USING OMNET++ TO SIMULATE A HETEROGENOUS DIGITAL VIDEO CLUSTER FABRIC

Martin N. Milkovits*
SeaChange International

Keywords: *simulation, networks, performance, contention*

Abstract

This paper describes the methodology used in developing a discrete event simulation model of a SeaChange® digital video cluster network. The SeaChange digital video cluster uses a complex fabric of heterogeneous network technologies to achieve the performance and reliability required for video on demand applications. The cluster network consists of three different interconnect technologies: PCI2.2, StarFabric and InfiniBand. I used OMNeT++ to handle the link contention, buffer contention and PCI arbitration challenges.

1 Introduction

The goal of this simulation is to predict performance and find the bottlenecks in a digital video cluster network. The SeaChange digital video cluster consists of massive storage arrays with redundancy, a high-speed internal network and many Gigabit Ethernet devices that transmit the video streams into the cable network and to the home [3].

The scope of the simulation will be traffic in the cluster network. The disk drive systems where the video is stored and the Gigabit Ethernet devices where the video is transmitted are abstracted to distribution models [2]. Therefore, the key areas of interest in the simulation will be the contention for the fabric links, the contention for the fabric buffers, and contention for the PCI bus which connects the two fabrics (StarFabric and InfiniBand) together.

2 Simulation Components

Simulation models created with OMNeT++ software are constructed of three primary components: modules, links and messages [5]. The modules are the agents in the simulation that create, send and receive messages. The modules will take information received in the message, and use it to change the state of the simulation. The messages themselves carry information between the modules across the links. The messages may be used for control or to represent an entity in the simulation. The links may be control path (and take no simulation time to transfer the message) or represent a data path. When a link is acting as a data path, the message is delivered after the necessary simulation time has passed to transmit that much data at the speed of the link.

The modules in this simulation which control the flow of messages throughout the fabric are InfiniBand switches, StarFabric bridges and switches and the PCI bus modules. The remaining RAID controller and Gigabit Ethernet modules simply create and destroy messages.

The data messages (*dMsg*) in this simulation represent 1024Byte data packets. The *dMsg's* contain their target node and card information and the modules use this information to determine the routing of the message through the system. There are two main types of control messages. The first (*rqst)* are used to request buffer and link resources between modules. The second (*qcheck)* prompt the PCI bus module to check the receive queues.

3 Managing Link and Buffer Contention

In order to transfer data across a network fabric, the device must have control of the transmission link and there must be a destination buffer that can accept the data. In this simulation model, each module in the system is responsible for accessing the necessary resources to transfer the *dMsg* and ensure a destination buffer. If the module also manages buffers, it must release the local buffers that the *dMsg* occupied when it is transmitted to the next component.

3.1 Sending a Message

Before a module transmits the *dMsg* message, it must gain access to the link or bus and the destination buffer. The *rqst* messages are used to manage these transactions over the control links. Each *rqst* message contains information about the access of the link/bus and destination buffer. For example, the module representing the disk controller will send the *rqst* message to the PCI bus module. If the bus is available, a component in the *rqst* message that represents the link access is set to **true** and is returned to the disk controller module. If the bus is not available, the PCI bus module would hold the *rqst* message until the bus becomes available. Holding a *rqst* message is the mechanism to manage flow control in the simulation network.

Once the *rqst* message is returned to signal link access, the disk controller module will attempt to grab a destination buffer on the StarFabric bridge (note there are no buffers in the PCI bus). Therefore, the *rqst* message is sent along the control link to the StarFabric bridge module. When the buffer is available, the StarFabric bridge module sets the buffer access message component to **true** and returns the *rqst* message. When both messages are sent and returned, the *dMsg* message may be sent. Any new *dMsg* messages that are received at the disk controller module during this process are simply added to the *queue* (see Figure 1).

3.2 Receiving a Message

When a *dMsg* message arrives, the receiving module checks to see if there are any *rqst* messages being held before releasing the link. If any *rqst* messages are found, they are returned to the requesting module. Likewise, when a *dMsg* message is sent from a module, the module checks if there is an outstanding *rqst* message before it releases the buffer. If so, the *rqst* message is returned and the resources remain used, otherwise, the buffer resources are released.

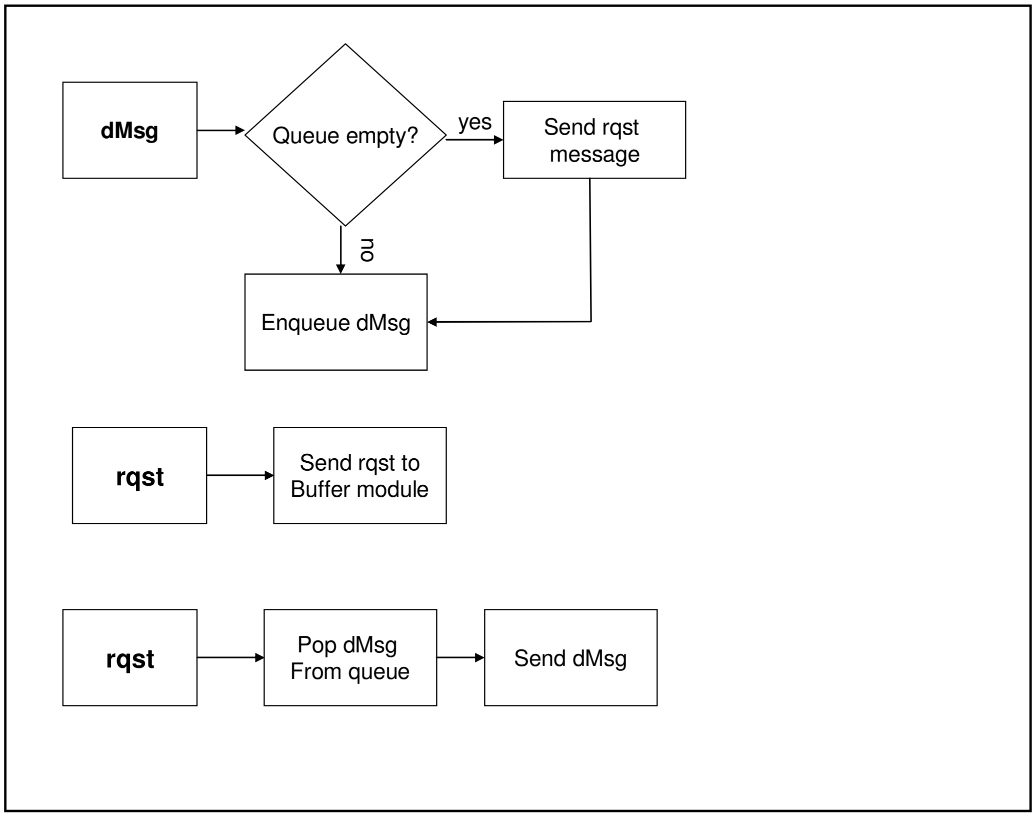

Figure 1: Disk Controller Module Buffer/Bus Access

4 Simulating the PCI Bus

The PCI bus is the common component between every fabric in the system and, hence, every module in this simulation. The PCI bus has the challenges of granting access to each device appropriately, of assuring that no one device hogs the bus, and of transferring the data to its destination.

These requirements make a simulation of the PCI bus as a process more appropriate than as a simple connection like the StarFabric or InfiniBand links. The PCI bus module was designed after the "Time-Shared Computer Model" on page 129 of *Simulation Modeling and Analysis* [1].

4.1 Transferring Data According to PCI bus speed

The *dMsg* has a component *transfer* that represents the length of in bytes message upon entry to the PCI bus module. Messages being transferred are held in the *work* array of the PCI bus module. While a message is being transferred, the *qCheck* message is scheduled for an interval of 240 ns (16 clock cycles). This simulates the time to transfer 128 Bytes of data. When the *qCheck* message fires, the *transfer* value on the *dMsg* is decremented by 128. This continues, (as long as there are no other messages arbitrating for the bus), until the *transfer* value is 0, at which time the *dMsg* message is sent to the destination module. If there are other messages arbitrating for the bus, they are moved to the *work* array and an additional 45nS is added to the *qCheck* time to account for PCI overhead (see Figure 3) [4]. See Section 4.3 for more information about PCI bus arbitration.

4.2 Granting Bus Access

The PCI bus module's *pciBus* table has an entry for each device connected to the PCI bus. When that device requests the bus, the array value for that device is set to 0. When the transaction for that device completes, the array value is incremented to 1. If a device requests the bus, but the *pciBus[devNum]* entry is set to 0, the request message is held until the previous transaction completes and the bus is released. At this point, the entry remains at 0, and the request message is returned to the device requesting the bus (see Figure 2).

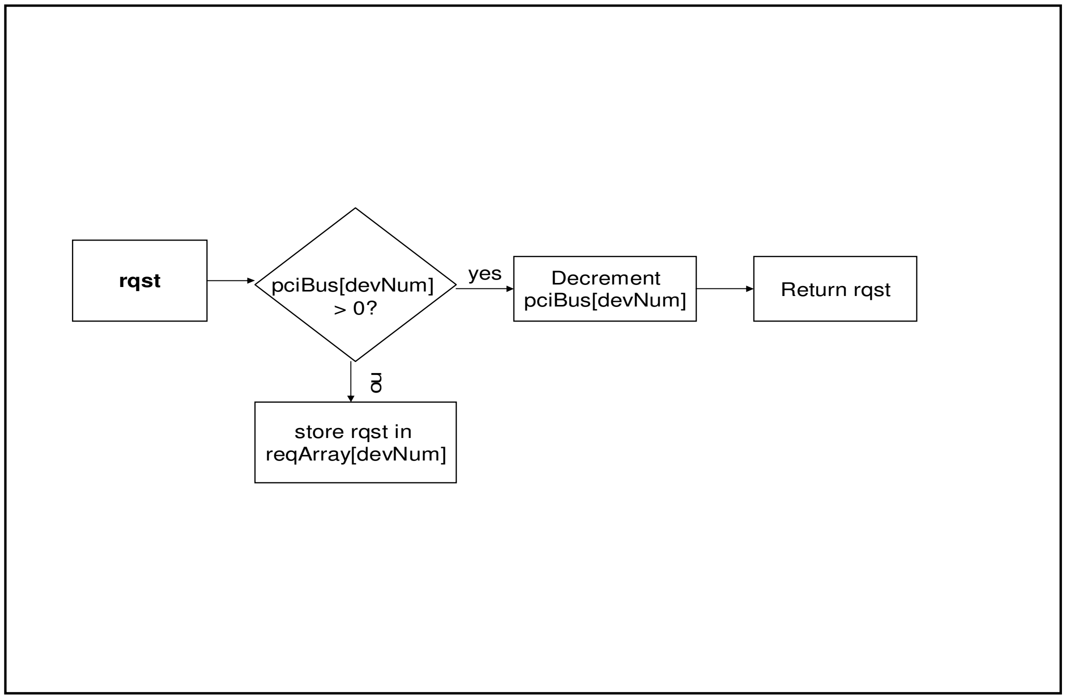

Figure 2: Granting Bus Access

4.3 Maintaining Bus Fairness

The *qCheck* message is used to maintain bus fairness. Note that every device can send a *dMsg* message to the PCI bus, but only one message may be transferred at a time. Each additional *dMsg* message is pushed to the *queue*. If the *dMsg transfer* is not 0 after the *qCheck* message fires, the *queue* is checked for *dMsg* messages. If there is a *dMsg* message in the *queue*, it is copied to the *work* array and the *dMsg* in the *work* array is pushed to the *queue* (see Figure 3).

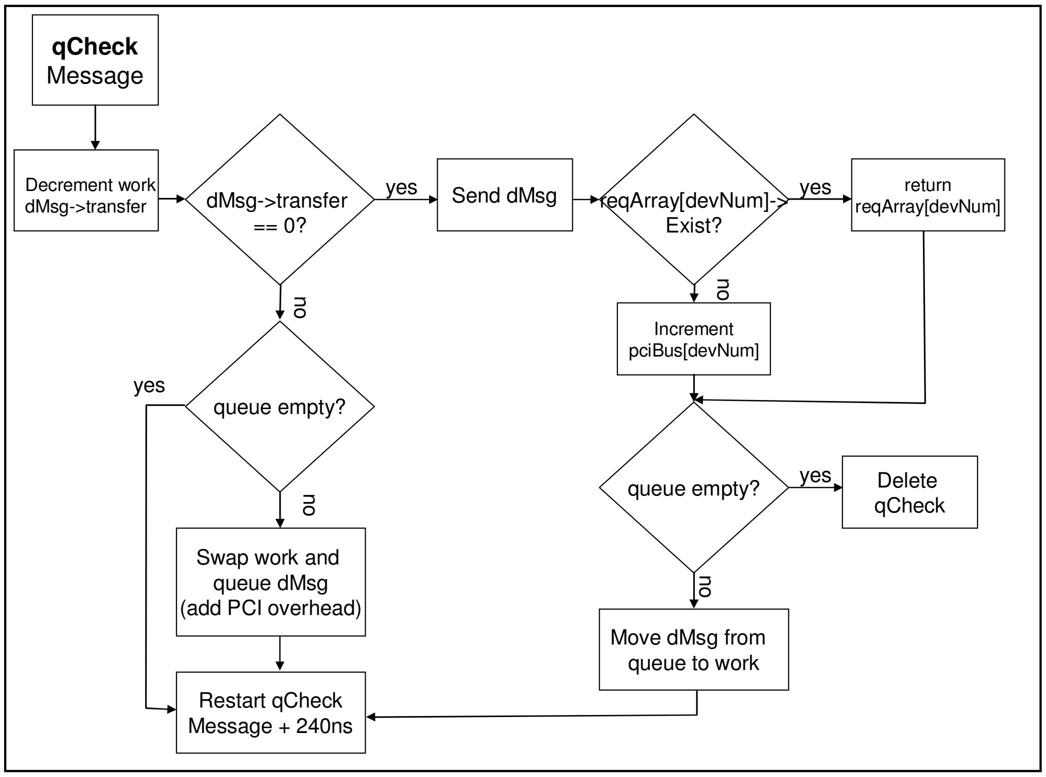
qCheck Message
Decrement work dMsg->transfer
dMsg->transfer == 0?
yes
Send dMsg
reqArray[devNum]-> Exist?
yes
return reqArray[devNum]
no
Increment pciBus[devNum]
no
yes
queue empty?
no
Swap work and queue dMsg (add PCI overhead)
Restart qCheck Message + 240ns
queue empty?
yes
Delete qCheck
no
Move dMsg from queue to work

Figure 3: Bus Fairness Algorithm

Conclusion

Using OMNeT++, a simulation model of a heterogeneous digital video cluster fabric was built. The simulation model handled buffer and link contention as well as PCI bus arbitration and contention. The results of this simulation model helped to better understand the relationships of these three networks and how the different buffer sizes, packet sizes and transfer rates affect multiple simultaneous streams of data [2].

References

[1] Law, Averill M., and Kelton, David W. *Simulation modeling and analysis.* McGraw-Hill, NY. 2003.

[2] Milkovits, Martin N. Digital Video Cluster Simulation. *Proceedings of the 2005 Winter Simulation Conference, M.E. Kuhl, N.M. Steiger, F. B. Armstrong and J. A. Joines, eds.* Orlando, Florida. 2005.

[3] SeaChange International, [online]. Available via <http://www.schange.com/products/mediacluster.asp> [accessed August 17, 2005].

[4] Shanley, Tom, and Anderson, Don. *PCI system architecture.* Mindshare Inc., Reading, MA. 1999.

[5] Varga, Andras. *OMNeT++ Version 3.0 user manual,* [online]. Available via <http://www.omnetpp.org/> [accessed August 17, 2005].

* **MARTIN N. MILKOVITS** received his B.A. in philosophy of mathematics from Colby College in Waterville, ME. He received his Master of Science in Computer Science degree from Rivier College in Nashua, New Hampshire, in 2005. He is currently a software engineer at SeaChange International. His research interests are in performance modeling and analysis. Feel free to contact Martin at mmilkovits@alum.colby.edu.