Addressing Data Compatibility on Programmable Network Platforms

Ada Gavrilovska
Center for Experimental Research in Computer Systems (CERCS)
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, 30332
ada@cc.gatech.edu

Karsten Schwan
Center for Experimental Research in Computer Systems (CERCS)
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, 30332
schwan@cc.gatech.edu

ABSTRACT

Large-scale applications require the efficient exchange of data across their distributed components, including data from heterogeneous sources and to widely varying clients. Inherent to such data exchanges are (1) discrepancies among the data representations used by sources, clients, or intermediate application components (e.g., due to natural mismatches or due to dynamic component evolution), and (2) requirements to route, combine, or otherwise manipulate data as it is being transferred. As a result, there is an ever growing need for data conversion services, handled by stubs in application servers, by middleware or messaging services, by the operating system, or by the network. This paper's goal is to demonstrate and evaluate the ability of modern network processors to efficiently address data compatibility issues, when data is 'in transit' between application-level services. Toward this end, we present the design and implementation of a network-level execution environment that permits systems to dynamically deploy and configure application-level data conversion services 'into' the network infrastructure. Experimental results obtained with a prototype implementation on Intel's IXP2400 network processors include measurements of XML-like data format conversions implemented with efficient binary data formats.

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

ANCS'05, October 26–28, 2005, Princeton, New Jersey, USA.

Keywords

Network processors, Streaming applications, Data Morphing

1. INTRODUCTION

Large-scale applications depend upon the efficient exchange of data among their distributed application components, including the receipt of data from heterogeneous sources and the transfer of data to clients ranging from high end server systems to embedded devices. Inherent to such data exchanges are (1) discrepancies among the data representations used by sources, clients, or intermediate application components (e.g., due to natural mismatches or due to dynamic component evolution), and (2) requirements to route, combine, or otherwise manipulate data as it is being transferred. As a result, there is an ever growing need for data conversion services, implemented by code running in application servers [25, 33], by middleware [7] or messaging systems [1], by the operating system, or by the network.

The approach of our research is to manipulate data while it is 'in transit' between distributed application-level services, morphing the data from forms used by senders to the forms required by recipients. The specific goal is to provide efficient data compatibility – data morphing – services for interactive or transactional applications, including real-time collaboration in the scientific domain [33] and the operational information systems (OIS) used by corporations [21]. Such applications routinely differentiate between external vs. internal data representations, and it is common to have differences in data formats in their continuous interactions with external partners or entities performed as part of distributed workflows. For instance, in the airline OIS we have been studying [14], data is represented differently at sources like the FAA vs. ticketing agents, internally within the company's OIS, or when exchanged with external clients like caterers. In addition, data merging occurs when responses to clients' reservation requests are combined with advertisements like those generated by DoubleClick for client-customized marketing. Further, in all such cases, applications evolve over time, with different subsystems evolving independently of one another, again causing issues with data compatibility.

Our approach to morphing data while it is 'in transit' between senders and recipients is motivated by multiple facts. First, modern network processors have reached the

maturity and computational power that has made it possible to perform application-specific operations on data as it moves across the distributed overlays used by applications [30, 34, 15]. In [12], for example, an Intel IXP 1200 is shown capable of executing low-level business rules at link speeds. Second, by manipulating data 'in transit', unnecessary memory or storage actions are avoided, permitting applications to focus on the data they actually and currently need instead of mining substantial data volumes in secondary stores. Third, particularly for operational applications like those considered in our work, it is not just raw performance like throughput, but also delay and predictability in terms of end-to-end delay that determine the quality of ongoing data transfers, to maintain contractual SLAs (Service Level Agreements) [19] or certain levels of QoS (Quality of Service). By combining data transport with data morphing, items of interest can be extracted from data with small delays, and such extractions can be done more predictably (in terms of delay) compared to similar actions performed by application-level overlays, the latter being subject to perturbation by changes in system loads. Third, 'in transit' data morphing can reduce loads on server systems and sometimes, on the network itself, by delivering exactly the data needed by end hosts in precisely the forms in which they require it. Results presented in [32], for instance, show manifold reductions in data volume when server systems customize data on a per-client basis. 'In-transit' data morphing can have the same effects, while also avoiding some of the undue server loads implied by dynamic data customization [28].

The specific objective of this paper is to explore the degree to which modern communication subsystems can provide efficient data morphing services. We focus on the execution of data compatibility services in programmable engines like Intel's IXP line of NPs (Network Processors). Services are run entirely on the NP, operating on packet streams of application data in one representation and translating them into different destination-specific representations, or they are performed jointly with the hosts to which NPs are attached. The latter case emulates the capabilities of future heterogeneous multi-core machines [8], in our case comprised of a general computational engines connected with a dedicated link (i.e., PCI bus) with the communications cores resident in the IXP NP.

Our technical contributions toward 'in transit' data morphing services are as follows. First, we develop an execution environment that permits applications, through middleware, to dynamically deploy 'chains' of application-specific data handlers onto network processors that execute them. The paper discusses the basic mechanisms required to provide such functionality, specifically focusing on compatibility services. Second, a prototype implementation of this environment uses Intel's IXP2400 network processor, both as a standalone data morphing engine and/or as a programmable network processor attached to a standard host, the latter emulating future multi-core machines. Third, experimental results attained on these platforms demonstrate the feasibility of executing morphing services on networking platforms, including chained format translations, as well as generating new data by merging inputs from multiple sources.

The remainder of this paper first provides background on existing approaches to enable application-specific services on programmable network devices (see Section 2). Section 3 describes in greater detail applications heavily dependent on efficient data compatibility services, current approaches to providing such services, and requirements on efficient service provision on programmable communication systems. The design of the proposed execution environment appears in Section 4, and experimental evaluations of the environment are described in Section 5. Concluding remarks and future directions appear in Section 6.

2. APPLICATION-LEVEL SERVICES ON NETWORKING PLATFORMS

Recent technology advances in network and communication systems have enabled the creation of programmable networking platforms to which developers of network services can deploy application-specific communication services. These services address the requirements of large-scale applications in the business, multimedia, and scientific domains, including multicast solutions [5], voice and video over IP [24], QoS for multimedia applications [15], security services [9, 22], etc.

Specific platforms with which researchers have experimented include network processors [13, 30, 34, 9], programmable network interface cards (NICs) [34, 23], and intelligent line cards [6]. Experimental results have demonstrated the utility of executing various compositions of protocol- vs. application-level actions in multiple processing contexts, specific examples including splitting the TCP/IP protocol stack across general purpose processors and dedicated network devices, performing selected protocol functionality in FPGA-based line card [6], using dedicated processors in SMP systems to execute selected protocol functionality [26], or splitting the application stack, as with content-based load balancing for an http server [4] or for efficient implementation of media services [27]. Similarly, in modern interconnection technologies, network interfaces offer separate processing context to enable protocol offload, direct data placement, and OS-bypass [35, 29].

Our approach leverages the work reviewed above in adopting the attitude that modern network processors or interfaces are capable of sustaining high link rates while also executing rich functionality on messages while in transit. We differ from prior work, however, in the degree to which applications can take advantage of network-level processing, our approach being to utilize middleware as an arbiter between applications and network. That is, middleware can interact with the underlying system to dynamically place certain application actions 'into' the network subsystem, where the network subsystem provides an execution framework for running such middleware-level services. This paper focuses on data compatibility services, outlining the network-level execution framework and then exploring the opportunities and limitations of morphing data 'in transit' in this context, for modern network processors like Intel's IXP 2400 programmable network router.

3. DATA COMPATIBILITY SERVICES – NEEDS AND REQUIREMENTS

The need for data compatibility – data morphing – services has increased substantially during the last decade. There are many reasons for this increase, including outsourcing requiring corporations to tightly couple their internal with external workflows across partners and subcontractors. Another reason is the increased componentization of business

processes or scientific applications to facilitate their development and adaptation to new needs. One outcome of interest to our work is the repeated and frequent need to adjust data formats and layouts to internal data representations used by different software components, both within single large-scale applications and across multiple cooperating applications. In the scientific domain, for instance, data translation is routinely needed in the complex simulation systems [20] being constructed from multiple, domain-specific simulations produced by different scientific sub-disciplines. Another outcome relevant to our research is the plethora of both general (e.g., based on XML schemas) and domain-specific data formats used in commercial and research settings, which requires dynamic format translation and puts increased pressure on middleware or messaging systems to provide automatic methods for morphing data to deal with dynamic data or format evolution across complex distributed systems [1].

Requirements imposed on data morphing services are derived from their inherently application- or domain-specific nature. In contrast to network-level services like multicast, data morphing services require (1) access to application-level information that describes data formats and their layout in message payloads and (2) the use of application-level code modules – handlers – that implement data morphing actions. The amounts of state needed to morph data depend upon the sizes and complexities of data formats and the footprints of the handlers applied to data messages.

Direct format translations may be described as $fout = h(fin)$, where handler state entirely depends upon the sizes on the input and output formats involved. An example of such a service evaluated in our work is the translation of data from a company's internal representation fin to a representation that can be shared with an external party $fout$. Data merging operations are described by $fout = h(fin_i)$, where fin-s are matched according to some data identifier or field value (i.e., merge key), and where the amount of state needed depends on the fan-in and the 'merge window'. Merging operations where inputs are more synchronous (i.e., data items related by the same key arrive closely together) will need to store data for the duration of those short intervals only, and therefore require less state. An example of a merge operation considered in our work is the generation of client-personalized responses to reservation requests, where the reservation response is merged with client-specific advertisement generated by a service such as DoubleClick. Finally, the state requirements of data splitting $fout_i = h_i(fin)$ operations depend upon their ability to reuse the original data representation to create output data, as opposed to copying relevant input fields to create each of the $fout$-s. Multicast can be considered as simple example of a splitting service, where the contents of the network header must be modified in a destination-specific manner [10].

The data morphing actions described above involve applying a single handler to a single or to multiple data streams. More generally, data morphing will require the repeated application of handlers to messages, resulting in handler chains that perform certain translation actions. This paper presents basic insights concerning handler chaining, with different length chains and differing amounts of state involved in chain execution.

The remainder of this paper builds upon the straightforward model of handlers, handler state, and handler execution presented in this section. Our general goal is to methodically explore the abilities of modern network processors to morph data with single or chained handlers. Our specific goals are to evaluate certain capabilities of NPs, including (1) the overheads of handlers that use general data formats descriptions and can therefore, deal with a variety of data formats received and sent by the NP vs. using handler code that is specialized for individual data formats, (2) the effects of format complexity and footprint size on handler performance, and (3) the ways in which multiple inputs/outputs and handler chaining affect the performance of stream processing.

Finally, a wide range of traditional networking services, such as NATs, protocol bridges, VoIP proxies, etc., also belong to the class of data translation services. In these cases, translation is primarily confined to the packet header, (e.g., update destination address, construct protocol-specific fields, etc.). Some of the requirements discussed in the following section can also apply to these classes of applications.

4. DATA TRANSLATION ENGINE

4.1 Enabling In-Transit Data Morphing

The following section discusses in greater detail the functionality necessary for efficient, in-transit execution of data morphing services. The objective is to identify the runtime components needed to implement in-transit data morphing, and to understand the handler chaining and state complexities involved.

4.1.1 Representing Application-level Data

Unlike traditional implementations of network-centric service on networking platforms, the ability to perform manipulations of application-level data requires that we (1) construct the application level message, and (2) apply processing to it whenever all appropriate data components become available. Therefore, we can no longer adopt the packet pipeline model often used by network-centric applications. Instead, we maintain application-level data as linked lists of packet buffers, where packet data may be dispersed throughout NP memory. Furthermore, data placement and data morphing are tightly coupled. Contiguous placement typically implies simpler data morphing code but may also require additional data copying. Dispersed placement implies more complex morphing code but avoids additional copies. We next describe the packet buffer structures used in the 'in transit' execution environment.

Because of the need to operate on variably sized portions of application-level messages, receive side processing cannot use a simple ring buffer structures to consecutively place packets in NP memory. Instead, our implementation relies on a freelist manager to identify memory into which packet data can be placed. Sufficient protocol code is executed to link each packet with the other packets belonging to the same application-level message (i.e., a lookup operation is performed on receipt to determine a packet's appropriate location in an application-level message). Pointers to messages currently under assembly are cashed into fast memory shared across different receive processing contexts. Similarly, the input to transmit side processing cannot simply be a head/tail of a ring-buffer of pointers to packets, but it is a two level ring of pointers to lists of packets. Enqueue/dequeue operations imply placement of single entries into ring buffers, where each entry now points to a linked

list of packets. Inputs to any additional processing in the 'pipeline' are packet lists. Outputs may be the same list, subsets of it, or they may require creation of new lists (whose elements may be combinations of existing or newly created packets). This implies that the freelist manager needs to service requests from both receive side and intermediate processing contexts.

4.1.2 Classification

The execution model described above requires repeated lookups to run data-specific handlers on application messages: (1) to determine the location of the data packet-list, (2) to determine the appropriate handler, or sequence of handlers, as well as (3) to determine the next network destination. Typical network services rely upon classification based on network fields, such as source/destination IP addresses, ports and protocol types. The data compatibility services addressed by our research require more flexible classification criteria, as combinations of both network and application-level fields. For the latter, we use 'format identifiers', which uniquely identify the binary formats describing application-level message contents [7]. Format descriptors permit us to write handler code that can be applied to multiple application-level messages that differ in format while requiring the same processing actions.

4.1.3 Handler Chaining

In order to enable to coexistence of data services that operate on different versions of the same data, or to give different contexts their own views of the same data, applications depend upon the ability to incrementally apply data translations from some original to some target representation. This process may use multiple intermediate representations, which are artifacts of the translation process, but are not otherwise important. For instance, to convert a passenger record from one containing SSNs and other private information to an indication of passenger meal preferences, one step may strip SSNs and similar private data, and a second step may compute composite records indicating flight meal preferences.

Any such sequence of incremental data translations can be implemented as a chain of handlers. A data translation engine, therefore, must be able to push data through a series of handlers. That is, it must support multiple execution contexts where handlers can be deployed and executed, and it must provide communication channels between such contexts.

4.1.4 Data Merging

As explained in Section 3, data morphing services often imply that new data formats are generated by merging data inputs from multiple independent sources. The resulting format may be a simple union of all fields (and content) of the input data items, or it may be produced as a function over all/some of their fields (e.g., arithmetic function such as sum, average, max, or a more complex select operation). Merging criteria may be expressed with temporal conditions, e.g., within every 5sec produce one output consisting of averages of recent temperature and pressure values [18]. In this case, the functionality required is the ability to maintain 'windows' of data items from each stream, so as to be able to produce the most recent composite value with the specified period.

A second class of merging operations may require that data is merged only if selected data identifying fields in each of the corresponding events match. For instance, in the airline example, a response from the airline business system is generated for a specific transaction, only after passenger profile content has been received from a separate subsystem. In this case, the data translation engine must have efficient lookup capabilities based on application-specific parameters like transaction identifiers.

4.1.5 Data Splitting

The performance of data splitting services is highly dependent upon their ability to reuse the original data representation to create the desired output data. This approach minimizes the state requirements imposed by the splitting service, and improves performance by avoiding costly data copying operations. Select classes of splitting services can benefit from the availability of a feedback mechanism which signals that shared portions of the data have been safely copied or transmitted, thereby avoiding copying of the entire data item, and yet maximizing the overlap of data modification and data transmission. Our group already deployed this mechanism for obtaining an efficient implementation of a destination-customized multicast.

4.1.6 Reconfiguration

The final set of requirements on efficient execution engines for application-level message handling concerns runtime reconfiguration. This is necessary to deal with runtime evolution of data formats or data services, which may be dealt with by dynamically modifying, reconfiguring, or deploying message handlers.

4.2 Handler Execution Environment

In order to better describe how to execute data translation services represented as chains of data handlers, we model the environment in which data morphing handlers run as a 'platform overlay' of execution contexts [11]. Execution contexts may be provided by hardware (e.g., processors, coprocessors, NICs) or by software (e.g., address spaces), and the set of processing contexts involved in executing a chain of handlers corresponds to multiple nodes in the platform overlay. Building on existing research on structuring computational paths as pipelines of elementary/basic operations [16, 31, 3], data is received through one of the platform's network interfaces, or from application components executing on host systems part of the platform via some system-level interconnect (e.g., PCI). Similarly, data may be transmitted onto the network, or it may be moved to the attached host systems' memory. Our packet processing architecture therefore permits packets to be processed in communication processors like Intel's IXP NPs and/or at communication end points, where packets may be processed in NICs and/or by host CPUs. In the latter case, the architecture simply assumes that NICs perform sufficient protocol code to strip packets of network headers, placing the application data contained in packet payloads into memory accessible to application code running on host CPUs.

Figure 1 represents a platform consisting of a general purpose core (i.e., a host node) and an IXP2400 NP, attached to it via the PCI interface. The processing contexts in this platform correspond to the IXP microengines and host-resident application components. Data items of format $F1$ traverse

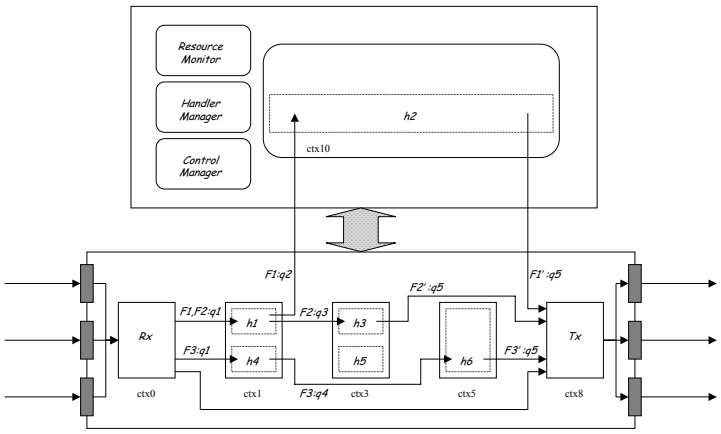

Figure 1: Platform Overlay.

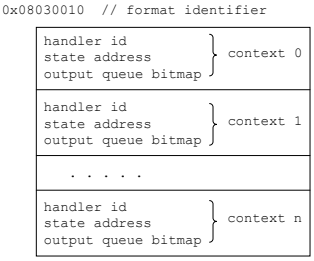

Figure 2: State maintained for each format identifier. At a given context, the appropriate handler is invoked. The handler uses the provided state at `state address` in a handler-specific manner. The output queue(s) is encoded via the output queue bitmap. The queue uniquely identifies the next context in the platform overlay where data with this format will be processed.

contexts 0, 1, 10, and 8, i.e., the data path crosses the host-IXP boundary, while data items of format $F3$ traverse contexts 0, 1, 5, and 8, i.e. the data path is fully contained on the network processor.

The basic functionality of any engine executing data translation services is the delivery of data from sources to destinations. Data morphing operations are executed jointly with the data forwarding functionality. As a result, in the execution environment, we identify a core data path through the overlay of processing contexts, from the context executing receive-side protocol processing, to those that transmit data (e.g., in Figure 1 all data items not belonging to formats $F1$, $F2$, or $F3$ are processed on the default data path by contexts 0 and 8). All other available processing contexts may be included in the data path, to perform additional data manipulations.

The context supported on the platform are interconnected via shared memory queues. Our current implementation assumes one input queue per context, which is shared for all data types, however the architecture can be easily extended to multiple input queues, where each queue may be associated with different service requirements, for instance. At each context data items the appropriate handler is applied to the data, based on its format identifier.

The above execution environment relies on additional support components, which include:

- *Resource Monitor* – gathers information for the current resource availability along data paths in the overlay;
- *Handler Manager* – responsible for managing the handler representations for different contexts, their current deployment, resource requirements and data format associations; and
- *Control Manager* – enables the (re-)configuration of the datapaths through the platform overlay and the exchange of relevant resource utilization information.

The implementation and functionality provided by these support components is further discussed in [10, 17].

4.2.1 Data Paths Through the Platform

For each format, the datapath through the platform overlay is encoded in shared configuration state accessible to all processing contexts. On the integrated host-IXP platforms used in our work, this state is replicated across both the IXP and the host, and the Control Manager ensures its synchronized updates. For data of format $F1$, this state includes configuration states for each of the contexts traversed by the data items. The configuration state consists of (1) a translation action, (2) state address, and (3) forwarding action. This state essentially enables the routing of the application data through the format handlers deployed in the platform overlay. Our current implementation uses network-level information, such as source-destination address pairs, or the address of single end point only (e.g., destination), or the format identifiers of the PBIO [7] binary data encoding, to distinguish between data flows. At each context, we presently use the context identifier to access the appropriate configuration state. However, a more scalable future implementation will simply pass the next-context state offset in the enqueued data handle. The translation action refers to the handler that can morph the data in a format recognized by the destination (and the contexts where that handler is currently deployed), or it simply indicates that the data is already in an acceptable format for further transmission. The state address identifies a fixed-size memory buffer to be used in a handler-specific manner (e.g., for counters, transient results, etc.). Handlers with large state requirements, such as handlers implementing merge operations, which need to maintain an entire window of data it their state, may request additional (slower) memory from the runtime, and maintain pointers to it in the preallocated state buffer. The forwarding action encodes the next context on the datapath through the overlay, by specifying the corresponding data queues.

For pipeline-fashioned translations (i.e., where data undergoes a series of transformations represented as a chain of handlers before being forwarded to its destination), data events are dispatched from the queues in a round robin manner, and the appropriate handler is applied to them. On

multi-core platforms, such as the IXP2400 used in our work, several possibilities exist regarding the mapping of format handlers to processing contexts. Ideally, one or more processing contexts are associated with a execution point in the chain, and all contexts are capable of executing any of the handlers. As a result, such an engine can benefit from hardware supported optimized data paths on the platform, such as next neighbor registers on the IXP2400, or fast shared memory available between adjacent processing elements on other network processing platforms [2]. Furthermore, the availability of multiple parallel processing contexts at each pipeline stage can be used to address variations in data processing times.

However, in this case, the length of the permissible chain length is fixed by the engine configuration. In addition, each chain reconfiguration may trigger a sequence of reconfigurations in order to match the new handler chain to the hardware-supported pipeline. Finally, platform limitations like the limited instruction store size per processing element may prevent the deployment of all format handlers from a chain stage at a processing context.

We use a more general approach, which permits arbitrary assignments of handlers to processing contexts. The approach takes into account handler resource requirements, in terms of instruction store, state, and computing cycles, and it considers resource availabilities at contexts. The resulting execution environment is one that provides data paths between all available contexts in the overlay, essentially by enqueuing data to the appropriate input queue(s) for each context. Whenever possible, the implementation of such data paths should be built on top of hardware-supported fast communication channels (e.g., utilizing the next neighbor registers for communication between adjacent microengines on the IXP2400). The data translation service may still be represented as a pipeline of format handlers, but their mappings to the underlying platform resources create data paths with arbitrary topologies. The Control Manager depicted in Figure 1 ensures synchronized accesses to the communication channels between contexts, thereby facilitating all intra-engine communication. For the classes of data morphing services evaluated on the IXP2400 platform in Section 5, we demonstrate that this approach still permits in-transit execution of these services.

4.2.2 Memory Issues

In addition to control managers for coordinating handler interactions, a data translation engine requires the ability to manage underlying memory resources. Memory regions are reserved for data handle queues, locks and mailboxes, necessary for communication. Additional state is necessary on a per format basis to maintain format-related state, such as for the 'routing tables' through the engine's contexts and handlers. The remaining available memory is available for storing data, both for received packets and for data generated by handlers. A freelist manager is used to manage this memory.

For morphing services that require merging of data inputs from multiple sources, one or more data items are temporarily stored until all remaining components become available. As a result, the runtime provides to such merging handlers lookup mechanisms for querying and updating the availability of all data components which are to be merged. This is implemented through a collection of per format structures.

Chain length	0	1	2	3	4	5
Throughput	980	981	980	978	978	978
Latency	0	350	351	352	353	355

Table 1: Throughput (Mbps) and per-stage average latency overhead (cycles) for rule chains of different length.

For merging services like temporal joins, these structures represent arrays of sizes dependent upon the time window requirements. For services where data items are merged based on unique keys, this key is used with hash tables to access the data components. In the event of multi-level memory hierarchy, these data structures should be implemented in the fastest available shared memory, so as to minimize the performance overheads. For the IXP2400-based implementation, we evaluated the possibility of placing such data in on-chip scratch memory, as well as SRAM, and the associated size/performance trade-off [12].

4.2.3 Reconfiguration

To address runtime application changes, the execution environment must support mechanisms for extending the runtime with new handlers, or modifying existing ones. Currently, we assume that such reconfigurations are application-driven. That is, application end-points provide format handlers either directly or in response to a request for such handlers from an application component that does not recognize the new format and perhaps, requires data to be translated to an older version. Reconfiguration is initiated through engine control and verification components residing in a separate processing context (e.g., the control host processor attached to the IXP). The functionality provided by these components ensures that any reconfiguration actions (1) can be satisfied with the currently available resources on the engine, and (2) will be consistent with any existing inter-dependencies among handler inputs and outputs. The specific reconfiguration actions include any combination of the following:

- modifying the data path through the runtime, so as to enable a handler currently deployed at the newly included context in the overlay;
- changing the behavior of an existing handler by specifying different parameters; or
- replacing a currently active handler with a new one (e.g., an improved implementation of the original one), by selecting among currently deployed handlers on the current data path, or by dynamically deploying new handler codes on available processing contexts, and hot-swapping existing ones.

For the IXP2400, these configuration methods have already been implemented and evaluated by our group [17, 11].

5. EVALUATION

The experiments presented in this section are conducted using a cluster of eight Dell 530s with dual 1.7GHz Xeon processors running Linux 2.4.18, outfitted with Radisys ENP2611 IXP2400-based boards, and interconnected via 1Gbps and 100Mbps links. We also use a cycle-accurate IXP2400 sim-

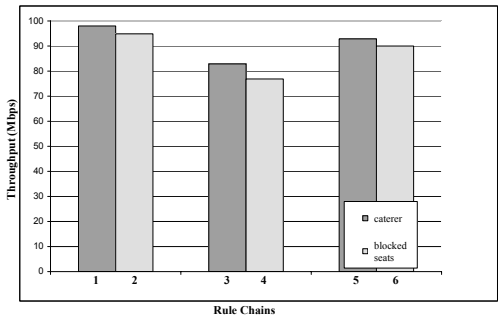

Figure 3: Throughput levels with different rule chains.

ulation package SDK4.0. The data streams used in experiments are generated with sequences of demo-replays of business data collected at a large company with whom we are working. The application components executed on the cluster nodes use our version of the RUDP protocol on top of raw sockets. The same protocol is used on the IXP microengines. We have evaluated the viability of executing various application-level services on the IXP NP as opposed to host nodes only (going directly to the host's ethernet link, without an IXP in the path), as well as the performance levels that can be achieved, by implementing different services with handlers that execute jointly on hosts and on their attached IXP2400s.

The experimental setup uses one IXP2400 NP as a translation engine, and data is streamed to/from this engine from/to other cluster nodes. The data streams consist of events ranging from 350 to 1150B. The results demonstrate the possibility of executing in-transit data translation services on programmable network systems. They also illustrate current limitations of such in-network data morphing services.

5.1 Handler Chains

The first set of results demonstrates the ability to apply chains of consecutive data translation handlers, so as to address progressive format evolution issues, or to address the needs of application subsystems for different and multiple data representations.

In order to demonstrate the feasibility of applying handler chains to data items in transit and to illustrate limitations, we present multiple data points. The results in Table 1 compare the sustained throughput levels and the increase in latency when applying chains of handlers to data forwarded through an IXP-based translation engine. A chain length of 0 corresponds to the baseline case where no additional data translation is performed in the forwarding path, and data is transmitted immediately upon receipt. For chain lengths greater than zero, each handler in the chain corresponds to a dummy handler. That is, we evaluate the raw overhead of performing the additional enqueue/dequeue and configuration state look-up operations at different contexts. The results demonstrate that the effect of adding handler stages on the attained throughput levels is negligible, primarily due to the hardware-supported parallelism available on the platform. The additional latencies due to added handler invocation are small, on the order of 1us on the 600MHz IXP microengines.

The detailed evaluation of the impact of handler chains demonstrates significant performance fluctuations based on the amounts of state manipulated by each handler and the data flow between handlers in the chain, the latter in turn is affected by the handlers' orders in the chain. For instance, applying a translation handler $h1$ that first extracts all needed fields from a data item, followed by handler $h2$ that reformats these fields in the required output format, results in significant performance improvements compared to a second possibility – where data is first reformatted by handler $h2'$, and then the content reduction is performed by $h1'$.

State manipulation is a more important contributor to chain execution costs. The bars in Figure 3 correspond to six chains of two handlers each. The first chain corresponds to a case with strong data reduction, similar to the one described above – the first handler extracts needed fields, and the second handler reformats them. The data is reduced from the original 350-1150 bytes to 72B. The output data flow represents 11% of the input. The layout of the data items is such that this transformation is accomplished by overwriting the first 72 bytes of the data with content extracted from subsequent fields, all of which are aligned on word boundaries. The second chain corresponds to a weak data reduction case – we extract from the original data only the fields that contain some company-sensitive information, in this case fields related to blocked airline seats. This reduces message size by 2-20%. The blocked seats fields, when present, appear near the very end of the PBIO data representation of the airline data, and as such can be discarded with limited amount of data copying.

The third and fourth chain reverse the handler order compared to the previous two examples. There is no reduction in the data flow between handler stages in these two chains. In order not to affect the computational complexity of the handlers in the chain, the reformatting handlers in each case simply read and write every field in the data item.

Finally, in order to asses the effects of state manipulation only, we modify the reformatting handler in the third and the fourth chain to discard 90 or 10% of the data items, respectively, thereby achieving similar data flow reductions between handler stages as in the first two handler chains.

The results demonstrate multiple facts. First, it is possible to construct and efficiently execute meaningful chains while sustaining high throughput levels and incurring only small latency overheads. Second, to address state concerns, it is important to develop compiler techniques to reduce the amount of state manipulations, including by determining the appropriate orderings of operations in handler chains.

5.2 Merging

Our second set of experiments evaluates more complex data morphing operations, those that require merging of data from multiple sources to produce some desired output formats. As discussed in Section 4.1.4, merging services require that data is temporarily stored in the translation engine, until all data components are received. The ability to efficiently perform merging operations is dependent upon the 'merge window'. The results in Figure 4 describe the ability of the IXP NP to perform data merging services.

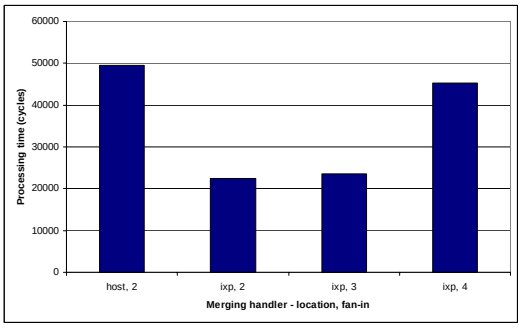

Figure 4: Evaluation of a merging handler for different fan-in values.

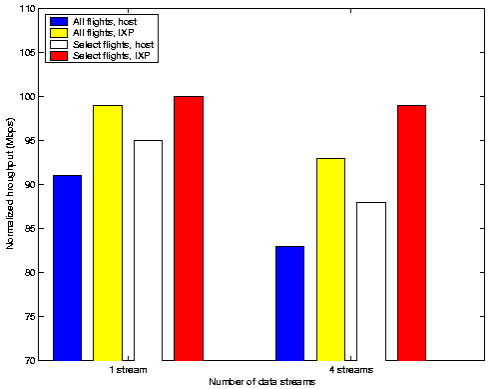

Figure 5: Importance of in-transit data morphing.

The merge operation is implemented on two microengines – the handlers executing on the first one perform the lookups and update the corresponding hash tables, and the handlers executed by the second one create the new data events. The results correspond to data merged from two, three, or four inputs. We also compare to these results a purely host-based implementations of the same data merging service, where data is delivered directly to and from the host's network interfaces, and processed by the host's version of the same protocol code executed by the IXP's microengines.

Our first observation is that the NP-based implementation of the merging operation is significantly faster than the host-based one: the IXP implementation improves performance by more than 50%. Second, as long as the memory footprint of the implementation can fit within available IXP memory, performance-levels scale well with increases in merge fan-in. There is only a 5% decrease in performance when merging three as opposed to two streams.

Results also expose the limitations of implementing merge-like operations on a network processor with limited memory resources. In our implementation, performance severely degrades when attempting to merge four data stream. The addition of the fourth stream stresses the limits of the available fast memory on the IXP, where hash tables associated which each stream reside. Furthermore, handling data from multiple streams decreases the synchrony between related events from each of these streams (i.e., the time delay between events with the same key). Therefore, data is stored in the IXP longer, which further decreases the engine's capacity.

5.3 Importance of In-transit Data Translation

Finally, we also demonstrate the importance of in-transit data translation when deployed on the communication subsystems at destination end-points. The results in Figure 5 represent the importance of offloading the data translation services from application components executed on standard host, onto their attached programmable network interconnect cards. We use the IXP2400 attached to host via their PCI interface to emulate such NICs. The results represent the performance levels attainable for host- vs. IXP-based execution of rule chains which translate data into the appropriate format (bars marked 'all flights'), or translate the data, and extract information currently needed by the application (bars marked 'select flights'). Results demonstrate that the in-network execution of these rule handlers results in improved performance, primarily due to offloading of the CPU, and removing loads from the host's I/O and memory subsystems, but avoiding unnecessary data copying and protocol stack traversals.

6. CONCLUSION AND FUTURE WORK

The paper addresses data morphing services – an important class of services required in large scale distributed applications, by enabling their execution on programmable network processors when data is 'in-transit' between application-level services. For these services, we demonstrate the importance and feasibility of their network-near execution, jointly with data forwarding and distribution services executed by communications hardware in the distributed infrastructure.

We discuss the functionality needed to implement data morphing services, and we describe a model of a network-level execution environment. A prototype implementation of a data-translation engine is evaluated with the Intel IXP2400 programmable network processors, using morphing services that involve chains of format translation handlers or services that merge data from multiple sources based on select merging criteria. The results demonstrate the importance and feasibility of enabling in-transit data morphing services, and they outline the limitations of the approach.

Our future work will focus on further analyzing the resource requirements presented by dynamically reconfigurable handler chains, and quantifying the performance levels attainable under specific platform conditions, in terms of computational loads, available memory, etc. In addition, we are working towards generalizing the proposed approach of 'in-transit' data manipulations to enable the execution of arbitrary middleware-level functionality, such as data customization or consistency services found in publish-subscribe systems. The objective is to enhance standard middleware systems by executing select operations on the communication subsystems in the distributed overlays, thereby enabling middleware-aware networking infrastructures.

Acknowledgments. We would like to acknowledge Sanjay Kumar, Paul Royal, Srikanth Sundaragopalan, and Joe Uhl for contributing to the implementation and evaluation of

select components described in this paper.

7. REFERENCES

[1] S. Agarawalla, G. Eisenhauer, and K. Schwan. Lightweight Morphing Support for Evolving Middleware Data Exchanges in Distributed Applications. In *Proc. of International Conference on Distributed Computing Systems*, 2005.

[2] Agere systems payloadplus network processors. http://www.agere.com.

[3] D. S. Alexander, P. B. Menage, A. D. Keromytis, W. A. Arbaugh, K. G. Anagnostakis, and J. M. Smith. The Price of Safety in An Active Network. *Journal of Communications and Networks (JCN), special issue on programmable switches and routers*, 3(1):4–18, Mar. 2001.

[4] G. Apostolopoulos, D. Aubespin, V. Peris, P. Pradhan, and D. Saha. Design, Implementation and Performance of a Content-Based Switch. In *Proc. of IEEE INFOCOM 2000*, volume 3, pages 1117–1126, Tel Aviv, Israel, Mar. 2000.

[5] P. Bhargava, S. Krishnan, and R. Panigrahy. Efficient Multicast on a Terabit Router. In *Hot Interconnects 2004*, Stanford, CA, 2004.

[6] F. Braun, J. Lockwood, and M. Waldvogel. Protocol wrappers for layered network packet processing in reconfigurable networks. *IEEE Micro*, 22(1):66–74, Jan./Feb. 2002.

[7] F. Bustamante, G. Eisenhauer, K. Schwan, and P. Widener. Efficient Wire Formats for High Performance Computing. In *Proc. of Supercomputing 2000*, Dallas, TX, Nov. 2000.

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

[9] C. Clark, W. Lee, D. Schimmel, D. Contis, M. Kone, and A. Thomas. A Hardware Platform for Network Intrusion Detection and Prevention. In *Proceedings of The 3rd Workshop on Network Processors and Applications (NP3)*, Madrid, Spain, 2004.

[10] A. Gavrilovska, S. Kumar, S. Sundaragopalan, and K. Schwan. Advanced Networking Services for Distributed Multimedia Streaming Applications. *Journal on Multimedia Tools and Applications*. to appear.

[11] A. Gavrilovska, S. Kumar, S. Sundaragopalan, and K. Schwan. Platform Overlays: Enabling In-Network Stream Processing in Large-scale Distributed Applications. In *15th Int'l Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'05)*, Skamania, WA, 2005.

[12] A. Gavrilovska, K. Schwan, and S. Kumar. The Execution of Event-Action Rules on Programmable Network Processors. In *1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure (OASIS 2004)*, Boston, MA, 2004.

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

[14] A. Gavrilovska, K. Schwan, and V. Oleson. Practical Approach for Zero Downtime in an Operational Information System. In *Proc. of 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, Vienna, Austria, July 2002.

[15] J. Guo, J. Yao, and L. Bhuyan. An Efficient Packet Scheduling Algorithm in Network Processors. In *IEEE Infocom*, 2005.

[16] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router, Aug. 2000.

[17] S. Kumar, A. Gavrilovska, K. Schwan, and S. Sundaragopalan. C-Core: Using Communication Cores for High Performance Network Services. In *Proc. of 4th Int'l Conf. on Network Computing and Applications (IEEE NCA05)*, Cambridge, MA, 2005.

[18] V. Kumar, B. F. Cooper, Z. Cai, G. Eisenhauer, and K. Schwan. Resource-Aware Distributed Stream Management using Dynamic Overlays. In *Proc. of 25th IEEE International Conference on Distributed Computing Systems (ICDCS-2005)*, Columbus, OH, 2005.

[19] V. Kumar, B. F. Cooper, and K. Schwan. Distributed Stream Management using Utility-Driven Seld-Adaptive Middleware. In *Proc. of 2nd IEEE International Conference on Autonomic Computing (ICAC 2005)*, Seattle, WA, 2005.

[20] Network for earthquake engineering simulation. www.nees.org.

[21] 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.

[22] M. Otey, R. Noronha, S. Parthasarathy, and D. K. Panda. NIC-based Intrusion Detection: A Feasibility Study. In *Proceedings of the Workshop on Data Mining for Cyber Threat Analysis*, 2002.

[23] V. Pai, A. Cox, V. Pai, and W. Zwaenepoel. A Flexible and Efficient APplication Programming Interface (API) for a Customizable Proxy Cache. In *Proc. of 4th USENIX Symposium on Internet Technologies and Systems*, Seattle, WA, 2003.

[24] 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. http://www.intel.com/design/network/casestudies/-path1.htm.

[25] B. Raman and R. Katz. An Architecture for Highly Available Wide-Area Service Composition. *Computer Communications Journal, special issue on "Recent Advances in Communication Networking"*, May 2003.

[26] 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 Symposium of Hot Interconnects*, Stanford, CA, 2003.

[27] 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.

[28] Y. Saito, B. Bershad, and H. Levy. Manageability, Availability, and Performance in Porcupine: A Highly Scalable Cluster-based Mail Service. In *Proc of 17th ACM SOSP, OS Review*, Kiawah Island Resort, SC, Dec. 1999.

[29] 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 (IPDPS '02)*, 2002.

[30] 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.

[31] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *International Conference on Compiler Construction (ICCC'02)*, Grenoble, France, Apr. 2002.

[32] M. Wolf, Z. Cai, W. Huang, and K. Schwan. Smart Pointers: Personalized Scientific Data Portals in Your Hand. In *Proc. of Supercomputing 2002*, Nov. 2002.

[33] Y. Xie, D. O'Hallaron, and M. Reiter. A Secure Distributed Search System. In *Proc. of 11th Symposium on High Performance Distributed Systems (HPDC-11)*, Edinburgh, Scotland, 2002.

[34] K. Yocum and J. Chase. Payload Caching: High-Speed Data Forwarding for Network Intermediaries. In *Proc. of USENIX Technical Conference (USENIX'01)*, Boston, Massachusetts, June 2001.

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