An Approach Towards Enabling Intelligent Networking Services for Distributed Multimedia Applications

Srikanth Sundaragopalan, Ada Gavrilovska, Sanjay Kumar, Karsten Schwan
{srikanth, ada, ksanjay, schwan}@cc.gatech.edu

Center for Experimental Research in Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

Abstract

An increase in network speeds and addition of new services in the Internet has increased the demand for intelligence and flexibility in network systems. This paper explores the extent to which an emergent class of programmable networking devices – network processors, can be used to deliver more efficient, innovative services to multimedia applications. We present and experimentally evaluate an architecture model, which enables the dynamic creation of application-specific services on programmable communication devices, using the Intel IXP2400 network processor and an image manipulation application.

1. Introduction

An increase in speed of networks and addition of new services in the Internet has increased the demand for intelligence and flexibility in network systems. Distributed multiplayer video games based on distributed virtual environments are becoming increasingly popular [12]. These applications involve periodic exchange of messages between the participating machines using some distributed agreement protocol [13]. On the other hand distributed streaming applications involve large volumes of audio/video data to be transmitted across a wide range of clients [8]. Both of these classes of applications need two kinds of strict requirements: (1) bandwidth to send large amount of data and (2) processing power in participating hosts to ensure timely processing and distribution of the content on continuous basis. These applications typically rely on the existence of an underlying communication infrastructure, such as application-level overlays or peer-to-peer networks. Application-specific processing actions are executed at intermediate nodes to implement data distribution functionality and ensure end-user QoS requirements. In addition, intermediate nodes may execute other application components. Therefore, the deployment of new distributed services must have predictable impact of the performance of existing CPU loads at these nodes.

Recent advancement in the network speed (Gigabit networks) has far reduced our concern about the bandwidth needed for such applications. However, the increase in sustainable bandwidth continues to present challenges. First, the growth in the network speed is fast approaching the memory interface speeds of general-purpose processors, making even a single memory reference impact the performance of the network application. Though the network is able to deliver data at high speeds, the overlay host machine becomes a bottleneck in processing and delivering them. Second, the increase in diverse mobile and wireless end-user platforms has created demand for continuous delivery of application services in resource poor environments (e.g., low-bandwidth connectivity, and limited computation power). In order to meet the high-bandwidth, low and predictable latency requirements of multimedia applications, additional processing needs to be applied to the data path in the overlays.

We argue that such processing and distribution of high bandwidth data must be carried out in the *fastpath* to get the maximum performance. The fastpath processing could be entirely supported by dedicated hardware, if only for their cost and inflexibility. With emergence of high-speed programmable network processors (NPs), software based solutions are considered feasible compared to their costly hardware based counterparts. Performance of previous implementations of software based router and QoS has strengthened the cause of exploring various services that can be added at network processor level [1,5,6,7,11]. Network processors

such as IXP 2xxx series are one such high-speed programmable processor, which meets these requirements with less cost and greater flexibility through their programmable nature.

This paper presents and evaluates a flexible architecture built for platforms with programmable network interfaces, which supports efficient execution of variety of services required in distributed media applications. Experimental results in Section 5 demonstrate the feasibility of implementing media services on communication cores, and the performance improvements, which this approach can deliver.

The remainder of the paper is organized as follows. Section 2 discusses the different classes of services needed in distributed media applications targeted by our work. It also briefly describes the Intel IXP NPs, used as an implementation platform in our work. Next, in Section 3 we describe the proposed architecture. Details regarding the IXP2400-based implementation are presented in Section 4. Experimental results appear in Section 5. Concluding remarks and outline of future research directions are given at the end.

2. Distributed Media Services

A wide range of multimedia services could be supported at the network level with the use of high-speed network processors. We can broadly classify these services as:

- *Path transforming services (PathX)* -- manipulation on the media stream, and
- *Data transforming services (DataX)* -- manipulation of the data content.

The first set includes services such as routing, multicasting, broadcasting, prioritizing, filtering, etc. For example, removing certain frames from an MPEG stream to produce an MPEG video with reduced, but still satisfactory quality can be easily supported on these platforms. All of these services would require access and manipulation of the protocol and application level headers, but still leave the data content unmodified.

The second category includes services such as transforming/transcoding the data in different format representation, down sampling image quality by modifying the image, cropping the image to match to client view, compression algorithms, encryption, etc. Applications which require such functionality at the network level, include visualization of remote experiments, or camera-captured data for a pool of clients with wide range of interests in the ongoing visualization, and/or which use devices with varying computational or networking capabilities [9]. These applications need services which (1) render appropriate view of imaging data on behalf of the clients, and (2) match the quality of the image stream to the client's interests, or device and connection capabilities.

Finally, consider services like application-level IP multicasting, needed in variety of distributed media applications [13]. In these services, once the data is received, it needs to be duplicated to 'n' other members, as indicated in the routing tables. Such duplication involves heavy copying, and imposes additional loads on the host's memory and I/O infrastructure. Moreover these packets need to be received by the network adapter, processed through the IP and UDP[1] stacks and then delivered to the application, which manages multicast routing functionality. In addition, for large application level data, we need to consider the delays due to fragmentation and reassembly. One could remove this multi layer approach by handling such duplication at the network level itself. One approach could be to program such multicasting logic into programmable networking devices such as the IXP NPs. Such programmable processors provide sufficient headroom for carrying out application level packet manipulation and still be in fast path [4,2].

2.1. IXP 2xxx Architecture

The proposed architecture assumes the availability of programmable communication cores or network cards at (at least some) nodes participating in the distributed infrastructure. Our implementation uses network processors from the Intel IXP family, specifically the IXP2400, attached to standard Linux-based hosts, to represent these future platforms. Some of the design here assumes that any network processor will have similar architecture and functionality of Intel IXP 2400. The feature that is important from the perspective of our design is the presence of multiple processing engines (such as the microengines of IXP2400), which can be pipelined in arbitrary manner to perform certain tasks.We briefly outline the key architectural features of the IXP2400 architecture. For more detailed information on the IXP2400 refer to [3].

The Intel IXP2xxx Network Processors have specialized hardware to support network

[1] Multicast is supported with UDP only

operations, which gives the ability to implement network services with high packet throughput and low latency [3]. In addition to the network centric operations, it also provides support for data processing in the form of CRC checksum calculations, integer arithmetic etc. Figure 1 shows the functional units of IXP2400 network processor.

The IXP2400 network processor offers an increased number of microengines at higher speeds, additional memory, a faster host-ANP PCI-based interconnect, and other technology improvements. The IXP2400 chip includes eight 8-way multithreaded microengines for data movement and processing, an Xscale core for management and other functions, local SRAM and DRAM controllers and an integral PCI interface with three DMA channels. The Radisys ENP2611 board [9] on which the IXP2400 resides includes a 600MHz IXP2400, 256MB DRAM, 8MB SRAM, a POS-PHY Level 3 FPGA which connects to 3 Gigabit interfaces and a PCI interface. An Xscale core, running Linux, is primarily used for initialization, management and debugging. The IXP2400 is attached to hosts running standard Linux kernels over a PCI interface. Data is delivered to and from the host-resident application components through the IXP's network interfaces.

There is wide range of development tools available which aid the application development for IXP Programmers. These include workbench simulators, SDK's, debugging utilities (data watch, memory watch etc), performance-gathering utilities, etc.

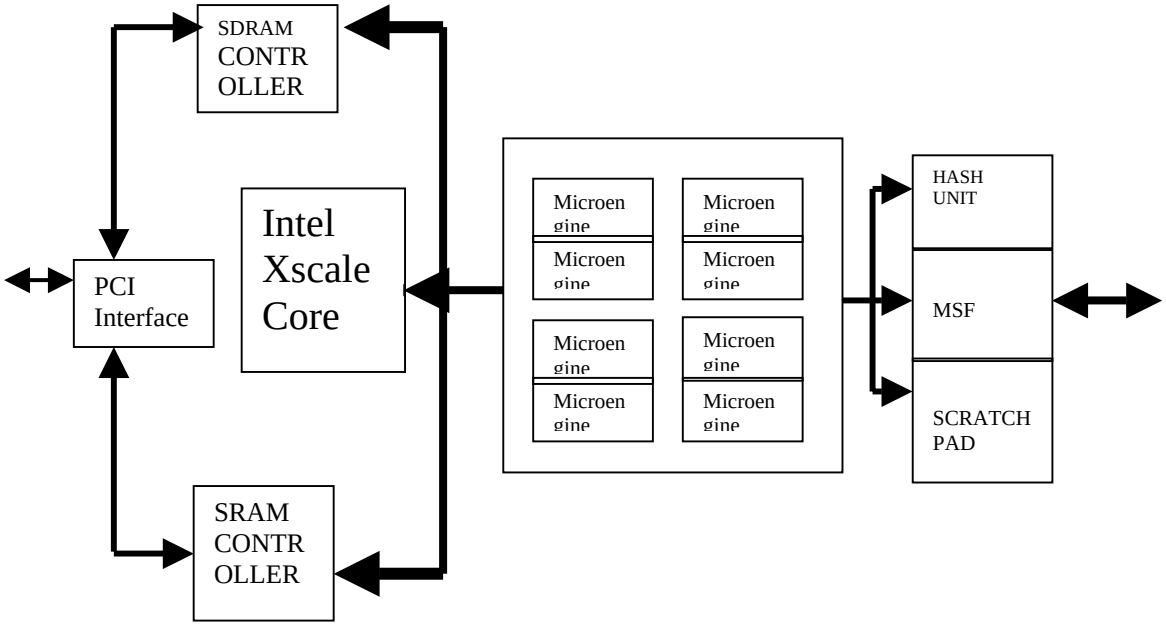

Figure 1. IXP2400 Block Architecture (source, Intel IXA)

3. System Architecture

Figure 2 presents the high-level view of the distributed infrastructure targeted by our work, and can be described as follows. Media data is exchanged on continuous basis on top of an application-level overlay. The data may traverse intermediate nodes on the path from sources to destinations. The processing at these intermediate nodes may involve solely overlay routing actions, or may require additional application-specific data content and path manipulations. Based on the processing actions required, or the existing loads at intermediate nodes, the data path may be mapped to their dedicated communication cores, at the network interface level. The mapping of processing actions to individual nodes in the overlay is responsibility of the middleware layer utilized by the distributed applications, and is not the focus

of this paper.

Individual nodes enhanced with programmable communication interfaces, similar to those discussed in [2] are depicted in Figure 3. In addition to data distribution functionality, the general purpose host CPU may execute other application components. The distribution of data in the overlay, and deployment of processing on behalf of overlay clients is managed by a Distribution Manager. This layer drives any reconfiguration actions through which the data path through this node can be offloaded onto the communication interface (e.g., programmable NIC, attached NP, dedicated communication core, etc.).

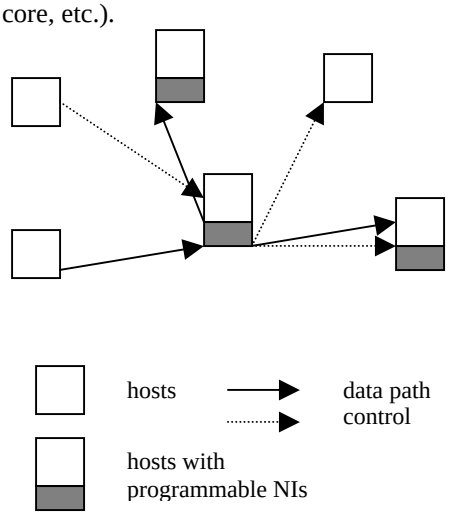

Figure 2. Distributed Application Overlays

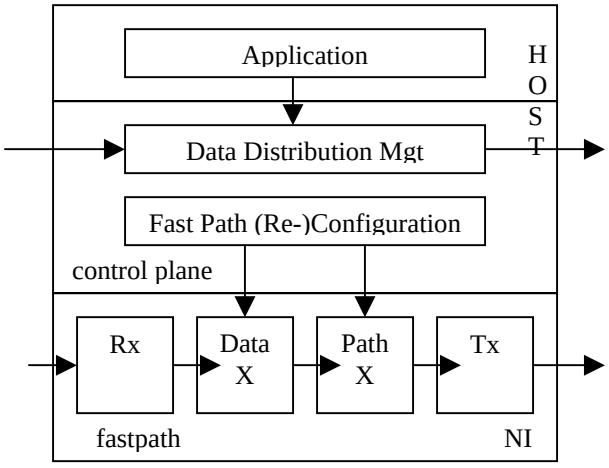

Figure 3. System Architecture

The core functionality implemented on the fast path is encapsulated by the Rx and Tx tasks, which implement receive- and transmit-side protocol processing. Additional DataX and PathX tasks may be deployed on the fast path to implemented new services, thereby forming a processing pipeline. Depending on the underlying platform architecture and resource availability, DataX and PathX tasks are mapped to separate execution contexts (e.g., threads, microengines, processing elements). This may be particularly needed for DataX tasks, which have greater computational requirements and include repeated access to data content stored in chip memory. PathX tasks may be combined and executed jointly with the Tx processing. Compositions of DataX and PathX tasks can be formed to represent variety of services, ranging from data filter, transcoding, or multicast [2,7].

The processing executed as part of the DataX and PathX blocks can be dynamically configured through the host-resident FastPath Configuration interface. Fast path state and resource utilization is available at the host, so that host-resident control components can make estimates regarding service levels sustainable with the current fast path configuration. Currently, we assume that all QoS-related data path reconfigurations are triggered by the middleware. Our future work will consider augmenting the architecture with functionality to detect QoS-violating conditions, and trigger appropriate actions.

4. Implementation Detail

4.1. IXP2400-based Architecture

Our current implementation assigns individual blocks from the fastpath pipeline to separate microengines on the IXP2400 NPs. Each task may be executed by one or more hardware supported context, i.e. microengine threads. Communication between pipeline stages is implemented through controlled access to shared ring buffers. SRAM based data descriptors describe the application level data, stored across multiple DRAM resident buffers. Communication with the host-resident control processes occurs via using shared mailboxes implemented on top of a PCI-based interface, similar to [10]. Fastpath reconfiguration requires involvement of the Xscale core, and has already been implemented for the previous generation IXP NPs with practically negligible service interruption (28-30 microseconds).

4.2. Image Processing Application

As an instance of the pipelined architecture, we have implemented a Packet Forwarding Application and an Image Filtering Application to benchmark application level data manipulation

in fast path. Figure 4 shows the implementation details of the Image Filtering Application.

As shown in the figure we have implemented two flavors of the filtering algorithm to compare memory performance. In the first algorithm we assemble the entire Image on to SRAM and then do the gray filtering. We do this instead of directly manipulating the buffers on the DRAM for two reasons. The first being that, there are certain application level processing which needs the complete data to act upon. This makes the data manipulation algorithm simple. Another reason is that, repeated access to the application data would amortize the cost for copying it on to the SRAM as there is a significance cycle count difference between a DRAM read and a SRAM read operation. Also, reading the fragmented data on DRAM may be costly if there is huge number of such data reads.

The second algorithm is complex in the sense; it needs to manipulate fragmented data in DRAM. But, avoiding the memory copy from DRAM to SRAM could trade off such a complexity. As said above, this kind of algorithm would make sense where there is a limited amount of application level data manipulation.

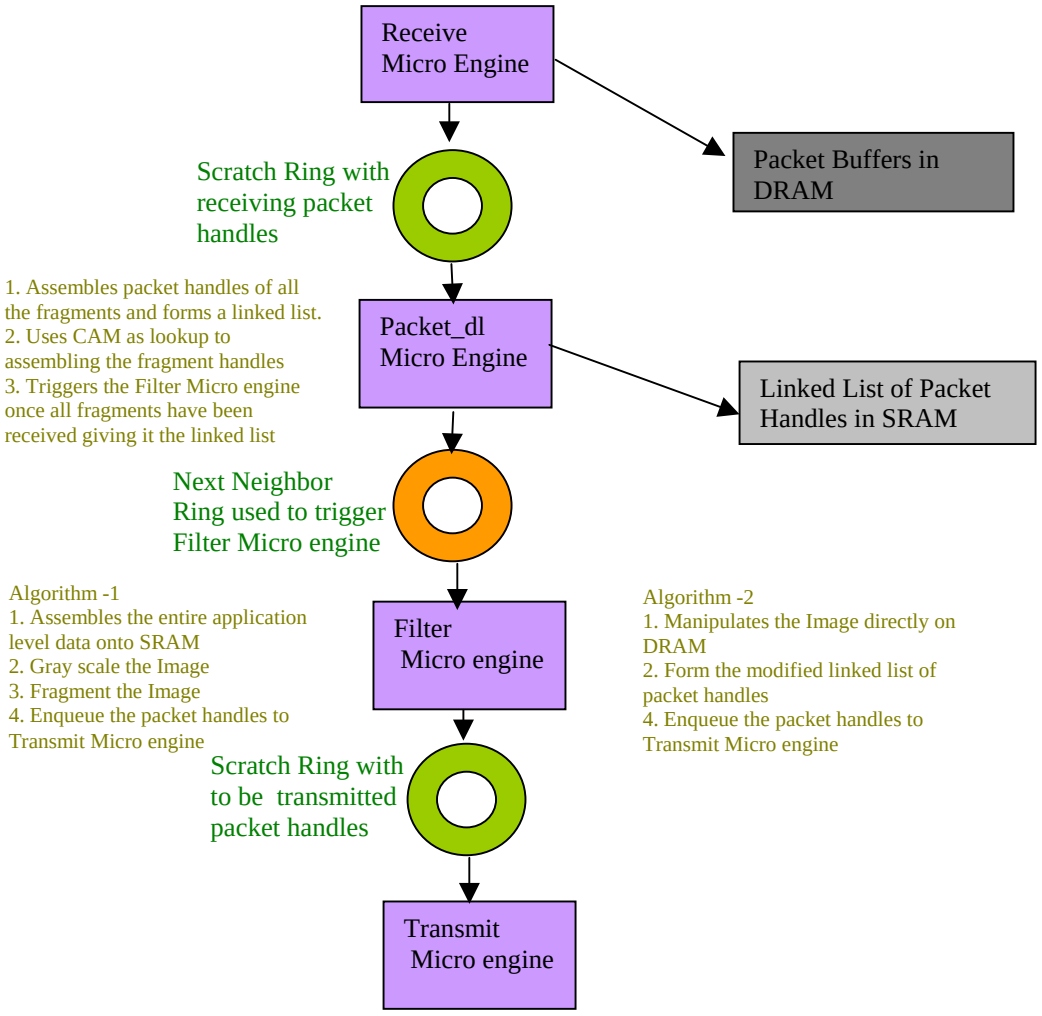

Figure 4. Image Processing Application

4.3. Efficient Multicast Implementation

The pipeline architecture assumes that each microengine is dedicated to a task. The data stream will pass through this pipeline, when it is in the fast path. The ability to assign the same task to multiple processing contexts permits us to exploit the IXP's hardware supported parallelism and create efficient implementations for variety of services.

Consider the support for application level multicasting on attached network processors. We could build the routing logic to be executed at the core and do the duplication at the micro engines. This means that the core will get involved in multicast routing algorithms to build the routing tables and store it in shared space which the micro engines can access. When packets arrive from the network, the micro engines can use the routing tables to route the packets to the destinations. A straightforward implementation would be to duplicate these packets n times and send it. But, such an operation would involve a huge number of memory copying. As an alternate approach, one could exploit the parallelism provided by these micro engines and do intelligent duplication.

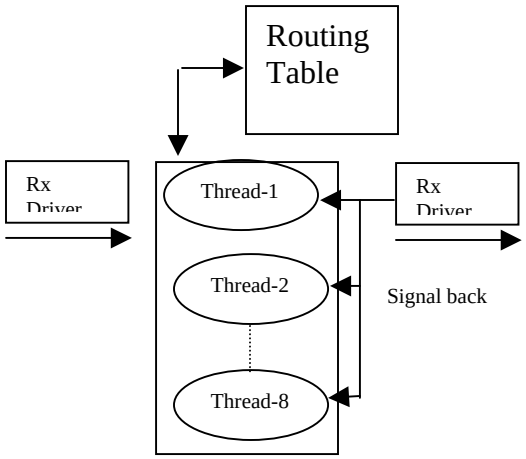

Figure 5. Parallel Packet Processing

Figure 5 shows such logic. As and when packets arrive, they are assembled and are ready to be transmitted to their respective destinations. Unlike the traditional way of duplicating the packets n times, we will work on individual packets exclusively and modify their headers to reflect the destinations. In our current implementation, each of the eight hardware assisted thread of the micro engine, works on individual packets. Each thread picks up a fully assembled packet and takes the first destination in the routing table to write into the IP header of the packet. It then places a transmit request for that packet to the Transmit driver. The transmit driver transmits the first 64 bytes of the packet and signals back the thread, which placed the request without freeing the packet buffer. Once the signal is received the thread goes ahead and picks up the next destination from the routing table and overwrites the IP header to the new value. This is absolutely acceptable, as the IP header has been transmitted in the first 64 bytes and thus achieving a near zero duplication of packets. Once the thread is done with all destinations, the packet buffer is freed. The next packet from the queue is picked up and the procedure is started all over again.

5. Performance Analysis

We have carried out different sets of experiments to analyze the performance of application level data manipulation.

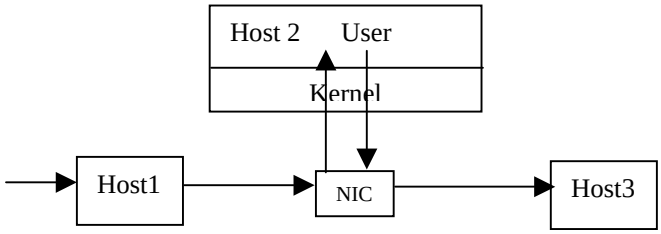

Figure 6. Host-based Test-bed

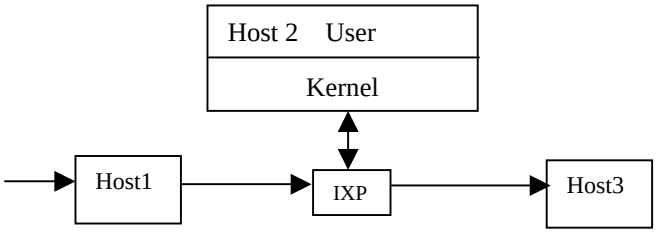

Figure 7. IXP-based Test-bed

5.1. General setup

We run the set of experiments in different setups. The first configuration, presented in Figure 6, is a purely host-based, in which the data path from source to destination traverses an intermediate general purpose host node. In the second setup, the data path traverses an IXP NP on its route to destination (see Figure 7). In each case, additional processing is applied to the data at the

intermediate node. Our host machines are 4 CPU with Intel Xeon processors, each with 2.4 GHz clock speed. The IXP test-bed uses the aforementioned Radisys ENP2611 boards, with eight microengines running at 600 MHz each. In order to simulate additional processing on the intermediate nodes in a distributed infrastructure, we run a CPU intensive process on the host machines, the `applu` application from the CPU2000 benchmark suite. The experiments use data streams generated from a flow of PPM images of varying sizes (which span across multiple Ethernet frames), and transmit these at maximum rates.

For the host based experiments, images are sent from first host to second host via UDP. Here, the user level application waits for the entire data to be arrived and then either forwards or grey scales the image and sends it to the third host. In the third host, we either use raw sockets to capture Ethernet frames or receive the data using UDP depending on the performance measurement needed. For latency measurements we use UDP itself in host-3, but for measuring the throughput we need to capture Ethernet frames of these UDP packets and so we use raw sockets in host-3.

For the IXP based test bed, the image is packetized into Ethernet frames and is pumped into the LAN through raw sockets. The IXP network processor receives them and either forwards or grey filters the Image before dispatching it to the third host. In host-3, we again use raw socket application to capture the Ethernet frames.

5.2. End-to-End latency

PPM Images are fragmented into 1500 bytes Ethernet frames and are put into the LAN using raw sockets API. The first timestamp is recorded when all the Ethernet frames are sent into the LAN from host-1 and the second one when we receive all the frames back on host-3. The difference between them is noted. Again, we repeat the same experiment with the host-1 and host-3 interchanged. This is done in order to remove the discrepancies in the two host machines clocks. We add the difference obtained in both the runs and divide it by 2 to get the actual end-to-end latency between these hosts. The experiment is carried out both on the host based test bed and the IXP based test bed.

Tables 1 and 2 show the end-to-end latency for image of different sizes for the host-based and the IXP-based configuration, respectively. The results show that, there is a substantial latency difference between host-based test bed and the IXP-based test bed for images of lesser size and significantly less difference when the image size is big. This is because the kernel/user space overhead and the stack-processing (UDP/IP) overhead are quite significant compared to the processing carried out on the data content itself.

Note that these latency measurements are for the image processing application, which requires computationally and data intensive processing at the intermediate node. For tasks with lower resource requirements, such as those implementing PathX services, the latency decreases are more significant for larger data sizes.

Image Size	**Number of Ethernet Frame (1490 bytes each)**	**End-to-end Latency (micro secs)**
57616 bytes	41 frames	8290
14436 bytes	11 frames	2956
3636 bytes	3 frames	1499
936 bytes	1 frame	1256

Table 1. End-to-end Latency for Host based Test bed

Image Size	**Number of Ethernet Frame (1490 bytes each)**	**End-to-end Latency (micro secs)**	**Percentage decrease in Latency**
57616 bytes	41 frames	7150	16 %
14436 bytes	11 frames	1935.5	52 %
3636 bytes	3 frames	621	141 %
936 bytes	1 frame	340	269 %

Table 2. End-to-end Latency for IXP based Test-bed

5.3. Throughput

The throughput measurements are carried out in with the same setup as above. In the host based approach, Image data is sent over UDP from host-1. host-2 receives the entire image and does grey filtering on it and dispatches the filtered image to host-3. In host-3 we run a raw socket application to capture the Ethernet frames of the

UDP packets. The timestamp when the first Ethernet frame is received is recorded and then again the timestamp when the last one is received is recorded. Again, we evaluate a PathX forwarding task, and a DataX grey scaling task.

The results are presented in Table 3. In both case the IXP-based implementation of the service outperforms the host-one. Simulation measurements indicate that the IXP-based implementation will significantly outperform the host-based implementation under increased network loads, in spite of the disparity of the computational capacity of the host CPUs vs. the IXP microengines.

	IXP-based (Mbps)	**Host-based (Mbps)**	**Gains (%)**
DataX	94.7	93.9	1
PathX	94.5	91.8	3

Table 3. Through measurements for the host- vs. IXP-based test-bed.

Other services evaluated in our work on this and previous IXP-based platforms include (1) filtering of entire application-level messages (e.g., image frames), to meet end-user quality requirements, (2) multicast, (3) image cropping to match the data delivered to the client's current interests, (4) format translation, and others. The results of these experiments indicate that programmable networking devices can provide support for performance improvements in distributed applications with strict bandwidth or latency/jitter requirements.

6. Conclusions

Technology advances have created the ability to bring intelligence into the networking infrastructure. This work explores one architecture, which using off-the-shelf programmable communication devices, creates enhanced platforms for flexible support for customized delivery of media streams. We demonstrate experimentally performance improvements attained on these platforms, for a range of services needed in distributed multimedia applications.

Our future work will continue to investigate mechanisms for exploiting the programmability of the networking infrastructure and delivering scalable and predictable services to different classes of applications. We are currently considering the deployment of middleware-level functionality onto programmable NPs, generalization of the host-NP control mechanisms, and techniques for efficient representation and caching of application-level meta-data regarding data formats and processing actions.

Acknowledgment: Ramkumar Gandhapuneni and Prashant Thakare worked on the original implementation of the imaging application.

References:

1. A. Gavrilovska, S. Kumar, K. Schwan, The Execution of Event-Action Rules on Programmable Network Processors, *Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure (OASIS 2004), held with ASPLOS-XI*, Boston, MA, Oct. 2004.
2. A. Gavrilovska, K. Schwan, O. Nordstrom, H. Seifu, Network Processors as Building Blocks in Overlay Networks, *Hot Interconnects 11*, Stanford, CA, Aug. 2003.
3. IXP Intel Network Processor Family, http://developer.intel.com/design/npfamily
4. T. Spalink, S. Karlin, L. Peterson, Y. Gottlieb, Building a Robust Software-Based Router Using Network Processors, *Proceedings of 18th SOSP 2001*, Banff, Canada, 2001.
5. Y-D. Lin, Y-N. Lin, S-C. Yang, Y-S. Lin, DiffServ over Network Processors: Implementation and Evaluation, *Proc. of Hot Interconnects 10*, Aug. 2002.
6. X. Zhuang, W. Shi, I. Paul, K. Schwan, Efficient Implementation of the DWCS Algorithm on High-Speed Programmable Network Processors, *Proc. of Multimedia Networks and Systems (MMNS)*, Oct. 2002.
7. S. Roy, J. Ankcorn, S. Wee, An Architecture for Componentized, Network-Based Media Services, *Proc. of IEEE International Conference on Multimedia and Expo*, Jul, 2003.
8. M. Wolf, Z. Cai, W. Huang, K. Schwan, SmartPointer: Personalized Scientific Data Portals in Your Hand, Supercomputing 2002, Nov. 2002.
9. Radisys Corporation. ENP-2611 Data Sheet. http://www.radisys.com
10. K. Mackenzie, W. Shi, A. McDonald, I. Ganev, An Intel IXP1200-based Network Interface, *Proceedings of the Workshop on Novel Uses of System Area Networks at HPCA (SAN-2 2003)*, Anaheim, CA, 2003

11. Path 1 Network Technologies, Professional Digital Video Gateways for the Broadcaster and Multi-Service Operator, *Delivered by Path 1 Network Technologies* and Intel® Network Processors. White Paper*, 2002.
12. M.R.Stytz, *Distributed virtual environments*, Computer Graphics and Applications, IEEE, Volume: 16, Issue: 3, May 1996
13. C. Diot, L. Gautier, *A distributed architecture for multiplayer interactive applications on the Internet*, Network, IEEE, Volume: 13, Issue: 4, July – August 1999.