Exploiting SCI in the MultiOS management system

Ronan Cunniffe, Brian A. Coghlan; *Dept. Computer Science, Trinity College Dublin;*
`ronan.cunniffe@cs.tcd.ie, coghlan@cs.tcd.ie`

Abstract--
A cluster management framework is described that takes advantage of SCI to transfer operating system images to and from cluster nodes. Users may schedule access to compute nodes via a management node, and denote the environment to be loaded. They may also take a snapshot during a session. This mechanism allows a cluster to be shared among what would otherwise be incompatible research projects. SCI allows this to be done without excessive overhead.

I. INTRODUCTION

This work is related to our MultiOS cluster management framework for OS and systems research [1], which allows ordinary users to download and upload their own operating system images to and from individual cluster nodes. Leaving aside the administrative issues, low network bandwidth has previously conspired against such an approach, and existing any packages [2] assume the limitations of a LAN-type network and permit only a small and fixed number of stable environments, updated only infrequently and then with manual supervision by system administrators. The MultiOS framework is intended to take advantage of SCI and to provide both automatic scheduling and interactive control for any number of environments, to support environments of any degree of stability, and to do this for clusters of any size.

II. OVERVIEW OF MULTIOS

The MultiOS framework provides two abilities: to reserve cluster nodes for exclusive use and to transparently switch over from one user's configuration to that of the next scheduled. It is not a single-system-image tool for a cluster [3] in the sense that it does not care what it is transferring, nor does it have any role once the transfers are complete.

Architecturally, the framework is a combination of a small number of core services running on a dedicated management node, a web-based interface and an image server to provide storage for the environment images, and these elements can be spread across a number of machines if that suits a particular cluster.

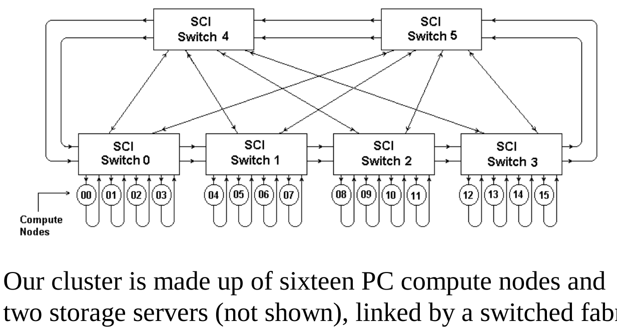
We will provide a simple scheduler for handling reservations for exclusive use of set of cluster nodes. We propose to use a single web-based interface, which is chosen for reasons of flexibility, platform independence, and universal access, for both interactive control and the reservation system.

The exact tools required to manage a particular cluster are very likely to be specific to that cluster, and what we propose is to provide explicit but filtered access to a modular toolset that can be extended or modified by an administrator. By filtering the commands, the MultiOS server can enforce the schedule and prevent user misbehaviour, such as triggering a command affecting a node reserved by somebody else.

In keeping with a modular design philosophy, both the web interface and the scheduler are essentially plug-ins to the core services, particularly as we foresee that the basic design goal of the MultiOS framework - the ability to manage and transport environment images for a cluster - could form a useful subsystem within a more elaborate management scheme.

III. IMPLEMENTATION OF MULTIOS

In the following, we will describe the framework with reference to our own cluster, but point out where it can be generalised for other configurations.



Our cluster is made up of sixteen PC compute nodes and two storage servers (not shown), linked by a switched fabric of SCI links. Exactly how the storage servers will connect is not yet finalized, possibly on the ringlets of nodes 0, 4, 8, and 12. Each compute node has 256MB of DRAM and a 2GB local hard-disk. The storage servers are connected to a large RAID, and all machines are connected to the external network through a firewall via 100Mb/s Ethernet. Also within the firewall are two NIS servers and a http server. All normal access to the cluster is through this network connection and physical access to compute nodes and servers is minimised. The MultiOS server will execute either on an extra server node, or on one of the existing servers. Over time, it is hoped to increase the number of compute nodes by increasing the number of nodes in each of the compute node ringlets.

The central idea behind the MultiOS framework is that between two successive zero-management ‘research’ sessions, there is a ‘management’ session, where all access is suspended, and the environment installed on the local hard-disks can be changed by management software according to the schedule or via user interaction with the MultiOS server. Implementing this requires that the MultiOS server be able to:

1. reset any compute node - independent of the running environment (i.e. in *hardware*)
2. gain control of a given compute node at boot-time.
3. optionally install and run management software on the compute node that does not use the local disk.

A hardware reset mechanism is required in order to enforce the schedule, since there is no guarantee that a running environment will shut down gracefully on request, and indeed highly experimental work is quite likely to crash or lock up the hardware it is running on. This mechanism is the key that allows the illusion of ‘bare-machinery’ for development environments and total transparency to the running software. It also provides a useful mechanism for allowing a researcher to recover a crashed node or nodes from their desktop. In our case, this mechanism will either be implemented using a LonWorks network [4] with the module in each compute node wired to the reset pin, or a parallel switched 100Mbps Ethernet fabric with modified wake-on-LAN [5]. Obviously, there are many alternative solutions, but the ability to reset a named node on request is one of the crucial elements that the MultiOS server relies on.

Ensuring that the MultiOS server gains control during boot can be done with a slightly non-standard use of standard mechanisms [6, 7] invented for booting diskless workstations, and several implementations are available [2, 8, 9]. Each compute node is set up to use its Ethernet card as a boot device. The network card queries the network for its own identity and for the name and location of a file to download and execute. The non-standard element is that the MultiOS server is allowed to switch the nominated file, which gives it *de facto* control over what is first executed on the requesting node. In the MultiOS framework, there are two executables. The first is the management environment. The other is a tiny executable which simply hands control to the boot block of the local hard disk, causing the installed OS to boot as though the network interrogation phase had not occurred.

It is difficult to write a single management environment with enough flexibility to handle any network topology, any network technology, or indeed combination of technologies. There are also variations on transport methods. The MultiOS approach is to use a fully featured operating system - Linux. The downloaded executable is a Linux kernel configured to run with filesystems only in memory and on remote disk. A Linux installation offers SCI drivers, raw disk I/O and UNIX commandline tools, such as *dd, gzip, diff, rcp*, etc., to which can easily be added new transport primitives. Almost any transport mechanism can be constructed with relative ease on these foundations.

IV. Moving OS images using SCI

The main feature of OS images is that they are large. In our case, in the absolute worst case, if all 16 compute nodes require loading/saving of raw disk images simultaneously, 32GB needs to be moved and then stored. Obviously, any steps available for reducing this quantity should be taken, particularly if the MultiOS framework is to scale with an increased cluster size. However, any comprehensive solution must allow for the worst case, and this is where the performance of SCI makes the MultiOS scheme workable even for larger clusters.

We assume the following performance figures with regard to our cluster:

- Disk read and write performance are identical.
- Best case transfer rate of the image storage server is 50MB/s.
- The observed sustained transfer rate of the compute node hard-disks is 9MB/s.
- SCI transfers at up to 75MB/s have been observed, (limited by the PCI bus).
- 100Mbps Ethernet has a best case limit of 8MB/s.

The term ‘sustained transfer rate’ has been used for hard disks both because the manufacturers usually specify burst bandwidth, and because hard disks frequently achieve real performance very close to their nominal maximum sustained transfer rate when executing large contiguous reads or writes, as will happen here.

2GB Raw using SCI

Nodes	Network traffic	Time
4	8GB@ 36MB/s	3.8 mins
8	16GB@ 50MB/s	5.5 mins
16	32GB@ 50MB/s	10.7 mins
32	64GB@ 50MB/s	21.3 mins

2GB Raw using 100Mbps Ethernet

Nodes	Network traffic	Time
4	8GB@ 8MB/s	17.1 mins
8	16GB@ 8MB/s	34.1 mins
16	32GB@ 8MB/s	68.3 mins
32	64GB@ 8MB/s	136.5 mins

The effect of the SCI bandwidth is clear in these two tables. Note that the bottleneck in the SCI table is a hard disk in each case, in the first row it is the combined writing speed of 4 node hard disks, and the other rows it is the server RAID performance limit.

While the above tables describe the worst case scenario, there are a number of ways of reducing the quantity of data transferred. If we still insist that the OS image on each compute node is unique, and must be handled separately, then there are two obvious ways of reducing the size of each: shrink the image using compression - trading computation against bandwidth - or reduce the uncompressed size of the installation to well below the full 2GB. Both compression and smaller-than-available-space approaches are already used by LAN-based OS managers [BPB] for reasons of bandwidth. The former is unlikely to be helpful with SCI since the computation time will dominate. The latter is generally unappealing as a restriction but may still be useful for an SCI cluster if the researchers are forced to pay for their image storage.

However, two installations of a given OS are likely to be very nearly identical, assuming the same configuration and that user data is not included. To exploit this, the MultiOS server retains a reference copy, downloads this to all compute nodes requesting their personal copy, then downloads a patch to each. In the reverse direction, the server downloads the reference copy again to each node, which generates a new patch by looking at the differences between the copy it is receiving, and what is on the local disk.

The most efficient way of transmitting the reference image is a multicast protocol. Although SCI does not implement multicast, similar behaviour and performance can be gained by sending data to one compute node, which then propagates it to another node, and so on in daisy-chain fashion. If the block size is kept small, the end-to-end propagation latency can be made very small compared to the time taken by the node's hard disk to read/write the data. The primary drawback is the huge amount of redundant network traffic, but if bandwidth and topology do not become a bottleneck, this process will be able to keep all hard-disk queues saturated, and therefore approach ideal multicast performance.

If we assume 100MB of differential information for each compute node:

100MB patches on 2GB multicast

Nodes	Network traffic	Storage	Time
4	2GB@9MB/s *+ 0.4GB@36MB/s	2.4GB	3.9 mins
8	2GB@9MB/s + 0.8GB@50MB/s	2.8GB	4.0 mins
16	2GB@9MB/s + 1.6GB@50MB/s	3.6GB	4.2 mins
32	2GB@9MB/s + 3.2GB@50MB/s	5.2GB	4.8 mins

This assumes that the patches are downloaded after the entire image has been written, although it is possible to download the patches in parallel with the image, since the image storage server has bandwidth to spare during this task. Either way, the 2GB download time dominates, even for 100MB patches for each of 32 compute nodes.

We note that since the sustained transfer rate for current disk drives is not far above the bandwidth of 100Mbps Ethernet, SCI may not perform dramatically better than Ethernet using IP multicast. The drawback to IP multicast is that it is essentially a broadcast protocol with local filtering, which will saturate the entire cluster. By contrast, SCI propagation multicast generates no extraneous traffic, making it far more suitable for use in a cluster whose structure allows for logical division into semi-independent subclusters.

However, the image server network connection is saturated, even if only modifying a fraction of the cluster. If this disk or its host server is used for other purposes, performance (as viewed by other cluster users) will deteriorate considerably. If this is not acceptable, then a separate dedicated image storage server may be required, together with enough independent bandwidth in an appropriate topology to support it.

V. PROPAGATION OVER A PARTITIONED CLUSTER

Although the drivers don't support doing this yet, a switched SCI fabric can be partitioned such that no traffic can cross boundaries established in the switches. The simplest way to do this is to modify the routing tables to fence requests at the second stage of SCI switches (fencing responses would also be needed for a complete solution). This yields a partitioning quantum of one first-stage switch (4 nodes in our case).

Partitioning allows systems research to co-exist with normal use. Rules can be imposed on the scheduler that ensure that some partition of the cluster is always running a default multi-user execution environment. In our case we intend to support a single default environment (Linux), which runs on all nodes at all times except when those nodes are explicitly scheduled to be running something else.

All the same, if a cluster has been divided into two or more logically separate subclusters, it may not always be possible to isolate propagation traffic completely. Any network whose OS images are only available from one point in the fabric, or which is broken into more partitions than there are OS image sources, is likely to have this problem, and all propagation traffic must travel through the subcluster in which the image server is located. The only alternative is to provide dedicated bandwidth that is reserved for exclusive use by the OS image server.

Our cluster is a good example of the problem. We intend to allow the cluster to be treated as four 4-node subclusters, partitioned per first-stage switch. If the image servers are only connected into switches 0 and 2, they cannot reach 1 or 3 without trespassing on local bandwidth. However, there is a compromise solution, where the full bandwidth of the server's SCI connection would not be exploited by the MultiOS framework. In this scheme, the OS image only propagates to a single node that is a member of the subcluster being considered, and more aggressive propagation strategies can be employed within the subcluster itself, on its internal SCI connections.

VI. Dynamic Propagation routing over an SCI network

It emerges from this that careful consideration of propagation routes is valuable for the MultiOS framework. But routes change depending on a large number of factors, including the number of image servers, their location, the number of compute nodes, the allowed subclustering, the bandwidth of the links, and any change in distribution of addresses.

Obviously, these routes could be encoded in a central configuration file, since the changes mentioned above all represent planned physical upheavals. However, if some node or set of nodes goes out of service, it would be immensely valuable if these nodes could be ignored and the routing so adapted that the system would cope with the failure to the extent that it does not affect other nodes.

Our proposed solution is to include dynamic routing as one of the core services of the MultiOS framework, via a propagation routing server that maintains a configuration database detailing the topology of the network. All the compute nodes query this server for information about which other nodes they must connect to under the propagation topology, and which other nodes they expect to connect to them. Failure to either query the server or make the expected contact implies a failed node, and this can be used to exclude the node. The setup algorithm is as follows:

```
query route server for upstream and
downstream nodes
REPEAT
    WHILE (all downstream nodes have not
    made contact) AND
          (not timed out)
    BEGIN
        wait for contact from
        downstream nodes
    END
    IF (timed out) THEN
    BEGIN
        report failures to server
        wait for new instructions
    END
UNTIL (not timed out)
contact upstream node.
```

Since contacting the upstream node is outside the loop, a failure anywhere causes a flurry of failure reports from all nodes upstream of the failure. This identifies the point of failure to the routing server, and also ensures that all affected nodes re-contact it for a new route. If there are no errors, then the propagation tree is built from the leaves towards the trunk, and the OS image server, which is at the root, is contacted only when the cluster is completely ready to transfer data.
Once the propagation tree is set up, the transfer can begin. The size of the transfer, transfer block size, where the data is to be written, whether there is a subsequent patch, etc. is assumed to be known. This information comes from the MultiOS server. The propagation algorithm is:

```
WHILE (data_blocks remaining)
BEGIN
    IF (this node is the 'root' or
    'source' node)
THEN
    load buffer from some local source
    ELSE
        send 'Clear-To-Send' to upstream
                                node
        wait for 'Done' from upstream node
    ENDIF

    IF (this node is not a leaf)
    THEN
    wait on 'CTS' from downstream node
    transfer buffer contents
    send 'Done' when transfer complete.
ENDIF

    do_local_processing
END
```

For linear propagation, the SISCI implementation of the algorithm (with error handling removed) for a transfer is as follows:

```
/*preamble*/
if (DnStreamNodeId)
{
  SCICreateDMAQueue(sd, &DmaQueue, adapter,
                    1,NO_FLAGS, &err);
}

/*propagate*/
do
{
  if (!UpStreamNodeId)
  {
    load_buffer(LocalSegment,SegmentSize);
  } else
  {
    SCISetSegmentAvailable(LocalSegment,
                           adapter
                           NO_FLAGS
                           &err);

    SendInterrupt(sd,adapter,LocalNodeID,
                  UpStreamNodeId, DMA_CTS);

    ReceiveInterrupt(sd, adapter,
                     LocalNodeID, DMA_ACK);
  }

  if (DnStreamNodeId)
  {
    ReceiveInterrupt(sd, adapter,
                     LocalNodeID, DMA_CTS);

    do
    {
      SCIConnectSegment(sd,
                        &DnStreamSegment,
                        DnStreamNodeID,
                        DnStreamSegID,
                        adapter,
                        NO_CALLBACK,
                        NULL,
                       SCI_INFINITE_TIMEOUT,
                        NO_FLAGS, &err);
      SleepMilliseconds(1);
    } while (err != SCI_ERR_OK);

    SCIEnqueueDMATransfer(DmaQueue,
                          LocalSegment,
                          DnStreamSegment,
                          LocalOfs,
                          DnStreamOfs,
                          SegmentSize,
                          NO_FLAGS,&err);

    SCIPostDMAQueue(DmaQueue,NO_CALLBACK,
                    NULL, NO_FLAGS,&err);

    SCIWaitForDMAQueue(DmaQueue,
                       SCI_INFINITE_TIMEOUT,
                       NO_FLAGS,&err);

    SCIDisconnectSegment(DnStrmSegment,
                         NO_FLAGS,&err);

    SendInterrupt(sd, adapter, LocalNodeID,
                  DnStreamNodeID, DMA_ACK);
  }
  do_local_processing(LocalSegment)
} while (more_data_to_come)

/*postamble*/
{
  SCIRemoveDMAQueue(DmaQueue,NO_FLAGS,
                    &err);
}
```

In these code fragments, local processing might involve writing the data to disk, writing it to a RAMdisk and patching against it, or some other function. This can be done in parallel with transferring the same data downstream.

VII. SUMMARY

We have discussed the exploitation of SCI within a framework for cluster management that is specifically tailored to OS and systems software research. Users may reserve one or more compute nodes via a web-console, denoting the environment image to be run. During their session they have the ability to take snapshots with highly customisable tools. The framework supports multiple users in parallel if the cluster topology permits, provides dynamic routing where appropriate to minimise the disturbance to other users. Hence a large range of researchers and research projects can be accomodated, enabling the capital investment of the cluster to be amortized over a greater number of funding sources.

Work on the framework, called MultiOS, began in October, 1999, and is still in progress.

VIII. ACKNOWLEDGEMENTS

Thanks to Prof.J.G.Byrne for support for the work described.

IX. REFERENCES

[1] Cunniffe, R., Coghlan, B. A., "Encouraging the Unexpected: Cluster Management for OS and Systems Research", submitted for publication. Contact authors.

[2] Rembo Technology, http://www.bpbatch.org

[3] Pfister, Greg; “in Search of Clusters”; Prentice Hall, 1998

[4] Foster, G.T., Glover, J.P.N., Warwick, K., "Flexible Distributed Control of Manufacturing Systems Using Local Operating Networks", Proc. LonUsers Internation Fall Conference, 1995

[5] Magic Packet Technology White Paper, AMD Publication no.20213, Advanced Micro Devices Inc., November 1995

[6] Wimer, W., "Clarifications and Extensions for the Bootstrap Protocol", IETF Request For Comments Document no.1542, October 1993.

[7] Sollins, K., "The TFTP Protocol (Revision 2)", IETF Request For Comments Document no.1350, July 1992.

[8] Free Software Foundation, "Grand Unified Bootloader" http://www.gnu.org/software/grub.en.html

[9] Yap, K; Savoye, R; "Network Interface Loader" http://nilo.sourcefourge.net/