A First Implementation of In-Transit Buffers on Myrinet GM Software*

S. Coll, J. Flich, M. P. Malumbres, P. López, J. Duato and F.J. Mora
Universidad Politécnica de Valencia
Camino de Vera, 14, 46071–Valencia, Spain
scoll@gap.upv.es

Abstract

Clusters of workstations (COWs) are becoming increasingly popular as a cost-effective alternative to parallel computers. In these systems, the interconnection network connects hosts using irregular topologies, providing the wiring flexibility, scalability, and incremental expansion capability required in this environment. Myrinet is the most popular network used to build COWs. It uses source routing with the up*/down* routing algorithm. In previous papers we proposed the In-Transit Buffer (ITB) mechanism that improves network performance by allowing minimal routing, balancing network traffic, and reducing network contention. The mechanism is based on ejecting packets at some intermediate hosts and later re-injecting them into the network. Moreover, the ITB mechanism does not require additional hardware as it can be implemented on the software running at Myrinet network adapters.

In this paper, we present a first implementation of the ITB mechanism on Myrinet GM software. We show the changes required in packet format and the modifications performed in the Myrinet Control Program (MCP). In addition, both the overhead introduced by the new code and the cost of extracting and re-injecting packets are measured. Results show that, even for this simple implementation, code overhead is only about 125 ns per packet and the message latency increase for messages that use the ITB mechanism is around 1.3 μs per ITB. This is the first attempt to implement this mechanism, showing that a real implementation of ITBs is feasible on Myrinet COWs, and the associated overhead does not restrict the potential benefits of this mechanism.

1. Introduction

Clusters Of Workstations (COWs) are currently being considered as a cost-effective alternative for small-scale parallel computing. COWs do not provide the computing power available in multicomputers and multiprocessors, but they meet the needs of a great variety of parallel computing problems at a lower cost.

In COWs, network topology is usually fixed by the physical location constraints of the computers, so the resulting topology is typically irregular. In networks that use source routing [1] (as opposed to distributed routing [5]), the path to destination is built at the source host and it is written into the packet header before delivery. Switches route packets through the fixed path found at the packet header, being simpler and faster than those in distributed routing, because no route decisions have to be made in them. However, the path followed by a packet can not be dynamically changed in order to avoid congested areas. In other words, the routing strategy is static and known before packets are sent. Myrinet network [1] is an example of a network with source routing.

Different routing algorithms suitable for irregular networks with source routing have been proposed. The most well-known routing algorithm is the up*/down* routing. It is used in Autonet [5] and Myrinet networks [1]. The up*/down* routing is quite simple. It is based on an assignment of direction labels to links. To do so, a breadth-first spanning tree is computed and then, the “up” end of each link is defined as: (1) the end whose switch is closer to the root switch in the spanning tree; (2) the end whose switch has the lower ID, if both ends are connected to switches at the same tree level. As a result, each cycle in the network has at least one link in the “up” direction and one link in the “down” direction. Thus, cyclic dependences between channels are avoided by prohibiting packets to traverse links in the “up” direction after having traversed one in the “down” direction.

In [2, 3] we evaluated the up*/down* routing algorithm in networks with source routing. From this study, we identified three major factors that limit performance:

- Non-minimal routing. As network size increases, routing algorithms based on spanning trees, like up*/down*, tend to use paths longer than the minimal
In previous papers [2, 3] we presented a new mechanism (In-Transit Buffer, ITB) for source routing networks. Basically, this mechanism avoids routing restrictions by ejecting some packets at intermediate hosts (in-transit hosts) and later re-injecting them into the network. For instance, when applied to up*/down* routing, an invalid path with one “down” → “up” transition is split into two valid up*/down* sub-paths. The dependence between “down” and “up” channels is broken by means of the in-transit buffer mechanism.

By avoiding routing restrictions, the ITB mechanism allows the use of minimal paths for every source-destination pair. As a consequence, packets do not need to frequently cross the root switch, allowing a good traffic balance, not achieved by routings based on spanning trees. One example of this is shown in Figure 1. There is a minimal path between switch 4 and switch 1 (4 → 6 → 1), but it is forbidden by the up*/down* routing because it uses an “up” link after a “down” link at switch 6. However, this path is allowed by the ITB mechanism by using one host at switch 6 as an in-transit host to break the dependence. In this case, the non-valid up*/down* path (4 → 6 → 1) is split in two valid up*/down* paths (4 → 6 and 6 → 1) by using the ITB mechanism. Notice that more than a single ITB can be needed in a path. In summary, by using ITBs, minimal routing can be guaranteed while keeping the deadlock-free condition.

Finally, by ejecting and re-injecting packets at some hosts, network contention is reduced. Ejected packets free the channels they have reserved, thus allowing other packets requiring these channels to advance through the network (otherwise, they would remain blocked). On the other hand, the mechanism can be easily implemented by modifying the network control program without changing the network hardware. Although it was originally proposed to be used for the up*/down* routing, it can be efficiently applied to any source-based routing algorithm [2, 3].

The main drawback of the mechanism is that ejecting and re-injecting packets at some hosts introduces an overhead that may increase message latency. An efficient implementation may help in keeping this overhead low. Obviously, as the number of ITBs used by a packet increases, packets may suffer a higher latency penalty. However, this latency penalty is only noticeable for short packets and at low network loads [2, 3].

2. Motivation

In [2, 3] we evaluated by simulation the behavior of the ITB mechanism using a network model based on Myrinet. Results showed that network performance can be improved. In particular, when compared with up*/down*, network throughput can be easily doubled and, in some cases,
trumped. In addition to this performance improvement, we claimed that only the software that runs on each Myrinet network adapter (Myrinet Control Program, MCP) needs to be modified. In particular, the MCP has to detect in-transit packets in order to handle the ejection and re-injection procedure. To minimize the introduced overhead, we proposed programming a DMA transfer to re-inject the packet as soon as the in-transit packet header is processed and the required output channel is free. Therefore, the delay to forward the packet will be the time required to process the header and start the DMA. In [2, 3], we used a delay of 275 ns (equivalent to 44 bytes received) to detect an in-transit packet and 200 ns (32 bytes received) to program the DMA that re-injects the packet. These timings were taken by using a customized MCP that only considered in-transit packets and analyzing message delays on a real Myrinet network.

In order to obtain with more accuracy these timings and to demonstrate that it is feasible to implement the ITB mechanism without degrading network performance, this paper presents a first implementation of the ITB mechanism on a current version of Myrinet GM software [4]. We have checked the correctness of the ITB implementation, verifying that the mechanism works correctly. On the other hand, we have taken measurements about the introduced overhead of the new code in the sending and receiving tasks of the MCP software. Also, we have measured the additional delay of a message when uses an ITB, obtaining that it is quite similar to the one assumed in our previous work [2, 3].

The rest of the paper is organized as follows. In section 3 we present the GM software organization and the network adapter architecture. In section 4 the implementation of the ITB mechanism is shown, describing the modifications performed on the GM/MCP software. In section 5 some evaluation results are presented, analyzing the viability of our ITB implementation. Finally, in section 6 some conclusions are drawn and future work is presented.

3 GM Software Organization and Myrinet Network Adapter Architecture

GM is a message-based communication system for Myrinet. The GM system provides protected user-level access to the Myrinet (secure in multiuser, multiprogramming environments), reliable and ordered packet delivery in presence of network faults, network mapping and route computation, and other features that support robust and error-free communication services. Other software interfaces such as MPI, VIA, and TCP/IP are layered efficiently over GM. The GM system includes a driver, the Myrinet Control Program (MCP), a network mapping program, and the GM API.

The MCP runs on the processor inside the network interface card (NIC). The NIC architecture is based on a custom-VLSI chip, called LANai, which is comprised of a network interface, two packet DMAs (send DMA and receive DMA), a host DMA, and a 32-bit RISC processor (see Figure 2). The host I/O bus, the packet receive DMA, and the packet send DMA can request a maximum of one memory access per clock cycle. The on-chip processor requests up to two memory accesses per clock cycle (instruction and data). However, memory bandwidth is limited to two memory accesses per clock cycle, based on the following priority (highest to lowest): host I/O bus, packet receive DMA, packet send DMA, and the on-chip processor. Since every host I/O bus memory access is granted, the LANai chip along with the memory on its local bus appears as a block of synchronous memory from the host point of view.
The MCP is loaded on the local memory of the network adapter by the device driver at boot time. This software interacts concurrently with both the host processor and the network. The MCP is composed of four state machines: SDMA, RDMA, Send and Recv. SDMA and RDMA control memory transactions between the host and the sending/receiving buffers located in the NIC memory. Send and Recv are responsible for controlling transactions to and from the network. On the other hand, the MCP is also responsible of setting up and detecting the completion of transactions on each interface. It can do this by means of start and finish events that modify the state of the system. The overall state of the system is managed through a number of status bits, being some of them maintained by the LANai hardware while the remaining ones being controlled by software. An event handler is in charge of monitoring the state of the MCP, giving control to the state machine that handles the highest priority pending event.

4 ITB Implementation on Myrinet GM Software

First of all, a new packet type (ITB packet) has to be created to distinguish between normal Myrinet packets and in-transit packets. New packet types are assigned by Myricom upon request. The format of a Myrinet packet is illustrated in Figure 3.a. The Myrinet mapper computes the paths among all hosts and stores them in the NIC SRAM of each network node. The MCP accesses the routing table every time a packet is stored in the send queue for delivery, stamping the path required to reach its destination in the packet header. When a packet enters a switch, the leading byte of the header is used to select the outgoing port. Once the output port is assigned to this packet the leading byte is removed from the packet. When a packet enters a NIC, the leading two bytes identify the type of the packet (a normal GM packet, a mapping packet, a packet with an IP packet in its payload or an ITB packet).

Figure 3. Packet format: (a) Original Myrinet and (b) with ITBs.

Figure 4. MCP states machine.
As it has been shown in previous sections, a path with in-transit buffers is split into several up*/down* paths. Figure 3.b shows how a path is accommodated in the Myrinet packet header for the case of using two up*/down* paths with one ITB. At each in-transit node, an ITB tag and the length of the remaining path are required by the MCP to identify and re-inject the packet as soon as possible. The Myrinet mapper has to be modified to calculate paths with the proposed mechanism.

In order to support in-transit packets, the MCP has been modified. The modifications have been made taking care of introducing the minimum overhead. Therefore, we need a fast detection mechanism of an incoming in-transit packet in order to reprogram a DMA transfer and re-inject it as soon as possible (even if the packet is still being received), thus providing virtual cut-through switching for ITB packets.

The MCP Recv state machine, which deals with the packet reception tasks, has to detect in-transit packets and check whether the Send state machine (actually the send DMA engine) is free. Figures 4 and 5 show the changes required in the MCP code. Once an incoming ITB packet is detected, if the Send state machine is free, the send DMA has to be programmed to re-inject the packet. Notice that in this case the Recv state machine is responsible for the in-transit packet re-injection in order to minimize the overhead (avoiding one dispatching cycle delay). This is shown in Figure 4 as dashed lines. If the Send state machine is busy, the packet will be sent as soon as it becomes free, as indicated in Figure 4 in dotted lines. Finally, note that when the Send state machine is re-injecting an ITB packet and this packet is blocked in the network, the Myrinet Stop&Go flow control will stop the transmission and as a consequence will temporarily stop the DMA send operation. In this situation, the rest of the ITB packet will remain in its buffer until the transmission is resumed again. If the ITB packet was not completely received in the above situation, the receiving process will not be stopped until the last byte of the ITB packet is received and stored in its corresponding buffer.

Notice that the length of both sending and receiving queues have been kept without changes from the original MCP (two buffers each). As we are going to evaluate ITBs on an unloaded network, we do not need more buffers. Although not implemented yet, we propose [2, 3] the implementation of a buffer pool for incoming packets. This buffer pool could be implemented as a circular queue (as many commercial network adapters do) managed with two pointers: one pointing the first incoming packet and the other pointing the next available buffer. So, when a new packet arrives, it is stored at the end of the queue, changing the last pointer to accept new incoming packets. In the case the circular queue became full and a new packet arrive, this packet will be flushed. The GM software has mechanisms to retransmit missing packets. However, this situation is very unusual [2, 3], and only happens with traffic injection rates beyond saturation. Moreover, the shipped memory on current Myrinet network adapters (8MB) seems to be enough to minimize these situations.

On the other hand, to detect as soon as possible an in-transit packet, a new high priority event has been included.
(EarlyRecvPacket event). It is triggered by the LANai hardware when the first four bytes of a packet are received. Then, the event handler activates the code associated with this event (which is included in the Recv state machine) checking if the incoming packet is an ITB packet. If not, dispatching process is resumed. If so, the availability of the send DMA engine is verified either to start the re-injection of the in-transit packet or to turn on a new state flag, ITB packet pending, for a later, but high priority, re-injection. When the ITB packet sending process is finished, resources are freed and next reception is programmed (if there is no other ITB packet pending). Note that this happens for each input buffer (two input buffers in the current implementation). Figure 5 illustrates in detail these modifications.

5. Evaluation Results

The implementation described in section 4 has been made on the GM-1.2pre16 software distribution for Linux. The testbed is comprised of three 450 Mhz Pentium III-based computers running SuSE Linux 6.1 with kernel 2.0.36. Computers are attached to a Myrinet network with two M2FM-SW8 switches (8-Port Myrinet Switch with 4 LAN ports and 4 SAN ports) according to Figure 6. Host 1 and in-transit host use M2L-PCI64A-2 NICs (universal, 64/32-bit, 66/33MHz, Myrinet-LAN/PCI interface, PCI short card - 2 MB) and host 2 uses a M2M-PCI64A-2 (universal, 64/32-bit, 66/33MHz, Myrinet-SAN/PCI interface, PCI short card - 2 MB).

Once the modified GM/MCP has been verified to deliver messages correctly, several tests have been made using the gm_allsize test distributed by Myricom in GM-1.2pre16. Firstly, the overhead introduced by the new code in the normal MCP operation has been measured. This test evaluates the impact of adding ITB support to the network. Notice that this overhead will be suffered in both normal and ITB packets, but only once per packet, as it only affects to the receiving part of the MCP. The test has been done comparing the point-to-point half-round-trip latency of the original MCP with the modified one when sending messages between hosts 1 and 2, using up*/down* routes. Latencies have been obtained by averaging 100 iterations for each message size. Figure 7 shows the average latencies measured versus message length when using both MCPs.

As it can be seen, overhead is very low. Difference in measured latencies does not exceed 300 ns and, on average, is equal to 125 ns. Notice that, as message latency increases (more switches to cross and/or longer packets) the relative impact of this overhead decreases. In this case, with packets traversing 2.5 switches, the relative overhead (not shown) ranges from 1 % for very short packets to 0.4 % for long packets.

In order to measure the overhead experienced by messages that use ITBs, a second test has been made. This test evaluates the delay associated with the detection of an incoming in-transit packet, its ejection and the re-injection into the network. It has been calculated by subtracting the point-to-point half-round-trip latency of messages being sent between hosts 1 and 2 using the up*/down* path.
Figure 7. Message latency overhead of the new GM/MCP code.

Figure 8. Message latency overhead of the in-transit buffer mechanism for in-transit packets.

To guarantee that this comparison shows only the overhead due to the ITB, both paths have been generated not only for messages to traverse the same number of switches but also to cross the same kind of ports (LAN or SAN). It has to be stated that the latency through a switch depends on the type of traversed ports.

Figure 8 shows the point-to-point half-round-trip latency for messages sent between host 1 and 2 without in-transit buffers (UD) and with one ITB (UD_ITB) versus message length. The resulting absolute overhead is also plotted. As it can be seen, the cost of detecting an ITB packet and handling its re-injection is around 1.3 $\mu$s. This value is higher than our estimations used in previous studies (around 0.5 $\mu$s) [2, 3]. However, notice that this increased delay only will be important when, after detecting an in-transit packet, the required output port is free. In other case, the in-transit packet has to wait until it is freed. Hence, we expect that results for medium and high network loads will not significantly change. On the other hand, the relative overhead introduced by each ITB decreases as message latency increases, and it ranges from 10% for short packets to 3% for long packets in our tests. Moreover, the measured message latency does not take into account the delay due to network contention. So, the actual relative overhead of ITBs will be lower.

6. Conclusions

The ITB mechanism was proposed to improve performance on networks with source routing and wormhole switching. In particular, we demonstrated by simulation that the ITB mechanism significantly improves network performance in Myrinet networks. In this paper, we have implemented the ITB mechanism in the Myrinet GM software in order to obtain accurate measurements of its associated delays. We have changed the MCP code...
taking into account that the overhead introduced by ITB code be as low as possible and the implementation keep the main structure of the MCP.

Once the MCP has been modified with the ITB code, several tests have been run in order to check its correct functionality. Then, we measured the temporal overhead that ITB code introduces in the normal operation of MCP, showing that is around 125 ns per packet, on average. Also, several tests have been performed to calculate the additional latency of a packet that uses an in-transit host. Results show that each ITB increases message latency by 1.3 μs. On the other hand, the relative importance of these overheads is reduced as the traversed distance in the network and/or message length is increased.

In summary, in this paper an implementation of the ITB mechanism has been developed. Several measurements and tests have been carried out, checking the correctness of the ITB implementation and obtaining its associated delays. This implementation is only valid to measure the overheads of using ITBs on a minimal network configuration and to demonstrate the viability of using ITBs on general Myrinet networks. Although we have tested the implementation on an empty network (no contention), it can be easily improved (with few additional instructions) to support medium and high network loads. So, the next step in our work will be improving this implementation in order to test it in medium-sized Myrinet networks. So, we definitively will prove the behavior of our mechanism analyzing the impact of using ITBs in the execution time of distributed applications.

References


