Utilizing Network Processors in Distributed Enterprise Environments

Paul Royal, Mitch Halpin, Ada Gavrilovska, Karsten Schwan
College of Computing
Georgia Institute of Technology
{*paul.royal, mphalpin, ada, schwan*}*@cc.gatech.edu*

Abstract

The integration of legacy systems and enterprise applications with novel communications, internet, or networking services creates the problems of mismatches, information integration, and possibly, problems from the evolution of the systems themselves. Enterprise services to solve these problems are currently implemented via commodity server hardware or Application Specific Integrated Circuits (ASICs), which suffer from either poor performance or a lack of flexibility. This paper presents an alternative to implementing these services by using Network Processors (NPs) as fast, flexible, and cost-efficient network appliances. It characterizes the hardware and software requirements of NP-based enterprise services, and presents a case study of a real-world enterprise problem. We implement a solution service to the problem on the Intel IXP2400, and present evaluation results that generalize the strengths, weaknesses, and requirements of the NP's role as a platform for enterprise service deployment.

1. Introduction

As legacy systems and enterprise networks are integrated with one another or with the internet, the simple client-server model is stretched to support the transactions that must actually take place among the (possibly very different) systems. The inherent problems in these complex transactions can generally be categorized into (a) mismatches, (b) information integration and (c) the evolution of the systems themselves.

Mismatches can be either physical, syntactic, or semantic, and correspond respectively to parts of the OSI network stack. A physical mismatch is at the lowest levels (up through the data link layer), such as when a link between two clusters is not as fast as the internal connections in the clusters themselves (e.g., [10]). A syntactic mismatch occurs in the network, transport, or session layers, such as when a legacy system cannot support modern protocols like TCP/IP (e.g., [9]). A semantic mismatch occurs in the presentation or application layers, as in cases where attributes used within one network are not understood (or worse, misinterpreted) by another network, or if the transfer of some aspects of the data is not advisable for security reasons or is subject to legal requirements, which may also differ from network to network. A violation of this last category has the potential to be very costly both to profit and public relations.

Information integration problems occur when information sent by various sources must be organized into a cohesive, single set of data for presentation at one or more destinations. This occurs routinely in the travel industry, where online services maintain or query information from a variety of corporations to create a single cohesive travel plan, potentially including flights, car rentals, hotels, or events. Even within a single source entity or network, data may need to be grouped together from a variety of systems, and the provision of this data may be governed by service-level agreements (SLAs) or regulatory requirements.

Finally, few enterprise systems are static; most undergo frequent evolution as new expectations, legal requirements, and business models arrive. As it is usually not financially viable to continually replace all affected systems, any solution should be both flexible and reconfigurable, but must also maintain compatibility with existing data sources, peers, or sinks.

In this paper we present the use of Network Processors (NPs) as efficient and cost-effective solutions to the mismatch and information integration problems previously described. In addition to providing high-level motivations towards *why* NPs are appropriate for addressing this class of problems, we present a case study of a real world problem in an enterprise environment that well-generalizes the benefits obtained by using NP-based service solutions. We implement a network service for the case study's problem domain using the Intel IXP line of

NPs, a programmable multicore architecture capable of utilizing a form of *pipeline parallelism* and compare its performance to that of the same service implemented on commodity server hardware.

The remainder of this paper is organized as follows. Section 2 provides motivation for the use of NPs in distributed enterprise environments. Section 3 uses the IXP architecture to describe the general development model of software services for network processors. A concrete implementation of an enterprise network service is presented in Section 4, and its experimental analysis appear in Section 5. Finally, Section 6 discusses future work and provides some concluding remarks.

2. Motivation

The simplest solution to the problems described in Section 1 is to modify the systems at one or more of the endpoints to translate as necessary to all the other endpoints with which they must communicate. However, this solution may not be possible or can introduce significant overhead or downtime. Therefore, a more common approach is to use *middleware* solutions (such as in [2]), to perform data translations *in-transit* between those endpoints. These transformations can be invisible in the sense that the sending host does not need to specifically target the middleware as its network destination; potentially, this transformation can happen as invisibly as a router changing the ethernet header of an IP packet when a packet travels between networks. However, despite this invisibility, middleware-level solutions implemented at the user-level on general purpose platforms come at a substantial cost due to factors such as operating system (OS) stack traversals and user/kernel switches, in addition to services competing for computational resources.

In contrast, NPs have the potential to offer a substantial improvement over commodity hardware (i.e., desktops and servers) for several reasons, primarily:

1. It is much easier for network traffic to be processed in real time by an NP. Unlike commodity hardware, traffic is not required to move through an OS or a network stack; additionally, the NP can dedicate all its time to the task at hand, rather than having to switch to other tasks (such as those of the OS).
2. NPs typically contain specialized hardware *accelerators* or instructions that are geared towards common networking operations, which can assist in the transmission or receipt of packets, as well as other areas, such as encryption or decryption.

These properties are likely to become more important, as the rate of information flow is increasing faster than the processing speeds of general purpose computing equipment [8]. Additionally, NPs can also offer a substantial improvement over commodity networking hardware (such as routers and hubs) for the following reasons:

1. NPs have the ability to filter and modify traffic based on criteria being applied to the contents (payload) of traffic (e.g., [12]), whereas most commodity equipment focuses primarily on the packet's headers.
2. NPs are programmable and can be dynamically reconfigurable; their behavior can be changed in real time based on directives from designated sources.

Additionally, many network operations lend themselves to parallel architectures such as NPs (e.g., [11]). Different cores can process separate packets or can focus on independent aspects of network processing – in the simplest case, receipt, acknowledgment, processing and retransmission of data in packets. This is explained in more detail in Section 3.

For a concretized case where an NP-based middleware solution can be advantageous, consider the problem of *merging* or joining data sent by multiple sources to a common destination. Two or more distinct pieces of information that are logically grouped are merged into one piece of outgoing data, whereas information that is not logically grouped with any other data is simply sent out as-is. As software cannot know if or when a packet might arrive, some limit must be enforced on the *merge/join window*. Once data is received, the software is unaware of whether further information will arrive that should be joined with previous data. Therefore, in some cases performance can be affected by the length of the merge/join window. For example, if the window length is increased, more memory will be required for temporary data storage, and more cycles may be required to perform *lookup* inside the data structure keeping track of currently held data. The number of streams going through the system can also significantly affect performance, as they directly affect the amount of memory required for data storage. The performance impact of both window length and the number of streams is comprehensively addressed in Section 5.

With all of this in mind, the purpose of our research is to evaluate the feasibility, utility, and limitations of using NP-based network appliances to provide functionality which addresses some of the issues raised in Section 1.

3. Network Processor Architecture

In this section we describe an architecture for structuring network service implementations on NP-based plat-

forms to address the types of mismatch, information integration, and system evolution challenges discussed in Section 1. For clarity in exposing the generalized requirements of a software architecture in this domain, we focus on the Intel IXP, a programmable, parallelizable device that operates directly at the network level. The IXP's hardware schematic shares commonalities with other NPs, such as IBM's PowerNP or Freescale's C-5 NP. For distributed enterprise environments, it well-represents the type of hardware template needed to solve various classes of problems in enterprise applications. For the sake of brevity, the overview provided is primarily conceptual; a reader with further interest in hardware particulars is directed to [1].

3.1. Hardware Architecture Support

At a high level, the IXP is a multicore hardware component that consists of one or more clusters of four processors (called *microengines*) and an XScale core. The XScale core is used mostly for administrative tasks and assumes a limited role (e.g., initializations) in fulfilling a service. The overwhelming majority of processing occurs in the *fast path*, which consists of one or more clusters of microengines that communicate with each other. Intercommunication is made fast by selection of appropriate types of memory from a complex hierarchy.

With this hardware template, the IXP microengines within a cluster, or even across clusters, can be configured like an assembly line, where each microengine represents a discrete stage in the overall process. A single microengine draws from the output of the previous stage, performs some processing, and passes the results to the next stage. Provided that work is partitioned with care, configuring a cluster in this fashion yields good performance by exploiting pipeline parallelism.

3.2. Software Architecture Support

Figure 1 depicts a common starting configuration for service implementations that use a single cluster. Each white block represents a microengine, with the first and last stages of the cluster selected for the general receipt and transmission of packets. The receive stage pulls data directly from the network and writes it into DRAM, while the transmit stage sends out packets as directed by the third microengine. The white ring between two adjacent microengines - a *scratch ring* - represents a small amount of fast memory used for their intercommunication. Appropriately named, the scratch ring serves as a circular queue for passing the output of one stage to the next. Its entries often comprise handles to received packets, the intermediate results of processing, or packets to send.

The other two microengines are unoccupied; they represent blank slates that can be programmed to do service-specific processing. In addition, the service they perform can be configured via an API their programming supports. In the realm of NPs, this API may take the form of special *configuration packets* that, when sent to the IXP, direct the service to modify its operating state according to the instructions provided. The degree to which a service is reconfigurable represents a potential tradeoff between flexibility and performance, and may require that additional care be given to the particulars of reconfiguration.

In the simplest of cases (i.e., for services that require no configuration or undergo reconfiguration infrequently), service code can simply be modified and reloaded [6]. More sophisticated configurability can be enabled by explicitly programming such support into the microengines, although this functionality is best conceptualized as a layer that sits on top of the underlying service. For moderately reconfigurable service classes, where the format of the data is static but the process criteria of the service (such as the fields to read) may need to be changed, a small amount of fast memory can be allocated for configuration data. In the most extreme cases where the domain context of a service may change completely, the service code comprises an engine that must be explicitly configured; designing configuration structures and efficiently working with configuration data become non-trivial problems. In either case, the *structure of communication* directing reconfiguration must itself be both flexible and efficient. This need can be met by employing lightweight binary formats such as Portable Binary I/O [2] to represent structured data, which permits us to describe the layout of fields in application-level data and, through the use of offsets, to efficiently describe data accesses and manipulations [5].

In summary, the software architecture must provide for the deployment of new services onto one (or more) of the available pipeline contexts, basic functionality for data receipt and transmission, storage/buffering and lookup/retrieval, and dynamic reconfiguration. Also, the IXP's utility is not limited to that of a vehicle for transforming received packets. That is, a processing stage may explicitly create new packets, or store the contents of received packets for a future purpose. For example, in [3], an IXP1200-based implementation of the Intrusion Detection System Snort generates a multicast alert if a packet's contents are deemed to be malicious. In addition, Section 4 depicts an IXP-based service implementation that stores received events in a data structure so that

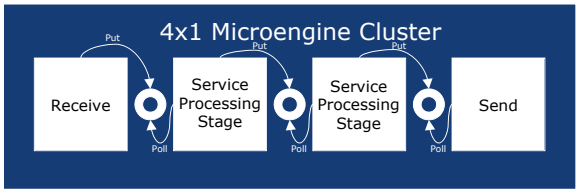

Figure 1. General IXP Software layout.

they may be matched with future arrivals.

4. NP-Based Service Implementation

Next we present a concrete example of a problem in a distributed, heterogeneous enterprise environment that can benefit from the use of a NP, and provide implementation details for its IXP-based solution.

4.1. Problem Background

The TPF OS, as used by Delta Airlines' Operational Information System (OIS) [9], provides an important array of information and transaction services such as passenger check-in, flight updates, and ticket pricing information. As a legacy system, however, it lacks components (i.e., until 1999, a TCP stack) important to interacting with modern-day operational infrastructures (e.g., web services). Moreover, the TPF has become computationally limited compared to modern hardware simply as a function of its age. Despite these deficiencies, the TPF represents a significant financial investment and encompasses key components of company operations; completely replacing the system is not considered an option. Delta has chosen instead to supplement the TPF's functionality by using modern-day hardware (e.g., web servers). As an example, in 2001 it was reported that Delta would begin using a server farm to handle its interface for fare searching instead of Deltamatic, which directly accessed the company's TPF [4]. Given that most operations and transaction information must still go through the TPF, there exists a clear need for a *bridge* that, at its most basic level, is capable of solving data format and protocol semantics mismatches.

Furthermore, in fulfilling client and/or external party's requests, responses may be generated by the TPF, the server farm, or additional value-added services, such as DoubleClick services used to generate customized marketing material. The information content of these responses must often be organized into a single presentation, which motivates the need for the bridge to perform information integration as well. Finally, regulations governing operation may change, and standards within the existing infrastructure may evolve. For an example specific to Delta, consider the case where an SLA changes to require a different number of price matches per request; in this scenario, the bridge must (perhaps through reconfiguration) handle an evolution of the system itself.

4.2. Current Solution

Currently, communication between the TPF and modern hardware is handled by custom Application Specific Integrated Circuits (ASICs). Concisely, Delta has created a bridge by having ASICs manufactured that specifically address its service needs. Although standard commodity hardware can be programmed with the appropriate logic to serve the same function, it is not sufficiently fast. Therefore, by placing service processing logic directly in hardware, performance requirements are maintained.

The use of ASICs, while diligently addressing performance needs, impacts both cost and flexibility. With regards to cost, ASICs are expensive because they are actual hardware that must be custom made to function for a set of contexts specific to a service. In addition, having software turned into a circuit inherently constrains adaptability. As an example, the implementation of a new service may indirectly require additional functionality from the bridge, meaning that a new ASIC will have to be made. Similarly, if the policies of an existing service are updated or altered, a new ASIC that accommodates those changes may need to be fabricated to replace one that already exists.

4.3. NP Alternative

The challenges faced by Delta are by no means unique; the need to implement fast yet flexible middleware services is a common theme for heterogeneous, distributed enterprise environments. Given that service solutions implemented on commodity hardware are flexible but slow, while ASIC-based implementations are fast but expensive and rigid, neither represents an exceptionally desirable approach. It is at this juncture that the utility of NP-based network appliances becomes exceptionally visible. An NP has the potential to capture the best parts of both commodity hardware and ASICs, or at the very least minimize each of their respective drawbacks.

To evaluate the viability of the NP alternative in efficiently arbitrating communication between the TPF and modern equipment, we implement a narrow but important subset of service-specific data transformations on the Intel IXP2400 NP. Called *temporal joins* [7], these transformations represent a form of data merging whereby incoming events (packets) are buffered, categorized over

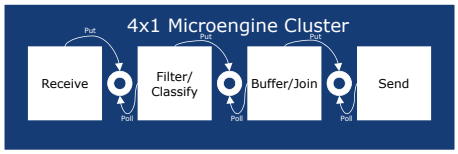

Figure 2. IXP Software layout for the Delta Event Merfer.

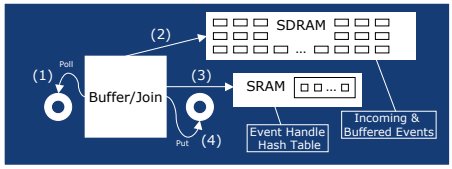

Figure 3. Process Overview of the Buffer/Join Stage.

some given criteria and when appropriate, joined with previously categorized events and sent to the appropriate destination.
Figure 2 displays the partition of work for the merging service. The stages can be described as follows:

- *Receive*: Detect all network traffic; validate bits as a legitimate frame, verify integrity of frame contents.
- *Filter/Classify*: Verify the IXP as the destination of a received frame; validate it as an event. Conceptually, at this stage the view of what was received is transitioned from being a frame to being an event.
- *Buffer/Join*: Categorize and further classify the event. If necessary, perform a join with a previously received event and queue the result to the Send stage; otherwise buffer the event.
- *Send*: Transmit joined events that have been queued to be sent.

The receive and send stages consist of generic components provided with Intel's IXA Software Development Kit. Constructing the filter/classify stage required simple service-specific modifications to another generic component. The buffer/join stage required substantive development and represents the most interesting aspect of fulfilling the event merging service. A visual overview detailing how this stage operates is shown in Figure 3.

In (1), an event (packet) handle placed by the Filter/Classify stage is sourced from the scratch ring. Using this handle, in (2) a pair of accesses is then made to SDRAM: one to acquire the absolute offset of the desired field within this event, and another to read the field's contents into DRAM transfer registers (which can then be addressed directly by the microengine). The reason why a read must be performed to obtain the absolute offset of a desired field within a particular event is because the event data format uses a form of relative addressing. Once read, the field's contents are hashed to a numerical value. This value is used to *query* a hash table stored in an available SRAM channel, although specialized hardware lookup structures (e.g., Content Addressable Memories, or CAMs) could be used enhance the performance of the query process (if available). Hash table entries themselves contain event handles and are indexed by the hash value of the field's contents. If there is no corresponding event to merge, an entry is created for the event handle and it is placed inside the hash table, as is shown in (3). If an existing entry is found containing an event handle to be merged, however, its contents are combined with the current event, a handle to the result is placed in the scratch ring polled by the Send stage, and the hash table entry of the event that matched is freed. This process is represented by (4).

5. Testing and Evaluation

In this section we present the results of testing the NP-based service implementation described in Section 4. Our analysis uses the results to, where possible, generalize the viability of using NPs for enterprise services. In addition to explicitly detailing the practical benefits and limitations of the NP-based model, we correlate performance measurements relevant to broader aspects (such as reconfigurability) in order to comment on their feasibility.

5.1. Testing Preliminaries

Although a direct comparison with the performance of the ASICs would be ideal, the sensitive nature of this data made it unavailable. Therefore, in order to quantitatively evaluate the performance of the IXP, we built a standard C programming language implementation of the event merging service. A 1.7Ghz Pentium 4 Xeon server running Linux 2.4.18 (representing a flexible commodity hardware-based service) was used to run this implementation. The source data used for event simulation is an actual Delta event stream consisting of 1.5 hours of flight status events. Represented in binary, most events are about 600 bytes in size. As mentioned in Section 4.3, although some fields within an event are accessible by an absolute offset, most use relative addressing.

Conceptually, the simulation environment represents three distinct components (see Figure 4):

- *Event Sender*: Read the Delta event stream, appropriately partition its contents into events and send them.

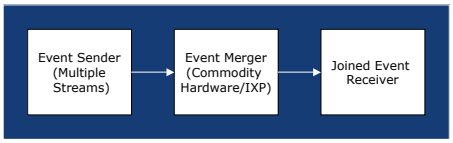

Figure 4. Component overview of the simulation environment.

- *Event Merger*: Receive, classify and categorize incoming events. Buffer or merge events as necessary.
- *Joined Event Receiver*: Receive and verify merged events.

Note that although the Event Sender is abstractly represented as one block, in the actual benchmark multiple machines (corresponding to multiple streams) were employed to simultaneously send events to the Event Merger component. This configuration, in addition to providing a more accurate representation of the operating environment, was used to prevent network throughput from becoming a pronounced bottleneck. Additionally, in order to minimize potential variability arising from overhead inherent in higher level protocols such as TCP, all network communications in the process were done at the data link layer.

5.2. Performance Measurements

In order to measure the performance of each of the implementations under various scenarios, we employ a metric called *complete time*; its definition is explained as follows. In a sequence of N incoming events, some fraction of those events will be merged, resulting in T merged events being sent. In the Joined Event Receiver the time is taken after receipt of the first merged event. After receipt of the T^{th} merged event, the time is again taken and their difference calculated; this number comprises the *complete time*.

Of the variety of fields contained within each event, *airport code* was selected as merge criteria for evaluating baseline performance. The reason it was chosen is that in the actual system, events likely to be merged (such as updates to a particular flight) will have a common arrival destination. In addition, for analyzing the performance impact of changing properties such as merge window size, airport code represents a field with sufficient median qualities.

5.3. Evaluation

5.3.1 Baseline Performance. Figure 5 shows the results of testing the event merging service with two streams (i.e., where two sources are configured to simultaneously send events to the service) and airport code as the merge criteria. The results indicate that the complete time for the service implementation on the IXP is approximately 50% lower than the commodity hardware implementation. Thus, at least for the operating context of this domain, an NP-based service indeed surpasses the performance of a reasonably equipped commodity server.

In analyzing why the NP-based implementation exhibits superior performance, observations on memory usage and control explain contributing factors. One related point that favors the NP-service implementation is that is operates directly in the network path. On the IXP, the receive stage places incoming packets (events) directly into DRAM accessible to any of the microengines; when buffered, events are not copied to another location, but simply kept in place. In the commodity hardware implementation, however, network data must propagate through the host OS before it can be processed. The actions involved in propagation (such as additional copying) can substantively add to the time required for a packet received by the network card to become available to the merging service for processing.

Another advantage has to do with placement of the central data structure used to facilitate the merging service. On the commodity hardware implementation, the event hash table must be stored in main memory, although some table entries for buffered events will appear in the processor's cache. In contrast, the NP-based implementation requires explicit programming of almost all memory operations, but this control also enables placement of the event handle hash table in an available SRAM channel (instead of DRAM). Thus, queries to and traversal of the hash table in the NP service are optimally fast, yielding a savings in processing time.

5.3.2 Additional Streams. To evaluate the performance impact of increasing the number of streams the NP-based service must handle, experiments using airport code as the merge criteria were conducted with three and four streams. To overcome a testing environment network bottleneck, for this evaluation performance measurements were obtained by running the service in a cycle-accurate IXP2400 simulator (available in Intel's IXA SDK 4.0). An experiment with two streams was also performed to show that simulator results indeed closely match endpoint-to-endpoint performance measurements obtained by using actual hardware.

Figure 6 shows the results of testing the service with two, three, and four streams using the simulator. For three streams, the hardware resources of the IXP are not yet ex-

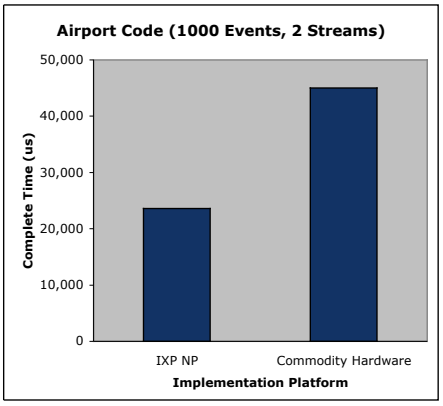

Figure 5. Baseline Performance.

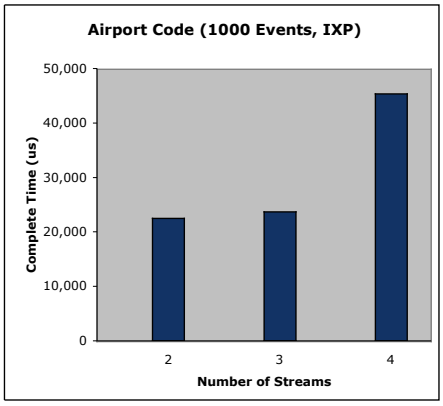

Figure 6. Performance vs. Streams.

hausted, and the service's performance scales well, as the complete time is only 5% higher as compared to merging over two streams. When four streams are used, however, the resources of the IXP available to fulfilling the service are exceeded, and performance drops significantly. In addition to available fast memory for hash table entries becoming saturated, the synchrony between related incoming events decreases. That is, the amount of time a buffered event resides on the IXP before it is merged increases. As a consequence, comparatively more (and expensive) memory operations must be performed to traverse hash table entry chains in order to find a potential match, which substantively impacts overall processing time.

The limitations exposed from evaluating the performance of the IXP with additional streams highlights a point crucial to the effective use of NPs in enterprise domains. It is that, due to the limited resources available to an NP, care must be given to selecting the appropriate hardware configuration for a particular problem context. More broadly, in constructing an NP-based enterprise service, there exists a clear need to well-understand the operating domain and the resources it needs. As a concise example for the merging service domain, the IXP2350 has twice the amount of SRAM and faster microengines than the IXP2400; both of these advantages could assist in combating resource exhaustion associated with managing four event streams. Furthermore, for domain specific NP appliances, traditional SRAM or DRAM components may be replaced by other types of memories (such as CAMs) for more efficient lookup. In addition, the above results may be used by NP appliance designers to include or enhance existing on-chip capabilities to provide better support for these classes of functionality.

5.3.3 Merge Window Length. Finally, we evaluate the performance impact of varying the length of the merge/join window. For long windows (where the time between two events to be merged is long), the performance of the IXP is already known, as managing four streams represents a case where window length is artificially inflated. To explain, separate streams decrease event synchrony, which effectively lengthens the merge window. Analogous consequences follow, in which buffered events reside on the IXP longer, available memory resources are stressed, and processing time dramatically increases.

For measuring the impact of short windows, the event merging service was modified to use the *airline code* field as merge criteria. As the event stream is from Delta, the overwhelming majority of events contain `DL` for the value of this field. The use of airline code therefore represents testing the merging service with a short merge/join window, as almost every two incoming events will be merged. For comparison, airport code was selected to represent the case of average window length.

Figure 7 shows a side-by-side performance comparison of the merging service with two streams and merge criteria representing average and short window lengths. The complete time for the short-length merge window is slightly less than the complete time for the average-length merge window, which provides further evidence that memory operations (and not computation) dominate total processing time. More importantly, this observation implies that *local* computation in a microengine exists as an otherwise untapped resource available for use.

The availability of unused local processing in a microengine speaks promisingly toward dynamic reconfiguration, an important aspect of what an NP-based enterprise service should provide. In particular, moderately reconfigurable services, whose configuration data is

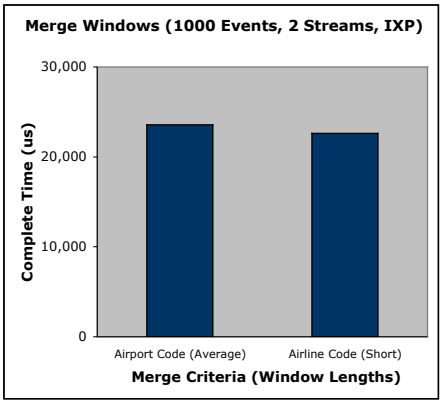

Figure 7. Performance vs. Window Length

small enough to fit inside the local memory of a microengine, can process that data at nearly zero cost. Therefore, moderately reconfigurable services are likely to incur no performance penalty over otherwise identical services that must be configured by service code modification and reloading; we will examine these issues further in future work.

6. Conclusion and Future Work

In this paper we have presented an examination of using network processors as platforms for service solutions which address the challenges of mismatches, information integration, and system evolution in distributed enterprise environments. By describing the differences that separate NPs from general purpose hardware, we presented a motivation for their use as network appliances for select classes of services. We generalized the hardware and software requirements that NP-based appliances must provide by focusing on the Intel IXP, a popular network processor. To concretize the NP's role in enterprise domains, we presented a case study of a real-world mismatch and information integration problem, whose solution was implemented on the Intel IXP2400. In evaluating the performance of the NP-based enterprise service implementation, we detailed the strengths and weaknesses of using the IXP to generalize about the viability of the NP model as an enterprise service. By analyzing results in the context of the IXP's hardware, we outlined general requirements and considerations for NP-based services important to their effective use.

In future work, we will evaluate the performance differences arising from the use of other NP configurations (such as the IXP2350). Further analysis will provide a basis for a more precise understanding of how various characteristics of NP hardware affect performance in different enterprise service domains. In addition, based on the observations made in Section 5.3.3, we plan to implement various classes of dynamic reconfiguration into enterprise services. Besides researching efficient ways to manage fully reconfigurable services, we intend to explore related issues important to general production environments, such as authentication-based reconfiguration.

References

[1] IXP2XXX Product Line of Network Processors. `http://www.intel.com/design/network/products/npfamily/ixp2xxx.htm`.

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

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

[4] Computerworld. Delta to change core technology. `http://www.computerworld.com/softwaretopics/software/appdev/story/0,10801,59639,00.html`.

[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] S. Kumar, A. Gavrilovska, K. Schwan, and S. Sundaragopalan. C-CORE: Using Communication Cores for High Performance Network Services. In *Proc. of Network Computing and Applications (NCA)*, 2005.

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

[8] P. Mehra. Information, The Final Frontier. In *High Performance Interconnects for Distributed Computing (HPIDC)*, 2005.

[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] R. Recio. Server I/O Networks Past, Present, and Future. In *Proc. of the Special Interest Group on Data Communications (SIGCOMM)*, Aug. 2003.

[11] J. Yao, Y. Luo, L. Bhuyan, and R. Iyer. Optimal Network Processor Topologies for Efficient Packet Processing. In *IEEE Globecom*, 2005.

[12] L. Zhao, Y. Luo, L. Bhuyan, and R. Iyer. SpliceNP: a TCP splicer using a network processor. In *Proc. of the Symposium on Architecture for Networking and Communications Systems (ANCS)*, Oct. 2005.