Using Multirail Networks in High Performance Clusters[*][†]

Salvador Coll, Eitan Frachtenberg, Fabrizio Petrini,
Adolfy Hoisie and Leonid Gurvits
CCS-3 Modeling, Algorithms, & Informatics Group
Computer & Computational Sciences Division
Los Alamos National Laboratory
{scoll,eitanf,fabrizio,hoisie,gurvits}@lanl.gov

Abstract

Using multiple independent networks (also known as rails) is an emerging technique to overcome bandwidth limitations and enhance fault tolerance of current high-performance parallel computers. In this paper we present and analyze various algorithms to allocate multiple communication rails, including static and dynamic allocation schemes. An analytical lower bound on the number of rails required for static rail allocation is shown. We also present an extensive experimental comparison of the behavior of various algorithms in terms of bandwidth and latency. We show that striping messages over multiple rails can substantially reduce network latency, depending on average message size, network load, and allocation scheme. The compared methods include a static rail allocation, a basic round-robin rail allocation, a local-dynamic allocation based on local knowledge, and a dynamic rail allocation that reserves both communication endpoints of a message before sending it. The last method is shown to perform better than the others at higher loads: up to 49% better than local-knowledge allocation and 37% better than the round-robin allocation. This allocation scheme also shows lower latency and it saturates at higher loads (for messages long enough). Most importantly, this proposed allocation scheme scales well with the number of rails and message sizes. In addition we propose a hybrid algorithm that combines the benefits of the local-dynamic for short messages with those of the dynamic algorithm for large messages.

Keywords: Communication Protocols, High-Performance Interconnection Networks, Performance Evaluation, Routing, Communication Libraries, Parallel Architectures.

[*] A shorter version of this paper and results for the static rail allocation can be found in [3] and [2].

[†] The work was supported by the U.S. Department of Energy through Los Alamos National Laboratory contract W-7405-ENG-36

1 Introduction

System-interconnection networks have become a critical component of computing technology, with a direct impact on the design, architecture, and use of high-performance parallel computers. Indeed, not only the sheer computational speed distinguishes high-performance computers from desktop systems, but also the efficient integration of the computing nodes into tightly coupled multiprocessor systems. Network adapters, switches, device-drivers and communication libraries are increasingly becoming performance-critical components in modern supercomputers.

One approach to building large-scale supercomputers, with as many as thousands of processors, is to use shared memory multiprocessors (SMPs) as building blocks. In such machines, it is very important to keep the ratio between computing power and communication capability properly balanced. One solution to the issues of limited bandwidth availability in network connections, and of fault tolerance, is the use of multiple parallel networks or "rails". This technique implies the utilization of several network interfaces per SMP node, attached to independent I/O buses. To the best of our knowledge, very little attention has thus-far been given in the literature to studies of communication protocols, performance characteristics, fault tolerance, and implementation of system software and libraries for multiple rails.

Aside from being a challenging scientific endeavor, the analysis of multirailed networks has direct practical implications as well. The Pittsburgh Supercomputing Center (PSC)[1], the most powerful supercomputer in the world for unclassified research at this time, is interconnected with two distinct network rails. Los Alamos National Laboratory and Compaq are currently developing an extreme-scale, multirailed cluster of SMPs, the 30Tops ASCI Q machine[2]. Both the PSC and the Q-machine are based on the Quadrics network (QsNet)[3], which consists of two building blocks, a 64bit/66MHz PCI card with a programmable network interface called Elan [10] and a low-latency high-bandwidth communication switch called Elite [11]. Elites can be interconnected in a fat-tree topology [6]. A recent performance evaluation of the QsNet shows that the network performance is seriously limited by the PCI bus [8]. In fact, the network can deliver almost 340 MB/sec at user-level (400 MB/sec of raw bandwidth), but the PCI implementation can sustain only 300 MB/sec, using the most efficient PCI chipset on the market. The presence of bidirectional traffic further degrades performance, limiting the aggregate communication bandwidth to 80% of the unidirectional bandwidth on most PCI chipsets (Intel 840, Serverworks He and LE, Compaq Wildfire). Though the next generation of the PCI interface, called PCI-X, will double the nominal performance, the new generation of QsNet will also double its performance, so this issue will not disappear. The same problem is also likely to appear with Infiniband, where high bandwidth between nodes can be achieved by grouping together several communication channels [1]. For example, the first implementations of Infiniband with the McKinley processor will be based on the Intel 870 chipset. This chipset provides a 4X Infiniband connection at 1GB/sec which is equalized to the bandwidth of the I/O bus in a single direction.

In this paper we present the basic properties of a multirailed network and analyze four approaches to multirail communication. These approaches try to minimize, or elim-

[1] http://www.psc.edu.

[2] http://www5.compaq.com/alphaserver/news/supercomputer_0822.html

[3] http://www.quadrics.com

inate, two distinct types of congestion.

1. *Conflicts at the destination node*. Multiple messages can be sent to the same destination from different sources at the same time. For example, if we split a message in two equally sized chunks and we send those chunks on two distinct rails, we expect to cut in half the delivery time of the whole message. But, if another message is sent to the same destination on one of the rails at the same time, then there is no performance advantage in using multiple rails.

2. *Conflicts on the I/O bus*. The recipient of a message can potentially use the same network interface to send another message in the other direction. Again, this can cause a substantial performance degradation[4] .

In the first approach, called *static rail allocation,* each network interface can either send or receive messages, and its direction is determined at initialization time, thus eliminating all conflicts on the I/O bus. Static allocation poses the problem of connectivity between nodes: we want to have a direct path in the network between any possible pair of nodes. The use of intermediate nodes could seriously degrade the latency achieved by zero-copy, user-level communication protocols, a key feature of most high-performance networks. In Section 2 we show that addressing this problem requires a large, possibly prohibitive number of rails. Also, the experimental results show that the performance obtained this way is sub-optimal.

We address these problems with *local-dynamic allocation*. In this scheme, rails are allocated in both directions, using local information available on the sender side. Messages are sent over rails that are not sending or receiving other messages, potentially striping a message over multiple rails when possible. Since this algorithm uses only local information, there is no guarantee that the traffic will be unidirectional, on both ends.

The *dynamic allocation* scheme tries to reserve both endpoints before sending a message and eliminates both types of conflicts. In its core there is a sophisticated distributed algorithm that ensures unidirectional traffic at both ends and avoids livelocks, potentially generated by multiple requests with a cyclic dependency. The implementation of this algorithm requires some processing power in the network interface card (NIC), which needs to process incoming control packets and perform the reservation protocol without interfering with the processors in the SMP. Fast response time in the NIC is essential to limit the overhead of this protocol for the protocol's overhead to be justified. This is the case of the QsNet [8], which is equipped with a thread processor that can read an incoming packet, do some basic processing and send a reply in as few as $2\mu s$.

Finally another dynamic allocation scheme is proposed, called *hybrid*, which allows bidirectionality for small messages, thus minimizing the protocol overhead for fine-grained communication. In the presence of large messages, the algorithm reserves both endpoints, maintaining unidirectional transmission on both ends as much as possible.

The experimental results, obtained using a circuit-level simulator of the network and network interface, explore the performance of these allocation algorithms under several traffic loads and message sizes. These results shed new light into the benefit of using multiple network rails and expose several trade-offs in the design of the allocation algorithms.

[4] All the algorithms presented in the paper can be easily generalized to the simpler case where bidirectional traffic can be efficiently handled by the network interface.

The rest of this paper is organized as follows: we start with the description and formal analysis of static rail allocation in Section 2. Section 3 presents the local-dynamic allocation and Section 4 offers a description of the dynamic and hybrid allocation approaches. The details of the experimental evaluation performed are described in Section 5 and the results obtained are presented in Section 6. Finally, we conclude in Section 7.

2 Static Allocation

In this section we describe the static allocation of network interfaces, in which each node-to-rail connection is exclusively a transmitter or a receiver. We obtain analytically the optimal allocation pattern and construct an algorithm for generating it. The terms network interface and rail are used interchangeably throughout this section.

2.1 Theoretical bound

The question we are trying to answer is, what is the maximum number of processing nodes that we can interconnect using a given number of rails, under the following constraints:

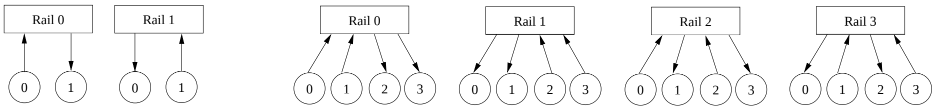
1. Each node can either transmit or receive on a given rail but not both. This ensures unidirectional access to the I/O bus.
2. Each node can transmit to every other node without passing through intermediate nodes.
3. Rails are independent: messages cannot pass from one rail to another.

Let us represent a static allocation using a binary matrix where columns represent nodes and rows represent rails, so that a value of '1' in the A_{ij} entry means that node *j* transmits on rail *i*, and a '0' means that node *j* receives on rail *i*. Figure 1 depicts static allocations examples and their equivalent allocation matrices. In the example shown in Figure 1(a), rail 0 can be used for sending by node 0 and receiving by node 1. Since the allocation is static, one more rail is required to allow communication from node 1 to node 0. As can be seen two rails are sufficient to ensure full connectivity between two nodes. When considering four nodes, at least four rails are required to ensure full connectivity. Figure 1(b) shows one possible allocation, and 1(d) the corresponding allocation matrix.

Our goal is to maximize the number of nodes *n* that can be fully connected by *r* rails, meeting the requirements listed above. A simple bound of $n \geq 2^{\frac{r}{2}}$, can be obtained with the static allocation described in Algorithm 1. While this allocation is simple, and clearly satisfies the constraints, it is not optimal. The optimality is contained in the following proposition:

Proposition 1. *Given r network rails, the number of nodes n that can be statically allocated to these rails with unidirectional communication in the network interface card (NIC) and full node connectivity cannot exceed*

$$n \leq \begin{pmatrix} r \\ \lfloor \frac{r}{2} \rfloor \end{pmatrix} \tag{1}$$



(a) Two rail allocation for two nodes

(b) Four rail allocation for four nodes

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

(c) Allocation matrix for two nodes

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

(d) Allocation matrix for four nodes

Figure 1: Simple static allocation examples for 2 (a) and 4 (b) nodes. Rectangles denote networks (rails); circles represent nodes and arrows denote the allocation of each rail to each node as either transmitter or receiver. The corresponding allocation matrixes are shown, respectively, in (c) and (d).

Algorithm 1 : Static rail allocation with $2\log_2 n$ rails.

```
procedure log_rail_alloc
begin
    for $i$ = 0 to $\log_2 n - 1$ do
    begin
        allocate nodes on rail $2i$ in consecutive groups of $2^i$, alternating
        between transmitters and receivers, starting with the transmitters.
    end
    for $i$ = 0 to $\log_2 n - 1$ do
    begin
        allocate nodes on rail $2i+1$ in consecutive groups of $2^i$, alternating
        between transmitters and receivers, starting with the receivers.
    end
end
```

Proof. Each node can use any given rail for either transmitting or receiving, but not both (unidirectional requirement). Let a binary vector represent the static allocation of nodes on a rail: the vector's *i*th entry is 0 if the *i*th node receives on this rail and 1 if it transmits on it. We can represent the static allocation of the entire system as a binary matrix *A* with *r* rows, each representing one rail, and *n* columns, each representing one node. Let A_{ij} denote the value at row *i* and column *j* of *A*, that is, the role allocated to the *j*th node on the *i*th rail. The problem can thus be formalized as determining the maximum number of columns *n* of a binary matrix with *r* rows for which the following property holds:

$$\forall\, x, y \in \{1..n\},\, x \neq y : \; \exists\, \rho \in \{1..r\} \;\; s.t.\; A_{\rho x} = 0,\, A_{\rho y} = 1 \tag{2}$$

For each matrix column *j* let S_j be the set of indices *i* for which $A_{ij} = 1$:
$S_j = \{1 \leq i \leq r \,|\, A_{ij} = 1\}$. Note that the property (2) of a matrix *A* is equivalent to the following property:

$$\forall\, x, y \in \{1..n\},\, x \neq y : \; S_x \not\subseteq S_y \tag{3}$$

The equivalence stems from the fact that if (3) doesn't hold, i.e.

$$\exists\, x, y \in \{1..n\},\, x \neq y \; s.t \; S_x \subset S_y$$

then for every row $\rho \in \{1..r\}$ for which $A_{\rho x} = 1$ we have also $A_{\rho y} = 1$ so (2) cannot hold. In the other direction, if (3) holds then for every two columns $x, y \in \{1..n\},\, x \neq y$ there would have to be at least one row $\rho \in \{1..r\}$ for which $A_{\rho x} = 0$, $A_{\rho y} = 1$, or else either $S_x \subset S_y$ or $S_y \subset S_x$. The maximum number of columns *n* for a matrix *A* with the property (3) is given by Sperner's lemma to be (1). A short proof of this lemma can be found in [7]. □

2.2 Allocation algorithm

We propose a constructive algorithm to allocate *r* rails to *n* nodes for any given *r* and *n* that satisfies (1). This algorithm is simple to implement and is optimal in the sense that it can allocate rails for all the nodes even when the bound is tight. The main idea behind it is to find *n* different binary vectors (representing the rail transmit/receive allocation for a single node), each having exactly $\left\lceil \frac{r}{2} \right\rceil$ 1's in them. The number of distinct vectors with this property is

$$\binom{r}{\left\lfloor \frac{r}{2} \right\rfloor}$$

so there is a sufficient number of vectors to allocate for *n* nodes. Also any two different vectors containing the same number of 1's satisfy condition (3), so by inference these vectors satisfy the requirement (2). Any enumeration that produces the different vectors can provide these vectors. For example, strings can be enumerated by lexicographic order (for $r = 4$ we could have 0011, 0101, 0110, 1001, 1010, 1100). Another simple procedure to enumerate such vectors is described in Algorithm 2.

Figure 2 shows the relationship between the number of nodes and the number of rails required to support them according to our requirements, using the two allocation algorithms described. An example allocation using Algorithm1 is depicted in Figure 3. We note that a maximum of 8 nodes can be allocated using 6 rails. Figure 4 is an

Algorithm 2 : Optimal static rail allocation.

```
{ build_rail_vectors is a recursive procedure that runs until n binary
  vectors of length r are output (n is the number of nodes and r is
  the number of rails), each representing an allocation of a single
  node. The procedure tries to allocate a 1 and then 0 for each vector
  location, and backtracks whenever a vector is completed. It should
  be first called from outside with the following parameters:
  build_rail_vectors (empty_vector, r, int(r/2))
}
Procedure build_rail_vectors
Input: vector being built (current_vector),
       rails left to allocate (rails_left),
       ones left for this vector (ones_left)
begin
  if n vectors were output then return  { Ending condition met -
                                          allocated for all nodes }

  if rails_left <= 0 then  { No. more rails to allocate means that - }
     output current_vector { the current vector (node) is completed. }
  else
  begin                           { Still have rails to allocate }
   if ones_left > 0 then    { Try to allocate a 1 if any left }
      build_rail_vectors (current_vector appended with 1,
                          rails_left - 1,
                          ones_left - 1)
   if (rails_left - ones_left) > 0 then    { Try to allocate a 0 -
                                            if any left }
       build_rail_vectors (current_vector appended with 0,
                           rails_left - 1,
                           ones_left)
  end
end
```

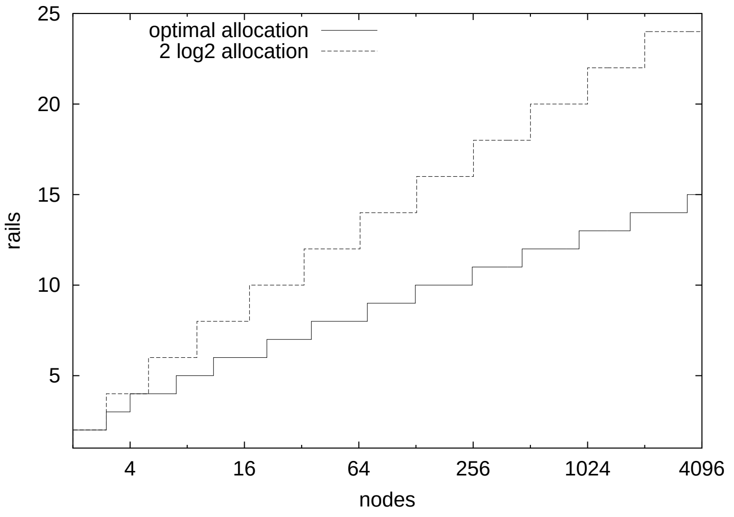


Figure 2: Required rails as a function of the number of nodes for both static allocation algorithms.

$$\begin{pmatrix}
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{pmatrix}$$

Figure 3: Example allocation for 6 rails and 8 nodes using Algorithm 1.

$$\begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1
\end{pmatrix}$$

Figure 4: Optimal allocation matrix for 6 rails and 20 nodes created using Algorithm 2.

example of an optimal allocation matrix created by Algorithm 2 for 20 nodes on 6 rails (20 nodes is the maximum for 6 rails). As it can be seen, the static needs a large number of rails to fully connect a given number of nodes.

3 Local Dynamic Allocation

With the dynamic allocation schemes, the direction in which each NIC is used by its node changes depending on the communication requirements. This allows to overcome the high rail requirement of the static allocation presented above and can make better use of network resources. Unlike static allocation, dynamic allocation does not predefine a communication direction for rails while still taking measures to minimize the amount of actual bidirectional traffic on a link.

In this section, a dynamic algorithm based only on local information (that available at the source node) is proposed. It can be applied to network configurations with any number of rails. Several desirable features are addressed namely, minimization of bidirectional traffic over the network interface, load balancing among rails, and high network utilization. The local-dynamic algorithm is used by each process to send a message over the network and is designed to stripe messages over multiple rails. Furthermore, when sending a message, it only selects NICs that are available. Thus, a send transaction will not produce bidirectional traffic in the source bus unless a message reception starts before the send transaction completes.

Algorithm 3 shows the local-dynamic scheme. The rail allocation policy selects a subset S of the set of free rails F for sending a message. All rails in S are then used for sending the message. The algorithm considers a rail as free if it is not sending or receiving. The local-dynamic algorithm uses a data structure (*NStatus*) which contains

Algorithm 3 : Local Dynamic Allocation

```
Procedure Local_Dynamic_Allocation
Input: message (M), destination node (dest), striping ratio (str_r)
begin
    repeat
        F ← {n | Nstatus[n]==FREE}
        S ← Select_Tx_NICs(F, str_r)
    until (F≠∅)
    send M to dest using NICs in S
end
```

the status of each NIC in a specific node. The state is updated by the NICs and can be RESERVED or FREE. The subset of free NICs which is selected depends on the desired striping ratio. This parameter fixes the number of free rails which is used to send a single message (striped in the appropriate number of fragments). Its value ranges between 0 (only one rail is selected) and 1 (all the available rails are chosen). The striping ratio is handled with the *Select_Tx_NICs* function, which employs a round-robin algorithm to ensure fairness when selecting a subset of the free NICs. The allocation of the NICs starts at the first free NIC just past the last one allocated in the previous transaction.

4 Dynamic Allocation

The dynamic allocation algorithm collects local- and remote-state information from the NICs for every communication operation. Its main goal is to guarantee that both the sending and the receiving side are free before injecting a message. This ensures unidirectional traffic at both ends.

In the dynamic allocation algorithm, we use two types of communicating processes. The first, the PE (processing element) process, is integrated with the underlying communication library and is run at user level by all the processes of a parallel job. The second runs on the NIC processors and handles local and remote requests. It should be noted that this distributed algorithm runs on every PE and NIC in the cluster.

4.1 PE process

This process, shown in Algorithm 4, runs on the PEs and is invoked when a message is sent. Rail reservation is employed prior to sending so that the network interfaces at source and destination are dedicated to unidirectional traffic at both ends. This reservation is performed by the sender in the following way: if local NICs are available, each request is temporarily assigned to all the available NICs. Then a *Request To Send* (RTS) is sent to the destination NICs (one destination NIC for each source NIC) to check for availability and reserve them. Destination NICs reply with a *Clear To Send* (CTS) if free and a *Negative Acknowledgment* (NACK) otherwise. Once the set of available paths (rails) is known at the sender side, another selection is done (by the *Select_Tx_NICs* function) in order to choose the actual set of rails for sending, based on the desired striping ratio. Rails initially allocated that are not eventually used are freed by sending an ABORT command. When the message is successfully delivered, the destination process sends a local ACK to its NIC, which on its turn forwards a remote ACK to the source NIC. A round-robin algorithm is used to guarantee a fair selection of NICs. Finally

Algorithm 4 : Dynamic Allocation (PE process)

```
Procedure Dynamic_Allocation_PE
Input: message (M), destination node (dest), striping ratio (str_r)
begin
  repeat
      F ← {n | Nstatus[n]==FREE}
      send local_RTS to the NICs in F
      Wait until all remote NICs reply or a timeout expires
      A ← {The set of NICs that replied with a CTS}
  until (A≠∅)
  S ← Select_Tx_NICs(A,str_r)
  Deallocate all NICs in A\S, sending an ABORT.
  send M to dest using NICs in S
end
```

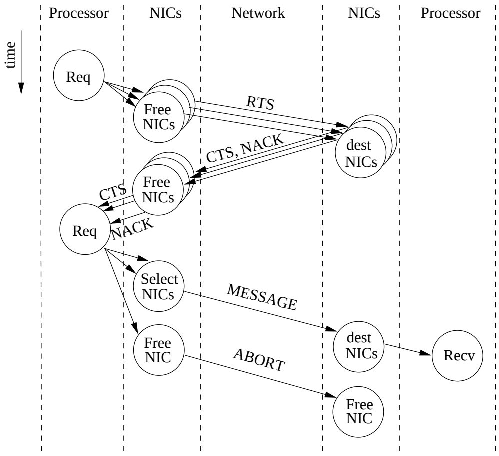


Figure 5: Dynamic allocation operation when selecting more than one NIC.

the message is striped, if possible, and sent over the selected set of NICs. A visual representation of the algorithm is depicted in Figure 5.

4.2 NIC Process

This process, shown in Algorithm 5 and Table 1, runs on the NIC and handles the requests issued by local and remote processors. As in the local-dynamic algorithm, we use a data structure (*NStatus*) containing the status of each NIC in a given node. In this case the status, which is only updated by the NICs, can be one of the following:

- FREE - the NIC is available.
- RESERVED - the NIC is reserved by a local requester, while trying to allocate the destination NIC.

Event\Status	Free	Receiving	Sending	Reserved	Receiving & Out_RTS
Local RTS	Remote RTS Reserved	Local NACK Receiving	Local NACK Sending	Local NACK Reserved	Local NACK Receiving & Out_RTS
Local ACK		Remote ACK Free			Remote ACK Reserved
Local ABORT			Remote ABORT Free		
Remote RTS	Remote CTS Receiving	Remote NACK Receiving	Remote NACK Sending	– *call livelock_avoidance*	Remote NACK Receiving & Out_RTS
Remote ABORT		– Free			– Reserved
Remote CTS	Local CTS Sending	Rmt ABORT & Lcl NACK Receiving		Local CTS Sending	
Remote ACK			– Free		
Remote NACK	Local NACK Free	Local NACK Receiving	Local NACK Sending	Local NACK Free	Local NACK Receiving

Table 1: Dynamic allocation - NIC process state table. The first row in each cell represents the message(s) to be sent and the second row represents the new state.

Event\Status	Reserved & Livelock
Local Winner	Send: Remote NACK; c=c-1 Reserved
Remote Winner	Send: Remote CTS; c=c+1 Receiving & Out_RTS

Table 2: Livelock avoidance state table.

- RECEIVING - the NIC is receiving a message.
- RECEIVING and Out_RTS - the NIC is receiving a message and has an outstanding RTS message.
- SENDING - the NIC is sending a message.

When a remote RTS is received and the NIC is free, the NIC is assigned to the requester and a CTS is issued. The requester can either use the reserved path to send a message or abort it. If the NIC is not free, a NACK is sent to the requester.

With regard to the local requests, if a local RTS is received and the NIC is free, it is assigned to the local requester and a remote RTS is sent to the destination NIC. If a CTS is received from the remote NIC (the path has been granted), a local ACK is sent to the local requester that decides whether to use the reserved path (sending a message) or to dismiss it (sending an ABORT). That depends on the applied striping ratio as stated in 4.1.

This procedure can livelock if a cyclic dependency is established between different NICs. As an example, let us suppose that each NIC in Figure 6(a) sends a request to another NIC so that a cycle of dependencies is generated. In this scenario, each NIC receives a request while having an outgoing request pending. Consequently, using the algorithm described above, every NIC sends a NACK (the NICs are busy as they have outgoing pending requests) and then all three NICs retry the connection. This leads to a livelock if no other mechanism is implemented.

In order to deal with this problem, a livelock-avoidance mechanism has been developed and included in Algorithm 6. For the sake of clarity, this mechanism is shown in

Algorithm 5 : Dynamic Allocation (NIC process)

```
Procedure Dynamic_Allocation_NIC
begin
  NStatus[i] ← FREE
  counter ← 0   {for livelock avoidance}
  while TRUE  { repeat forever }
    case event of
      local_RTS:
        if (NStatus[i]==FREE) then
            NStatus[i] ← RESERVED
            send RTS to remote node
        else
            send NACK to local process

      remote_CTS:
        if ((NStatus[i]==RESERVED) OR (NStatus[i]==FREE)) then
            NStatus[i] ← SENDING
            send CTS to local process
        else if (NStatus[i]==RECEIVING) then
            send NACK to local process
            send ABORT to remote node

      remote_RTS:
        if (NStatus[i]==FREE) then
            NStatus[i] ← RECEIVING
            send CTS to remote requester
        else if (NStatus[i]==RESERVED) then
            call livelock_avoidance
        else
            send NACK to remote requester

      local_ACK:
        if (NStatus[i]==RECEIVING) then
            NStatus[i] ← FREE
        else if (NStatus[i]==RECEIVING AND Outstanding_RTS) then
            NStatus[i] ← RESERVED
        send ACK to remote requester

     remote_ACK:
        if (NStatus[i]==SENDING) then
            NStatus[i] ← FREE

     remote_NACK:
        if (NStatus[i]==RESERVED) then
            NStatus[i] ← FREE
            send NACK to local process
        else
            send NACK to local process

     local_ABORT:
        if (NStatus[i]==SENDING) then
            NStatus[i] ← FREE
            send ABORT to remote requester

     remote_ABORT:
        if (NStatus[i]==RECEIVING AND Outstanding_RTS) then
            NStatus[i] ← RESERVED
        else if (NStatus[i]=RECEIVING) then
            NStatus[i] ← FREE
end
```

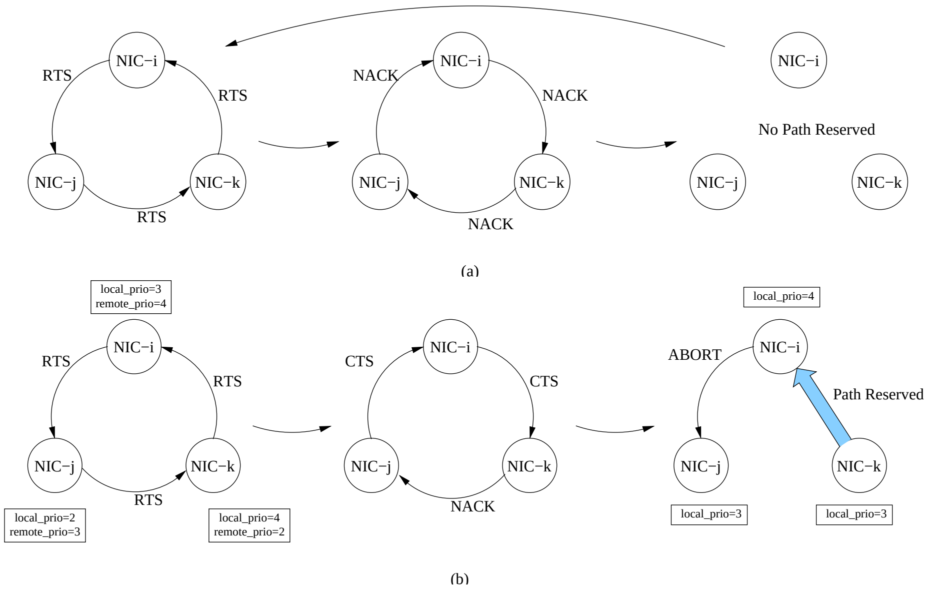


Figure 6: Livelock example.

Algorithm 6 : Livelock avoidance procedure

```
Procedure livelock_avoidance
begin
  if ((local_counter>remote_counter) OR
      ((local_counter==remote_counter) AND
       (local_node_id>remote_node_id))) then   { local request receives priority: }
     counter ← counter - 1
     send NACK to remote requester
     NStatus[i] ← RESERVED
  else                       { remote receives priority }
     counter ← counter + 1
     send CTS to remote requester
     NStatus[i] ← RECEIVING & OUTSTANDING_RTS
end
```

a separate procedure (Algorithm 6 and Table 2). This priority-based algorithm is run by each NIC whenever a livelock is possible, which is every time an incoming request is received while an outgoing request is pending. At initialization time, every NIC is assigned a default priority level. Each time a potential livelock is detected the priorities of the remote NIC (incoming request) and the local NIC (outgoing request) are compared. The request with lower priority is aborted. If the priorities are identical, the identifiers of the local and remote node are used instead. Finally, in order to ensure fairness, the local priority is updated in the following way: if the local request wins, the local priority is decremented, otherwise it is incremented.

An example is shown in Figure 6(b). In this example, the potential livelocked situation appears when each node sends an outgoing request, and while this is still pending, it receives an incoming one. NIC-i and NIC-j have lower priority than the source NIC of their incoming requests (NIC-k and NIC-i, respectively), so they send a CTS to the requester NICs. On the other hand, NIC-k has a higher priority than its requester NIC (NIC-j), so it sends it a NACK. Eventually, every NIC receives a reply. NIC-i receives a CTS and rejects it since it has granted a connection to the higher priority NIC-k. NIC-j receives a NACK for its request and ignores it since it has been previously granted a path to NIC-i. NIC-k receives a CTS which grants it the path for the requested sending. Finally, NIC-j receives an ABORT from NIC-i and becomes free again. NIC priorities are updated as stated above, NIC-i and NIC-j increment their priorities, and NIC-k decrements its one. All the possible states and transitions are depicted in Tables 1 and 2.

4.3 Hybrid algorithm

The rail reservation protocol employed by the dynamic algorithm incurs an overhead for every message sent. For short messages, this overhead could become significant, compared to the time it takes to send the message. We therefore implemented a third, hybrid approach, shown in Algorithm 7. The status of the NIC is not modified when sending short messages, thus, additional messages might be simultaneously received. Moreover on the network side of the NIC an incoming short message is always accepted even if the NIC is sending another message. These messages may cause bidirectional traffic during short periods of time (the time needed to send or receive a short message, in the worst case). A short message is never striped, since the striping overhead is not justified in this case. Rather, it is sent on a single rail which is chosen in a round-robin fashion to ensure fairness.

Algorithm 7 : Hybrid allocation (PE process)

```
Procedure Hybrid_Allocation_PE
Input: message (M), destination node (dest), striping ratio (str_r)
begin
  if |M| ≤ SHORT_MESSAGE_LENGTH then
     F ← {n|Nstatus[n]=FREE}                    { Set of free NICs }
     select s∈F using round-robin
     send M to dest using NIC s
  else
     call Dynamic_Allocation_PE (M, dest, str_r)
end
```

The threshold used by the algorithm to distinguish between long and short messages is an important parameter. This value has to be carefully selected to provide the best performance. If the value is too small, the dynamic algorithm could be applied to messages for which striping and guaranteed unidirectional bus traffic would not be effective. If too large, the allocation policy approximates the basic algorithm. Several experiments have been carried out in order to analyze the influence of this parameter on network performance and determine its optimal value, and the results are shown in Section 6.

5 Experimental Framework

This section offers details on our simulation platform, the workloads that were simulated, and the metrics of interest.

5.1 Simulation model

In the experimental evaluation, we focus our attention on a family of fat-tree interconnection networks, ranging from 32 to 128 SMPs, with four processors per SMP. Unless otherwise stated, a configuration with 4 rails is used. Since the performance bottleneck is usually the PCI bus, the network topology is not relevent and similar results are expected for other topologies. The simulation model tries to capture the most important characteristics of the QsNet at the granularity of the clock cycle. The simulator models wormhole flow-control, with two virtual channels on each physical channel. The input buffers on each virtual channel can contain up to 128 flits [4], each consisting of two bytes. A flit can be transmitted over a physical channel in a single clock cycle, while a packet can be routed through an Elite switch in six clock cycles.

The simulator also models a thread processor in the NIC, which can process incoming control and data packets and can send a reply in a few hundreds of clock cycles. Another important characteristic is the unidirectionality of the I/O bus, which can transmit data in one direction at a time. We also assume that the bus bandwidth is equalized with the external network bandwidth (an optimistic set of assumptions, given the current state of the art).

This model is evaluated in the SMART (Simulator of Multiprocessor ARchitectures and Topologies) environment [9]. Implemented in C++, SMART is an object-oriented, discrete-event simulation tool for evaluating parallel architectures and high performance interconnection networks.

5.2 Communication patterns

In our model each process generates packets independently, using three random variables:

- the message size, which is exponentially distributed with a given mean value,
- the inter-arrival time, also exponentially distributed around a given mean value,
- and the destinations, which are randomly chosen with equal probability between the processes.

We consider a set of communication algorithms, including a baseline *basic algorithm*, and the dynamic algorithms described in Sections 3 and 4. The basic algorithm does not use any protocol; whenever a node needs to send a message, it sends it on one rail, choosing it in round-robin fashion. This base case can serve to illustrate the effects of both the overhead of other protocols and the penalties of bidirectional traffic.

5.3 Metrics

The performance of an interconnection network under dynamic load is usually assessed by two quantitative parameters, the *accepted bandwidth*, or *throughput*, and the *latency*. Accepted bandwidth is defined as the sustained data delivery rate given some offered bandwidth at the network input. Two important characteristics are the saturation point and the sustained rate after saturation. Saturation is defined as the minimum offered bandwidth where the accepted bandwidth is lower than the global packet creation rate at the source nodes. It is worth noting that, before saturation, offered and accepted bandwidth are the same. The behavior above saturation is important because the network and/or the allocation algorithms can become unstable, leading to a sharp performance degradation. We usually expect the accepted bandwidth to remain stable after saturation, for example in the presence of burst-mode applications that require peak performance for a short period of time [5].

The experimental results of each traffic are presented using two graphs, one to display the accepted bandwidth and the other to display the network latency. In both graphs, the x-axis corresponds to the offered bandwidth normalized with the unidirectional bandwidth of the links connecting the processing nodes to the network switches. This makes the analysis independent of the link bandwidth and the flit size.

We report the latency in cycles rather than absolute time, in order to make our analysis insensitive to technological changes. Given that the I/O bus in the network interface can only allow unidirectional traffic, the maximum achievable throughput under uniform traffic is only 50% of the nominal injection bandwidth. The intuition behind this limit is the following: let us consider for example a cluster with only two SMPs and single network rail; under uniform traffic, only one SMP can send to another at any given time, due to the unidirectionality constraint in the endpoints.

6 Experimental Results

In this section, we try to provide insight into some important aspects of the multirail allocation algorithms. We first study the impact of network load, message size, and striping on the basic and dynamic algorithms using 4 rails. Then, we analyze how the algorithms perform when the number of nodes and the number of rails are scaled up, and we integrate these results in the evaluation of the hybrid algorithm.

6.1 Bandwidth and latency

The following results were obtained by simulating 128 SMPs (nodes), four rails and four PEs per SMP. Figures 7-10 compare the accepted bandwidth and network latency as a function of the offered bandwidth. Two different values for the average packet size, 4

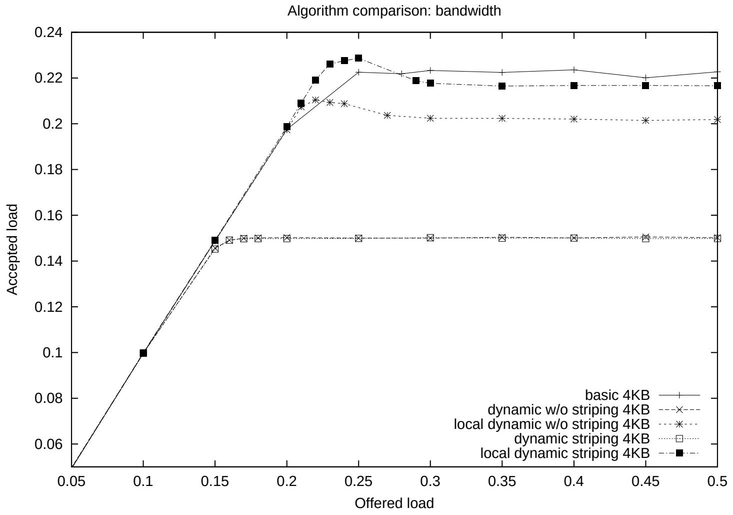


Figure 7: Bandwidth for 4KB average message size.

and 64 KB, are compared in the experiments. These graphs show the performance for the basic, local-dynamic and dynamic algorithms. Given that the message latency goes to infinity after the saturation point, we only report the latency when the network is not saturated.

We can see that the basic algorithm performs relatively well on short messages, but its performance decreases as the message size increases. The dynamic algorithm behaves in the opposite manner, performing poorly on short messages, and increasing in performance as the message size grows. The local-dynamic algorithm exhibits similar performance to the basic algorithm (although it achieves lower latencies when striping is used), performing better than the basic for larger messages and worse for shorter messages. This suggests that we may benefit from using the hybrid approach, where short messages are sent using the basic protocol and long messages using the dynamic protocol.

6.2 Effect of striping

Figures 7-10 also depict the effect of message striping whenever possible in the dynamic and local dynamic approaches. Figure 11 shows the latency vs. the average message size for an offered load of 0.15. We used an aggressive approach for striping, using only full striping and no intermediate values. Results not shown here indicate that it is always best to stripe as much as possible.

Striping does not seem to have a significant effect on any protocol's accepted bandwidth. However, it does reduce the latency of sending messages, especially as the message size grows (which makes the striping overhead less significant) and load diminishes (which allows a higher effective striping ratio). It can be seen, for example, that for an average message size of 64KB and a load of 5% (Figure 10), striping reduces the dynamic and local-dynamic latencies by approximately 65% and 72% respectively.

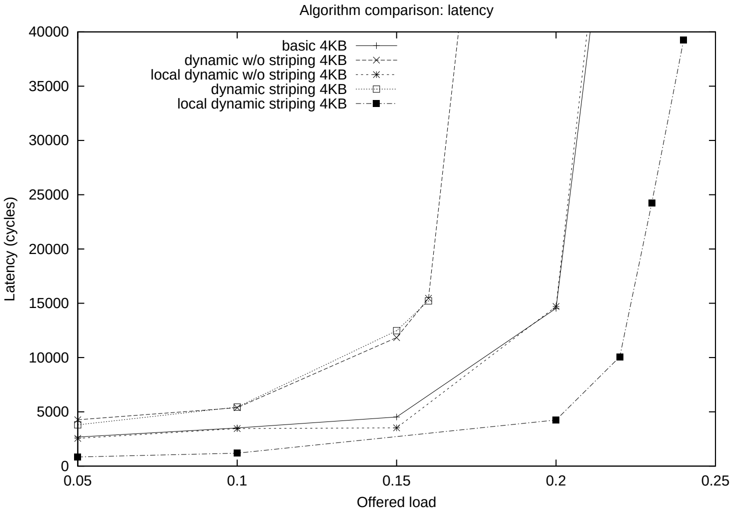


Figure 8: Latency for 4KB average message size.

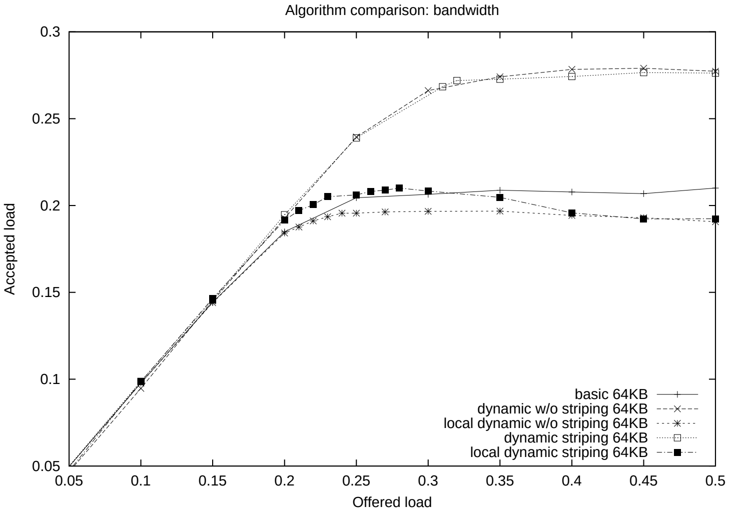


Figure 9: Bandwidth for 64KB average message size.

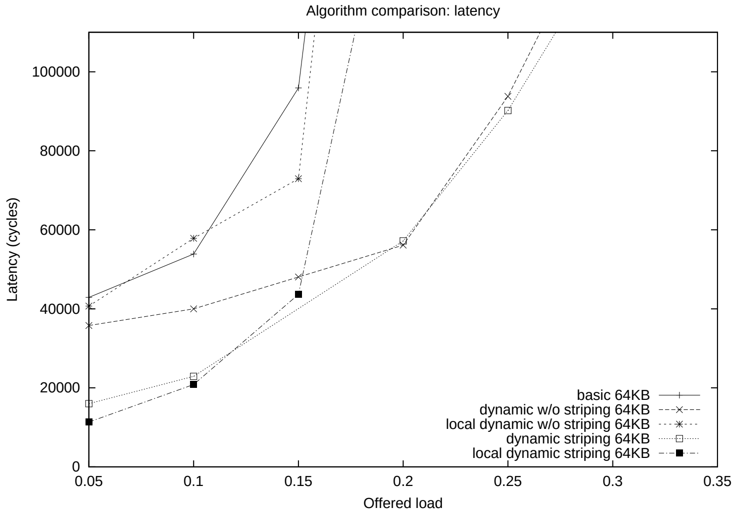


Figure 10: Latency for 64KB average message size.

The better local-dynamic results at low loads arise from the lower overhead associated with sending a message (there is no path reservation in the local-dynamic algorithm). At higher loads (above 15%) the dynamic approach outperforms the local-dynamic because the protocol overhead is compensated with the low latency provided by the reserved path. In both cases, striping is useful with low loads that offer a high probability that rails will be free.

From Figure 11 it can be seen that for short average message sizes (below 16KB) the best results are obtained with the local-dynamic allocation algorithm with full striping. For longer messages the dynamic allocation algorithm with full striping provides the best performance.

6.3 Node scalability

The effect of increasing the number of nodes on the maximum accepted load is shown in Figure 12 for an average message size of 32KB. The dynamic algorithm outperforms the basic algorithm by 36% for 32 nodes and 29% for 128 nodes. These algorithms scale reasonably well, with a loss of 7%-12% in maximum accepted bandwidth when the network size is quadrupled from 32 to 128 nodes.

6.4 Rail scalability

In order to understand the behavior of the algorithms as a function of the number of rails, we tested configurations of one, two, and four rails with 32 nodes, each having four PEs, and using average message sizes in the range 1KB-256KB. The results are desplayed in Figure 13.

For the dynamic allocation we show full striping only, since the maximum bandwidth is hardly affected by striping (due to the low probability of reserving more than one rail

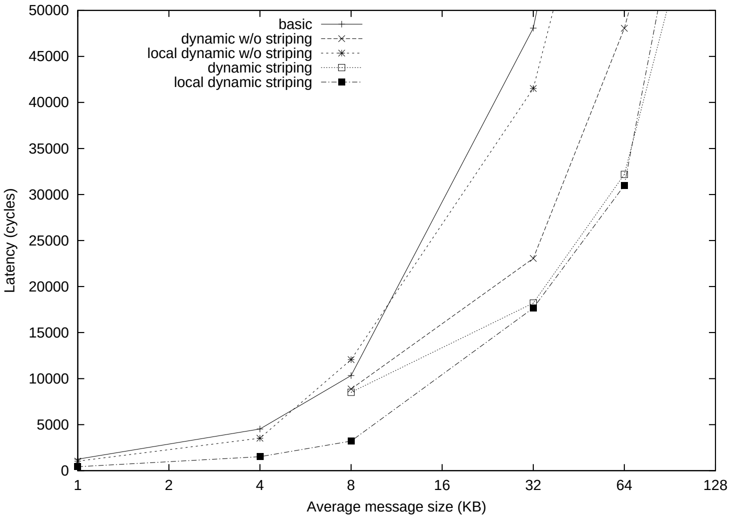


Figure 11: Latency vs. message size with an offered load of 0.15. The latency for messages shorter than 8KB is not shown for the dynamic allocation because the network is saturated with this load.

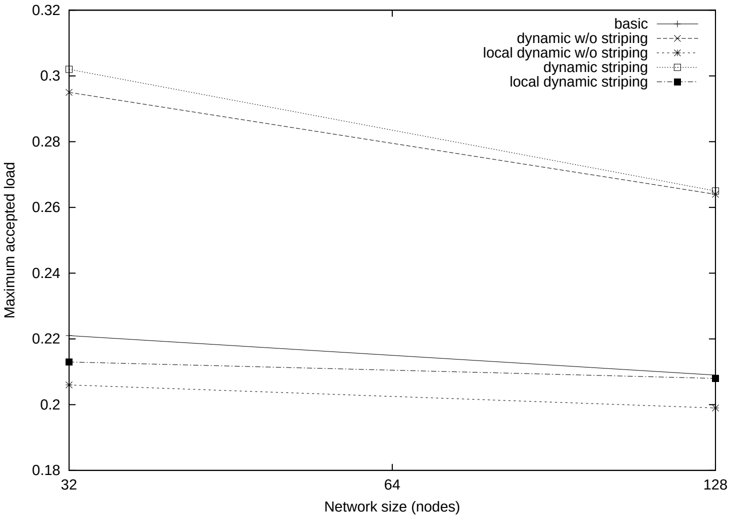


Figure 12: Maximum accepted load vs. network size using 4 rails and average message size of 32KB.

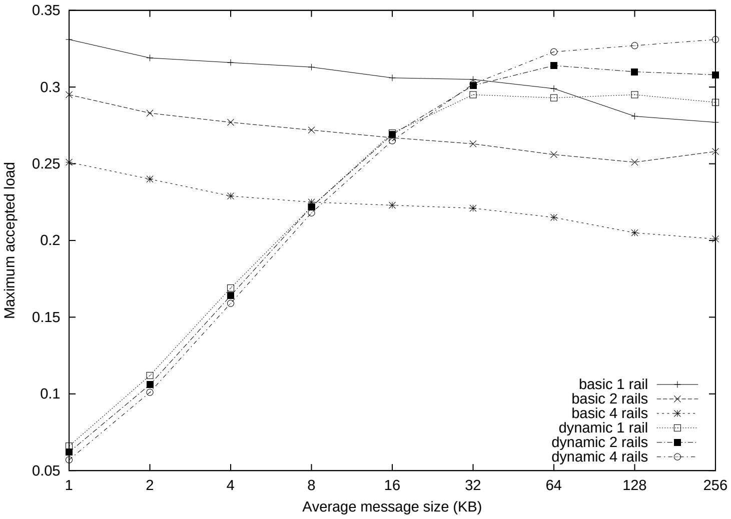


Figure 13: Maximum accepted load vs. average message size for 32 nodes.

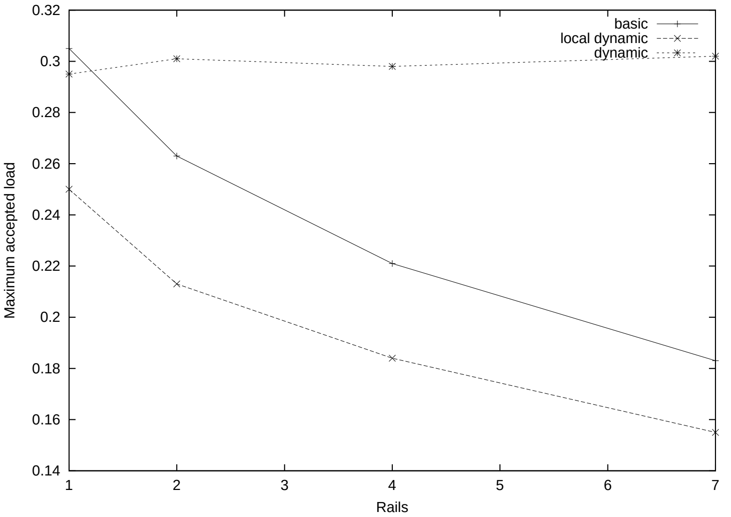


Figure 14: Maximum accepted load vs. number of rails for 32 nodes and average message size of 32KB.

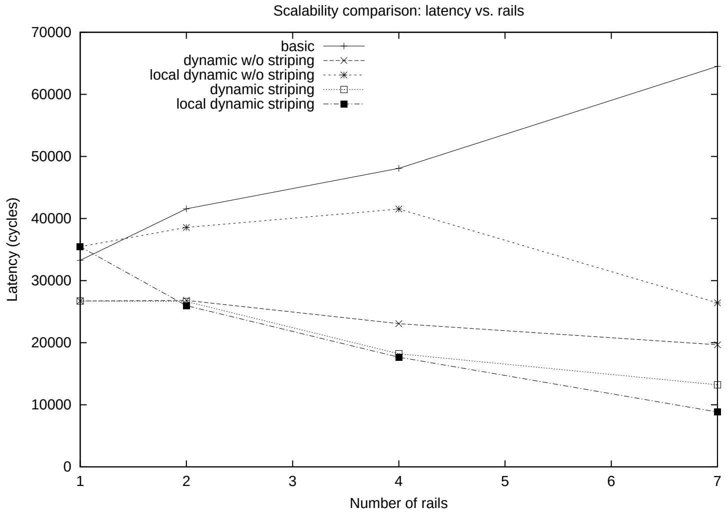


Figure 15: Scalability analysis for latency with injection load of 0.15 and average message size of 32KB.

for high injection rates). The offered load is normalized by dividing it by the number of rails, so that the resource requirement matches the increase in available resources, thus giving a clearer view of the network's scalability. Again, we see the dynamic algorithm's performance increasing with message size, for any number of rails, while the basic algorithm's performance decreases, this result supporting the idea of a hybrid approach. More importantly, we see that the maximum bandwidth obtained using the dynamic algorithm is almost constant for any number of rails (and even improves when adding more rails, for messages larger than 16KB). This can be clearly seen in Figure 14 which shows the maximum accepted load vs. number of rails (up to seven) for an average message size of 32KB. This graph confirms that the dynamic allocation algorithm slightly improves its bandwidth when the number of rails is increased. On the other hand, the basic algorithm degrades significantly when compared with the single-rail configuration (a 40% bandwidth reduction in the maximum accepted load with seven rails when compared to the single-rail topology). The reason for this is that as the number of rails grows, so does the average sending load of each processor (the number of processors is fixed). The basic approach uses a round-robin rail selection method, ignoring the state of the NICs. It therefore becomes more probable for the processors to self synchronize the choice of the rails, leading to a performance loss.

In Figure 15 we can observe the effect of the number of rails on latency. The data were obtained from experiments with an injection load of 0.15, using 32 nodes (four PEs per node) and an average message size of 32KB. The basic algorithm's latency actually increases with the number of rails, due to the inefficiency of the round-robin method, as discussed above. This is confirmed in the simulation traces that show the injection latency to be the source of the latency growth. As expected, striping reduces the latency when the number of rails is increased for the dynamic algorithms, with an advantage to the local-dynamic algorithm. It is interesting to note that even with no striping, both

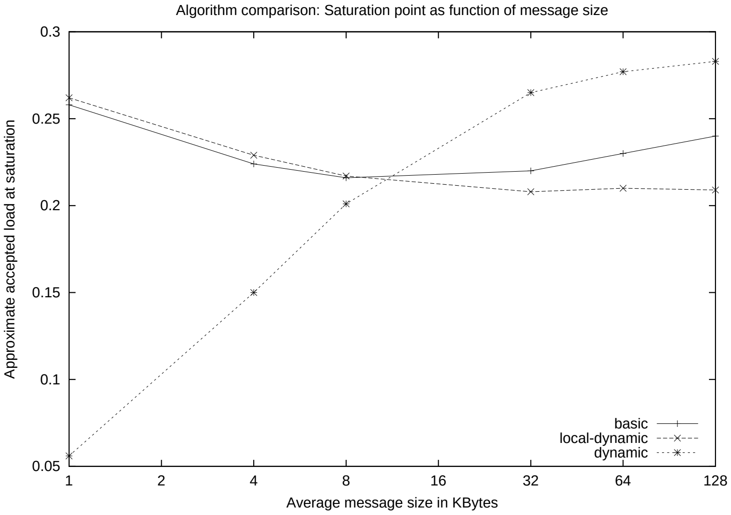


Figure 16: Saturation point as a function of message size.

dynamic algorithms scale well with the number of rails.

6.5 Effect of message size on saturation point

Another important feature of the allocation algorithms is the saturation point for different message sizes. The experimental data set that was used to obtain the saturation points for each message size is the same as in 6.1. The results are shown in Figure 16.

We can see that the dynamic algorithm's saturation point increases with the message size, while the basic and local-dynamic algorithms retain a near-constant saturation point. These results suggest that the dynamic algorithm scales better with the message size than do the other two. One possible explanation for this is that the dynamic algorithm ensures that no conflicts will occur on any rail. These conflicts are more likely as the message size increases and rails are unavailable for longer periods of time.

6.6 Hybrid approach

The results observed in 6.1 and 6.5 indicate that the basic algorithm performs better on shorter messages, while the dynamic algorithm performs better on longer messages. It may therefore be useful to try a hybrid approach, that uses the basic algorithm for messages shorter than a given threshold, and the dynamic algorithm otherwise. (This threshold is implemented in Algorithm 7 as SHORT_MESSAGE_LENGTH).

Several short message size thresholds were tested and compared in the dynamic and basic algorithms. We used 128 nodes of four PEs each with four rails, an average message size of 32KB, and short message size thresholds of 1, 4, 8, 16, and 32 KB. Figures 17-20 show the bandwidth and latency obtained with and without striping.

It can be clearly seen from these results that the hybrid approach outperforms both the dynamic and the basic approaches in terms of bandwidth for almost all the thresholds

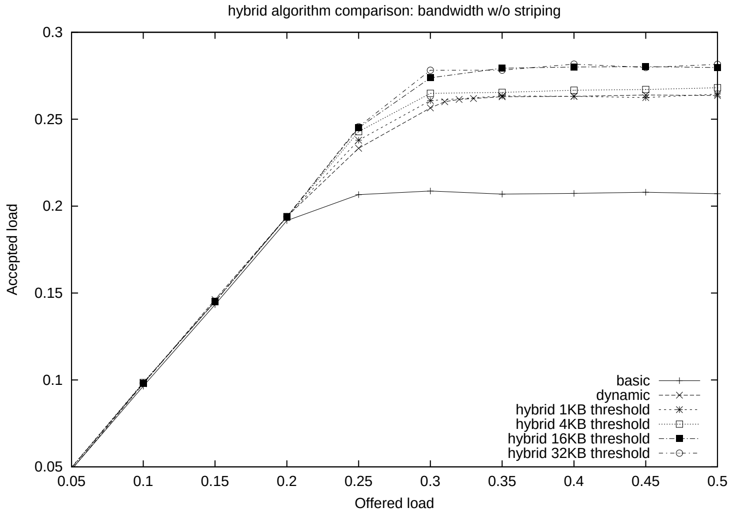


Figure 17: Hybrid bandwidth without striping.

chosen, regardless of striping. With the exception that at a threshold of 32KB, hybrid performs somewhat worse than dynamic for low injection rates when striping is used, hybrid otherwise outperforms both the dynamic and basic approaches for latency, regardless of striping, similar to the observation for bandwidth. This may stem from the fact that messages shorter than the threshold are sent with no striping (as in basic), so the latency for relatively large messages can be lower if striping is used (Figure 20). On the other hand, when no striping is used, the dynamic algorithm performs worse than the hybrid methods for low injection rates, and almost the same or better for higher injection rates. This can be explained by the fact that the dynamic approach has a larger saturation point for average message size of 32KB than the basic approach (see 6.5), and the hybrid approach uses the basic algorithm for short message sizes.

7 Conclusions

One of the novel methods that can be used to increase communication performance and enhance fault tolerance in a cluster of workstations is to use parallel independent networks (rails). In this paper, we explored various aspects of multirail interconnects and presented several rail allocation algorithms for efficient usage of the rails. We have shown that the dynamic algorithm can perform relatively well in terms of bandwidth for sufficiently large message sizes, and can handle a relatively high load before saturating. Furthermore, it has been shown that this algorithm is scalable due to its adaptive nature - increasing the number of rails from one to seven increases the maximum relative bandwidth in a linear manner. Superlinearity is achieved for messages larger than 8KB. Furthermore, the bandwidth increases as the message size increases, unlike the case for other approaches. Incorporating protocol-free short message handling was shown to increase the maximum bandwidth by up to 7.5% more than the pure dynamic algorithm, and up to 36.6% and 48.7% more than the basic and local-dynamic approaches respec-

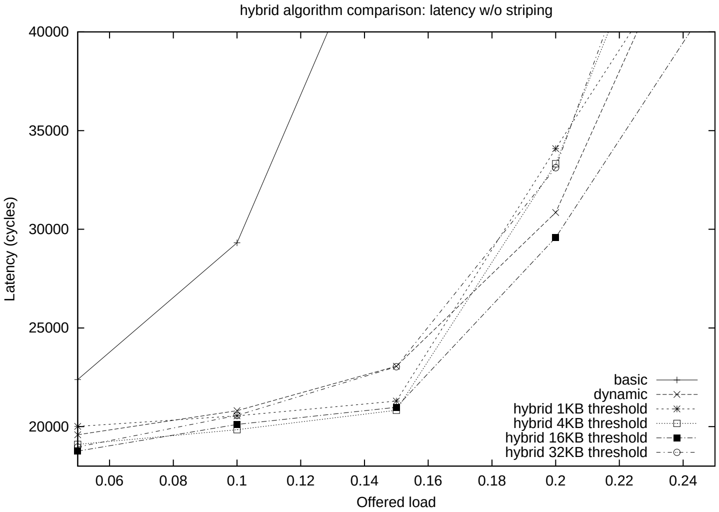


Figure 18: Hybrid latency without striping.

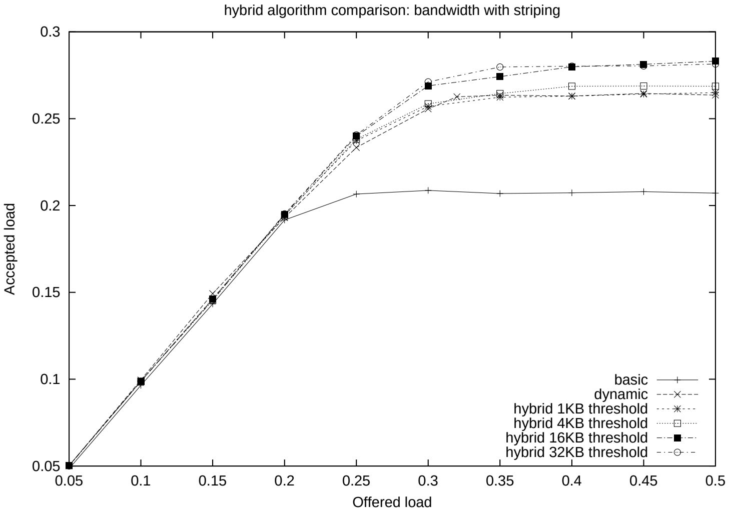


Figure 19: Hybrid bandwidth with striping.

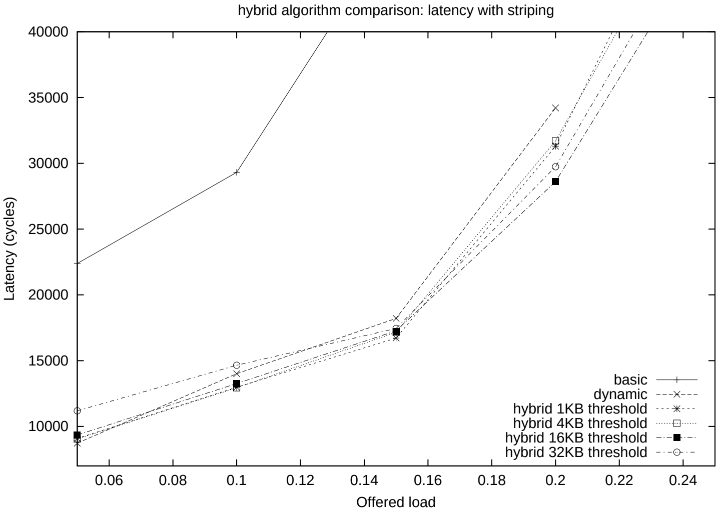


Figure 20: Hybrid latency with striping.

tively. We have also shown that striping a message over several rails can be used to obtain a significant reduction of latency when load is low.

Acknowledgements We thank José Duato for spearheading the project, for pointing out the limitations of the static approach and suggesting the dynamic allocation strategy as a promising venue of research.

References

[1] Daniel Cassiday. Infiniband architecture tutorial. Hot Chips 12 Tutorial, August 2000.

[2] Salvador Coll, Eitan Frachtenberg, Fabrizio Petrini, Adolfy Hoisie, and Leonid Gurvits. Static Allocation of Multirail Networks. Technical report, Los Alamos National Laboratory, Los Alamos Unclassified Report 01-3896, 2001.

[3] Salvador Coll, Eitan Frachtenberg, Fabrizio Petrini, Adolfy Hoisie, and Leonid Gurvits. Using Multirail Networks in High Performance Clusters. In *Third IEEE International Conference on Cluster Computing (Cluster'01)*, Newport Beach, CA, USA, October 2001.

[4] William J. Dally. Virtual Channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, March 1992.

[5] José Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks: an Angineering Approach*. IEEE Computer Society Press, 1997.

[6] Charles E. Leiserson. Fat-Trees: Universal Networks for Hardware Efficient Supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, October 1985.

[7] D. Lubell. A short proof of Sperner's theorem. *Journal of Combinatory Theory*, 1(299), 1966.

[8] Fabrizio Petrini, Adolfy Hoisie, Wu chun Feng, and Richard Graham. Performance Evaluation of the Quadrics Interconnection Network. In *Workshop on Communication Architecture for Clusters (CAC '01)*, San Francisco, CA, April 2001.

[9] Fabrizio Petrini and Marco Vanneschi. SMART: a Simulator of Massive ARchitectures and Topologies. In *International Conference on Parallel and Distributed Systems Euro-PDS'97*, Barcelona, Spain, June 1997.

[10] Quadrics Supercomputers World Ltd. *Elan Reference Manual*, January 1999.

[11] Quadrics Supercomputers World Ltd. *Elite Reference Manual*, November 1999.