Platform Overlays: Enabling In-Network Stream Processing in Large-scale Distributed Applications

Ada Gavrilovska
ada@cc.gatech.edu

Sanjay Kumar
ksanjay@cc.gatech.edu

Srikanth Sundaragopalan
srikanth@cc.gatech.edu

Karsten Schwan
schwan@cc.gatech.edu

Center for Experimental Research in Computer Systems (CERCS)
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, 30332

ABSTRACT

The purpose of this research is to explore the capabilities of future, multi-core heterogeneous systems, with specialized communication support, to be used as efficient and flexible execution platforms in distributed streaming applications. On such platforms, we create overlays of hardware- and software-supported execution contexts – *platform overlays*. Stream manipulations, represented via stream handlers, are deployed on top of such overlays, based on the ability of individual contexts to perform handler operations. As a result, stream processing is dynamically mapped to those platform resources best suited for it, and it can even be fully contained to the networking subsystems, thereby enabling in-network stream processing. Experimental results demonstrate the benefits of our approach towards meeting application-specific quality requirements.

Categories and Subject Descriptors

D.4.1 [**Input/Output and Data Communications**]: Data Communication Devices—*Processors*; C.2 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Design, Performance

Keywords

Network processors, Streaming applications

1. INTRODUCTION

Data-intensive streaming applications challenge distributed execution platforms with their large data volumes, dynamic quality of service constraints, and/or highly variable runtime behaviors. Typically deployed on top of application-level overlays, application components executing in these distributed environments depend upon underlying communication services for meeting end user performance and quality needs. QoS requirements range from simple rate or timeliness guarantees to those that dynamically link application-specific notions of quality like 'data resolution' to lower-level metrics like delay and throughput.

In order to address dynamics in application requirements and policies, in inputs and outputs, and in platform resources, a common approach is to execute application-specific computations 'along' the data paths between distributed sources and sinks [15, 16, 10]. Nodes on the stream data path perform application-specific stream customizations and/or route data through the overlay. Since such overlay-centric actions compete with other computational or communication tasks executed on overlay nodes, it is important (1) to isolate the computations needed for data movement in the overlay from those of application components that execute 'core' functionality, and (2) to increase the overlap of computation and communication for data received/transmitted from these application components.

Emerging networking technologies have significantly increased the bandwidths available in distributed infrastructures. In addition, techniques like OS-bypass or RDMA-based approaches have provided improved computation/communication overlap [8, 13]. However, the ability to directly place data where it is needed by the application remains subject to hardware restrictions like I/O bus address range. Furthermore, data layout, content, or format may not meet application's expectations, thereby not eliminating the need to perform additional data copying or receive-side processing.

Recent developments in programmable networking devices, network interfaces and dedicated communications cores in future heterogeneous multi-core platforms, create possibilities to concurrently execute communications and computation on separate hardware supported contexts on these devices/systems. Distributed streaming applications can benefit from these platforms by (1) deploying all overlay related functionality (e.g., application-specific multicast) onto the communications subsystem, and thereby not perturbing the ongoing computation, (2) dynamically tuning the processing actions executed on the overlay path through the communication core to meet application interests and platform resources, and (3) dynamically specifying the layout, content, and subset of data which is to be delivered to application components executing on the computational subsystem.

This paper explores the capabilities of heterogeneous multi-core systems, with specialized communication cores to be used as execution platforms for distributed data streaming applications. To emulate such systems we use hosts with attached network processors (NPs), where the host represents the general purpose processing core, and the NP the communication core. We model such heterogeneous systems as overlays of processing contexts, interconnected via system-level interconnects and shared memory communication channels, termed *platform overlays*. A context in such a platform overlay corresponds to some hardware- or software-supported processing element available on the modeled platform. The platform overlay model permits multiple data paths to be deployed simultaneously, delivering data either to/from application components running on general purpose computational nodes on the same platform, or to remote end-points in the distributed application overlay. Application-specific computations, represented via stream handlers [5], are dynamically deployed onto contexts in the platform overlay, thereby increasing the computation/communication overlap in the system, reducing processing loads to the general computational resources, and delivering improvements through utilization of specialized networking hardware.

The intent of platform overlays is to permit applications, via middleware, to dynamically map different stream processing actions to those platform resources best suited for them, and even fully contain such actions to the networking subsystem. The outcome is in-network stream processing along the entire stream path from source to destination. Advantages derived from this approach include the following. First, at communication end-points, processing actions executed jointly with data receipt/transmission can transform the stream and its data items, placing these items into memory in exactly the forms and layouts required by the application. Second, at both end-points and intermediate nodes in the application overlay, the ability to offload processing from the general computation nodes can reduce perturbations to application computations and improve overall system and overlay performance. Third, communication cores have optimized hardware enabling more efficient implementations of certain types of processing actions, such as data forwarding and multicast, byte swapping, and certain low-level application-specific operations on stream data.

The remainder of this paper is organized as follows. Section 2 discusses the target application domain and the services suitable for in-network execution. We describe our model for representing application-level services in Section 3 and the platform overlays execution model in Section 4. Results from experimental analysis appear in Section 5. This is followed by a brief survey of related work and concluding remarks.

2. DISTRIBUTED STREAMING APPLICATIONS

Distributed streaming applications depend upon the ability of the underlying infrastructure to provide services for online data analysis, for pre-processing and/or customization of a stream before it reaches its destination, or for stream manipulations necessary for the implementation of different quality or fault-tolerance properties. Examples of such applications abound, ranging from scientific collaborations, multimedia and visualization applications, operational information and event notification systems, etc.

The application from which we draw illustrative examples is a distributed scientific collaboration, termed SmartPointer (see Figure 1). Here, data streams generated by a data-intensive molecular dynamics (MD) simulation are delivered to an imaging server. Based on the stages of the experiment and the scientists' interests in different types of molecules, the imaging server generates renderable images that represent different views of the ongoing simulation's output. The generation of these images is itself a computationally intensive task, involving floating point and matrix arithmetic, and imposes substantial loads on the server CPU and on its I/O and memory infrastructure. Once computed, images are forwarded to multiple clients or groups of clients. Depending on the clients' networking and platform resources, or interests, the imaging server needs to further manipulate the image representation, such as downsampling the color encoding, performing cropping operations to match the client's view point, etc. Another class of streaming applications used extensively in our work which benefit from the movement of stream manipulations onto dedicated communications hardware are the Operational Information Systems used by companies like Delta Airlines OIS [9], where data events regarding passengers, crew, flights, luggage, etc. are collected and processed on continuous basis, 24/7, to support company's daily operations.

The stream processing actions involved in these applications are executed at the stream end points, sources and destinations, as well as at intermediate nodes in the application-level overlays. Such actions are specific to each application, and typically, they require access to, interpretation, and manipulation of the application-level contents of the messages traversing the overlay. The wide range of stream manipulations shown to be well-suited for in-network execution can be classified as follows:

- *content-based routing* – necessary for content-based load balancing [1], or for wide area event systems, as in IBM's Gryphon system [19], where boolean or simple query-based selection operations are performed on events, to route events or to deliver suitable event subsets to groups of end users;
- *data sharing* – required for data mirroring and replication services, used in publish/subscribe infrastructures and transactional services such as those executed in OISs;
- *selective data filtering* and *data reduction* – needed to ensure delivery of only those data items and those elements of items that are of current interest to the application, as with image cropping that matches the current end user's viewpoint in remote scientific collaborations, or with downsampling of an MPEG stream, to reduce the amount of data placed on the network so as to match the current networking conditions;
- *data transcoding* operations, used in OISs or scientific collaborations, for instance, to enable data exchanges that have the required format or necessary level of detail;
- *priority scheduling* – necessary to implement QoS-centric stream manipulations, e.g., in order or classify the data items being transported as in for critical data delivery in multimedia or sensor application;
- *stream merging* – needed to coalesce data from multiple sources, either to create a single collective stream containing all individual substreams, or to perform selective joins on inputs from multiple sources which satisfy specific requirements, such as temporal joins;
- ancillary tasks like data stream *monitoring* and *control* used for timely detection of critical events in sensor applications, or for status services such as online averages and sums; and
- *security* services that cover the wide range of security processing either already embedded into high end networking products (e.g., the crypto processing in the IXP2850) or under consideration for such machines (e.g., authentication, intrusion detection, attack forensics, etc.).

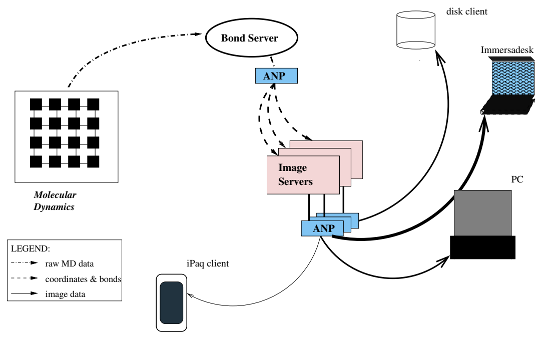

Figure 1: **SmartPointer application.**

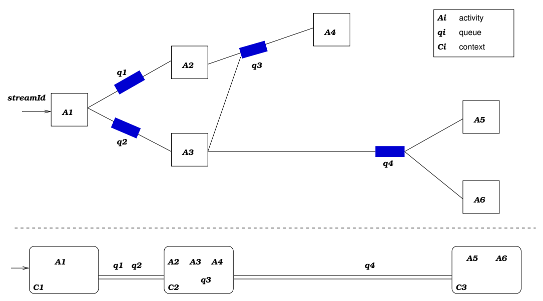

Figure 2: **Platform overlay model.**

3. ENABLING IN-NETWORK STREAM PROCESSING

In order to enable the execution of stream manipulation actions in the network, we introduce a model for representing application-specific processing actions that can be embedded and executed jointly with core forwarding functionality on networking hardware. Our approach relies on (1) a data abstraction to represent application-specific information about the stream data items to be manipulated; (2) stream operands represented through stream handlers [5], computational units to represent application-specific processing actions applied to stream data; and (3) service paths – the collection of stream operands applied to the end-to-end stream data path and the processing contexts where these operands are executed. The remainder of this section discusses each of these abstractions.

Data abstractions. The basic data abstraction is a *self-describing application-level data unit*. A data unit is self-describing through information embedded in the data unit's header, which can specify the data size and format, i.e. layout, offsets, and sizes of the application-level data fields that comprise the data unit, as well as the data unit's membership and position in a set of similar data units from the same source(s) or for the same destination(s). This information defines a *data tag* with two parts: one that defines the application-level data represented by the data unit, the other that determines its membership in a group, based on network- and/or middleware-level protocols. In the SmartPointer application, data units correspond to binary representation of simulation outputs, which also carry information about the event's type in the form of format identifier. The format identifier, along the the lower-level protocol headers make up the data tag.

A data unit may be represented as a sequence of one or more *data fragments*, all of which, except for the last one, are of size *fragment size*. Each fragment is tagged with a *fragment tag* that uniquely binds a fragment to a single data unit, and to a specific position in the sequence of fragments that compose the data unit. The first fragment in the sequence is the only one that has to contain the data tag corresponding to the application-level data unit. A single application-level data unit can have multiple representations as a sequence of fragments. The combined use of fragment and data tags permits us to apply communication actions typically associated with different levels of a communication stack. Examples are those associated with network header information, with system services, such as `sendfile` that operate of kernel buffers [7], and application-level processing actions. The actions applied are determined by a combination of data tag and fragment tag content, and they are applied to individual fragments' contents.

The collection of data items for which certain fields in the data tag have certain values, or sets of values, define a *data stream*. The fields in the tag that unify the data items in a stream can be solely the application-level portion of the data tag, such as format identifier, or they can be information about the path endpoints, such as source and sink addresses, or a combination of both. The subset of the data tag shared by all items in a stream represents a *stream identifier*.

Stream operands. Our previous work has introduced stream handlers, which are lightweight, parameterizable units of application specific computation that can be executed jointly with communication codes to implement manipulations of stream data. We use stream handlers to provide implementations of stream operands that can be embedded and executed at multiple locations on the stream data path from source to destination [5]. In the context of the data abstraction introduced above, a stream operation may be executed on the entire data item, or may be incrementally applied to subsequent fragments of stream data. For instance, a data selection operand may be applied once the entire stream data is delivered to a particular execution context, or may it may be executed as early as the fragment(s) containing the appropriate fields used by the select operation become available. Finally, a stream operand will have distinct implementations depending on the processing contexts where it is executed (e.g., to deal with platform heterogeneity), or depending on data fragment to which it is applied.

Service paths. The sequence of stream operands applied on the stream end-to-end path, described via the execution contexts which execute these operations, and the representation of stream data accepted by these operands, represents a service path. Service paths are partially ordered graphs, with graph nodes corresponding to stream operands, and edges to the flow of stream data items (see Figure 2).

4. PLATFORM OVERLAYS

In order to better understand how to execute stream manipulation services represented as compositions of stream operands, on heterogeneous, multi-core platforms, we model such platforms as *platform overlays*. Hardware (e.g., processors, coprocessors, NICs) or software (e.g., address spaces) processing contexts correspond to nodes in the platform overlay. The model builds on top of existing research on structuring computational paths as pipelines of elementary/basic operations [6, 15].

Multi-core platform assumptions. Several assumptions regarding future multi-core heterogeneous platforms underlie our research. First, we assume that these platforms will consist of a collection of general purpose and specialized cores, such as for communication, storage or graphics tasks. The IBM Cell processor is an example of such a multi-core system [3]. Next, we assume that individual

CPUs and memory are organized in a way that does not create interference from excessive access to shared memory modules or shared system interconnect. This assumption implies that processes bound to a CPU (for computation or communication) have exclusive access to private memory, 'local' to that particular CPU. Accesses to 'remote' memory are needed to implement intra-platform communication paths, and are controlled through the OS components executing on separate cores. Finally, since we focus on the communication capabilities of these systems, we assume that communication cores are dedicated to executing networking-related functionality, i.e. protocol processing [11], and have architectures specialized to efficiently execute communication stacks and handle multiple network interfaces, similarly to modern network processors.

Platform overlay runtime. Data is delivered to/from a context as a sequence of fragments (e.g., network packets, memory location, sequence of memory buffers), which is property of the context. The runtime at each context may translate the data from one representation to another (e.g., from network packets to application-level data item in memory). Within each context, one or more activation points may be identified, which correspond to locations where additional data accesses and manipulations can be applied (e.g., points along the layers in the protocol stack processing, kernel/user interface, etc.). Finally, a context may support concurrency, i.e., be multi-threaded. In a single data-multiple threads model, the application is responsible for providing multithreaded representation of the application-specific operation in order to benefit from the supported concurrency. In a multiple data-single thread model, synchronized access to shared state is enabled through compiler- and OS-level techniques and underlying hardware support.

Stream operands represented via stream handlers are executed at contexts' activation points. The runtime uses configuration state and stream meta data, implemented as hash tables, CAMs or other look-up structures, to invoke the appropriate handler (or sequence of handlers). In addition, the runtime provides resources for handler execution, such as handler internal memory, needed for parameters and state. If data is delivered to/from the context as a sequence of fragments with respect to which the operation implemented by the handler is incremental, it can be executed 'immediately after'/'just before' the data fragment is received/forwarded. For incremental operations, or operations that do not depend on special hardware or software components, there may be multiple deployment options in the overlay. The 'cost' of each of these options is determined through the attainable performance levels. However, certain application components are bound to specific context and activation points (e.g., an operation that performs floating point matrix arithmetic on unaligned data can only be invoked at message boundaries in a context where a floating point unit exists).

Platform overlay nodes are interconnected via communication channels, represented as queues. Each queue can have multiple inputs and outputs, i.e., arbitrary communication paths may be established among platform nodes, similarly to distributed network overlays. Communication is enabled via two basic operations: enqueue and dequeue. Queue elements consist of data and/or data descriptors, containing information regarding location and sender of the data, as well as additional meta-data, such as data tag, or integer counter for efficient implementation of operations such as zero-copy multicast, described in Section 5. Actual implementations of the queues and the supporting operations may vary. For instance, across contexts with separate memories, the movement of data requires copying. In such cases, enqueue and dequeue are implemented on top of underlying hardware-supported data movement operations, memory read/writes or direct I/O, and entail additional use of signaling mechanisms to exchange meta data and synchronize. In the prototype system described in Section 5, communications between the general purpose CPU and the communication subsystem involve DMA and programmed I/O across the PCI interconnect. Enqueue/dequeue operations for platform nodes with shared memory may involve only manipulation of the data descriptor, and avoid unneeded copies.

Meta information contained in data descriptors is used to provide mechanisms for backwards control-flow, needed for acknowledgments, reference counting, or to enable efficient, zero-copy implementation of data sharing operations, such as the multicast evaluated in the following section. Here, upon successfully updating and processing the data fragment containing the network header, each transmitting handler signals a notification that the subsequent use of the same data copy can proceed concurrently.

The use of platform overlays to abstract complex underlying hardware, coupled with stream handlers to represent stream manipulations as sequences of stream operands, permits us to construct service paths where specific operations are mapped to specific platform contexts, based on available resources or on application requirements. In this manner, in-flight stream customizations, as needed in distributed streaming applications, can be embedded with communications-related tasks on dedicated communications cores. At data sources, data sharing services can be implemented more efficiently by moving them onto networking hardware components, optimized for repeated execution of data transmissions. At stream destination nodes, preprocessing of stream data on hardware contexts not running core application processing, can reduce loads due to unneeded data, or can perform stream data transformation according to application-specific requirements, and eliminate the need for additional data copying due to format/encoding mismatch.

Finally, in order to deal with the dynamic nature of the applications considered, service paths in platform overlays must be reconfigurable. Towards this end, applications must have the ability to dynamically deploy new codes, and perform reconfiguration with zero-perceivable down time. We demonstrate that this is feasible on standard off-the-shelf programmable networking platforms.

5. EVALUATION

The next section demonstrates the importance of abstracting heterogeneous multi-core platforms as overlays of processing contexts with different capabilities, and of using application-specific information to dynamically map stream manipulation operations to the contexts best suited for their execution. Our results demonstrate that the flexibility offered by this approach, which permits applications to embed computations with communications and execute them on platform components dedicated for communications-related processing, can lead to performance gains derived from multiple factors. These include (1) reductions in processing loads on CPUs dedicated to core application processing, (2) mappings of operations to the contexts best suited for their execution, and (3) improved integration of computation and communication to ensure that the data being transported matches current application requirements.

In order to represent multi-core systems with communication cores, we use pairs of standard hosts and attached network processors (NPs). The integrated platforms used in these experiments consist of standard Linux hosts, interconnected via the PCI interface to Intel IXP NPs (IXP1200 and IXP2400), and are further described in [5]. In the host-based version of the experiments evaluated below, all data traffic is directed to one of the host's network interfaces. In the IXP-based results, data streams are delivered to the IXP's network interfaces, and, if necessary, data is delivered to the host-resident application component via the PCI interconnect.

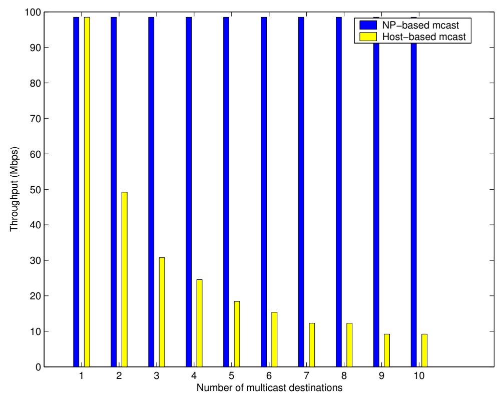

Figure 3: Multicast result.

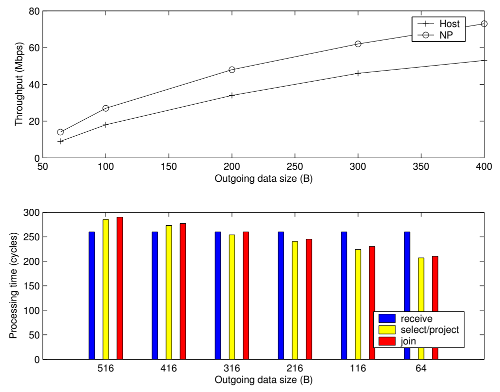

Figure 4: Operand results for input stream data sizes of 600B.

Efficient execution of communication-related services. The first set of experiments demonstrates the importance of dedicating communications-related processing to the communication cores available on future platforms. We implement an efficient multicast stream operand, which utilizes meta information in data descriptors to synchronize and overlap data transmissions to multiple clients.

We first compare the processing times of performing the multicast operation on the general purpose host vs. the communications processor. We record the timestamps when a packet is received and after the packet is multicast to n (in this case up to 10) different destinations. Measurements show that the multicasting done at the communication core (120 us) outperforms its host-based (209 us) implementation.

We next analyze the ability of both hosts and communication processors to maintain service levels. Results in Figure 3 are gathered for a stream of 1500B Ethernet packets, delivered at the rate of 98.5Mbps, and multicast to n destinations. Results indicate that the communication core is able to sustain our maximum input rate of 98.5 Mbps, for all n destinations. The host-based approach, however, exhibits a steady decrease in throughput sustenance as the number of destinations increases. These results demonstrate the inability of standard hosts to efficiently execute certain communications-related tasks, needed in distributed applications. Hence, in future multi-core systems, the platform overlay model presented in this paper will permit applications to identify and map such operations to communication subsystems, thereby significantly improving end user performance.

Ability to apply in-network stream operands. Next, we demonstrate the ability of specialized communication cores to efficiently and flexibly execute application-specific stream processing. The experiment uses two data streams carrying typical data from an airline's OIS. Experiments compare the execution of a set of stream operands of the NP vs. the general purpose host. The results shown in Figure 4 clearly demonstrate that the NP (i.e. the communication core) performs better than the general purpose host processor for all packet sizes. This is in part because the communication core has hardware support for operations like packet receiving, transmitting, queuing, pipelined multiprocessors, etc. Results also show that we can sustain operations that produce the same size sub-streams as the input data stream. Provided that typically, 'select' queries tend to produce smaller data-streams than the original stream, these 'worst case' results show that our pipeline implementation can sustain most queries. Moreover, the system is scalable because as the number of sub-streams increases, additional communication cores can be used for stream querying, and some of the query processing load can be shifted to host processors under high system loads. Finally, by performing stream processing on the communication core, the host processor is freed to carry out the more general processing actions for which it has been designed. In summary, these results show that stream processing on communication cores is feasible, scalable, and produces higher performance than when such actions are carried out by general purpose processor cores.

Importance of 'smart' data delivery. For the SmartPointer application described in Section 2, we evaluate the following customizations of the stream data: (1) filtering of unnecessary data, implemented as a set of data field comparisons, and (2) reformatting the incoming stream data to meet application-specific format requirements. We compare the overheads in terms of execution time of executing the specific customizations on the communications subsystem, i.e. on the IXP, vs. the gains observed for the host-resident application component. The host-side gains result from both reducing unnecessary loads to the host's I/O infrastructure and eliminating application-level data copies as a result of the format mismatch. In all cases, measurements indicate a net performance gain for our approach, up to 40% in certain cases (see Figure 5).

Additional experimental results. For brevity, we do not discuss in detail additional measurements that demonstrate the importance of embedding application-specific processing onto communication cores. These results include CPU utilization measurements gathered using the `linpack` tool, which demonstrate the utility of off-loading select functionality from the computational core. We have also evaluated the feasibility of supporting flexible platform reconfiguration mechanisms, which range from choosing among pre-loaded handlers, to using parameters to specify the stream handler actions, to dynamic hot-swapping of new codes. For the host-IXP platforms used in our work, dynamic reconfiguration can be achieved with practically negligible overheads, which reach up to 28 to 30us in the hot-swapping case.

6. RELATED WORK

The work presented in this paper builds on our previous research to create integrated platforms of hosts and attached network processors, so as to enable the execution of application-specific services onto the programmable NP and closer to the network [5]. Programmability of network processors has been widely exploited in both industry and academia, for delivering more flexible network- and application-level services [14, 17]. Our current work assumes that these integrated host-NP platforms exemplify future heterogeneous, multi-core systems, and aims to explore the additional benefits available through such tight integration.

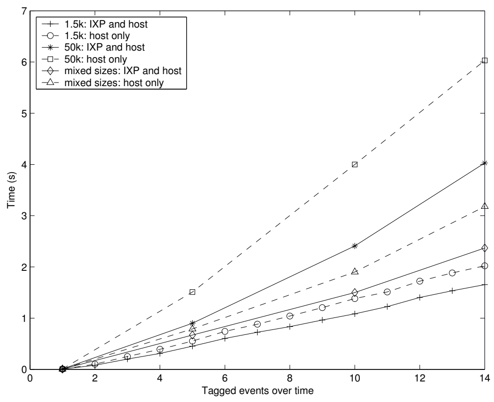

Figure 5: **Importance of customized data delivery**

The utility of executing compositions of various protocol- vs. application-level actions in different processing context is already widely acknowledged. Examples include splitting the TCP/IP protocol stack across general purpose processors and dedicated network devices, such as network processors, FPGA-based line cards, or dedicated processors in SMP systems [2, 11], or splitting the application stack, as with content-based load balancing for an http server [1] or for efficient implementation of media services [12]. Similarly, in modern interconnection technologies, network interfaces represent separate processing context with capabilities for protocol off-load, direct data placement, and OS-bypass [18, 13]. In addition to focusing on multi-core platforms, our work differs from these efforts by enabling and evaluating the joint execution of networking and application-level operations on communications hardware, thereby delivering additional benefits to distributed applications.

Finally, there are several existing models for representing computation in streaming applications, such as StreamIt or Spidle [15, 4], or models derived from active networking research. We differ from these approaches by providing a framework that can be uniformly applied to represent both network- and/or application-level manipulations to streaming data. For streaming operations modeled with these or other approaches, typical performance improvements are derived through OS-enhancements, and techniques such as specialized kernel calls, proxy services or OS-bypass [7]. Our work is complementary to these efforts, as we specifically focus on the additional performance gains which can be attained through judiciously mapping select streaming operations onto communications hardware. Our future work will evaluate the benefits from integrating these two techniques.

7. CONCLUSIONS

This paper evaluates the ability of future heterogeneous multicore platforms, with specialized communications support, to support efficient and scalable services for streaming applications. We argue that by using such platforms, distributed streaming applications can achieve performance improvements due to processor offload and the use of specialized networking hardware. Additional gains can be attained by permitting the deployment of application-specific computations onto communication cores, due to the efficient in-network execution of selected stream manipulations, or due to the improved ability to address the mismatch between received and expected data content, layout or format at communicating endpoints.

8. REFERENCES

[1] G. Apostolopoulos, D. Aubespin, V. Peris, P. Pradhan, and D. Saha. Design, Implementation and Performance of a Content-Based Switch. In *Proc. of INFOCOM 2000*.

[2] F. Braun, J. Lockwood, and M. Waldvogel. Protocol wrappers for layered network packet processing in reconfigurable networks. *IEEE Micro*, Jan./Feb. 2002.

[3] Cell Processor Architecture Explained. http://www.blachford.info/computer/Cells/Cell0.html.

[4] C. Consel, H. Hamdi, L. Reveillere, lenin Singaravelu, H. Yu, and C. Pu. Spidle: A DSL Approach to Specifying Streaming Applications. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, Erfurt, Germany, Sept. 2003.

[5] A. Gavrilovska, K. Schwan, O. Nordstrom, and H. Seifu. Network Processors as Building Blocks in Overlay Networks. In *Proc. of HotI 11*, Stanford, CA, Aug. 2003.

[6] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[7] J. Kong and K. Schwan. KStreams: Kernel Support for Efficient Data Streaming in Proxy Servers. In *Proc. of NOSSDAV'05*, Skamania, WA, 2005.

[8] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *Int'l Conference on Supercomputing*, 2003.

[9] V. Oleson, K. Schwan, G. Eisenhauer, B. Plale, C. Pu, and D. Amin. Operational Information Systems - An Example from the Airline Industry. In *First Workshop on Industrial Experiences with Systems Software (WIESS)*, Oct. 2000.

[10] B. Raman and R. Katz. An Architecture for Highly Available Wide-Area Service Composition. *Computer Communications Journal*, May 2003.

[11] G. Regnier, D. Minturn, G. McAlpine, V. Saletore, and A. Foong. ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine. In *Proc. of HotI 11*, 2003.

[12] S. Roy, J. Ankcorn, and S. Wee. An Architecture for Componentized, Network-Based Media Services. In *Proc. of IEEE International Conference on Multimedia and Expo*, July 2003.

[13] P. Shivam, P. Wyckoff, and D. Panda. Can User Level Protocols Take Advantage of Multi-CPU NICs? In *Int'l Parallel and Distributed Processing Symposium*, 2002.

[14] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a Robust Software-Based Router Using Network Processors. In *Proc. of 18th SOSP'01*, Chateau Lake Louise, Banff, Canada, Oct. 2001.

[15] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *International Conference on Compiler Construction*, Apr. 2002.

[16] D. Xu and X. Jiang. Towards an Integrated Multimedia Service Hosting Overlay. In *ACM Multimedia*, 2004.

[17] K. Yocum and J. Chase. Payload Caching: High-Speed Data Forwarding for Network Intermediaries. In *Proc. of USENIX Technical Conference*, Boston, Massachusetts, June 2001.

[18] X. Zhang, L. N. Bhuyan, and W.-C. Feng. Anatomy of UDP and M-VIA for Cluster Communications. *Journal on Parallel and Distributed Computing*, 2005.

[19] Y. Zhao and R. Storm. Exploiting Event Stream Interpretation in Publish-Subscribe Systems. In *ACM Symp. on Principles of Distributed Computing*, Aug. 2001.