A Network Co-processor-Based Approach to Scalable Media Streaming in Servers *

Raj Krishnamurthy, Karsten Schwan, Richard West

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{rk, schwan, west}@cc.gatech.edu

Marcel-Cătălin Roşu

IBM T.J. Watson Research Center
mrosu@watson.ibm.com

Abstract

This paper presents the embedded construction and experimental results for a media scheduler on i960 RD equipped I2O Network Interfaces (NI) used for streaming. We utilize the Distributed Virtual Communication Machine (DVCM) infrastructure developed by us which allows run-time extensions to provide scheduling for streams that may require it. The scheduling overhead of such a scheduler is $\approx 65\mu s$ with the ability to stream MPEG video to remote clients at requested rates. Moreover, placement of scheduler action 'close' to the network on the Network Interface (NI) allows tighter coupling of computation and communication, eliminating traffic from the host bus & memory subsystem, allowing increased host CPU utilization for other tasks without being affected by host-CPU loading. Architectures to build scalable media scheduling servers are explored - by distributing media schedulers and media stream producers among NIs within a server and clustering a number of such servers using commodity hardware and software.

1. Introduction

Background. The scalable delivery of media and web services to end users is a well-recognized problem. At the network level, researchers have designed multicast techniques, media caching or proxy servers, reservation-based communication services[9] and media transmission protocols. For server hardware, scalability is sought by using extensible SMP and cluster machines[1, 5]. Scalability for server software is attained by using dynamic load balancing across parallel/distributed server resources, by using admission control and online request scheduling[28] to control resource usage, improve throughput, and guarantee service for resources like CPUs[17, 12], network links [26, 25, 28], and disks[5]. In addition, developers employ application-level or end-to-end solutions[13] that adapt server and/or client behavior in response to user needs and resource availability. For instance, for clients, media caching/buffering and runtime variation of delivered service quality[20] are two of many techniques that attempt to deal with limitations in client resources and with fluctuations in the service offerings experienced by clients.

Scalable Cluster Services. Our group is developing software solutions that aim to improve scalability in media servers, where servers are assumed to be clusters of processor/storage nodes connected via high performance system area networks[24]. A cluster is comprised of a switch and of network interfaces (NIs) connected to the cluster nodes. Each NI has a high performance host CPU-NI interconnect (e.g., a PCI bus), direct connections to the switch, a programmable CoProcessor supporting protocol processing, and local memory with direct connections to disk devices and other peripherals. The NIs used in our research include ATM FORE[19], Myrinet[24], and I2O-compliant network interface boards (Intelligent I/O Industry Consortium)[15, 11].

This paper employs a server configured as 16 quad Pentium Pro nodes connected via I2O-based NIs, each of which has two 100Mbps Ethernet links, a PCI interface to the host CPU, and two SCSI interfaces directly attached to disk devices. Consequently, host-to-host

*This work is supported in part by the Department of Energy under the NGI program and the National Science Foundation under a grant from Divison of Advanced Networking Infrastructure and Research, by hardware donations from Intel Corporation and software donations from WindRiver Systems (VxWorks).

communications are supported by I2O board-resident protocols (like TCP and UDP), and media streams may flow from server disks to clients via host CPUs or directly via the I2O boards (see Figure 1)[15, 11].

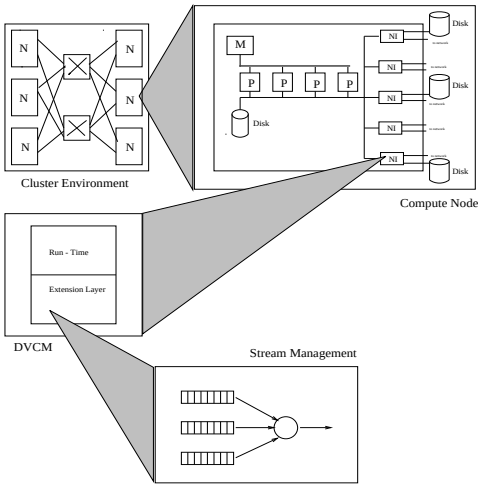

Figure 1. Cluster Hardware and Interconnect.

The approach to server organization we advocate is one that views a server like the one depicted in Figure 1 as an information processing, storage, and delivery engine that is programmed at two levels of abstraction, reflecting the hardware configuration being employed :

1. A cluster-wide, programmable *distributed virtual communication machine* (DVCM) executes 'close' to the network, on the CoProcessors as shown in Figure 1. The cluster-wide services executed by this machine are available to nodes' application programs as *communication instructions*.

2. High-level node-specific services are implemented on host nodes. However, such host-resident application programs may also *extend* the DVCM with additional 'instructions' to support their specific needs. As a result, the services implemented by the DVCM vary over time, in keeping with the needs of current cluster applications.

In summary, host nodes use the DVCM's services to efficiently implement end user information services. The DVCM's functionality resides on the NIs and is accessed from host CPUs via communication instructions. The DVCM is extended and specialized much like extensible OS kernels developed for high performance server systems like SPIN and Exokernel[3, 7]. The architecture of DVCM is described in more detail in Section 2. Next, we present some of the specific DVCM extensions implemented and evaluated in this paper.

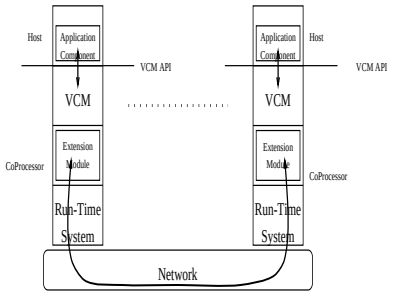

Figure 2. Distributed VCM Architecture.

Contributions. The DVCM idea, its realization for CoProcessor-based NI architectures, and its utility for attaining high performance on cluster machines have been explained and evaluated in previous papers[19]. This paper's novel contributions are the following:

- We demonstrate the feasibility of implementing the DVCM architecture on COTS (Commercial Off-The-Shelf) runtime support resident on NIs, namely, the VxWorks embedded real-time OS from Wind River Systems[27] running on Intel i960RD I2O boards. We show that performance-critical communication extensions can be executed with high performance on such standard CoProcessor platforms.

- We demonstrate the utility of placing certain services onto NIs vs. hosts. Such placements can improve performance in several ways. First, the service's *device-* or *network-near* functions typically run faster on the CoProcessor, because their execution does not involve I/O busses, host memory, and host CPUs. Second, on the host CPU, the time-critical execution of device interactions is easily jeopardized by the CPU's need to also run higher-level application services, and because the context switches typically required by such interactions are expensive due to the CPU's deep cache hierarchy and due to cache pollution. Third, a service running on an NI like the I2O boards not only removes load from the CPU but more importantly, it *eliminates traffic* across both the host's I/O busses and the communication network, thereby freeing up these valuable resources for other server actions. This is discussed in more detail next. Also, services like media schedulers running on host-CPUs are easily affected even by transient loading conditions whereas media sched-

ulers running directly on I2O NIs are immune to host-CPU loading.

The traffic elimination we demonstrate for media applications utilizes window- and time-constrained scheduling techniques[26, 25]. Specifically, when a server streams video to some number of clients, packet scheduling is performed in order to guarantee differential packet rates and deadlines to meet clients' individual QoS needs. By performing packet scheduling on the NI rather than the host, traffic is eliminated for the media streams emanating from the disks attached to it (and to the host), thereby offloading the server's I/O busses, CPU, and memory resources.

Section 2 describes the architecture of the DVCM. Network CoProcessor based media scheduling is described in Section 3 with algorithm, design and construction of the DWCS embedded scheduler. Section 4 experimentally evaluates the embedded scheduler we have built and provides microbenchmarks, demonstration of streaming capabilities, impact of server load, comparison with an equivalent host-based scheduler and discussion of results. Related work is described in Section 5 and Section 6 concludes the paper.

2. DVCM Architecture

The architecture of the DVCM is comprised of three sets of functions (see Figure 2). The first set implements the DVCM's API, which gives each node's application programs efficient access to DVCM instructions. This communication-centric API is supported by extensions of the operating system kernel resident on each node. In the implementation described in this paper, these extensions are implemented as device drivers interacting with the I2O boards via PCI interfaces. In a previous implementation, a SUN Solaris kernel extension was used to implement DVCM for FORE ATM boards linking a distributed SUN Sparc-based cluster machine[19, 21].

The second set of DVCM functions comprises low-level runtime support on the NI, used for implementation of DVCM instructions. The I2O-based implementation of DVCM as described in this paper, utilizes an embedded system configuration of the VxWorks real-time operating system[27] offering support for memory management, task creation, deletion, and scheduling, and device access for the network links and SCSI interfaces resident on the NI. The following functionality has been added to VxWorks in order to exploit this NI's specific hardware and to implement the cluster-wide actions of DVCM: fixed-point library to implement operations cheaply, driver front-ends to initialize controllers/storage, timestamp counter rollover management, circular queues and heaps.

The third set of DVCM functions are the extensions that support specific applications' needs. An example of this would be a scheduler for streaming media provided as an application specific extension by the DVCM and resident on the i960 RD I2O cards. Section 3 describes an application-specific extension namely, a scalable scheduler for media streaming (Figure 2).

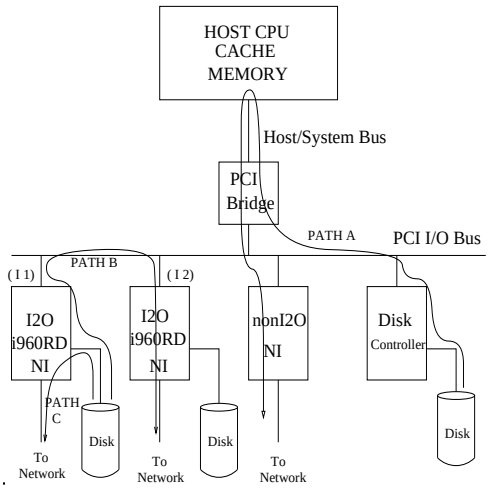

Figure 3. Frame Transfer Paths.

3. Network Co-processor Based Scalable Media Scheduling

The DVCM architecture allows run-time extensions for inclusion of services required for streaming media. This Section describes a service implemented by us for use in scheduling media streams. The algorithm, architecture and implementation issues are described in this Section.

Stream Scheduler Architecture. The media scheduler is based on the DWCS (Dynamic Window-Constrained Scheduling) packet scheduler described in [26, 25], where the scalability of the DWCS scheduler is demonstrated with respect to streaming media frames. One possible scalable target architecture is shown in Figure 3. Multiple NIs (in this case i960 RD I2O cards) are present on the I/O bus. One or more NIs may run the media scheduler and also have disks attached. The Figure 3 shows three paths - A, B and C. Path A represents the transfer of frames from a disk attached to a SCSI controller card to a separate non-I2O NI. An application thread on the

host CPU that has opened the MPEG file for reading must transfer frames from the disk to the filesystem buffer cache (with possibly major portions in host CPU memory). The first part of this transfer involves traversal of memory hierarchies and bus domains (I/O bus to system bus). The second part of path A involves transfers of frames from host CPU memory to the network via the NI with suitable protocol encapsulation. Again, this path involves traversal of memory hierarchies and bus domains. Path B shows the transfer of frames from disks attached to an i960 RD card (I1) to another i960 RD media scheduler card (I2). This path only involves the I/O bus and completely eliminates host CPU or memory. Path C involves transfer of frames from a disk attached to an i960 RD network card to media scheduler input queues resident on the same card. Here, the path eliminates the I/O bus, host CPU and memory. Also, the media scheduler executes on the i960 RD card and does not consume host CPU cycles. Being closer to the network, the scheduler may be reconfigured based on network condition parameters. This again will not require message traversals across the I/O bus.

Embedded Scheduler Construction. The DWCS scheduler code module is embedded in the i960 RD I2O NI with the bootable system image of the VxWorks Operating System, usually resident on flash-ROM on the i960 RD I2O NI card. On system boot, all devices on the PCI bus boot up, which, brings up the VxWorks Operating System on each NI. Initialization code in the kernel is used to spawn the scheduler thread with code image resident on flash-ROM. A detailed description of embedded scheduler design considerations for media scheduling can be found in [18].

The embedded construction of the DWCS scheduler is shown in Figure 4. Frames or packets are stored in circular buffers on a per-stream basis. Head-of-line packets in each stream form loss-tolerance and deadline heaps and encode stream priority values. The scheduler must pick the stream with the lowest priority according to rules described in [26, 25]. The host-based scheduler implementation uses System V shared memory between processes[26, 25] while the embedded NI CoProcessor implementation of the scheduler is lightweight and uses VxWorks tasks (threads) with physically pinned memory for data sharing between tasks. Frame producers on the host CPU or other PCI NIs transfer frames to the NI-based scheduler. The i960 RD I2O NIs are equipped with 4MB of on-board installed memory and may be expanded to 36MB. To conserve memory, we maintain a single copy of frames in NI memory and allow scheduling analysis and dispatch to manipulate addresses of frames. Storing frames directly in NI memory (rather than in host memory) reduces the overall scheduling analysis and dispatch latency of a single frame, the jitter experienced by frames and also may reduce the mean queuing delay of frames in a given stream as the frames are directly available in local NI memory and do not require a 'pull' from host memory.

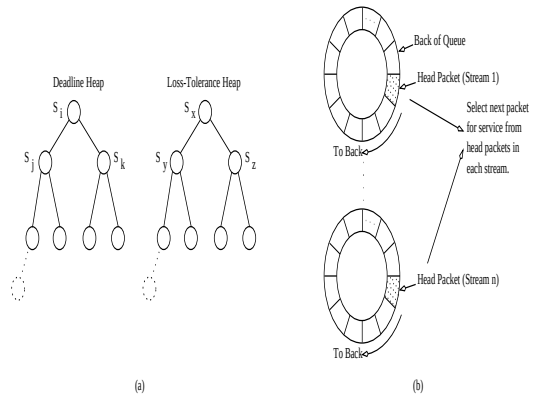

Figure 4. (a) This implementation of DWCS uses two heaps: one for deadlines and another for loss-tolerances.(b) Using a circular queue for each stream eliminates the need for synchronization between the scheduler that selects the next packet for service, and the server that queues packets to be scheduled.

DWCS is designed to maximize network bandwidth usage in the presence of multiple packets each with their own delay constraints and loss-tolerances. The per-packet delay and loss tolerances must be provided as attributes after generating them from higher-level application constraints. The algorithm requires two attributes per packet, as follows:

- *Deadline* – this is the latest time a packet can *commence* service. The deadline is determined from a specification of the maximum allowable time between servicing consecutive packets in the same stream.
- *Loss-tolerance* – this is specified as a value x_i/y_i, where x_i is the number of packets that can be lost or transmitted late for every *window*, y_i, of consecutive packet arrivals in the same stream, i. For every y_i packet arrivals in stream i, a minimum of $y_i - x_i$ packets must be scheduled on time, while at most x_i packets can miss their deadlines and be either dropped or transmitted late, depending on whether or not the attribute-based QoS for the stream allows some packets to be lost.

At any time, all packets in the same stream have the same loss-tolerance, while each successive packet in a stream has a deadline that is offset by a fixed amount from its predecessor. A detailed description of the DWCS scheduler is provided in [26, 25].

4. Experimental Evaluation

This Section presents an experimental evaluation of the system presented in Section 3. We demonstrate that offloading the media scheduler from the host CPU to the NI is feasible and efficient because of fewer overheads and *network-near* operation not involving host/PCI bus domain traversals. We also compare the NI-based scheduler with an equivalent host-based scheduler. The experimental infrastructure will be described first, followed by the experiments. A detailed description of the experimental evaluation can be found in [18].

Experimental Setup. The experimental setup consists of a Quad Pentium Pro server (4 X 200MHz) running Solaris 2.5.1 X86 with 128 MB of memory. Three I2O cards are placed in separate PCI slots on the same bus segment. One I2O card hosts the scheduler code and scheduler data structures. The other two cards serve as stream sources for MPEG traffic. Disks are directly attached to the I2O cards that store MPEG files used to source streams. An MPEG segmentation program developed in [26, 25] is used for segmenting an MPEG encoded file into I, P and B frames and serves as a stream producer. This emulates the MPEG file segmentation process in an MPEG player. MPEG stream producer processes or threads on the host or I2O cards inject frames into the scheduler queues on the scheduler I2O card using PCI bus transfers. The scheduler then picks frames based on scheduling criteria and dispatches frames on the network. Client machines running MPEG players may attach to the scheduler card for MPEG stream delivery over an Ethernet network[26, 25].

Microbenchmarks. The following benchmarks have been constructed to show that off-loading scheduling from the host to the NI is feasible. The experiments described above show that scheduler functionality may be off-loaded to i960 RD NIs with a scheduling overhead of $\approx 65\mu$s. This corresponds to around half an Ethernet frame time ($\approx 120\mu$s) on a 100Mbps link. Also, the output of the scheduler is *network-near* at the exit point to the network and does not require traversals across the I/O bus to the network. Microbenchmarks have been developed that measure system primitive performance. For the purposes of the microbenchmarks, we start the scheduler after all frame descriptors have been written into the circular buffer (we store addresses of frame descriptors in the circular buffer). 'Total Sched Time' records the time to schedule all the frames on the network. 'Avg Frame Sched Time' records the average time to schedule a single frame on the network. Similarly 'Total time w/o Scheduler' measures the cumulative time to transmit all the frames on the network without the scheduler. We simply re-route execution in the code to a point where the address of the frame to be dispatched is readily available and does not need scheduler rules. Similarly, 'Avg Frame Time w/o Sched' records the average time to transmit a single frame without the scheduler.

Microbenchmark	Time (μSecs)
Total Sched time	14295.60
Avg frame Sched time	94.60
Total time w/o Scheduler	4195.68
Avg frame time w/o Scheduler	27.78

Table 1. Scheduler Microbenchmarks (Data Cache Enabled)

The scheduler overhead as reported in [26, 25] for the host-based DWCS scheduler is $\approx 50\mu$s measured on a 300MHz Sparc. Results from Table 1 show a scheduler overhead of $\approx 66.82\mu$s (difference between Average Sched time with scheduler and Average frame time without the scheduler).

Some of the scheduler operations require fractional values to one or two decimal places and we have used a fixed-point implementation of the scheduler for better performance. The VxWorks software floating-point (FP) library adds almost $\approx 20\mu$s for each scheduler computation per frame and we have used our own fixed-point computations instead of the software FP library (the i960 RD I2O card does not have a floating-point unit). Also, we have enabled the data cache (disabled by the VxWorks SCSI driver) as this improves scheduler computation time per frame by $\approx 15\mu$s. Enabling the data cache allows stream priority values and descriptor addresses to be cached and updated every scheduler cycle without additional memory latency. We attempted to improve the scheduling time per frame by placing the frame descriptors in the i960 RD I2O 'Hardware Queues' (1004 32-bit memory-mapped registers) but found that the performance was comparable to the results presented in Table 1[11, 15, 27].

A detailed exploration of implementation options

Expt	Frame Transfer Path (1000 byte frame)	Frame Transfer Time (msec)
I	Path A: Disk-Host CPU-I/O Bus-Network (no load w/ cache)	8
II	Path C: NI Disk-NI CPU-Network (no cache)	5.4
III	Path B: Disk-I/O Bus-NI CPU-Network (no cache)	5.415 (4.2disk+1.2net+0.015pci)

Table 2. Critical Path Benchmarks comparing host(I) and NI(II/III): Same Filesystem and Disk mounted

including fixed- and floating-point implementations of the DWCS scheduler along with benefits of data caching and mapping descriptors to 'Hardware Queues' can be found in [18].

Media streaming from NI attached disks. This Section presents results from critical path benchmarks recorded for three different configurations of frame transfers. All benchmarks measure the latency of a 1000 byte frame transfer from disk to remote client (over an Ethernet network) averaged over 1000 transfers. The measurements in Table 2 record the latency of a single frame transfer. Consider Experiment I in Table 2. This is similar to Path A in Figure 3. An MPEG file on internal system disk attached to a disk controller on the PCI bus is streamed to a remote client. The second experiment (Path C in Figure 3) consists of a single i960 RD NI with a disk attached. Bus activity is reduced to a minimum by disabling other cards on the same bus segment. A thread running on the i960 RD NI reads frames from the locally attached disk and serves frames to a remote client attached to the NI over an Ethernet link. The third experiment III (Path B in Figure 3) consists of two NI cards. This path involves transfer of frames from disk (attached to an NI or separate disk controller) over the PCI I/O bus (using PCI peer-peer DMA transfer) to a separate i960 RD I2O NI which transfers frames to the remote client across the network. This transfer does not involve consumption of host memory, host CPU cycles or host system bus bandwidth.

The latency component common to Experiment I, II and III is the disk access time which is $\approx 4.2ms$ for a single frame (4.2disk in Table 2). Transfer time to the remote client is $\approx 1ms$ for a single frame (1.2net in Table 2, data cache disabled). This represents the latency from end-to-end including traversal of network stacks at either end and wire transmission time. Also, transfer time from I2O NI card to I2O NI card across the PCI bus is $\approx 15\mu$s for a single frame (we used DMA writes from card-to-card to achieve this). The results in Experiment I show a frame latency time of $\approx 8ms$ when the VxWorks filesystem was mounted on Solaris. These were obtained on a Solaris 2.5.1 system using an Intel 82557-based NI . The system disk attached to the system disk controller was used to serve frames using the Intel NI. The VxWorks filesystem is a dos-based filesystem and this was mounted on the Solaris host. To mitigate the effects of filesystem and disk layout performance variation, the same filesystem and disk was used with Experiment I, II and III.

It is interesting to note that when the Solaris UFS filesystem is mounted (for Experiment I) and MPEG files are stored on disk, the frame latency time is only 1 ms. UFS uses a logical block size of 8K, may cache and prefetch blocks for better performance. VxWorks does not support UFS filesystems and we were unable to mount UFS with VxWorks for Experiments II and III[6, 15].

Media stream transfer directly from NI-attached disks to remote clients is advantageous because these transfers may occur with lower overhead than streams being transferred from host system disks under the control of the host CPU and host OS. This as shown in Table 2 can effectively bring down the frame transfer latency time significantly. Inter-card (peer-peer) transfers do not involve the host CPU or memory subsystem and can increase host CPU utilization for other tasks by traffic elimination.

Comparison of Host- and NI-Based Scheduling. A packet scheduler running on the host, may schedule frames on the network along path A in Figure 3. Frame transfer to scheduler input queues, scheduling and dispatch are the three main actions involved in host-based scheduling. These activities involve consumption of host filesystem buffers, system bus bandwidth, I/O bus bandwidth, host memory and kernel/user space buffer usage. Depending on the frame transfer path to scheduler input queues (path B or path C in Figure 3), NI-based schedulers may completely avoid consumption of I/O bus bandwidth (path C in Figure 3), host memory and host bus bandwidth[11, 15, 27] and are unaffected by host

CPU load. A host-based scheduler is easily affected by the host Operating System's need to run higher-level application services (even a minimal installation runs system daemons; database and cluster-connectivity enabled machines run additional services). With a large number of tasks competing for the host CPU, effects of CPU contention on the host-based scheduler may be seen as degradation in scheduling quality (expressed as decrease in output bandwidth available for a stream and increase in frame queuing delay).

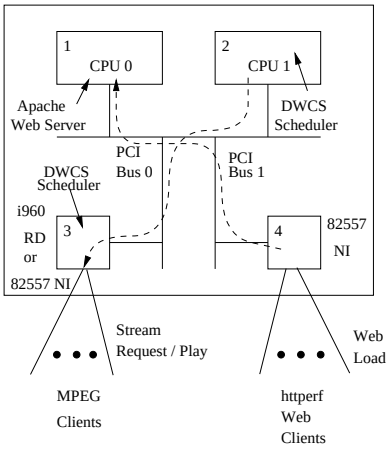

Figure 5. Server Loading Architecture - Web and Media Traffic.

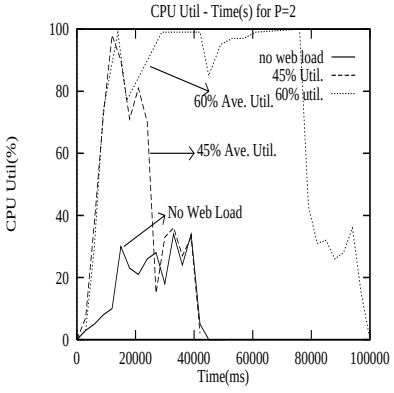

Figure 6. CPU Utilization Variation with Server Load.

Impact of Server Load on Scheduling. The experimental setup consists of a Quad Pentium Pro server (4 X 200Mhz) running Solaris 2.7 X86 with 128 MB of memory. This machine has two separate PCI bus segments and we place NIs on each of the bus PCI segments as shown in Figure 5.

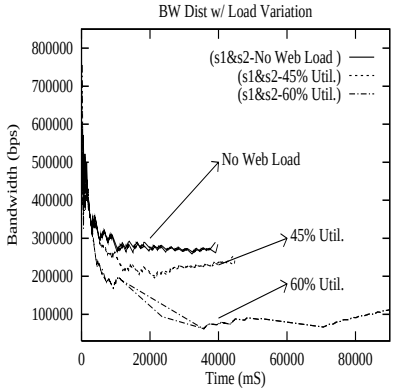

Figure 7. Bandwidth Variation with Load.

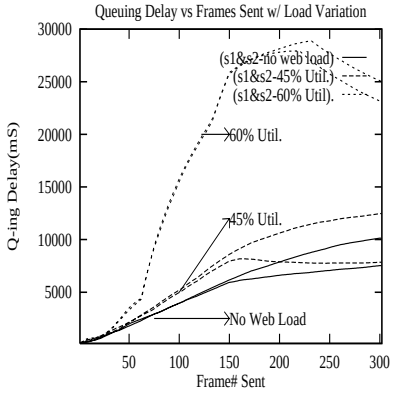

Figure 8. Queuing Delay Variation with Load.

For host-based scheduling load experiments, Intel 82557 100Mbps transceiver based NIs are placed in separate slots on each of the two bus segments (components 3 and 4 in Figure 5). For NI-based scheduling load experiments, one of the Intel 82557 NIs is replaced with a i960 RD I2O NI (component 3 in Figure 5). The machine runs the Apache web server version 1.3.12 (with a maximum of 10 server processes and starting process pool with five server processes)[2]. The web server is loaded using 'httperf' (version 0.6)[14] from remote Linux-based clients. Flexible specification of load from remote clients is allowed by 'httperf' - web pages may be requested at a certain rate by a number of connections with a user-specified ceiling on the total number of calls. The experimental infrastructure is shown in Figure 5. Availability of two separate NIs on separate bus segments allows separation of web load and stream traffic. One of the NIs (with IP address bound to the Intel 82557-based NI) is used to load the

web server using 'httperf' client traffic while, the other NI (with a different IP address bound to the NI, 82557-based or i960 RD I2O NI) is used to request and source stream traffic.

The first set of experiments involves the host-based scheduler version of the DWCS algorithm as described in [26] for a study of system loading effects on scheduler performance. Here, the DWCS scheduler runs completely on the host-CPU and data structures are maintained in host-CPU memory. For these experiments, two of the CPUs are brought off-line for a total of two on-line CPUs. The Apache web server in a configuration described above is brought up and bound to an IP address (of one of the NIs). The DWCS scheduler is initiated and bound to one of the processors (using the 'pbind' Solaris facility)[6] with client requests accepted on a separate IP address bound to a different NI . This allows 'httperf' web clients to connect and load the Apache Web server using a specific IP address (bound to a specific NI). Similarly, MPEG clients may connect to a different IP address (bound to a different NI) for stream delivery. This experiment involves components 1, 2, 3 and 4 as shown in Figure 5 with component 3 as an Intel 82557 NI. Two MPEG clients shown as streams s1 and s2 connect to the system.

Figure 6 shows the total CPU utilization (measured using Solaris Perfmeter)[6] when the host-based scheduler is run without any load imposed by the remote web clients, a peak of around 35% is seen with an average utilization of 15%. Corresponding bandwidth variation and mean queuing delay of frames is shown for two streams s1 and s2 in Figure 7 and 8.

The system is then loaded using the remote web clients as shown in Figure 6 and stream requests are made to the scheduler simultaneously. Figure 7 and Figure 8 represent the corresponding bandwidth and queuing delay variations for the same two streams. The utilization represents the total utilization including CPU utilization because of streaming to clients. Load from web clients is applied at two levels - at the 45% utilization level and the other at the 60% utilization level (in two separate experiment sets as seen in Figure 6). For the 45% average utilization load, a decrease in bandwidth to 200,000 bps is seen at the 15s-20s time mark in the presence of load and the bandwidth can settle to only 230,000 bps (see Figure 7). The queuing delay graph in Figure 8 also shows the effects of loading, frames suffer additional queuing delay of around 2s in the presence of load. For the 60% average utilization case, severe degradation is seen. The bandwidth variation in Figure 7 shows a decrease in bandwidth to around 100,000 bps when the CPU utilization is in the excess of 80% (see Figure 6) during the period from 40s-80s. The bandwidth settles to less than 125,000 bps - half of the bandwidth seen in the absence of web server load (see Figure 7). Frames experience excessive queuing delay in the presence of load (60% average utilization), upto three times (30,000 ms) that seen in the absence of load (10,000 ms).

The next set of experiments involves the NI-based scheduler. For purposes of this experiment, one CPU is brought off-line (for a total of one on-line CPU) with one 82557-based Intel NI used for web server loading and a i960 RD based I2O NI used for MPEG streaming (DWCS runs on the NI) . This experiment involves components 1, 3 and 4 with component 3 as an i960 RD I2O NI which can stream to clients directly. Initially, streams are played to MPEG clients in the absence of any web server load with bandwidth variations and queuing delay measured on the NI-CPU. Next, the system is loaded using the load profile shown in Figure 6 (for 60% average utilization). The i960 RD I2O NI is placed on a separate bus segment and MPEG frames are streamed to clients by the NI-based scheduler. Loading of the web server is done using the other NI placed on a separate bus segment.

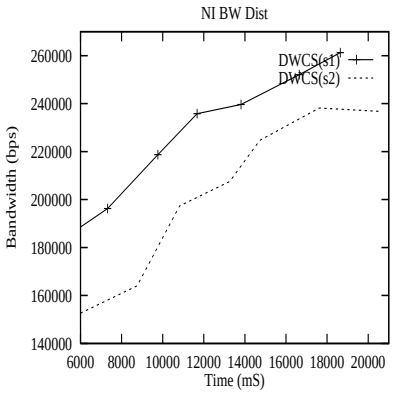

Figure 9. NI Bandwidth Distribution Snapshot: Unaffected by System Load.

The NI based scheduler is completely immune to web server loading and the bandwidth variation and queuing delay experienced by frames is shown in Figure 9 and 10 for both the cases of loaded and unloaded server (for streams s1 and s2). A settling bandwidth of around 260,000 bps is seen (for stream s1) which, is similar to the settling bandwidth achieved by the host-CPU based scheduler in the absence of load (250,000 bps in Figure 7). This represents traffic completely eliminated from the host system bus. A maximum queuing delay of 11,000 ms (stream s1) is seen (cf. 10,000 ms seen in the case of Figure 8).

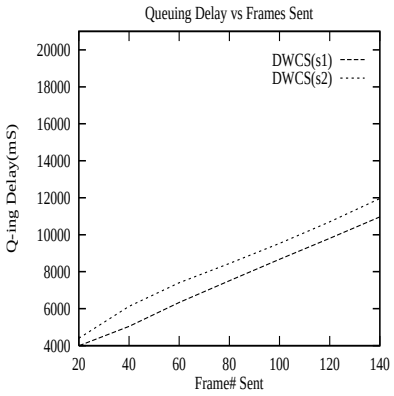

Figure 10. NI Queuing Delay Snapshot: Unaffected by System Load.

Discussion of Results. The scheduling overhead of the host-based DWCS scheduler as reported by us in [26, 25] is of the order of $\approx 50\mu$s. This result was obtained on an Ultra Sparc CPU (300 MHz) with quiescent load. The scheduling overhead of the i960 RD I2O card (66 MHz) based scheduler is around $\approx 65\mu$s. These results are comparable, although the i960 RD is a much slower processor (by a factor of 4). The results from Figure 7 and 9 above show that scheduling performance (in terms of bandwidth and queuing delay) is comparable between host-CPU based schedulers (in the absence of load) and NI-CPU based schedulers. This makes NIs attractive candidates for offloading stream scheduling from host-CPUs. The presence of even transient load adversely affects host-CPU based schedulers with performance degradation seen for CPU utilizations as low as 45% and severe degradation seen for CPU utilizations of 60% and higher. This shows that packet or frame scheduling is an activity that must be offloaded and embedded into the NI for enhanced streaming of multimedia. Inter-card transfers between peers on the same I/O bus segment enable traffic elimination from the host CPU and memory subsystem while allowing frame producers to transfer frames to schedulers.

5. Related Work

A number of NI-based research projects have focused on providing low-latency message passing over cluster interconnects like ATM, Myrinet, FDDI and HIPPI[8, 16, 23, 22] using intelligent NIs equipped with programmable CoProcessors[4, 11, 15, 19]. Our DVCM communication machine implementation on FORE SBA-200 (i960CA) cards allows run-time extension of NI functionality and enables computation directly on the NI[19]. The SPINE project at the University of Washington involves construction of a custom OS for the NI with a focus on safety ([10]) and support for video and protocol processing on the NI.

The I2O industry consortium has defined a specification for development of I/O hardware and software. It allows portable device driver development by defining a message-passing protocol between the host and peer I/O devices[11, 15, 27].

There has also been a significant amount of research on the construction of scalable media servers. If DWCS performed its scheduling actions using a reservation-based CPU scheduler like that described in [12], it would be able to closely couple its CPU-bound packet generation and scheduling actions with the packet transmission actions required for packet streams. Similarly, DWCS could also take advantage of the stripe-based disk and machine scheduling methods advocated by the Tiger video server, by using stripes as coarse-grain 'reservations' for which individual packets are scheduled to stay within the bounds defined by these reservations[5].

6. Conclusions and Future Work

We have built an embedded NI CoProcessor based scheduler with a focus on scalability using commodity hardware and software. The tradeoffs of storing descriptors in appropriate data structures for scalability and embedded construction is considered. Our experiments indicate that the performance is comparable to an equivalent host-based scheduler. Computing packet scheduling decisions on the NI and integrating this tightly with the network is shown to have benefits. These may be realized in terms of increased host-CPU utilization by traffic elimination from the host bus and memory subsystems. Also, packet schedulers running on host-CPUs are easily affected even by transient loading conditions as demonstrated by us, whereas packet schedulers running directly on NIs are immune to host-CPU loading. Offloading frame/packet scheduling on the NI is a viable option.

Future Work. Experimentation is underway for studying bandwidth allocations for a large number of streams streamed by the scheduler. We are looking at ways of improving scheduling decision time using FPGAs(Field Programmable Gate Arrays) to augment CoProcessor functionality[24].

Acknowledgements. We would like to thank the reviewers for their many invaluable suggestions. Our thanks to other contributors to this research, including

Prof. Sudhakar Yalamanchili, Sarat Manni and S. Roy.

References

[1] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW Team. A Case for Networks of Workstations: NOW. *IEEE Micro*, Feb. 1995.

[2] Apache http Server Project. Apache Software Foundation. http://www.apache.org/httpd.html.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, and B. E. Chambers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, December 1995.

[4] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet – A Gigabit-per-Second Local-Area Network. *IEEE MICRO*, Feb. 1995.

[5] W. J. Bolosky, R. P. Fitzgerald, and J. R. Douceur. Distributed schedule management in the tiger video fileserver. In *Sixteenth ACM Symposium on Operating System Principles*, volume 31, pages 212–223. ACM, December 1997.

[6] Solaris Online Documentation. http://www.docs.sun.com.

[7] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th Symposium on Operating System Principles*, December 1995.

[8] E. W. Felten, R. D. Alpert, A. Bilas, M. A. Blumrich, D. W. Clark, S. Damianakis, C. Dubnicki, L. Iftode, and K. Li. Early Experience with Message-Passing on the SHRIMP Multicomputer. *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996.

[9] D. Ferrari, A. Banerjea, and H. Zhang. Network support for multimedia. a discussion of the tenet approach. *TR-92-072*, 1992.

[10] M. E. Fiuczynski, B. N. Bershad, R. Martin, and D. Culler. SPINE - An Operating System for Intelligent Network Adapters. (TR-98-08-01), Aug. 1998.

[11] Intelligent I/O Special Interest Group. www.i2osig.org/architecture/techback98.html.

[12] M. B. Jones, D. Rosu, and M.-C. Rosu. Cpu reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Sixteenth ACM Symposium on Operating System Principles*, volume 31, pages 198–211. ACM, December 1997.

[13] C. Mercer, S. Savage, and H. Tokuda. Processor capacity reservation for multimedia operating systems. In *IEEE International Conference on Multimedia Computing and Systems*. IEEE, May 1994.

[14] D. Mosberger and T. Jin. httperf – a tool for measuring web server performance. In *Proceedings of the 1998 Workshop on Internet Server Performance , held in conjunction with Sigmetrics 1998*, June 1998.

[15] Intel I2O Page. http://www.developer.intel.com/iio.

[16] S. Pakin, M. Laura, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. *Supercomputing*, Dec. 1995.

[17] X. G. Pawan Goyal and H. M. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *2nd Symposium on Operating Systems Design and Implementation*, pages 107–121. USENIX, 1996.

[18] Raj Krishnamurthy, K. Schwan, Richard West and M. Rosu. A network co-processor-based approach to scalable media streaming in servers. Technical Report GIT-CC-00-03, Georgia Institute of Technology, 2000.

[19] M.-C. Rosu, K. Schwan, and R. Fujimoto. Supporting Parallel Applications on Clusters of Workstations: The Intelligent Network Interface Approach. *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing*, Aug. 1997.

[20] D. Rosu, K. Schwan, S. Yalamanchili, and R. Jha. On Adaptive Resource Allocation for Complex Real-Time Applications. *18th IEEE Real-Time Systems Symposium*, Dec., 1997.

[21] M.-C. Rosu and K. Schwan. Support for Recoverable Memory in the Distributed Virtual Communication Machine. In *Proceedings of the International Parallel and Distributed Processing Symposium(IPDPS 2000)*. May, 2000.

[22] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Dec. 1995.

[23] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

[24] R. West, R. Krishnamurthy, W. Norton, K. Schwan, S. Yalamanchili, M. Rosu, and S. Chandra. QUIC: A Quality of Service network interface layer for communication in NOWs. In *Proceedings of the Heterogeneous Computing Workshop, in conjunction with IPPS/SPDP*, San Juan, Puerto Rico, April 1999.

[25] R. West and K. Schwan. Dynamic window-constrained scheduling for multimedia applications. In *6th International Conference on Multimedia Computing and Systems, ICMCS'99*. IEEE, June 1999. Also available as a Technical Report: GIT-CC-98-18, Georgia Institute of Technology.

[26] R. West, K. Schwan, and C. Poellabauer. Scalable scheduling support for loss and delay constrained media streams. Technical Report GIT-CC-98-29, Georgia Institute of Technology, 1998.

[27] WindRiver Systems. *VxWorks Reference Manual*, I edition, February 1997.

[28] H. Zhang and S. Keshav. Comparison of rate-based service disciplines. In *Proceeedings of ACM SIGCOMM*, pages 113–121. ACM, August 1991.