# Scalable Hardware-Based Multicast Trees*

Salvador Coll[1] José Duato[2] Fabrizio Petrini[3]
Francisco J. Mora[1]

[1]Digital Systems Design (DSD)
[2]Parallel Architectures Group (GAP)
Technical University of Valencia
Aptdo. 22012 Valencia 46071, SPAIN
{scoll,fjmora}@eln.upv.es, jduato@gap.upv.es

[3]Performance and Architecture Laboratory (PAL)
Computer and Computational Sciences (CCS) Division
Los Alamos National Laboratory, NM 87545, USA
fabrizio@lanl.gov

**Abstract**

This paper presents an algorithm for implementing optimal hardware-based multicast trees, on networks that provide hardware support for collective communication. Although the proposed methodology can be generalized to a wide class of networks, we apply our methodology to the Quadrics network, a state-of-the-art network that provides hardware-based multicast communication. The proposed mechanism is intended to improve the performance of the collective communication patterns on the network, in those cases where the hardware support can not be directly used, for instance, due to some faulty nodes. This scheme provides significant reduction on multicast latencies compared to the original system primitives, which use multicast trees based on unicast communication. A backtracking algorithm to find the optimal solution to the problem is presented. In addition, a greedy algorithm is presented and shown to provide near optimal solutions. Finally, our experimental results show the good performance and scalability of the proposed multicast tree in comparison to the traditional unicast-based multicast trees. Our multicast mechanism doubles barrier synchronization and broadcasts performance when compared to the production-level MPI library.

## 1 Introduction

High-performance and scalable collective communication is an important factor to achieve good performance and improve resource utilization of a parallel computer. On the one hand, this becomes particularly important if it is considered that scientific codes spend a considerable part of their run time, in some cases up to 70%, executing collective communication [14]. On the other hand, resource management greatly conditions the efficient utilization of a parallel machine. Many recent research results show that job scheduling algorithms can substantially

*The work was supported by the Spanish CICYT through contract TIC2000-1151-C07-05



SC'03, November 15–21, 2003, Phoenix, Arizona, USA

improve scalability, responsiveness, resource utilization, fault tolerance and usability of large-scale parallel computers [1, 8, 11]. In [9] it is shown that, using an optimized implementation of a small set of mechanisms, typical resource management operations can be significantly improved (i.e. scheduling two orders of magnitude faster than the best previously reported results). Those mechanisms, which are soon to be common in modern high-performance interconnects, rely on a number of hardware features provided by the underlying network. Among them, high-performance hardware-based collective communication primitives have been proven to be particularly important [9].

The Quadrics interconnection network (QsNET) [12], is being used by some of the most powerful computers in the world (six among the top 10 in the Top500 list[1]). As shown in [13], although the QsNET shows an outstanding collective communication performance, it is limited to the case where the destination nodes are consecutive. A single gap in the destination nodes for a collective communication, i.e. due to a single faulty node or to a job scheduling decision, makes the hardware support for collective communication completely unusable. A unicast-based multicast is used instead, with a significant performance penalty (two times slower barrier, and eight times slower broadcast on a 32-node cluster [4]). This effect increases with the network size, since the hardware mechanisms provide very good scalability while the software-based primitives show a logarithmic performance degradation.

The above limitation becomes particularly relevant if we consider that clusters of workstations with thousands of nodes are becoming increasingly popular as high-performance computing platforms. Currently, 6 machines among the top 10 in the Top500 list are clusters. Those systems, with so many components, have a MTBF which is proportionally reduced as the number of components increases. As an example, in [10] it is shown that fault-tolerance becomes a key factor in systems such as BlueGene/L, with 64K processors.

The above scenario shows, on the one hand, that high-performance and scalable collective communication support is becoming increasingly important in the high-performance computing arena. On the other hand, the network performance must not degrade drastically when the nodes allocated to a job are not consecutive; i.e. due to the presence of faults or when the machine is fragmented after having been allocated to multiple parallel jobs.

In order to overcome these constraints, a new multicast mechanism, which is intended to enhance multicast transmissions in those cases where the hardware support is not directly usable, is presented in this paper. This technique uses a multicast tree to distribute the data among all the destinations, but with the particular feature that each message is a hardware multicast. Several open problems have been addressed: first, multiple broadcasts in parallel may lead to a deadlock [7]; and, second, the network serializes all the broadcasts through a tree [13]. On the other hand, it is shown that our strategy is likely to provide performance figures very close to those provided by the original hardware mechanisms, significantly outperforming the behavior of the unicast-based multicast when the number of destinations is high.

The rest of the paper is organized as follows. Section 2 presents the point-to-point routing mechanisms used by the QsNET, while Section 3 describes the network support for collective communication. Section 4 presents the hardware-based multicast trees developed by us and Section 5 proposes two algorithm implementations to obtain multicast trees. Section 6 discusses the experimental results and provides insight onto several aspects of the proposed algorithms. Finally, in Section 7 some conclusions are drawn and future directions are given.

[1] http://www.top500.org/list/2003/06/

## 2 Routing issues

The Quadrics network is a butterfly bidirectional multistage interconnection network based on 4 x 4 switches (called Elite), which can be viewed as a quaternary fat-tree. A quaternary fat-tree belongs to the more general class of the k -ary n -trees [15]. A quaternary fat-tree of dimension $n$ is composed of $4^n$ processing nodes and $n \cdot 4^{n-1}$ switches interconnected as a delta network, and can be recursively built by connecting four quaternary fat trees of dimension $n-1$. It uses wormhole switching, with two virtual channels per physical link, source-based routing, and adaptive routing. Some of the most important aspects of this network are: the integration of the local memory (either in the NIC or in the host) into a distributed virtual shared memory, the support for zero-copy remote DMA transactions and the hardware support for collective communication.

The routing information is attached to the header before injecting the packet into the network and is composed of a sequence of switch link tags. As the packet moves inside the network, each switch removes the first routing tag from the header, and forwards the packet to the next switch in the route or to the final destination. The routing tag can identify either a single output link (used for unicast communication) or a group of adjacent links (used for multicast communication).

The transmission of each packet is pipelined into the network using wormhole switching. At link level, each packet is partitioned in smaller units called flits (flow control digits) [5] of 16 bits. The header flit opens a circuit between source and destination, and this path stays in place until the destination sends an acknowledgement to the source. At this point, the circuit is closed by sending an EOP token.

Minimal routing between any pair of nodes is accomplished by sending the message to one of the nearest common ancestor switches and from there to the destination. In this way, each packet experiences two routing phases: an adaptive ascending phase (in forward direction) to get to a nearest common ancestor, where the switches forward the packet through the least loaded link; and a deterministic descending phase (in backward direction) to the destination.

## 3 Collective communication on the Quadrics network

The hardware multicast capability of the network allows packets to be sent to multiple destinations. The Elite switches can forward a packet to several output ports, with the only restriction that these ports must be contiguous. In this way, a group of adjacent nodes can be reached by using a single hardware multicast transaction.

The routing phases for multicast packets differ from those defined for unicast packets. With multicast, during the ascending phase the nearest common ancestor switch for the source node and the destination group is reached. After that, the turnaround routing step is performed and, during the second routing phase, the packet spans the appropriate links to reach all the destination nodes.

A situation where multiple multicast packets proceed in parallel may lead to deadlock [7]. For this reason, in order to guarantee deadlock-freedom, several limitations apply to the routing of multicast packets. These restrictions are based on the concept of *Broadcast Tree* and *Root Switch*:

**Definition 1** *A **Broadcast Tree,** in a fat-tree network topology, is composed of the links and switches that provide a descending path between the top leftmost switch in the network and all the nodes.*

*It can also be defined as the links and switches that provide an ascending path between all the nodes and the top leftmost switch.*

Figure 1(a) shows the default broadcast tree for a 16-node network (dashed lines).

**Definition 2** *A **Root Switch** for any given group of nodes refers to the nearest common ancestor switch which belongs to the broadcast tree for that group of nodes.*

Figure 1(c) shows the root switch (filled) for the groups highlighted in gray and black.

The Elite switches serialize incoming multicast packets if there is some overlap among the requested output ports, otherwise they can proceed in parallel. When sending a multicast packet, the root switch is used to perform the turnaround routing. Thus, at this point, the packet starts to be replicated to several output ports. In the QsNET, a multicast packet can only take paths included in the broadcast tree. According to this, only switches that are included in the broadcast tree are allowed to be used as root switches by the system routing tables and the switches configuration. In this way, simultaneous multicasts are serialized through the root switch and, therefore, deadlocks are avoided.

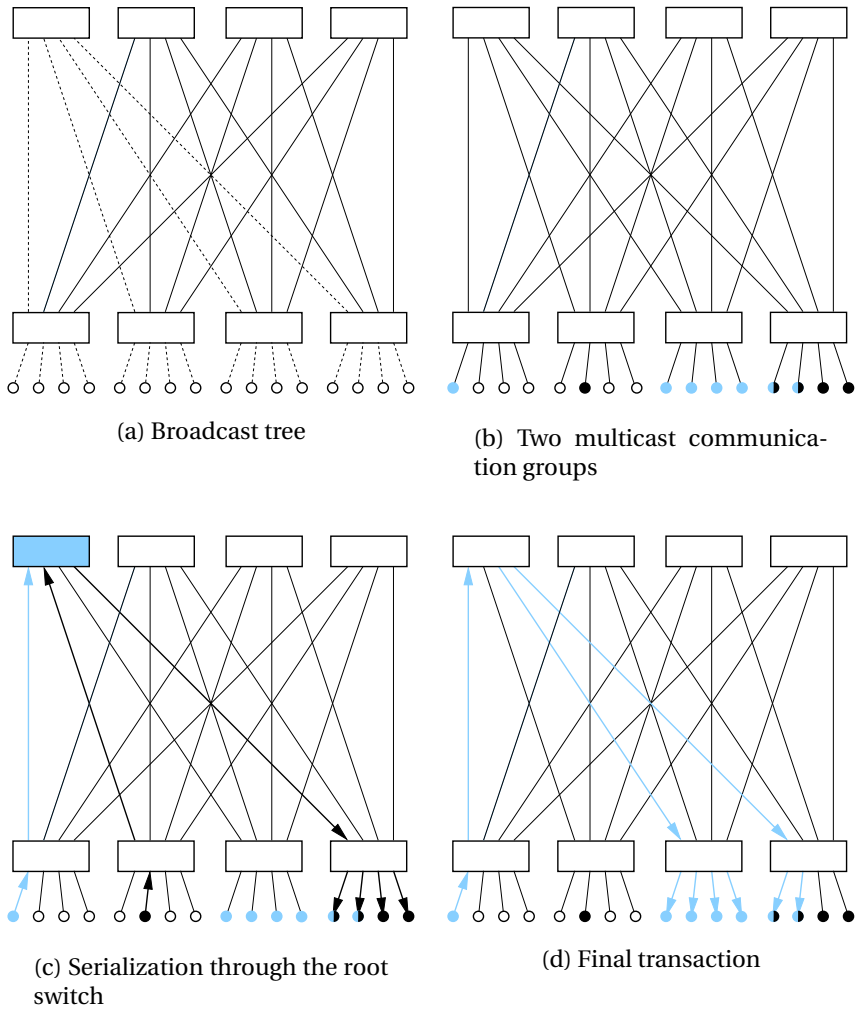
A communication example which involves two multicast groups is presented in Figure 1 (b). The gray and black nodes represent two groups with a partial overlap of destinations, the leftmost node in each group being the source. Let's assume that both sources send a packet to its group, and both packets arrive to the root switch (second stage leftmost switch) at the same time with the same priority. As there is overlap on the requested output ports, the switch arbiter only allows one packet to proceed (black packet, Figure 1(c)), while the other one is blocked until its resources are available (gray packet, Figure 1(d)).

For a multicast packet to be successfully delivered, a positive acknowledgement must be received from all the recipients of the multicast group. The Elite switches combine the acknowledgements, as pioneered by the NYU Ultracomputer [2, 16], returning a single one to the source. Acknowledgements are combined in a way that the "worst" ack wins (a network error wins over an unsuccessful transaction, which on its turn wins over a successful one), returning a positive ack only when all the partners in the collective communication complete the distributed transaction with success.

# 4 Hardware-based multicast trees

As stated in Section 3, the Quadrics network hardware support for collective communication can only be used when all the destination nodes are contiguous. Otherwise, a software-based multicast algorithm based on a balanced tree is used [13]. This approach provides significantly lower performance than the hardware-based multicast since it is based on point-to-point messages.

In order to overcome the hardware multicast limitations and provide high-performance collective communication, a hardware-based multicast tree technique has been presented in [6]. This technique uses a multicast tree to distribute the data among all the destinations, but with the particular issue that each message is a hardware multicast. In this way, the source node sends a multicast message to a subset of contiguous destinations, each of which forwards the message to another subset. Eventually, all destinations will receive the message. A key factor to achieve the best possible performance is the parallel transmission of as many multicast transactions as possible, once a subgroup of adjacent destinations has received a copy of the message. It is worth noting that, for the correct operation of this mechanism, deadlock-freedom has to be guaranteed. This mechanism is likely to provide significant performance improvements when



(a) Broadcast tree

(b) Two multicast communication groups

(c) Serialization through the root switch

(d) Final transaction

Figure 1: Hardware Multicast

the number of destinations is high and there are a few gaps in the group allocation (e.g., due to some faulty nodes).

Figure 2 shows an example comparing the behavior of the original software-based tree (Figure 2(a)), used by the Quadrics system software, and the proposed hardware-based tree (Figure 2(b)). The example considers a multicast group composed of 13 non-adjacent nodes. In the software tree, once each node receives the message from its parent, it forwards one copy to each one of its children. In the hardware tree, the source node sends a hardware multicast to one subgroup of adjacent nodes and, after that, each node in the subgroup forwards the message to another subgroup. The sequence of multicast steps for the hardware-based tree is shown in Figure 3. Figure 3(a) shows the first step and Figure 3(b) shows the second step. As can be seen, the hardware tree provides, in general, lower tree depth and thus lower latency. Nevertheless, the performance of the hardware-based tree relies on the possibility of parallelizing the hardware multicasts that are to be sent during the same tree step.

In order for several hardware multicast messages, with different sources, to proceed in par-

allel without collisions among them, several routing restrictions have to be taken into account:

- All multicast messages must follow the broadcast tree once the replication to several output ports at some switch starts. In other words, the root switch for any given multicast subset must belong to the multicast tree. In this way deadlocks are avoided. The root switches used during the two multicast steps of the described example are highlighted in Figure 3.
- Given two multicast operations with different sources and destinations, they can proceed in parallel (that is, with no serialization at any switch or link) only if they do not share any link. That is the case for the three parallel hardware-based multicasts shown on Figure 3(b).

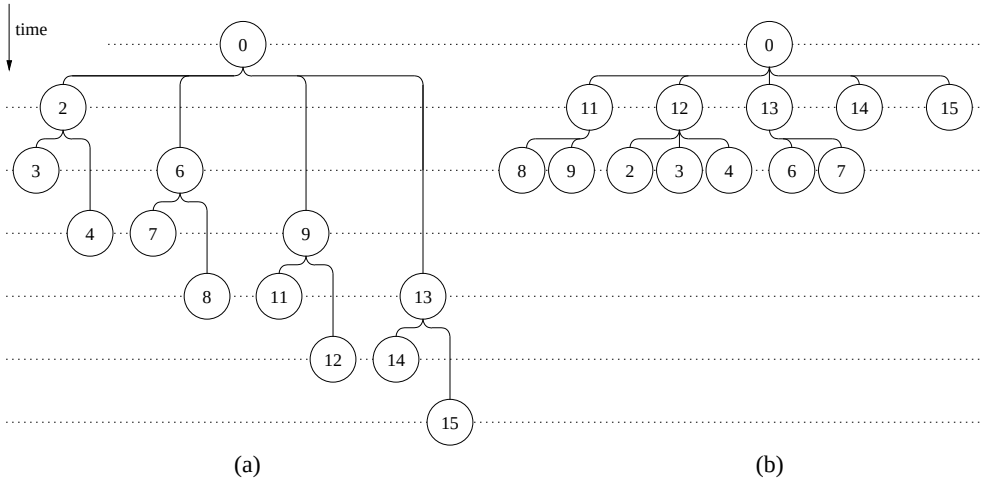



Figure 2: (a) Software- and (b) hardware-based multicast trees.

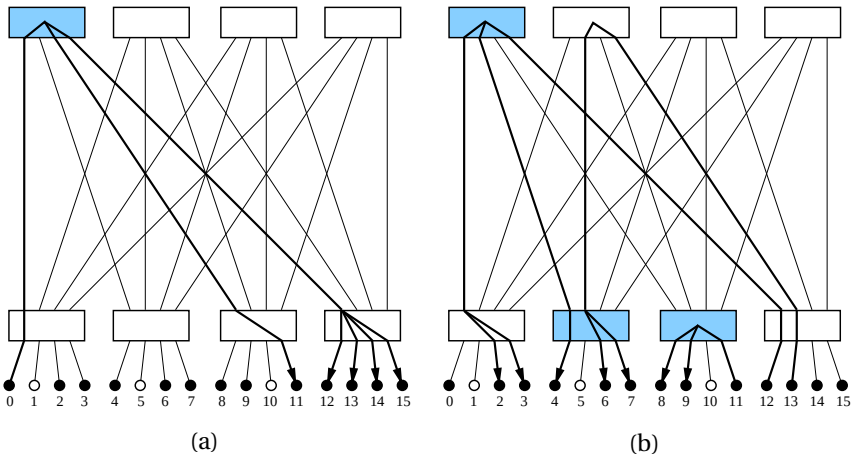



Figure 3: Hardware-based multicast tree: (a) first step; (b) second step.

# 5 Hardware-based multicast tree algorithm

Obtaining the optimal (minimum depth) hardware-based multicast tree for any given problem is not a simple task. There is no rule to select the set of destination nodes, to be reached at each tree step, that guarantees an optimal solution. For this reason, two different algorithms have been developed: a backtracking algorithm and a greedy algorithm. The backtracking approach explores all the solutions for any given problem in order to find the optimal one. The computational cost of such an algorithm is exponential, and thus impractical for a real implementation. On the other hand, the greedy approach applies some heuristics to find a good solution, but does not guarantee that the optimal tree will be found. Nevertheless, the number of iterations of the greedy algorithm is proportional to the tree depth, leading to a practical implementation with an extremely low computational cost.

## 5.1 Preliminary considerations

As it has been described in Section 4, a hardware-based multicast tree step consists, in general, of a set of parallel hardware multicasts sent by a number of nodes to disjoint sets of contiguous destinations. After the necessary number of steps, all the destinations will be reached. An optimal tree, in terms of latency, has the minimum number of steps, as shown in [3].

For an optimal solution to be found, all the alternative multicast trees should be explored. That is, at each tree step, all the alternative destination sets of contiguous nodes, reachable in parallel by the nodes that have a copy of the message during that step, have to be checked. And from those destinations, all the combinations of destinations that can be reached during the next step, and so on. This approach has an intrinsically exponential cost. A preliminary analysis of the problem indicates that there is no algorithm (or rule) that makes it possible to find, in general, an optimal multicast tree with logarithmic cost, that is, proportional to the tree depth. For this reason, two different algorithms have been developed:

- a backtracking algorithm, Algorithm 1, that explores all the solutions, providing an optimal solution to the problem (Section 5.2). This algorithm is optimized by stopping and backtracking as soon as the current partial solution cannot improve the best solution already found.
- a greedy algorithm, Algorithm 2, that takes a decision at each tree step, providing an optimal or near-optimal solution with small computational cost (Section 5.3). This algorithm uses an heuristic to select the appropriate sequence of groups to be reached during each step of the multicast tree.

## 5.2 Backtracking algorithm

As a refinement of exhaustive search, a backtracking algorithm builds a feasible solution incrementally. A solution is a sequence of decisions that defines the set of groups of adjacent nodes that are reached at each tree step (an example is shown in Figure 2(b)). After a first solution to the problem is found, only partial solutions that improve the current best solution are explored. In this way, the depth of the search tree is reduced.

As it has been stated, a hardware-based multicast tree is composed of one or more steps. During each step of the tree, one or more hardware-based multicast transactions are transmitted in parallel from the nodes that already received the message to groups of adjacent nodes

still waiting for the message. The proposed backtracking algorithm (Algorithm 1) is based on the sequential checking of all possible combinations of groups of adjacent nodes reachable in parallel from the nodes that received the message at a previous tree step. In this way, the algorithm selects a combination of as many groups as the number of source nodes (nodes that already received the message) out of the set of pending destination groups (tag ❶ in Algorithm 1). It is worth noting that the proposed multicast mechanism relies on the parallel transmission of hardware-based multicast messages during each tree step. For this reason, at this point it is checked if the selected combination of destination groups can be reached in parallel from the sources (there is no overlap among their routes) ❷. If the selected groups are not reachable in parallel, they are split into smaller groups in such a way that the routing overlaps are removed and an overlap-free set of destination groups is chosen ❸ ❹. In this way, at each tree step as many destination groups as nodes which already received a copy of the message are reached in parallel ❺. Later, the source and destination lists are updated accordingly, and, if there are pending destinations, a recursive call is made to explore the next tree step ❻. Eventually, the algorithm finds a sequence where all the destinations are reached, that is, a solution to the problem ❼. Next, additional solutions to the problem are explored by repeating the above procedure with different combinations of groups ❽.

The algorithm is implemented in such a way that, once the first solution is obtained, only solutions that may improve the best solution found are explored ❾. A further refinement of this algorithm is obtaining a good first solution. In this way, the better solutions space is significantly reduced. The heuristic used to find a good solution for the first iteration of the algorithm, consists of sorting the original list of groups of adjacent nodes in decreasing order of sizes (groups with equal sizes are reverse-ordered according to their root switch level). This heuristic is described in detail in Section 5.3, since it was originally devised for the greedy algorithm.

As stated in the Algorithm 1 description above, in order for the parallel reachability of a given set of destination groups from a given set of source nodes to be checked, two conditions are verified: *backward routing overlap* and *forward routing overlap*. If any link overlap exists, either during the forward routing phase or during the backward routing phase (Section 2) of two or more hardware-based multicast transactions, then the set of pre-selected destinations would not be reachable in parallel from the sources (the transactions would be serialized through the overlapping switches). These dependencies can be removed by splitting one or more groups of adjacent nodes into smaller groups. The overlap resolution mechanism could have been implemented by trying all the possible combinations of groups splitted into all the possible ways. But this would have increased the computational cost of the backtracking algorithm to even higher levels. An approach that takes into account the specific properties of the overlaps has been used instead (the detailed procedure for doing so is described below).

### 5.2.1 Backward routing overlap

Backward routing overlap occurs when there is overlap among the broadcast trees of the selected groups of nodes. It has to be mentioned that the backward routing overlap does not depend on the source nodes, since it appears on the backward routing phase, which always follows paths included in the broadcast tree (Section 3) of the destination groups.

Let $[x - x']$ denote a group of adjacent nodes from node $x$ to node $x'$, $[x - x'] = \{i \mid x \leq i \leq x'\}$.

Let $[x_1 - x'_1], [x_2 - x'_2], \ldots, [x_n - x'_n]$ denote a set of $n$ groups of adjacent nodes.

Given two groups of adjacent nodes $[x - x']$ and $[y - y']$, a backward routing overlap exists at level $l$ if and only if:

Algorithm 1: optimal tree using backtracking

```
Procedure optimal_tree
Input:  the source nodes list (srcs), the destination nodes list (dests),
        the sublist of destination nodes to be tested (dests_to_try),
Output: the partial solution being obtained (sol),
        the steps of the partial solution being obtained (steps),
        the best found solution (best_sol),
        the steps of the best found solution (best_steps)
Note: the destination nodes list is sorted before the initial call
      according to number of nodes and root switch level
begin
  copy srcs, dests, dests_to_try lists to local variables
  while (!last_combination && ❾(steps < best_steps-1))
    {
  ❶ dests_to_try = next combination of |srcs| groups out of dests_to_try
                   (update last_combination)
  ❷ if backward_routing_overlap( dests_to_try )
      {
      /* generate two alternative lists where the overlap is removed */
    ❸ backward_routing_overlap_resolution(dests_to_try, left_try, right_try)
      /* check two alternative branches of the solutions tree */
      optimal_tree(srcs, dests, right_try, sol, steps, best_sol, best_steps)
      optimal_tree(srcs, dests, left_try, sol, steps, best_sol, best_steps)
      }
    ❷ else if (forward_routing_overlap(srcs, dests_to_try, limited_level))
        {
        foreach group in dests_to_try with level >= limited_level
          {
          /* generate an alternative list where the overlap
             is likely to be removed                          */
        ❹ dests_to_try = split_a_group(dests_to_try,group)
          optimal_tree(srcs, dests, dests_to_try, sol, steps, best_sol, best_steps)
          }
        }
        else
          {
          /* groups in dests_to_try are reachable in parallel,
             account for a new tree step */
        ❺ steps++
          /* update dests and srcs according to reached groups */
          update_dests(dests, dests_to_try)
          update_srcs(srcs, dests_to_try)
          update_solution(sol)
          if (dests != ∅)  /* still pending destinations */
            {
          ❻ optimal_tree(srcs, dests, dests, sol, steps, best_sol, best_steps)
            }
            else if (steps < best_steps)
              /* a better solution is found */
            ❼ update_solution(best_sol, sol)
            }
          }
    /* explore an alternative branch of the solutions tree */
  ❽ steps--
    copy srcs, dests, dests_to_try lists to local variables
    }
end
```

- both groups use at least two links of the broadcast tree at level $l$, $\left(\lfloor \frac{x}{4^l} \rfloor \neq \left\lfloor \frac{x'}{4^l} \right\rfloor\right) \wedge \left(\lfloor \frac{y}{4^l} \rfloor \neq \left\lfloor \frac{y'}{4^l} \right\rfloor\right)$, and,
- both groups share one of those links, $\left(\left\lfloor \frac{x'}{4^l} \right\rfloor = \lfloor \frac{y}{4^l} \rfloor\right) \vee \left(\lfloor \frac{x}{4^l} \rfloor = \left\lfloor \frac{y'}{4^l} \right\rfloor\right)$.

where $\lfloor x \rfloor$ indicates the immediate lower integer of $x$, and $l$ refers to a switch level, being level $0$ that where the nodes are attached to. The condition for backward routing overlap is given by:

$$\left(\lfloor \frac{x}{4^l} \rfloor \neq \left\lfloor \frac{x'}{4^l} \right\rfloor\right) \wedge \left(\lfloor \frac{y}{4^l} \rfloor \neq \left\lfloor \frac{y'}{4^l} \right\rfloor\right) \wedge \left(\left(\left\lfloor \frac{x'}{4^l} \right\rfloor = \lfloor \frac{y}{4^l} \rfloor\right) \vee \left(\lfloor \frac{x}{4^l} \rfloor = \left\lfloor \frac{y'}{4^l} \right\rfloor\right)\right) \tag{1}$$

As an example, if Condition 1 is applied to the groups $[2,5],[7,15]$ (depicted on the top left corner of Figure 4), backward routing overlap occurs at level one as the next condition is true:

$$\left(\left\lfloor \frac{2}{4^1} \right\rfloor \neq \left\lfloor \frac{5}{4^1} \right\rfloor\right) \wedge \left(\left\lfloor \frac{7}{4^1} \right\rfloor \neq \left\lfloor \frac{15}{4^1} \right\rfloor\right) \wedge \left(\left(\left\lfloor \frac{5}{4^1} \right\rfloor = \left\lfloor \frac{7}{4^1} \right\rfloor\right) \vee \left(\left\lfloor \frac{2}{4^1} \right\rfloor = \left\lfloor \frac{15}{4^1} \right\rfloor\right)\right) =$$

$$= (0 \neq 1) \wedge (1 \neq 3) \wedge ((1 = 1) \vee (0 = 3)) = 1 \wedge 1 \wedge (1 \vee 0) = 1$$

Since the overlapping groups cannot be reached in parallel by hardware-based multicast transactions, the selected combination of groups is not part of a valid multicast tree (Section 4). Nevertheless, the overlap can be removed by splitting some groups into smaller groups in such a way that the selected combination of groups (or part of it, if not enough source nodes are available at a particular tree step) can be reached.

To do so, a backward routing overlap removal mechanism has been implemented in the following way: the combination of pre-selected groups (which is implemented using a list) is checked for overlaps among groups, by following the groups from left to right (checking every group against all the others). When the first overlap is found, two lists of pre-selected groups are generated by splitting some groups: first, the latest group checked is splitted to remove dependencies (right group); second, the group/s already checked is/are splitted (left group/s). The remaining groups in the list are not rearranged at this point, if additional overlaps exist, they will be detected later. After that, two recursive calls are made with the two newly generated lists (that is, two branches of the solution tree are explored) and the procedure is executed until an overlap-free list of groups is selected. This technique guarantees that an overlap-free combination of size the number of source nodes from the list of pending destinations will be selected.

Figure 4 shows the solutions tree explored by the algorithm on a 16-node network with the following conditions: source node 8; original groups of adjacent destination nodes, $[0,0]$, $[2,5]$, $[7,15]$. The circle represents the source node, rounded boxes denote group/s of adjacent nodes, and arrows represent algorithm decisions. Any path between the tree root to a leaf represents a solution to the hardware-based multicast tree problem. For instance, following the rightmost branches the solution is: during the first step of the tree node 8 sends the packet to group $[7,15]$, during the second step two nodes in $[8,8]$, $[7,15]$ send two copies of the packet to groups $[0,0]$ and $[2,5]$, respectively. As can be seen, the number of rounded boxes between the root node and a leaf is equal to the tree depth, which is equivalent to the number of steps of the multicast tree (two in the example).

The dotted ellipse highlights the point in a partial solution where a backward routing overlap is detected. Group $[2,5]$ and group $[7,15]$ share a link in their broadcast trees at switch level 1, as it is indicated in the network representation on the top left corner of the figure (the overlapping

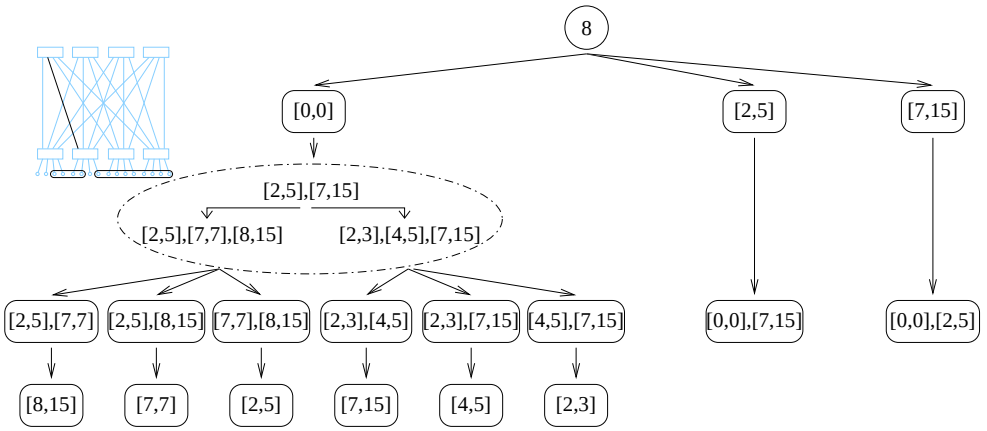



Figure 4: Example of the backtracking algorithm operation with backward routing overlap.

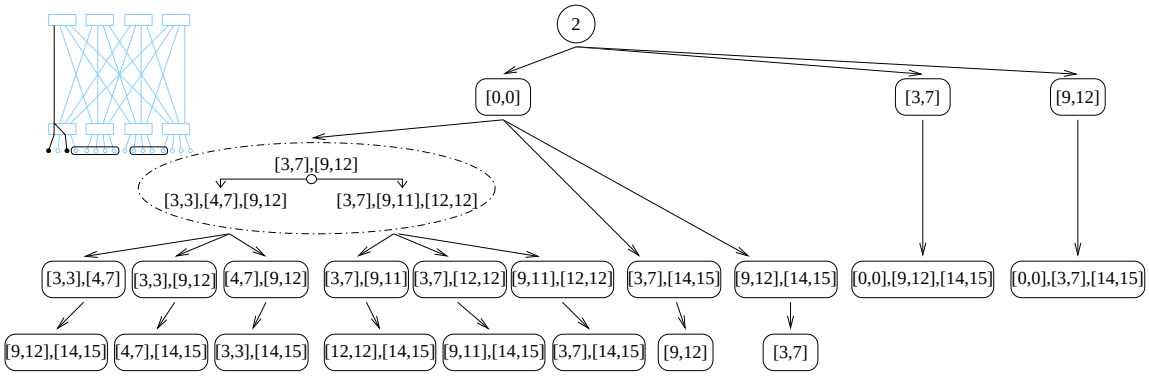



Figure 5: Example of the backtracking algorithm operation with forward routing overlap.

groups are highlighted). The overlap resolution mechanism implemented performs two different rearrangements of the pre-selected groups ($[2,5]$, $[7,15]$), as described above: first, group $[7,15]$ (right group) is splitted into groups $[7,7]$ and $[8,15]$; second, group $[2,5]$ (left group) is splitted into groups $[2,3]$, $[4,5]$. From this point two different solutions sub-trees are explored, as indicated in the figure, by performing two recursive calls to the backtracking procedure with the new two lists ($[2,5]$, $[7,7]$, $[8,15]$ and $[2,3]$, $[4,5]$, $[7,15]$). As can be seen, by splitting the overlapping groups in this way, two alternative overlap-free group arrangements are explored.

### 5.2.2 Forward routing overlap

After arranging the selected groups of destinations in a backward routing overlap-free combination, it has to be verified if that particular combination of groups is reachable from the source nodes. Forward routing overlap occurs when there are not enough disjoint routes to reach the root nodes of the pre-selected destination groups from the available source nodes. As in the case of the backward routing overlap, splitting the destination groups into smaller groups in the appropriate way (described below) allows the overlaps to be removed.

The forward routing overlap detection depends, not only on the destination groups, but also on the source nodes and which nodes try to send to each destination group. In order to pro-

vide a simple solution to this problem, this is addressed in the following way. The algorithm accounts for the number of root switches (Section 3) a particular list of source nodes can reach in parallel at each switch level, starting with the highest level, for short *capabilities*. Each node is accounted only once, that is, the sum of the capabilities equals the number of nodes. Besides, it calculates the number of root switches per switch level a particular list of destination groups needs for them to be reached in parallel, for short *needs*. By comparing, level by level (in decreasing order), the routing capabilities of the sources with the routing needs of the destinations, limitations (which correspond to link overlaps) are detected. At each level, if the capabilities are greater than the needs, the remaining capabilities are moved to the immediate lower level and the comparison continues at that level. If the needs are greater than the capabilities a forward routing overlap is detected. This mechanism allows overlaps to be detected, while it is avoided to check all the possible combinations of source nodes against the destination groups.

The problem can be formalized in the following way:

Let array $c = (c_{L-1},\, c_{L-2},\, ...,\, c_0)$ denote the capabilities of a set of nodes in a network with $L$ switch levels, where $c_i$ denotes the capabilities of that set of nodes for level $i$.

Let array $n = (n_{L-1},\, n_{L-2},\, ...,\, n_0)$ denote the needs of a list of destination groups of adjacent nodes in a network with $L$ switch levels, where $n_i$ denotes the needs of those node groups at level $i$.

Let $d = (d_{L-1},\, d_{L-2},\, ...,\, d_0)$ be the array of differences between $c$ and $n$ calculated in the following way:

$$d_{L-1} = c_{L-1} - n_{L-1} \qquad d_{i,\, i<L-1} = \begin{cases} c_i - n_i & if\ d_{i+1} \leq 0 \\ c_i - n_i + d_{i+1} & if\ d_{i+1} > 0 \end{cases} \tag{2}$$

A forward routing overlap exists when the following condition holds:

$$\exists\, l \in \mathbb{N},\, 0 \leq l < L \,|\, d_l < 0 \tag{3}$$

Overlaps are checked in decreasing order of levels. When an overlap is detected, the forward routing overlap removal procedure selects all the groups at and above the level where the overlap is detected. Then, one of the groups is splitted to produce subgroups with the root switch at an immediate lower level. After that, a new call to the backtracking procedure is performed. If no additional overlaps are detected the algorithm proceeds as stated in Section 5.2.

This procedure is repeated for all the groups with root switches at the level where the limitation is detected and higher, one by one. This is done in this way because there is no knowledge about the particular groups that overlap, since no matching between source nodes and destination groups is done. This has been thought to be the less costly approach since, in the worst case, all the groups will be splitted; otherwise, all the possible pairs source node - destination group should be tested. If the resulting list of destinations has additional overlaps, they will be detected and removed on a later recursive call.

Figure 5 shows the solutions explored by the algorithm on a 16-node network with the following conditions: source node 2; original groups of adjacent nodes, $[0, 0]$, $[3, 7]$, $[9, 12]$. The ellipse indicates the point in a partial solution where a forward routing overlap is detected. In this case, group $[3, 7]$ and group $[9, 12]$ share a link at switch level 1 because the source nodes are attached to the same level 0 switch (nodes 2 and 0, in the example). This is indicated in the network

representation on the top left corner of the figure (the rounded boxes indicate the overlapping groups). This overlap is detected with the mechanism described above based on source capabilities and destination needs. The source nodes at this point of the solutions tree are: node 2, and node 0 with capabilities $c = (1, 1)$. As can be seen in the figure, from nodes 0 and 2 in a 16-node network one root switch at level 1 can be reached, and additionally a root switch at level 0. As an alternative, two root switches at level 0 could be reached in parallel. The destination groups are: $[3, 7]$, $[9, 12]$ with needs $n = (2, 0)$. The differences array according to Equation 2 is $d = (-1, 1)$. As there is a limitation at level 1, the list of destination groups has a forward routing overlap.

At this point the backtracking algorithm explores two solution sub-trees, each one resulting from splitting one of the two overlapping groups into smaller subgroups to remove the dependencies. First, group $[3, 7]$ is splitted into groups $[3, 3]$ and $[4, 7]$; second, group $[9, 12]$ is splitted into groups $[9, 11]$, $[12, 12]$. After that, two recursive calls to the backtracking procedure with the new two pending destination lists ($[3, 3]$, $[4, 7]$, $[9, 12]$ and $[3, 7]$, $[9, 11]$, $[12, 12]$) are performed.

As described above and indicated in Algorithm 1, the routing overlap removal mechanisms are applied in a pre-defined order. When both overlap types exist in a particular set of destination groups to be checked, backward routing overlaps are removed first and, after that, forward routing overlaps are removed. As the backward routing overlap takes place in a later stage in packet routing, when the overlapping groups are splitted the resulting groups are bigger than the groups that would be obtained if the forward routing overlap was removed first. As the goal is finding the minimum steps multicast tree, reaching bigger groups as soon as possible will provide better solutions.

### 5.3 Greedy algorithm

A greedy hardware multicast tree algorithm (Algorithm 2) has been developed to provide an implementation with practical computational cost. The structure of this algorithm is based on the backtracking approach presented in previous section. The main difference is that no additional solutions are explored after the first solution is found. This solution will be demonstrated to be a good solution to the problem of finding the optimal hardware multicast tree.

The key factor of the greedy algorithm is an heuristic for choosing the group (or set of groups) to be reached during each step of the multicast tree. At any step of the hardware multicast tree, the optimal subset of groups of destination nodes to be selected is not known. Nevertheless, two properties of any given destination group give an indication of the potential capabilities of that group to reach adittional destinations during subsequent multicast tree steps:

- the number of nodes in the group: the bigger the groups, the more destinations they are capable to reach in parallel. This indicates the maximum number of multicasts that might be transmitted in parallel in the next step by the corresponding group. This maximum number can be reached if there is no routing overlap.

- the level of the root switch for the group: this parameter gives an indication of the routing alternatives from that group to a given root switch (and, hence, to groups of adjacent nodes). This is related to the concept of routing capabilities presented in the previous section and is further explained below using the example in Figure 6.

Figure 6 provides some insight into this issue by comparing the routing capabilities of two groups composed of two nodes in a 16-node system. Figure 6(a) shows a 2-node group (group $[6, 7]$) with the root switch located at level 0. The capabilities array for this group is $c = (1, 1)$.This

Algorithm 2: multicast tree using a greedy algorithm

```
Procedure greedy_tree
Input:  the source nodes list (srcs), the destination nodes list (dests),
        the sublist of destination nodes to be tested (dests_to_try),
Output: the partial solution being obtained (sol),
        the steps of the partial solution being obtained (steps)
Note: the destination nodes list is sorted before the initial
      call according to number of nodes and root switch level
begin
  copy srcs, dests, dests_to_try lists to local variables
  dests_to_try = next combination of |srcs| groups out of dests_to_try
  if backward_routing_overlap( dests_to_try )
    {
    /* generate two alternative lists where the overlap is removed */
    backward_routing_overlap_resolution(dests_to_try, left_try, right_try)
    /* check one branch of the solutions tree */
    greedy_tree(srcs, dests, right_try, sol, steps)
    }
    else if (forward_routing_overlap(srcs, dests_to_try, limited_level))
      {
      select a group in dests_to_try with level >= limited_level
      /* generate an alternative list where the overlap
         is likely to be removed                          */
      dests_to_try = split_a_group(dests_to_try,group)
      greedy_tree(srcs, dests, dests_to_try, sol, steps)
      }
      else
        {
        /* groups in dests_to_try are reachable in parallel,
           account for a new tree step */
        steps++
        /* update dests and srcs according to reached groups */
        update_dests(dests_to_try, dests)
        update_srcs(dests_to_try, srcs)
        update_solution(sol)
        if (dests != ∅)  /* still pending destinations */
          {
          greedy_tree(srcs, dests, dests, sol, steps)
          }
        }
   }
end
```

configuration allows reaching only one group with the root switch at level 1 and another group with the root switch at level 0 (two groups with the root switch at level 0 are reachable too). Figure 6(b) shows a 2-node group with the root switch at level 1 (group $[7,8]$). The capabilities array for this group is $c = (2,0)$. In this case, two groups with their root switch at level 1 are reachable in parallel, provided they do not share any link, as stated in Section 4. All the combinations of groups with root switches at lower levels are possible as well, that is: one group with the root switch at level 1 and one group with the group switch at level 0, or two groups with the root switch at level 0. As can be seen, the routing capabilities of the group in Figure 6(b) include those provided by the group in Figure 6(a).

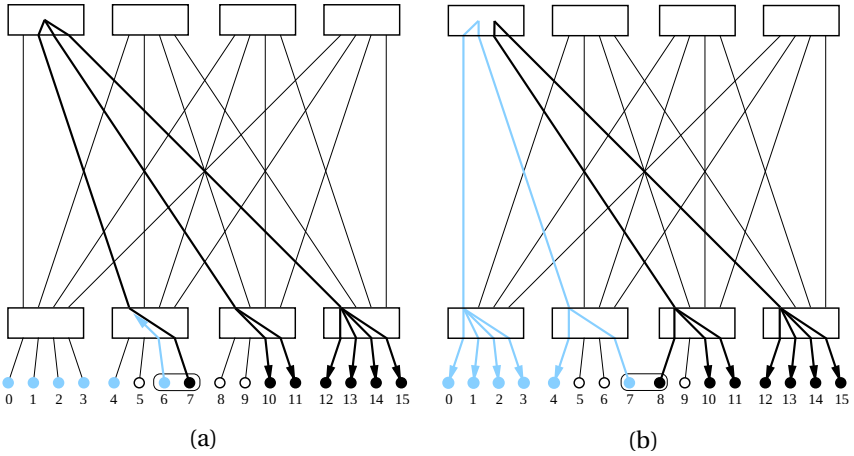



Figure 6: Hardware-based multicast capabilities of two subgroups composed of 2 nodes.

Considering two disjoint groups of adjacent nodes, in order to compare their capabilities to reach additional destinations, several situations are possible:

- One of the groups has higher root switch level and more nodes. That group is the best one because its routing capabilities include and overcome all the others'.
- Two groups have the same root switch level. In this case, it is clear that the best group is the one with more nodes.
- Two groups have the same amount of nodes and different root switch level. The best one (in terms of multicast capabilities) will be the group with the higher-level root switch.
- In all the other cases the best group depends on the remaining destinations.

Taking into account the above considerations, the heuristic applied in the proposed algorithm consists of sorting the original list of destination groups in decreasing order of nodes and root switch level. In this way, the algorithm, which is based on the backtracking implementation, selects destination groups in the established order.

During the initial algorithm iteration, the first group in the list is selected as the destination for the first step of the multicast tree. Since this is the first step, with only one source and one destination group, the destinations can always be reached in parallel using a hardware-based multicast. For the following steps of the multicast tree, the next sublist of destination groups

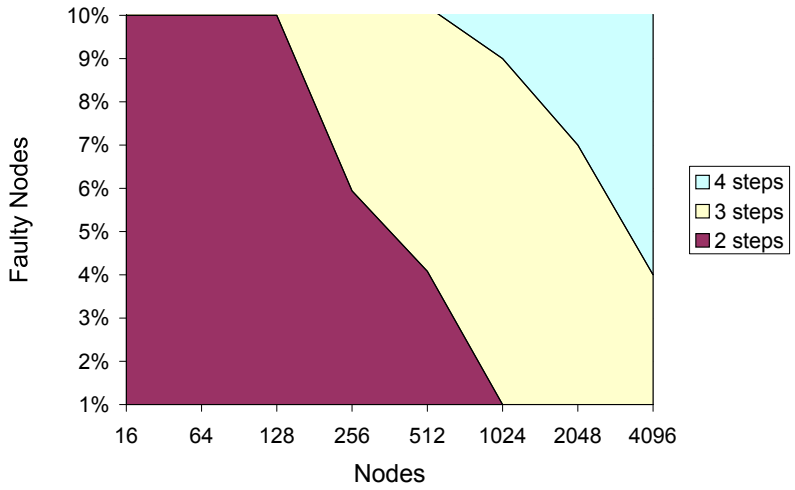



Figure 7: Maximum steps required by a hardware-based multicast tree.

of size the number of source nodes (nodes that already received the message) out of the list of pending destination groups is selected. In this way, potentially better groups are selected first. If any routing overlap problem arises, it is solved in the same way used for the backtracking approach, with the difference that only the first alternative is explored.

The computational cost of the greedy algorithm is proportional to the multicast tree depth, and thus to the logarithm of the number of groups of adjacent destination nodes. This provides an approach that can be implemented in practice with low computational cost, with the limitation that finding the optimal solution is not guaranteed.

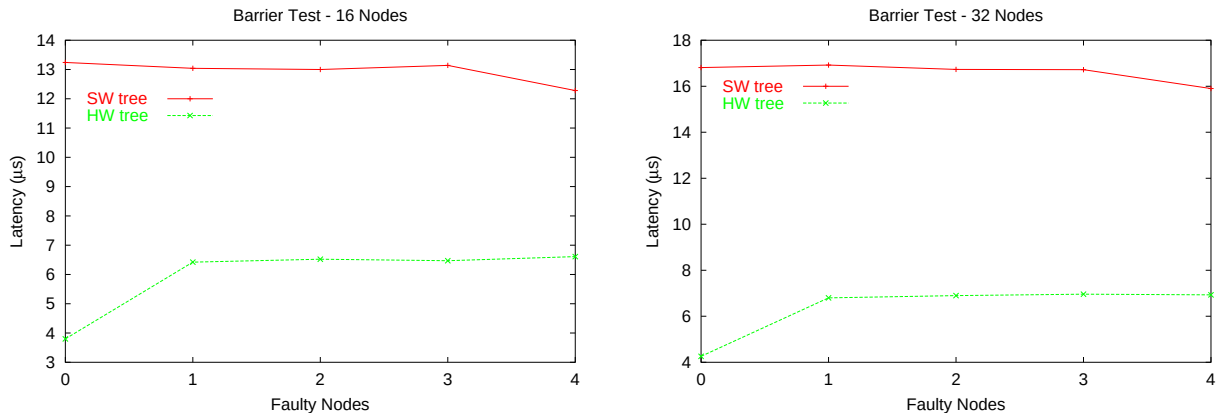
# 6 Results

As first step, the developed greedy algorithm has been tested in order to verify the optimal solution matching ratio. This has been done by comparing the solutions the greedy algorithm obtains with the optimal solution found using the backtracking algorithm. Tests have been conducted on configurations ranging from 16 to 4096-node networks, with faulty node percentages ranging from 0.1% to 10%. One thousand random faulty node mappings were generated for each configuration. Our tests show that the greedy algorithm finds an optimal solution for 99% of the cases. This percentage increases as the amount of faulty nodes decreases, being 100% when less than 1% of the nodes are faulty. A summary of the results is shown on Figure 7. Each colored area covers the configurations where the worst case tree depth (2, 3 or 4) is the same. It has to be mentioned that, as Figure 7 shows worst case values, the closer a given configuration is to its left area limit, the higher the probabilities for that configuration to be solved in one step less.

Detailed results of multicast trees on large-scale QsNET clusters are shown in Table 1. These tests have been designed to analyze the greedy algorithm performance when a fraction of the nodes is not operational. Networks with 1024, 2048 and 4096 nodes have been simulated. The

| Network Size | Faulty Nodes | | | | |
|---|---|---|---|---|---|
| | 0.2% | 0.4% | 0.6% | 0.8% | 1% |
| 1024 | 2 | 2 | 2 | 2 | 2 |
| 2048 | 2 | 2 | 2 | 2 | 2 |
| 4096 | 2 | 2 | 2 | $2/0.998$<br>$3/0.002$ | $2/0.582$<br>$3/0.418$ |

Table 1: Optimal hardware-based multicast tree steps for different configurations.




(a) (b)

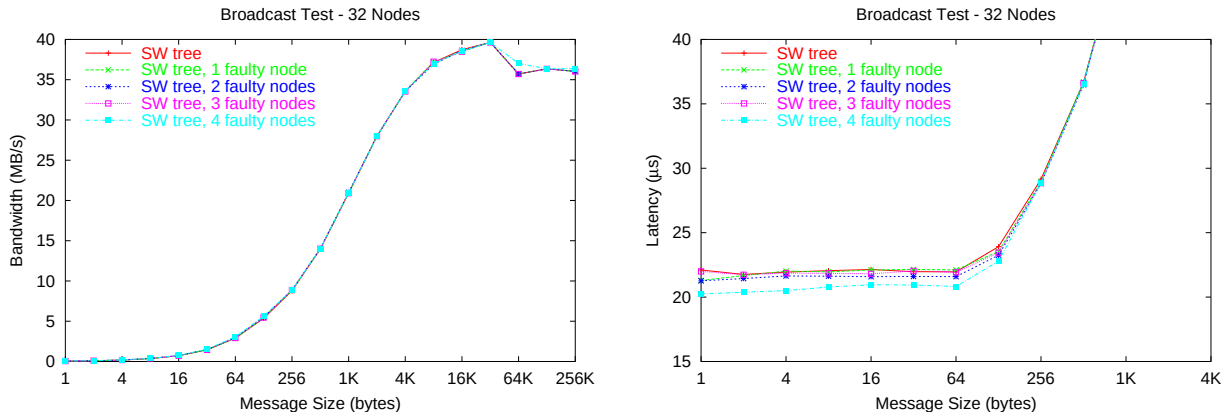
Figure 8: Barrier synchronization latencies.

number of nodes not taking part in the multicast transaction has been varied from 0.2% to 1% in 0.2% intervals. One thousand random problems for each particular configuration have been solved using the algorithms presented in this report. The number of steps for the optimal trees are summarized in the table, where the notation $x/y$ denotes a $x$-step tree with probability $y$. The results show that any practical situation, with a realistic amount of faulty nodes, is likely to be solved using a hardware-based multicast tree with only 2 steps. Only the largest configuration tested, 4096 nodes, will require 3 steps for 42% of the cases when 1% of the nodes are faulty.

On the one hand, these results indicate that the proposed greedy algorithm provides an optimal solution to most of the cases. On the other hand, since the tree depth is very short (a maximum of 4 steps are required in the worst case for the configurations tested) the QsNET collective communication support can be significantly improved.

In order to measure the impact of the proposed mechanism on the QsNET collective communication, several experiments have been conducted. The experimental evaluation has been performed on a 32-node cluster of Dell 1550, running Red Hat 7.1 Linux. Each node has two 1.13 GHz Pentium-III with 1GB of ECC RAM, and a Quadrics QM-400 Elan3 NIC attached to the network through a 66MHz/64-bit PCI bus.

Figure 8 shows the average time to perform a barrier synchronization on an empty network. The tree construction time is not taken into account, since it is computed once for each configuration. Results for 16- and 32-node network configurations are shown versus the number



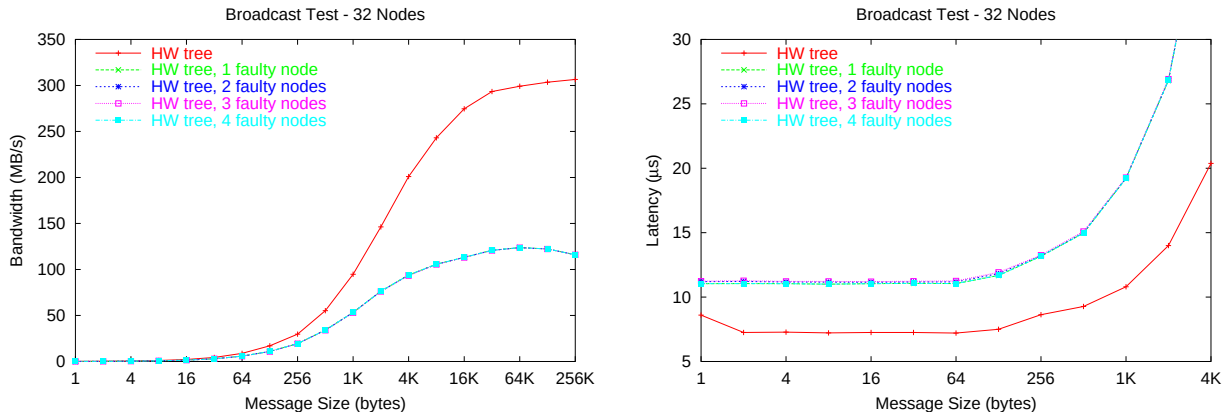

(a) (b)

Figure 9: Software-based broadcast.

of faulty nodes. As expected, the latency of the software-based barrier used by the QsNET is insensitive to the number of nodes not taking part in the collective communication, when this is low. This is due to the fact that a slight reduction on the destination nodes has no effect on the multicast tree depth [4]. For a 32-node configuration with less than four faulty nodes, the latency is constant at approximately $17\mu s$; while it slightly decreases when four faulty nodes are considered ($15.9\mu s$). On the other hand, it can be seen that the synchronization based on the hardware tree developed by us, significantly outperforms the software tree used by the QsNET. The latency required to barrier synchronize with no faulty nodes is $4\mu s$ in both configurations tested, which corresponds to a single multicast step. The time required to barrier synchronize the network with four faulty nodes or less is constant at $6.5\mu s$ for 16 nodes and $7\mu s$ for 32 nodes, since only two multicast steps are required. As can be seen, when the number of faulty nodes is low, our methodology provides between 50% and 60% faster synchronizations, for the configurations tested. We expect this performance gap to broaden with larger network configurations, since, as shown in Table 1, the scalability of our multicast mechanism is very good.

Figure 9 shows the results obtained with software-based broadcasts over a 32-node network, and buffers globally allocated in main memory (that is, with the same virtual address in all processes). As it has been shown for the barrier experiments, the performance of the software-based multicast is insensitive to the number of faulty nodes when there are only a few. Only the configuration with four faulty nodes, as can be seen from Figure 9(b), shows a slight improvement in latency. The peak bandwidth of 40MB/s is obtained for 32KB messages, while messages shorter than 64 bytes are delivered in less than $22\mu s$.

When all 32 nodes in the network take part in the collective communication, the hardware based multicast can be directly used. In this case a peak bandwidth of 306MB/s has been measured for 256KB messages (top curve on Figure 10(a)). On the other hand, if several faulty nodes are considered, our hardware-based multicast tree is used. The results obtained are shown on Figure 10. As can be seen, the performance of the multicast tree developed by us is insensitive to the number of faulty nodes for the configurations we tested, since the destinations can always be reached in two multicast steps. The measured bandwidth of the hardware-based multicast tree reaches a peak of 124MB/s for 64KB messages. For the tested message sizes, the bandwidth




(a) (b)

Figure 10: Hardware-based broadcast.

provided by the hardware-based multicast tree, with faulty nodes, ranges between 60% and 40% of the bandwidth obtained with no faulty nodes. These results show that the hardware-based multicast tree significantly outperforms the software-based multicast tree, providing a collective communication that is twice as fast in the worst case.

# 7 Conclusions

This paper presented hardware-based multicast trees as an alternative to overcome the limitations of the hardware support for multicast provided by the Quadrics network. A backtracking algorithm to calculate minimum latency hardware-based multicast trees is presented (Algorithm 1). As the backtracking implementation has an exponential cost, a greedy algorithm (Algorithm 2) has been developed in order to provide an alternative with lower computational cost; and, hence, suitable to be implemented in the QsNET system libraries.

The proposed mechanism has been applied to solve a number of random configurations for different network sizes. The greedy algorithm has been shown to find optimal solutions for the majority of the tested configurations. In addition, on systems where a realistic number of faulty nodes is considered, any multicast transaction can be performed using a hardware-based multicast tree with only two steps. That means that the hardware-based multicast tree mechanism would significantly outperform the behavior of the software-based multicast, used by the QsNET, on a large-scale network.

Results for barrier synchronization and broadcast, obtained on a 32-node cluster, show that the hardware-based multicast tree mechanism significantly outperforms the QsNET software-based multicast (barrier latencies are reduced to a half, while broadcast bandwidths are doubled). This effect is likely to increase as the network size increases due to the poor scalability of the software-based multicast when compared to the hardware-based multicast tree we propose. These results show that hardware-based multicast trees provide a scalable and fault-tolerant alternative to software-based multicast trees, while overcoming the limitations of the hardware support for collective communication.

As future work the authors plan to fully integrate the proposed mechanism into the QsNET software. A further experimental evaluation of the mechanism with larger network configurations, and the impact of this technique on real applications and job scheduling tools, are future venues of research too.

**Acknowledgments**

The authors would like to thank the Quadrics team, in particular David Addison and Mike Hinds, for their invaluable support.

# References

[1] Andrea Carol Arpaci-Dusseau. Implicit coscheduling: Coordinated scheduling with implicit information in distributed systems. *ACM Transactions on Computer Systems*, 19(3), August 2001. Available from `http://www.cs.wisc.edu/~dusseau/Papers/tocs01.ps`.

[2] Gordon Bell. Ultracomputers: A teraflop before its time. *Communications of the ACM*, 35(8):27–47, August 1992.

[3] Salvador Coll. A Parameterized Communication Model for QsNET. Technical report, Digital Systems Design Group, Technical University of Valencia, February 2002. Available from `http://ttt.upv.es/~scoll/papers/comm_model.pdf`.

[4] Salvador Coll, José Duato, Francisco J. Mora, Fabrizio Petrini, and Adolfy Hoisie. Collective Communication Patterns on the Quadrics Network. In *Proceedings of the Performance Analyisis and Grid Computing Seminar*, Dagstuhl, Germany, August 2002. Available from `http://ttt.upv.es/~scoll/talks/padc2002.pdf`.

[5] William J. Dally and Charles L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[6] José Duato. Improved Multicast Support for the Quadrics Network. Technical report, Parallel Architectures Group, Technical University of Valencia, August 2001.

[7] José Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks: an Engineering Approach*. Morgan Kaufmann, August 2002.

[8] Dror G. Feitelson and Morris A. Jette. Improved utilization and responsiveness with gang scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *3rd Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 238–261, Geneva, Switzerland, April 5, 1997. Springer-Verlag. Available from `http://www.cs.huji.ac.il/~feit/parsched/p-97-11.ps.gz`.

[9] Eitan Frachtenberg, Fabrizio Petrini, Juan Fernandez, Scott Pakin, and Salvador Coll. Storm: Lightning-fast resource management. In *IEEE/ACM SC2001*, Baltimore, MD, November 2002. Available from `http://ttt.upv.es/~scoll/papers/sc02.pdf`.

[10] Manish Gupta. Challenges in Developing Scalable Software for BlueGene/L. In *Scaling to New Heights Workshop*, Pittsburgh, PA, May 2002. Available from `http://www.psc.edu/training/scaling/gupta.ps`.

[11] Fabrizio Petrini and Wu chun Feng. Buffered coscheduling: A new methodology for multitasking parallel jobs on distributed systems. In *Proceedings of the International Parallel and Distributed Processing Symposium 2000 (IPDPS 2000)*, volume 16, Cancun, Mexico, May 1–5, 2000. Available from `http://ipdps.eece.unm.edu/2000/papers/petrini.pdf`.

[12] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, January/February 2002. ISSN 0272-1732. Available from `http://www.computer.org/micro/mi2002/pdf/m1046.pdf`.

[13] Fabrizio Petrini, Salvador Coll, Eitan Frachtenberg, and Adolfy Hoisie. Hardware- and software-based collective communication on the Quadrics network. In *Proceedings of the 2001 IEEE International Symposium on Network Computing and Applications (NCA 2001)*, Cambridge, Massachusetts, October 8–10, 2001. Available from `http://ttt.upv.es/~scoll/papers/nca01_ps.gz`.

[14] Fabrizio Petrini, Darren Kerbyson, and Scott Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Proceedings of SC2003*, Phoenix, Arizona, November 10–16, 2003. Available from `http://www.c3.lanl.gov/~fabrizio/papers/sc03_noise.pdf`.

[15] Fabrizio Petrini and Marco Vanneschi. Performance analysis of wormhole routed $k$-ary $n$-trees. *International Journal on Foundations of Computer Science*, 9(2):157–177, June 1998. Available from `http://www.c3.lanl.gov/~fabrizio/papers/ijfcs98.ps.gz`.

[16] Gregory F. Pfister and V. Alan Norton. "Hot spot" contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

# Buffered Coscheduling: A New Methodology for Multitasking Parallel Jobs on Distributed Systems *

Fabrizio Petrini† and Wu-chun Feng†§
{fabrizio, feng}@lanl.gov

† Computing, Information, and Communications Division
Los Alamos National Laboratory
Los Alamos, NM 87545

§ School of Electrical & Computer Engineering
Purdue University
W. Lafayette, IN 47907

## Abstract

*Buffered coscheduling is a scheduling methodology for time-sharing communicating processes in parallel and distributed systems. The methodology has two primary features: communication buffering and strobing. With communication buffering, communication generated by each processor is buffered and performed at the end of regular intervals to amortize communication and scheduling overhead. This infrastructure is then leveraged by a strobing mechanism to perform a total exchange of information at the end of each interval, thus providing global information to more efficiently schedule communicating processes.*

*This paper describes how buffered coscheduling can optimize resource utilization by analyzing workloads with varying computational granularities, load imbalances, and communication patterns. The experimental results, performed using a detailed simulation model, show that buffered coscheduling is very effective on fast SANs such as Myrinet as well as slower switch-based LANs.*

**Keywords***: distributed resource management, parallel job scheduling, distributed operating systems, coscheduling, gang scheduling.*

## 1. Introduction

In recent years, researchers have developed parallel scheduling algorithms that can be loosely organized into three main classes, according to the degree of coordination between processors: *explicit coscheduling*, *local scheduling* and *implicit or dynamic coscheduling*.

Explicit coscheduling [5] ensures that the scheduling of communicating jobs is coordinated by creating a static global list of the order in which jobs should be scheduled and then requiring a simultaneous context-switch across all processors. Unfortunately, this approach is neither scalable nor reliable. Furthermore, it requires that the schedule of communicating processes be precomputed, thus complicating the coscheduling of applications and requiring pessimistic assumptions about which processes communicate with one another. Lastly, explicit coscheduling of parallel jobs also adversely affects performance on interactive and I/O-based jobs [10].

Conversely, local scheduling allows each processor to independently schedule its processes. Although attractive due to its ease of construction, the performance of fine-grain communicating jobs degrades significantly because scheduling is not coordinated across processors [7].

An intermediate approach developed at UC Berkeley and MIT is implicit or dynamic coscheduling [1, 4, 12, 16] where each local scheduler makes decisions that dynamically coordinate the scheduling actions of cooperating processes across processors. These actions are based on local events that occur naturally within communicating applications. For example, on message arrival, a processor speculatively assumes that the sender is active and will likely send more messages in the near future.

In this paper, we present a new methodology that conjugates the positive aspects of explicit and implicit coscheduling using three techniques: communication buffering to amortize communication overhead (a technique similar to

*This work was supported by the U.S. Dept. of Energy through Los Alamos National Laboratory contract W-7405-ENG-36.

*periodic boost* [11]); strobing to globally exchange information at regular intervals; and non-blocking, one-sided communication to decouple communication and synchronization. By leveraging these techniques, we can perform effective optimizations based on the status of the parallel machine rather than on the limited knowledge available locally to each processor.

The rest of the paper is organized as follows. Section 2 describes the motivation and features of buffered coscheduling. Preliminary results are presented in Section 3. Finally, we present our conclusions in Section 4.

# 2 Multitasking Parallel Jobs

Our study of resource utilization in SPMD programs inspired our buffered coscheduling methodology which consists of communication buffering, strobing, and optionally non-blocking communication. This methodology allows all the communication and I/O which arise from a *set* of parallel programs to be overlapped with the computations in those programs.

## 2.1 Motivation

Figure 1 shows the global processor and network utilization during the execution of an FFT transpose algorithm on a parallel machine with 256 processors connected with an indirect interconnection network using state-of-the-art routers [3]. Based on these figures, we observe an *uneven and inefficient use of system resources*. These characteristics are shared by many SPMD programs, including unclassified ASCI application codes such as Sweep3D [8]. Hence, there is tremendous potential for increasing resource utilization in a parallel machine.

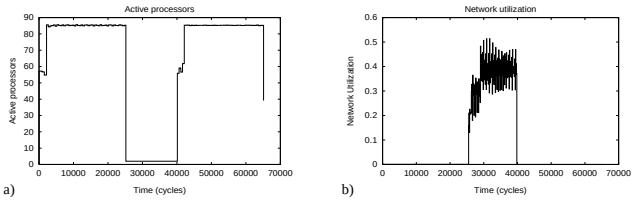



**Figure 1. Resource Utilization in an FFT Transpose Algorithm.**

Another important characteristic shared by many parallel programs is their access pattern to the network. The vast majority of parallel applications display *bursty communication patterns* with alternating spikes of impulsive communication with periods of inactivity [13]. Thus, there exists a significant amount of unused network bandwidth which could be used for other purposes.

## 2.2 Communication Buffering

Instead of incurring communication and scheduling overhead on a per-message basis, we accumulate the communication messages generated by each processor and amortize the overhead over a set of messages. By delaying the communication, we allow for the global scheduling of the communication pattern. And because we can implement zero-copy communication, this technique can theoretically achieve performance comparable to OS-bypass protocols [2] without using specialized hardware.

## 2.3 Strobing

The uneven resource utilization and the periodic, bursty communication patterns generated by many parallel applications can be exploited to perform a total exchange of information and a synchronization of processors at regular intervals with little additional cost. This provides the parallel machine with the capability of filling in communication holes generated by parallel applications.

To provide the above capability, we propose a strobing mechanism to support the scheduling of a set of parallel jobs which share a parallel machine. At a high level, the strobing mechanism performs an optimized total-exchange of control information which then triggers the downloading of any buffered packets into the network.

The strobe is implemented by designating one of the processors as the *master*, the one who generates the "heartbeat" of the strobe. The generation of heartbeats is achieved by using a timeout mechanism which can be associated with the network interface card (NIC). This ensures that strobing incurs little CPU overhead as most NICs can count down and send packets asynchronously.

On reception of the heartbeat, each processor (excluding the master) is interrupted and downloads a broadcast heartbeat into network. After downloading the heartbeat, the processor continues running the currently active job. (This ensures computation is overlapped with communication.) When all the heartbeats arrive at a processor, the processor enters a strobing phase where its kernel downloads any buffered packets to the network[1].

Figure 2 outlines how computation and communication can be scheduled over a generic processor. At the beginning of the heartbeat, $t_0$, the kernel downloads control packets

[1] Each heartbeat contains information on which processes have packets ready for download and which processes are asleep waiting to upload a packet from a particular processor. This information is characterized on a per-process basis so that on reception of the heartbeat, every processor will know which processes have data heading for them and which processes on that processor they are from.

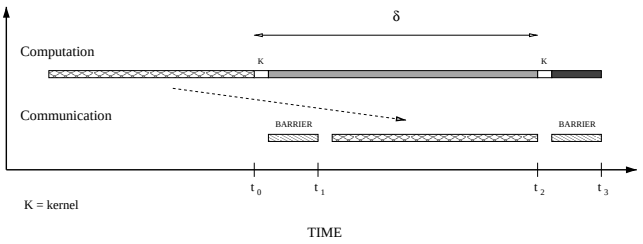



**Figure 2. Scheduling Computation and Communication. The communication accumulated before $t_0$ is downloaded into the network between $t_1$ and $t_2$.**

for the total exchange of information. During the execution of the barrier synchronization, the user process then regains control of the processor; and at the end of it, the kernel schedules the pending communication accumulated before $t_0$ to be delivered in the current time slice, i.e., $\delta$. At $t_1$, the processor will know the number of incoming packets that it is going to receive in the communication time-slice as well as the sources of the packets and will start the downloading of outgoing packets. (This strategy can be easily extended to deal with space-sharing where different regions run different sets of programs [5, 9, 17]. In this case, all regions are synchronized by the same heartbeat.)

The total exchange of information can be properly optimized by exploiting the low-level features of the interconnection network. For example, if control packets are given higher priority than background traffic at the sending and receiving endpoints, they can be delivered with predictable network latency[2] during the execution of a direct total-exchange algorithm[3] [14].

The global knowledge of the communication pattern provided by the total exchange allows for the implementation of efficient flow-control strategies. For example, it is possible to avoid congestion inside the network by carefully scheduling the communication pattern and limiting the negative effects of hot spots by damping the maximum amount of information addressed to each processor during a time-slice. The same information can be used at the kernel level to provide fault-tolerant communication. For example, the knowledge of the number of incoming packets greatly simplifies the implementation of receiver-initiated recovery protocols.

[2]The network latency is the time spent in the network without including source and destination queueing delays.

[3]In a direct total-exchange algorithm, each packet is sent directly from source to destination, without intermediate buffering.

# 3 Experimental Results

As an experimental platform, our working implementation includes a representative subset of MPI-2 on a detailed (register-level) simulation model [15]. The run-time support on this platform includes a standard version of a substantive subset of MPI-2 and a multitasking version of the same subset that implements the main features of our proposed methodology. It is worth noting that the multitasking MPI-2 version is actually much simpler than the sequential one because the buffering of the communication primitives greatly simplifies run-time support.

## 3.1 Characteristics of the Synthetic Workloads

As in [4], the workloads used consist of a collection of single-program multiple-data (SPMD) parallel jobs that alternate phases of purely local computation with phases of interprocess communication. A parallel job consists of a group of $P$ processes where each process is mapped onto a processor throughout its execution. Processes compute locally for a time uniformly selected in the interval $(g - \frac{v}{2}, g + \frac{v}{2})$. By adjusting $g$, we model parallel programs with different computational granularities; and by varying $v$, we change the degree of load-imbalance across processors. The communication phase consists of an opening barrier, followed by an optional sequence of pairwise communication events separated by small amounts of local computation, $c$, and finally an optional closing barrier.

We consider three communication patterns: *Barrier*,*News*, and *Transpose*. *Barrier* consists of only the closing barrier and thus contains no additional dependencies. We can therefore use this workload to analyze how buffered coscheduling responds to load imbalance. The other two patterns consist of a sequence of remote writes. The communication pattern generated by *News* is based on a stencil with a grid, where each process exchanges information with its four neighbors. This workload represents those applications that perform a domain decomposition of the data set and limit their communication pattern to a fixed set of partners. *Transpose* is a communication-intensive workload that emulates the communication pattern generated by the FFT transpose algorithm [6], where each process accesses data on all other processes.

For our synthetic workload, we consider three parallel jobs with the same computational granularities, load imbalances, and communication patterns arriving at the same time in the system. The communication granularity, $c$, is fixed at $8\ \mu s$. The number of communication/computation iterations is scaled so that each job runs for approximately one second in a dedicated environment. The system consists of $32$ processors, and each job requires $32$ processes (i.e. jobs are only time-shared).

### 3.2 The Simulation Model

The simulation tool that we use in our experimental evaluation is called SMART (Simulator of Massive ARchitectures and Topologies) [15], a flexible tool designed to model the fundamental characteristics of a massively parallel architecture. The current version of SMART is based on the x86 instruction set. The architectural design of the processing nodes is inspired by the Pentium II family of processors. In particular, it models a two-level cache hierarchy with a write-back L1 policy and non-blocking caches.

Our experiments consider two networks with 32 processing nodes, representative of two different architectural solutions. The first network is a 5-dimensional cube topology with performance characteristics similar to those of Myrinet routing and network cards [3]. This network features a one-way data rate of about 1 Gb/s and a base network latency of less than a $\mu$s. The second network is based on a 32-port, 100-Mb/s Intel Express switch, a popular solution due its attractive performance/price ratio.

### 3.3 Resource Utilization

Figures 3 and 4 show the communication/computation characteristics of our synthetic benchmarks on a Myrinet-based interconnection network and an Intel Express switch-based network, respectively, as a function of the communication pattern, granularity, load imbalance, and time-slice duration. Each bar shows the percentage of time spent in one of the following states (averaged over all processors): computing, context-switching and idling.

For each communication pattern in the Myrinet-based network, we consider time-slices of 0.5, 1, and 2 ms. In contrast, for the switch-based network, we consider time-slices of 2, 4, and 8 ms due to the larger communication overhead and lower bandwidth. In both cases, the context-switch penalty is 25 $\mu$s.

In each group of three bar graphs, the computational granularity is the same, but the load imbalance is increased as a function of the granularity itself, i.e., $v = 0$ (i.e. no variance), $v = g$ (the variance is equal to the computational granularity) and $v = 2g$ (high degree of imbalance).

Figures 3 (l)-(n) and 4 (l)-(n) show the breakdown for the *Barrier*, *News*, and *Transpose* workloads when they are run in dedicated mode with standard MPI-2 run-time support. For Figures 3 (a)-(i) and 4 (a)-(i), a black square under a bar denotes a configuration where buffered coscheduling achieves better resource utilization than MPI-2 user-level communication, and a circle indicates a configuration where the performance loss of buffered coscheduling is within 5%.

Based on Figures 3 and 4, we make the following observations. First, the performance of buffered coscheduling is sensitive to the context-switch latency. As context-switch latency decreases, resource utilization and throughput improve. Second, as the load imbalance of a program increases, the idle time increases. Third, and most importantly, these initial results indicate that the time-slice length is a critical parameter in determining overall performance. A short time-slice can achieve excellent load balancing even in the presence of highly unbalanced jobs. The downside is that it amplifies the context-switch latency. On the other hand, a long time-slice can virtually hide all the context-switch latency, but it cannot reduce the load imbalance, particularly in the presence of fine-grained computation.

In Figures 3 (a), (d), and (g) which use a relatively small time-slice in a Myrinet-based network, buffered coscheduling produces higher processor utilization than when a single job runs in a dedicated environment in over 55% of the cases and produces higher (or no worse than 5% less) resource utilization in nearly 75% of the cases.

Taking a big picture view of Figure 3, we conclude that for high-performance Myrinet-like networks that buffered coscheduling performs admirably as long as the average computational grain size is larger than the time-slice and the time-slice in turn is sufficiently larger than the context-switch penalty. In addition, when the average computational grain size is larger than the time-slice, the processor utilization is mainly influenced by the degree of imbalance.

With a less powerful interconnection network, we find that buffered coscheduling is even more effective in enhancing resource utilization. Figures 4 (a), (d), and (g) show that in a 100-Mb/s switch-based interconnection network, buffered coscheduling outperforms the basic approach in 16 out of 18 configurations with *Barrier*, 13 out 18 with *News*, and 10 out 18 with *Transpose*. In this last case, the performance of buffered coscheduling can be improved by increasing the time-slice.

What makes buffered coscheduling so much more effective in a less powerful interconnection network? The answer lies in the "excessive" communication overhead that is incurred in these commodity networks when each job is run in dedicated mode with MPI-2 run-time support; the overhead is high enough to adversely impact the resource utilization of the processor and network. For example, by comparing the graphs for the 500-$\mu$s computational granularity in Figures 3 (l)-(n) and Figures 4 (l)-(n), respectively, we see that the resource utilization for the switch-based network is significantly lower than the Myrinet network when running in dedicated mode. Consequently, there is substantially more room for resource-utilization improvement in the switch-based network, and the buffered coscheduling methodology takes full advantage this by overlapping computation with potentially long communication delays/overhead, thus hiding the communication overhead.

Irrespective of the type of network, for the cases where jobs are perfectly balanced, i.e., $v = 0$, running a sin-

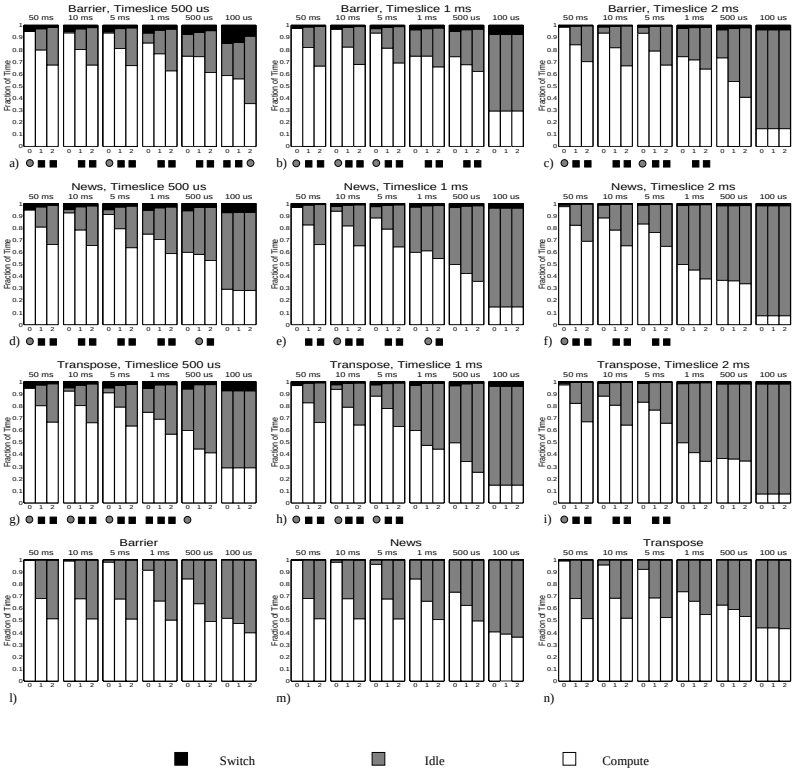



**Figure 3. Resource Utilization on a Myrinet-Based Interconnection Network.**

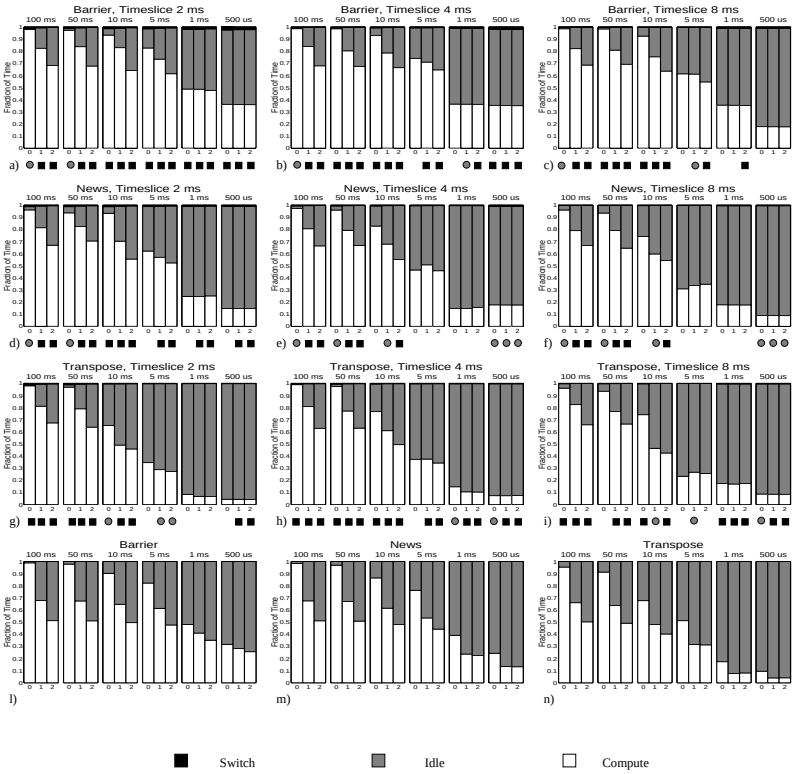



**Figure 4. Resource Utilization on a Switch-Based Network.**

gle job only results in *marginally* better performance because buffered coscheduling must "pay" the context-switch penalty without improving the load balance because the load is already balanced. On the other hand, in the presence of load imbalance, job multitasking can smooth the differences in load, resulting in both higher processor and network utilization.

As a final note, our preliminary experimental results do not account for the effects of the memory hierarchy on the working sets of different jobs. As a consequence, buffered coscheduling requires a larger main memory in order to avoid memory swapping. We consider this as the main limitation of our approach.

## 4. Conclusion

In this paper, we presented buffered coscheduling, a new methodology for multitasking jobs in parallel and distributed systems. This methodology significantly improves resource utilization when compared to existing work reported in the literature. It also allows for the implementation of a global scheduling policy, as done in explicit coscheduling, while maintaining the overlapping of computation and communication provided by implicit coscheduling.

We initially addressed the complexity of a huge design space using three families of synthetic workloads — *Barrier*, *News*, and *Transpose* — and two types of networks — a high-performance Myrinet-based network and a commodity switch-based network. Our experimental results showed that our methodology can provide better resource utilization, particularly in the presence of load imbalance, communication-intensive jobs, or a commodity network.

In the future, we intend to examine the throughput and response time of parallel jobs when using buffered coscheduling and then comparing its performance to implicit coscheduling or a space-sharing commercial solution such as LSF. We will also consider the effects of the memory hierarchy in a real application rather than in synthetic workloads as presented here.

## References


[1] A. C. Arpaci-Dusseau, D. Culler, and A. M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proceedings of the 1998 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, Madison, WI, June 1998.

[2] R. A. F. Bhoedjang, T. Rühl, and H. E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, November 1998.

[3] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawick, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, January 1995.

[4] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proceedings of the 1996 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, Philadelphia, PA, May 1996.

[5] D. G. Feitelson and M. A. Jette. Improved Utilization and Responsiveness with Gang Scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[6] A. Gupta and V. Kumar. The Scalability of FFT on Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):922–932, August 1993.

[7] A. Gupta, A. Tucker, and S. Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. In *Proceedings of the 1991 ACM SIGMETRICS Conference*, pages 120–132, May 1991.

[8] A. Hoisie, O. Lubeck, and H. Wasserman. Scalability Analysis of Multidimensional Wavefront Algorithms on Large-Scale SMP Clusters. In *The Ninth Symposium on the Frontiers of Massively Parallel Computation (Frontiers'99)*, Annapolis, MD, February 1999.

[9] M. A. Jette. Performance Characteristics of Gang Scheduling in Multiprogrammed Environments. In *Supercomputing 97*, San Jose, CA, November 1997.

[10] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph. Implications of I/O for Gang Scheduled Workloads. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[11] S. Nagar, A. Banerjee, A. Sivasubramaniam, and C. R. Das. A Closer Look At Coscheduling Approaches for a Network of Workstations. In *Eleventh ACM Symposium on Parallel Algorithms and Architectures, SPAA'99*, Saint-Malo, France, June 1999.

[12] W. E. W. Patrick Sobalvarro, Scott Pakin and A. A. Chien. Dynamic Coscheduling on Workstation Clusters. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 231–256. Springer-Verlag, 1998.

[13] F. Petrini. Network Performance with Distributed Memory Scientific Applications. Submitted to the Journal of Parallel and Distributed Computing, September 1998.

[14] F. Petrini and W. Feng. Scheduling with Global Information in Distributed Systems. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'00)*, April 2000.

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

[16] P. Sobalvarro and W. E. Weihl. Demand-Based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors. In *Proceedings of the 9th International Parallel Processing Symposium, IPPS'95*, Santa Barbara, CA, April 1995.

[17] K. Suzaki and D. Walsh. Implementing the Combination of Time Sharing and Space Sharing on AP/Linux. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 83–97. Springer-Verlag, 1998.