Continuous Discrete-Event Simulation of a Continuous-Media Server I/O Subsystem

Michael Weeks
University of Teesside, SST
Middlesbrough, TS1 3BA, UK
E-mail: michael.weeks@tees.ac.uk
Fax: +44 1642 342401

Chris Bailey
Department of Computer Science
University of York,
York, UK
E-mail:chrisb@cs.york.ac.uk

Reza Sotudeh
Department of Electronic, Communication, and Electrical Engineering
University of Hertfordshire
Hatfield, UK
E-mail:r.sotudeh@herts.ac.uk

Abstract

When designing computer systems, simulation tools are used to imitate a real or proposed system. Complex, dynamic systems can be simulated without the cost and time constraints involved with real systems. Experimentation with the simulation enables the system characteristics to be rapidly explored and system performance data to be generated, so encouraging modification to improve performance.

This paper details the experiences encountered when designing and building a proprietary continuous discrete-event object-oriented simulation in order to further investigate the performance of a proposed continuous-media server I/O subsystem. Previous investigations of the proposed architecture have been based upon mathematical models in order to calculate comparative performance. However, such static models do not take into account the dynamic properties of a system. A simulation tool was therefore built in order to assess quality of service under high system load conditions. The resulting simulation rapidly produced more realistic performance figures, in addition to providing a flexible simulator infrastructure for other unrelated projects.

1 Introduction

The object of our research is improved cost/performance ratios for the input/output (I/O) sub-systems of continuous--media servers. Previous investigations of the proposed architecture have been based upon static mathematical models. These models were used to gain comparative performance figures for various continuous-media server I/O subsystems, with the performance measure being based upon the maximum number of concurrent constant bit-rate (CBR) streams that can be supported by the I/O subsystem. This research highlighted bottlenecks in the servers design, the results from which were used to propose new configurations[9]. However, static mathematical models do not take into account the effect of buffering, bus and memory contention, and other dynamic properties of the system. A static model is also unable to provide data regarding the quality of service (QoS) of a streams delivery. This latter performance metric is of vital importance for continuous media data, if the client or viewer is to receive jitter-free video and/or audio data.

To investigate further the proposed I/O subsystem and to gain more realistic performance figures that are based upon quality of service criteria, a simulation tool was required. There are many computer-based simulation systems available both commercial and free-ware. A quick survey of available packages was unable to find a system that would meet our requirements of low-cost, short learning curve, and flexibility. It was decided that we would create a proprietary simulator and tailor it to our needs. This would also provide invaluable experience in building computer simulators. C++ was chosen as the simulation language as it is a fast, flexible, object-oriented language that is freely available, well-documented and highly portable.

1089-6503/00 $10.00 © 2000 IEEE

2 Simulator Requirements

There are three categories of simulation model; continuous-time, discrete-time, and continuous-time discrete-event. The latter category is the most suitable for a simulation based upon the interaction of function-based entities operating at differing, unevenly spaced and/or arbitrary intervals.

In continuous-time discrete-event simulations, the time parameter is conceptually continuous and the observation period is a real interval starting at zero. The operation path is completely determined by the sequence of event times and the discrete changes in the system due to these events. The simulator can advance simulated time directly from one discrete event to another, taking any necessary steps that are required to advance the state accordingly.

The simulators infrastructure should be;

Flexible and easy to learn The simulator should have a steep learning curve. It should be flexible and extensible so that it is relatively easy to add new functionality to the system.

Efficient Due to the large amounts of data that traverse a media servers I/O sub-system, the expected simulation will be highly processor intensive. For instance, a 33MHz PCI bus will require 66 million events per second (i.e. every clock edge) for itself, as well as every attached master and slave interface. The design under investigation utilises three PCI buses, therefore efficient simulator code is necessary in order to limit the time taken to complete an observation period. Therefore, the core scheduling program and individual objects are coded as efficiently as possible, in terms of speed and resource utilisation.

Portable Due to the varied computer resources available, the simulator must be platform and architecture independent. It should be possible to use any standard C++ compiler and operating system combination without any major changes to the code.

Object-Oriented Inheritance is used at the high-end of active simulation objects, in order to allow the scheduler to have common interface access to various entity types. Entities also use passive components such as queues, as well as more complex active-entities such as servers and interfaces. These enable the simulator to be more flexible and extensible, allowing new objects to be added without affecting the overall simulator infrastructure.

Documentation and Reporting Good documentation is necessary for several reasons. When building and debugging the simulator code, it's vital to be aware of the capabilities and software interfaces to the simulations objects. The operation of each object, and its interaction with others must be clearly specified.

Each object must also have the ability to report on its behaviour and performance, in order to prove its correct operation. Significant performance data, where appropriate, must also be logged to data files for analysis at a later date.

3 Simulation Strategy

The basic premise of simulation is that a complex system is replaced with a model that is simpler to observe, but has all the original systems major characteristics intact. To realise this, we must decide which components of the real system should be considered for the model. This abstraction requires certain initial assumptions that will decrease the complexity of the model, this in turn simplifies implementation and observability. This can be achieved by determining the key features of the simulation; i.e. what is it we are trying to achieve and/or measure, and what characteristics have an effect on them?

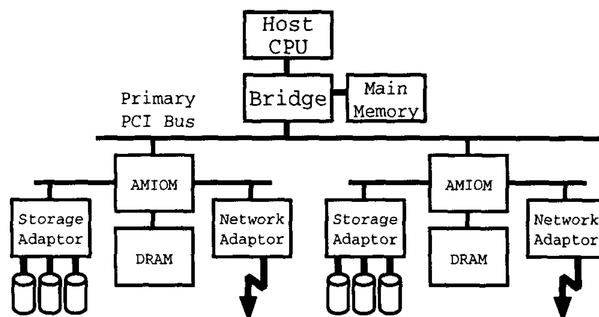

Figure 1. Modular bus-based streaming architecture

For the media server simulation, this work has been examined in a previous paper that derived the mathematical model for the system [9]. Figure 2 shows the architecture under investigation. The I/O module, under supervision from the host CPU system, controls concurrent data streaming from the storage device, via the temporary streaming memory buffers, to the network adaptor. In order to provide local scalable processing, the module incorporates an on--board CPU whose code and data stream are separate from the media data streams.

From previous studies, we know that bottlenecks occur on the data streaming pathways due to the high transport bandwidth that multiple continuous-media data streams require. These pathways must therefore be included in the

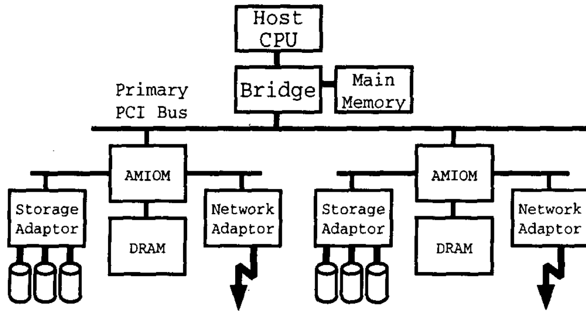

Figure 2. Modular bus-based streaming architecture

model. The stream pathway, and hence the simulation, consists of multiple network and storage adaptors, two subordinate I/O interconnect buses, and the streaming module.

In order to quantify the performance of the system, the measure of performance will be the maximum number of streams that the I/O subsystem can support before quality of service suffers due to contention. The latter shall be realised by monitoring the delay in servicing deterministic network requests.

Due to the size and complexity of the scenario under investigation, the quantity of data that the simulator can produce can be overwhelming for both simulation analysis tools and observer alike.

Therefore, only significant events must be logged. In this simulation we need to know all of the PCI bus occupancy rates, and the delay incurred when issuing network I/O requests. The former is achieved by logging any change in a bus's PCI Grant signal. The latter is recorded as the time difference between a stream-process request being issued, and the time it gains control of the network interface. Both resulting data files require some post-processing, using proprietary data-processing utilities, in order to make use of the data.

Several factors require consideration when executing a simulation run. The first is the level of resolution and accuracy of the observation points. For our purposes, the critical components of the simulation are the subordinate PCI buses. These are clocked at 33MHz, which means 66 million events per second due to the two phases of the buses clock. The observation points are therefore set at 1 nanosecond intervals, which means that each clock phase requires 15 simulator ticks. This gives an accuracy of 99% of the real PCI timing value.

The **observation period** determines the length of real-time that the simulation is modelled. The rule of thumb [6] for this period is at least ten cycles of every distribution. In our case, the storage request period is the least frequent distribution. Therefore, an observation period of approximately two seconds is required for 10Mbps streams.

The **warmup** period determines the length of time taken by the simulation to settle into a stable state. The complex and time-consuming nature of the model makes this very important. It is therefore imperative that the warmup period is kept as short as possible in order to minimise overall processing time. Several stream start-up scheduling schemes were employed in order to minimise the warm-up period. The most successful at low stream loading levels, introduced new streams to the simulation at equally-spaced time intervals over the period of a storage request. The effect of this traffic pattern must be carefully observed under high stream loading, however. We have chosen a warmup period length of four storage request periods.

In order to obtain accurate results from the simulation, several replications must be effected for each set of configuration data. A rule of thumb [6] requires three to four replications.

4 Simulator Infrastructure, Objects, and Analysis Tools

The simulation is fundamentally composed of a collection of active temporal-based entities and a scheduler, details of which are given below. Upon implementation, each object was verified inside a test harness for correct performance and operation, and the results documented.

4.1 The Scheduler

The scheduler is the most important part of the simulator infrastructure. It co-ordinates the entire simulation throughout the observation period, by monitoring active and passive entities in order to activate them at the correct time. The objects cannot interfere with the scheduler directly, but can do so only indirectly through modifications of the scheduler queue.

There are two approaches for scheduling simulator events; event-oriented and process oriented. For event-oriented systems, a procedure or function-member in the case of C++, is associated with each type of event in the system. The event procedure performs a required action to handle that type of event and it is invoked by the scheduler every time such an event occurs. A process-oriented approach, as used by the C++SIM Simulation Package [4], utilises a number of interacting processes that are executed in parallel. For our simulator tool we have opted for the event-oriented approach due to its ease of implementation. Also, the process-oriented approach raises the issue of portability, as process handling varies with operating system platforms.

The scheduler's initial task is to create entities and define their interactions at run-time as specified by configuration data. An internal "clock" is used to monitor the passing of simulated time. The scheduler then queries an entity's event time and compares it with the current time. If the comparison is favourable the scheduler requests the entity to activate and respond. Entity event responses are executed in pseudo-parallel. At any instance of real time, only one entity is serviced, but in terms of simulated time, many entities are serviced concurrently. Simulated time is discretely advanced when all entities have been processed for the current instance of simulation time.

4.2 Active Entities

There are two levels of entities; simulation devices and simulation components. Examples of simulation devices are high-level complex devices such as a bus, a bridge, a disk, or a CPU. A simulation component on the other hand can be active, such as an interface or server, or passive, in the case of queues, lists, arrays, etc. The use of the abstract terms high-level and low-level depend upon the context within which they are used. For instance, several simulation devices could be easily converted to components, then grouped together to form a **module** simulation device. The effect of this grouping is to make configuration far simpler. This technique however makes the simulator less configurable, but more importantly in our case, it saves precious time. The major difference between active simulation devices and components is that simulation devices inherit interface compatibility from an abstract *'simobj'* class, allowing the scheduler to dynamically create and reference different entities at run-time.

All active entities require at least two vital function members; nextEvent() and event(). The member nextEvent() returns the time of the next discrete event for that entity. The event() member activates the entity to respond as determined by the functionality of the object it simulates. Figure 3 illustrates the structure of a simulation object. Note that the upper level simulation object is accessed by the abstract class 'simobj', and that the component objects are accessed from within the simulation objects member functions.

4.3 Active Simulation Components

The following subsections give an overview of the active entities that are used in the simulation (figure 4).

Each object has a verbosity level which enables entities to report on their individual status. This is manually set at compile-time and allows observation of correct operation and to catch any problems. Differing verbosity levels enable varying report levels to be coded into objects. On certain objects, such as the PCI bus, the verbosity level can control the logging of significant information to files for later analysis.

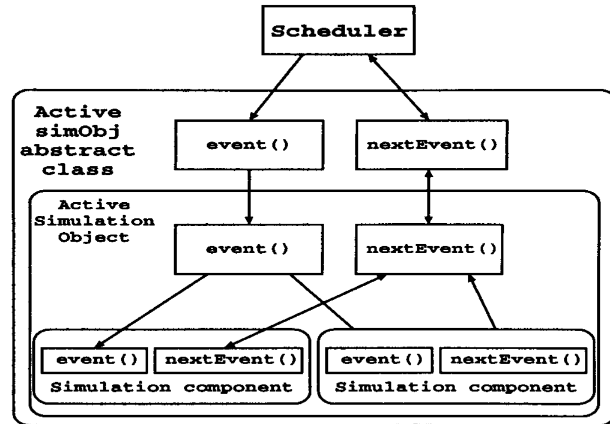

Figure 3. Member function calls within a composite simulation object

Documentation for the project is produced by commenting the source-code and using the 'Doxygen'[1] automatic documentation processor for C and C++. This package can generate on-line html pages, or a reference manual from specially formatted source code comments. By specifying a web server account and the source-code paths in a configuration file, on-line documentation can be automatically generated with minimal effort.

4.3.1 PCI-Interface

A PCI interface consists of two entities; a PCI master entity and a PCI slave entity. The purpose of these two entities are to provide a standard means of communication over a shared PCI bus entity.

The master and slave objects are designed to be components within a PCI device object. The interface objects connect to the shared PCI-bus via pointers to the PCI bus objects function members. A PCI device connects to the interfaces via data queues that are accessed via interface function members.

Both interfaces consist of a state machine, internal registers, and input and output queues. The state machine operates on both rising and falling edges of the PCI bus clock. Typically the interface reads data from the PCI bus on the clocks rising edge, and writes data to the bus on the falling edge, though there are a few exceptions.

The model specification requires PCI READ and WRITE commands only, therefore no other commands have

[1] http://www.stack.nl/ dimitri/doxygen/index.html

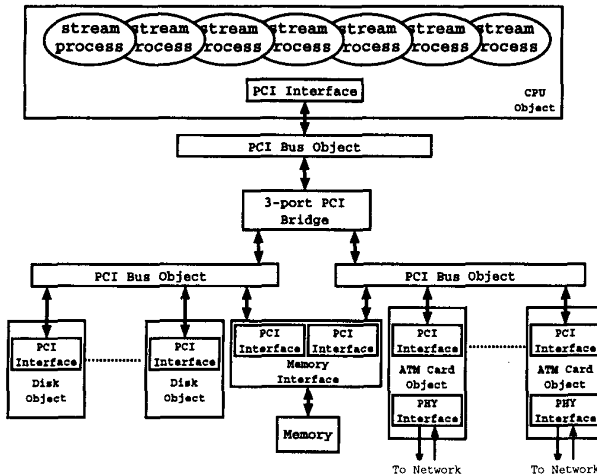

Figure 4. Software model of single bus-based I/O module

been coded into the master/slave interface state machines. The PCI interface entities match real PCI READ/WRITE transaction performance, under optimal conditions, and also when either master or slave inserts wait-states.

4.3.2 ATM Physical Interface

The ATM interface is a component in an ATM Network Interface Card (NIC) or device. It is a DMA-like object consisting of two state machines for ATM network input and output. No underlying technology, such as SONET, will be used for the physical connection. It will be assumed that a direct peer-to-peer path is made between between connecting ATM interfaces. In order to increase the speed of the simulation, the physical path widths will be one cell wide.

4.3.3 CPU Stream Process Object

For each media-stream being transmitted to the client, there will be a **CPU Stream Process** entity that controls it. It exists within a CPU entity (figure 5), and is intended to simulate a software process. A stream-process entity consists of two inter-linked state machines, each having access to all of the components contained within the CPU entity. One state machine is used for controlling the attached storage controllers, the other being used to control the attached network controllers.

The trigger for both state machines is a Poisson-event that is based upon the average I/O request rates of the specified stream. Upon receiving an I/O request event, the state machines issue requests to the relevant devices request queue in the CPU entity. When the device is granted, the state machine must request the PCI bus interface in a similar manner. With PCI access granted to a stream processes state machine, initialisation data (source and destination addresses, block size, etc.) can be written to the I/O device, before initiation data is written to the device to instruct it to act in semi-autonomous DMA mode. The stream process then releases the PCI interface for use by other processes.

When the semi-autonomous peripheral has concluded its transaction it raises an interrupt. The stream process with current control over that device then requests PCI access again. The stream process then clears the interrupt, so making the peripheral device ready to complete another transaction.

This PCI bus request/grant technique has a dual purpose. Without it, all stream processes would execute concurrently. A single stream with access to the PCI interface, blocks all other stream processes, therefore simulating real serial processor operation. To simulate context switching, a Poisson delay is introduced when PCI access is granted to a new stream process. This overhead is based upon data given in configuration files.

4.4 Active Simulation Devices

4.4.1 PCI-Bus Model

The PCI-bus object consists of passive bus lines, and an active arbiter. Real-world PCI bus arbiters decide on a "one cycle at a time" basis, and can preempt transactions when higher priority requests arrive. The C++ scheduler, however, works in pseudo-parallel, therefore an entity's priority depends upon its location within the scheduler queue. To solve this problem, all PCI-bus requests are given equal priority, and are stored in a FIFO request queue.

For every bus clock cycle, the arbiter checks for a request on one of the request lines. If a request line is asserted, the arbiter places that request in a FIFO arbiter queue and de--asserts the request line. When the PCI bus becomes idle, as indicated by 'Frame' and 'Grant' being de-asserted, the arbiter grants the bus to the device whose request is at the front of the arbiter queue. This technique allows the arbitration delay on consecutive transactions to be hidden. If multiple requests are made within the same clock cycle, all requests are processed with priority being given to the device with the highest request line.

In order to collect PCI bus utilisation data from the model, the PCI bus entity must record bus traffic patterns. The time that the PCI bus is in use can be found by monitoring the PCI master grant lines, and the PCI bus object therefore has the ability to record this data to a file. From these data files, two utility programs can provide performance figures.

The first, **'Gnuplot'**[2], is a command-line driven interactive function plotting utility that enables visualisation of mathematical functions and data. The second is a proprietary utility that extracts statistical data from the grant data files. This utility provides data such as the bus utilisation, bandwidth utilisation, bus efficiency and the number of transactions over a specified period.

4.4.2 CPU Object

The CPU object consists of PCI master and slave interfaces, multiple *CPU stream process entities*, and I/O request queues for each storage and network peripheral under its supervision (figure). Under our scenario, the continuous media data is stored on disk using a data-striping technique. This requires the use of multiple SCSI controllers, for which there exists one I/O request queue per controller. There also exists one I/O request queue per ATM controller. With one PCI master interface being shared by multiple *CPU stream process object's* a PCI request queue also exists. All queues in the object grant CPU resources using a FIFO policy.

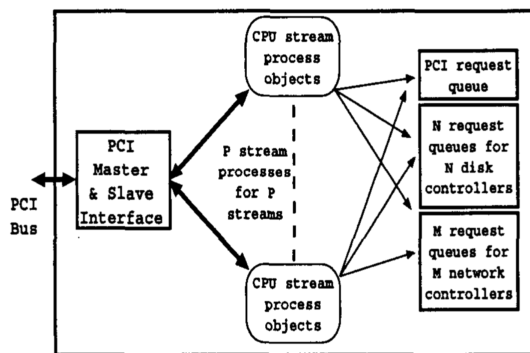

Figure 5. CPU object

4.4.3 3-Port PCI-Bridge Object

The PCI Bridge is a 3-port PCI-bridge based upon the operation of the Pericom Multi-Ported PCI-to-PCI Bridge Chip [1]. The object consists of three modified PCI slave and master interfaces (one each per port), and a bridge object that links them. Figure 6 shows the graphical layout of the object.

4.4.4 PCI-based Disk/Controller object

The disk object simulates a PCI-SCSI controller attached to a SCSI hard disk drive, and is based upon the operation of

[2]http://www.cs.dartmouth.edu/gnuplot_info.html

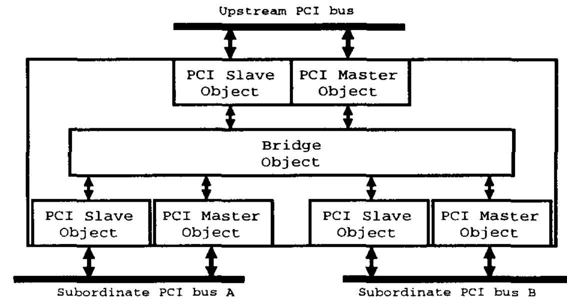

Figure 6. 3-port PCI Bridge object

the Symbios Logic SYM65C875A PCI-SCSI I/O Processor [8]. A disk entity (figure 7) consists of a state machine, internal registers, PCI master and slave interfaces and random or random Poisson distributed events to simulate disk access times.

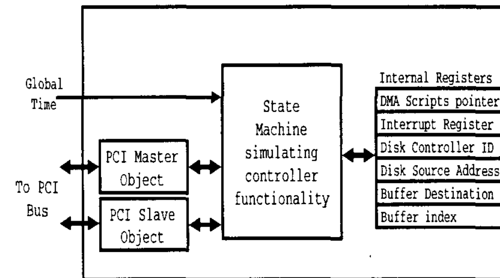

Figure 7. PCI-based disk object

The disk object has two basic modes of operations. The first is a "disk slave" operation, whereby PCI transactions access internal registers. The disk entity's data can be accessed via an external PCI master device. The disk entity's PCI slave interface transfers the data to/from the internal registers as determined by the PCI transactions address phase. The "disk master" operation provides PCI bus mastering capability, so 'stored data' can be streamed to another PCI device. The switch for this operation is an internal register flag (DMA Scripts Pointer). When in disk-master mode, the device transfers a quantity of disk-block sized packets over the PCI bus, to the streaming memory. Configuration data such as the disk block size and the number of blocks per I/O request can be varied to suit performance.

Once finished the device will set its corresponding interrupt line on the PCI bus. To clear this, the CPU streaming process must read then write to the disk objects internal interrupt register.

Mechanical disk latencies have been modelled in the disk object. The rotational latency of the disk is modelled by a

random number generator [5] that varies between zero and the full disk rotation time. Seek time is dependant upon mechanical properties and disc access patterns [7] and is less easy to model. Therefore, seek-time is modelled using a random Poisson distribution based upon the average seek-time. The average seek-time is determined by using a simple elevator-based disk-scheduling algorithm. This algorithm services stream requests as the disk arm sweeps across the surface of the disk (figure 8). Assuming that all streams are of equal data-rate, and the distance between the two furthest stream requests is equal to the maximum seek time, the total quantity of stream requests, S, are serviced in twice the maximum seek-time, $T_{seek_{max}}$. The mean seek-time, $T_{seek_{mean}}$, is therefore quantified in equation 1.

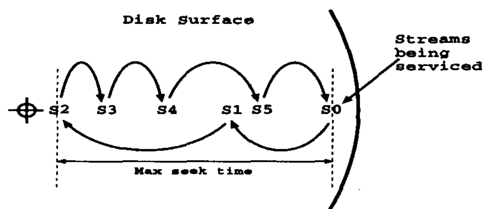

Figure 8. Elevator disk-scheduling algorithm

$$T_{seek_{mean}} = \frac{2 \cdot T_{seek_{max}}}{S} \quad (1)$$

Each word of the media stream contains time-stamped information in order to provide QoS performance data on the server and its components. With possibly hundreds of 5--10 Mbps data streams, this will produce a massive amount of performance data as well as requiring vast amounts of simulation memory. Therefore, in order to reduce this data overload problem, certain streams can be monitored as they are transferred through the server I/O subsystem.

4.4.5 PCI-based ATM Network Interface

Similar in concept to the disk object. The interface consists of a state machine with bus mastering capability, internal registers, internal data queues, PCI master and slave interfaces, and input and output physical interfaces. Its operation is based upon that of the IDT 77201 NICStAR [2] device.

4.4.6 Streaming Memory Object

This object has dual-PCI slave interfaces, so that it can be accessed by master devices on either downstream subordinate bus. The memory object simulates the operation of streaming memory buffer and its controller. To increase bus efficiency, PCI READ and WRITE operations can be simultaneously performed from both subordinate buses. To minimise bottlenecks at the memory interface, it operates at twice the PCI bus frequency.

5 Results of Simulation

The results from the mathematical model [9] describing the proposed architecture gave a streaming performance of eighty two streams. Performance was constrained by the bandwidth of the network subordinate bus. Batch retrieval of ATM cells from memory by the ATM adaptor has increased maximum streaming performance to approximately 100 streams. It remains the limiting factor of the server, but as shown in figure 9, it closely matches the performance of the storage bus.

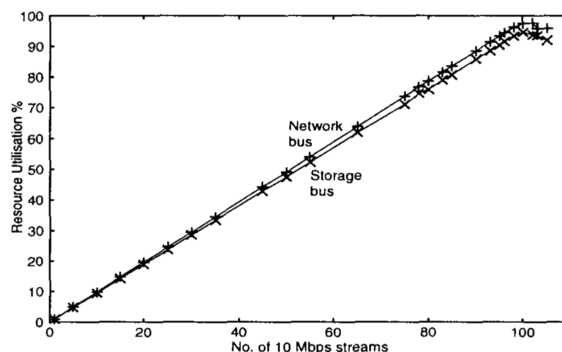

Figure 9. Subordinate PCI bus utilisation

A continuous-media server must also supply a specific QoS to the client. To give an indication of QoS, the delay when issuing network I/O requests has been recorded. This delay is plotted in figure 10 as a percentage of the overall network I/O request period. By assuming that the client's end system has adequate buffering capability, an arbitrary maximum threshold has been set at 1% of the mean network request rate. With this upper limit, maximum streaming performance is quantified at 98 streams. At this performance level, bus utilisation is high at approximately 94% and 96% respectively for the storage and network buses.

6 Conclusions

The simulator has been shown to meet our requirements for a fast and flexible environment. It has provided more realistic performance metrics regarding our proposed system design, in a short period of time. The simulation is currently being used to simulate multiple I/O modules in an independent-proxy media server configuration [3], whilst running a distributed streaming application (figure 11). The

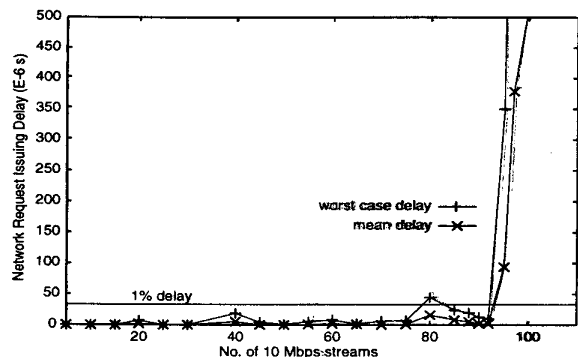

Figure 10. Network I/O request delay as a percentage of the network I/O request rate

simulator infrastructure has also proven flexible enough to be used by the author in separate unrelated projects.

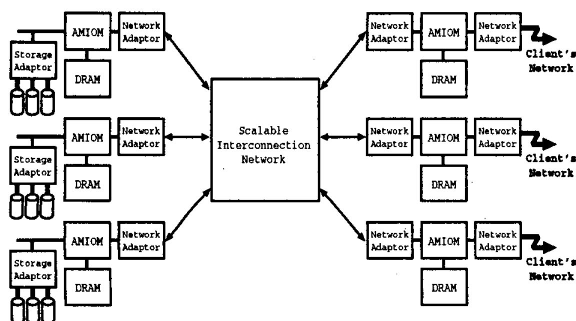

Figure 11. I/O modules in an Independent-Proxy Media Server

The execution time for a two second observation period depends upon stream loading. The execution time for a 100 stream scenario requires approximately 8 hours processing time[3], or 6.4ms of simulated time per stream per second. Since we have no comparison to compare with, and considering the complexity, and large volumes of data processed by the simulation, this performance appears reasonable.

The simulator was coded in order to be architecture and operating system neutral. To date, the simulator compiles and executes successfully, without modification, on systems using the GNU g++ compiler on both Linux (x86), and Solaris (UltraSparc-II) systems. It would be useful to port the simulator to other architectures, such as Windows NT.

Automatic object documentation has proven useful when constructing new simulation scenarios, as has object verbosity. Currently, verbosity levels are set haphazardly between differing entities and their components. A more consistent approach to verbosity levels throughout the simulation will lead to reduced debugging time.

Further modifications are under development to improve the simulator. Currently, the simulator uses compile-time entity declaration. Dynamic run-time entity declaration would remove the need for re-compilation of the scheduler under varying scenarios and allow batching. This would also maximise processing time, and minimise supervision. Configuration data initialised via text-based files has been re-organised. A separate object has been created for containing popular variables. Similarly, a separate object has been created for collating data for reporting purposes. The development of a graphical user interface, possibly using tcl/tk, would be useful for removing configuration errors. Also graphical tools are under consideration for displaying data, such as a waveform viewer to create state level diagrams.

[3]8 hours on a 250 MHz Solaris UltraSparc-II server under light loading

References

[1] K. Annamalai. Multi-ported PCI-to-PCI bridge chip. In *Wescon'97 Conference Proceedings, Santa Clara, CA*, pages 426–433. IEEE, Nov. 1997.

[2] Integrated Device Technology, Inc. *IDT77201 NICStAR User Manual Version 2*. Santa Clara, CA, USA, Nov. 1995.

[3] J. Y. B. Lee. Parallel video servers – a tutorial. *IEEE MultiMedia*, 5(2):20–28, Apr.–June 1998.

[4] M. Little and D. L. McCue. Construction and use of a simulation package in c++. Technical Report 437, Computing Science Technical Report, University of Newcastle upon Tyne, July 1993.

[5] W. H. Press, S. A. Teukolsky, W. T.Vetterling, and B. P.Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, second edition, 1992.

[6] S. Robinson. *Successful Simulation: A Practical Approach to Simulation Projects*. McGraw Hill Publishing, 1994.

[7] E. Ruemmler and J. Wilkes. An introduction to disk drive modelling. *IEEE Computer*, X(X):17–28, Mar. 1994.

[8] Symbios Logic, Inc. *SYM65C875A PCI-SCSI I/O Processor Data Manual Revision 3.0*, Mar. 1996.

[9] M. Weeks, H. Batatia, and M. Maierhofer. Autonomous streaming architectures for continuous media. In *2nd International Conference on Information, Communications and Signal Processing*, Singapore, Dec. 1999.