterns. In this way, communication phases can be detected and logically formulated within the application. This scheme also provides the power to easily manipulate the granularity of communication phases using a set of proposed operations on the patterns. However, we present a specific phase partitioning algorithm used to maximize phase duration and minimize connections that cannot be satisfied in the circuit switch network.

Additionally, we extend the classification of static and dynamic communication patterns and operations to include persistent communications. Persistent communications cannot be determined statically, however, they remain unchanged for large segments of the ap-



(a) COMOPS256.



(b) SP256.

Fig. 18. Completion time examples with various network configurations.

plication execution. An experimental compiler has been developed to implement this framework using the SUIF compiler infrastructure.

Applying our experimental compiler on the NAS parallel benchmark suite, we found that a large portion of communications which were previously classified as dynamic [8] are actually persistent. This provides opportunities for pre-configuring the network to reduce communication overhead. Simulation results show that competitive performance can be achieved through combination of compiled communication and circuit switching interconnection networks for multiprocessor systems. In particular, message delay using compiled communication and circuit switching can reduce message delay by 37% to 63% using no more than 6 circuit switches when compared to an idealistic buffered crossbar based packet switch network.

Finally, we note that the compiled communication concept can be expanded to include non-message-passing parallel programming models such as distributed shared memory models that give the appearance of shared memory but require actual messages to traverse the network.

References

[1] G. Broomell and J. R. Heath, "Classification categories and historical development of circuit switching topologies," *ACM Computing Surveys (CSUR)*, vol. 15, no. 2, pp. 95–133, 1983.

[2] J. Duato, S. Yalamanchili, and L. Ni, *Interconnection Networks: An Engineering Approach*. Margan Kaufmann, 2003.

[3] F. Cappello and C. Germain, "Toward high communication performance through compiled communications on a circuit switched interconnection network," in *Proc. of the Int. Symp. on High Performance Computer Architecture (HPCA)*, 1995, pp. 44–53.

[4] X. Yuan, R. Melhem, and R. Gupta, "Compiled communication for all-optical TDM networks," in *Proc. of SC*, 1996.

[5] T. Gross, "Communication in iwarp systems," in *Proc. of SC89*. ACM/IEEE, 1989, pp. 436–445.

[6] K. J. Barker, A. Benner, R. Hoare, A. Hoisie, A. K. Jones, D. J. Kerbyson, D. Li, R. Melhem, R. Rajamony, E. Schenfeld, S. Shao, C. Stunkel, and P. A. Walker, "On the feasibility of optical circuit switching for high performance computing systems," in *Proc. of SC*, 2005.

[7] J. Shalf, S. Kamil, L. Oliker, and D. Skinner, "Analyzing ultra-scale application communication requirements for a reconfigurable hybird interconnect," in *Proc. of SC*, 2005.

[8] A. Faraj and X. Yuan, "Communication characteristics in the NAS parallel benchmarks," in *Proc. of the Parallel and Distributed Computing and Systems Conf. (PDCS)*, 2002.

[9] J. Vetter and F. Mueller, "Communication characteristics of large-scale scientific applications for contemporary cluster architectures," *Journal of Parallel and Distributed Computing*, vol. 63, no. 9, pp. 853–865, September 2003.

[10] N. R. Adiga *et al.*, "An overview of the bluegene/l supercomputer," in *Proc. of Supercomputing (SC)*, 2002.

[11] S. L. Scott, "Synchronization and communication in the t3e multiprocessor," in *Proc. of ASPLOS-VII*, 1996.

[12] S. Habata, K. Umezawa, M. Yokokawa, and S. Kitawaki, "Hardware system of the earth simulator," *Parallel Computing*, vol. 30, no. 12, pp. 1287–1313, 2004.

[13] V. Gupta and E. Schenfeld, "Combining message switching with circuit switching in the interconnection cached multiprocessor network," in *Proc. IEEE Int. Symposium on Parallel Architectures, Algorithms and Networks*, 1994.

[14] ——, "Task graph partitioning and mapping in a reconfigurable parallel architecture," *Parallel Processing Letters*, vol. 5, no. 4, pp. 563–574, 1995.

[15] D. Chiarulli, S. Levitan, R. Melhem, J. Taza, and G. Gravenstreter, "Partitioned optical passive star (pops) multiprocessor interconnection networks with distributed control," *IEEE Journal of Lightwave Technology*, vol. 14, no. 7, pp. 1601–1612, 1996.

[16] G. Gravenstreter and R. Melhem, "Realizing common communication patterns in partitioned optical passive stars (pops) networks," *IEEE Transactions on Computers*, vol. 47, no. 9, pp. 998–1013, 1998.

[17] P. Dowd *et al.*, "Lightning network and system architecture," *Journal of Lightwave Technology*, vol. 14, pp. 1371–1387, 1996.

[18] A. K. Kodi and A. Louri, "Rapid: Reconfigurable and scalable all-photonic interconnect for distributed shared memory multiprocessors," *IEEE/OSA Journal of Lightwave Technology*, vol. 22, no. 9, pp. 2101–2110, 2004.

[19] ——, "Design of a high-speed optical interconnect for scalable shared memory multiprocessors," *IEEE Micro*, vol. 25, no. 1, pp. 41–49, 2005.

[20] ——, "A new technique for dynamic bandwidth re-allocation in optically interconnected high-performance computing systems," in *IEEE Symposium on High-Performance Interconnects*, 2006.

[21] P. C. et al, "Design and nonlinear servo control of mems mirrors and their performance in a large port-count optical switch," *Journal of Microelectromechanical Systems*, vol. 14, no. 2, pp. 261–273, April 2005.

[22] T. Yamamoto, J. Yamaguch, R. Sawada, and Y. Uenishi, "Development of a large-scale 3d mems optical switch module," *NTT Technical Review*, vol. 1, no. 7, pp. 37–42, October 2003.

[23] Z. Ding, R. Hoare, A. Jones, D. Li, S. Shao, S. Tung, J. Zheng, and R. Melhem, "Switch design to enable predictive multiplexed switching in multiprocessor networks," in *Proc. of the Int. Parallel and Distributed Procssing Symp. (IPDPS)*, 2005.

[24] S. Shao, A. K. Jones, and R. Melhem, "A compiler-based communication analysis approach for multiprocessor systems," in *Proc. of the Int. Parallel and Distributed Procssing Symp. (IPDPS)*, 2006.

[25] D. Shires, L. Pollock, and S. Sprenkle, "Program flow graph construction for static analysis of mpi programs," in *Proc. of Int. Conf. on Parallel and Distributed Processing Techniques and Applications(PDPTA)*, June 1999.

[26] S.-Y. Ho and N.-W. Lin, "Static analysis of communication structures in parallel programs," in *Proc. of the Int. Computer Symp.(ICS)*, 2002, pp. 215–221.

[27] *MPI: A Message-Passing Interface Standard*, Message Passing Interface Forum, June 1995.

[28] D. Bailey, T. Harris, W. Sahpir, and R. van der Wijingaart, "The NAS parallel benchmarks 2.0," Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Tech. Rep. NAS-95-020, December 1995.

[29] H. G. Dietz and T. Mattox, "Compiler techniques for flat neighborhood networks," in *Proc. of 13th Int. Wrokshop on Languages and Compilers for Parallel Computing*, 2000.

[30] J. Liang, A. Laffely, S. Srinivasan, and R. Tessier, "An architecture and compiler for scalable on-chip communication," *IEEE Trans. on Very Large Scale Integration Systems (TVLSI)*, vol. 12, no. 4, pp. 711–726, July 2004.

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

[32] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina, "Architectural requirements of parallel scientific applications with explicit communication," *ACM Computer Architecture News*, vol. 21, no. 2, pp. 2–13, May 1993.

[33] R. R. Hoare, Z. Ding, and A. K. Jones, "Level-wise scheduling algorithm for fat tree interconnection networks," in *Proc. of Supercomputing*, 2006.

[34] V. Delaluz, M. Kandemir, N. Vijakrishnan, A. Sivasubramaniam, and M. J. Irwin, "Dram energy management using software and hardware directed power mode control," in *IEEE International Symposium on High-Performance Computer Architecture*, 2001, pp. 159–169.

[35] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarsinghe, J. M. Anderson, S. W. K. Tjiang, S. W. Liao, C. W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "Suif: An infrastructure for research on parallelizing and optimizing compilers," *ACM SIGPLAN Notices*, vol. 29, no. 12, pp. 31–37, December 1994.

[36] P. Pavlo, G. Vahala, and L. Vahala, "Higher order isotropic velocity grids in lattice methods," *Physics Review Letters*, vol. 80, p. 3960, 1998.

[37] A. MacNab, G. Vahala, P. Pavlo, L. Vahala, and M. Soe, "Lattice boltzmann model for dissipative incompressible mhd," in *28th EPS Conference on Contr. Fusion and Plasma Phys*, vol. 25A, June 2001, pp. 853–856.

[38] J. Kim, W. J. Dally, B. Towles, and A. K. Gupta, "Microarchitecture of a high radix router," in *Proc. of ISCA*, 2005, pp. 420–431.

[39] C. B. Stunkel, J. Herring, B. Abali, and R. Sivaram, "A new switch chip for ibm rs/6000 sp systems," in *Proc. of SC*, 1999.

[40] T. Hoefler, P. Kambadur, R. L. Graham, G. Shipman, and A. Lumsdaine, *A Case for Standard Non-blocking Collective Operations*, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4757, pp. 125–134.

[41] L. Gharai, C. Perkins, and T. Lehman, "Packet reordering, high spped networks and transport protocol performance," in *Proc. of the IEEE International Conference on Computer Communications and Networks (ICCCN)*, 2004, pp. 73–78.

[42] P. Balaji, W. Feng, S. Bhagvat, D. K. Panda, R. Thakur, and W. Gropp, "Analyzing the impact of supporting out-of-order communication on in-order performance with iwarp," in *Proc. of SuperComputing (SC)*, 2007.

[43] LLNL, "The asci comops benchmark code," Lawerence Livermore National Laboratory Website, `http://www.llnl.gov/`.

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

PLACE PHOTO HERE

Shuyi Shao received a B.S. and a M.S. degree in computer science from Xi'an Jiaotong University, China, in 1996 and 1999, respectively. He is currently pursuing the Ph.D. degree at the University of Pittsburgh, PA. His research interests include high performance computing, compiler, and architecture. He is a student member of IEEE.

PLACE PHOTO HERE

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 Northwestern University. He is currently an Assistant Professor of Electrical and Computer Engineering and Computer Science at the University of 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 twice. Dr. Jones research interests include compilation techniques for behavioral and low-power synthesis, embedded systems, radio frequency identification (RFID), and high performance computing. He is the author of over 50 publications in these areas. Dr. Jones has served on several conference program committees including the International Conference on Parallel Processing, the Parallel and Distributed Computing and Systems Conference, and the Workshop for Large Scale Parallel Processing at IPDPS. He has served as associate editor for several journals including ACM Transactions on Design Automation for Electronic Systems and Parallel Processing Letters. He is currently a member of the IEEE and the ACM and is serving on the executive committee of the ACM Special Interest Group in Design Automation.

PLACE PHOTO HERE

Rami Melhem received a B.E. in Electrical Engineering from 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 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 interests include Optical Networks, High Performance Computing, Real-Time and Fault-Tolerant Systems 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 Springer 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.