The Execution of Event-Action Rules on Programmable Network Processors

Ada Gavrilovska, Sanjay Kumar, Karsten Schwan
Center for Experimental Research in Computer Systems (CERCS)
Georgia Institute of Technology
Atlanta, Georgia, 30332
{*ada, ksanjay, schwan*}*@cc.gatech.edu*

Abstract

This paper evaluates the ability of programmable network processors (NPs) to perform application-specific processing that is structured as sets of interdependent event-action rules sharing joint state. Our intent is twofold: (1) to assess the ability of NPs to deal with the complex application-specific code and state maintained by event-action rules like those used in business processes also termed (termed business rules), and (2) to create system solutions that permit developers to dynamically map such application-level service code to appropriate sites in target distributed platforms comprised of hosts, network processors, and embedded server systems.

A specific outcome of our work presented in this paper is the creation of a simple, efficient dynamically reconfigurable rule engine for a network processor, able to execute rules at the Gigabit speeds required by the network links attached to it. Business rules like those found in the Operational Information Systems used by companies like Delta Air Lines are used to demonstrate rule engine capabilities and overheads. A second outcome of our work is its demonstration of the flexibility and ease of reconfiguration associated with a network processor-resident rule engine, where rules can be added and removed whenever appropriate (i.e., hotswapping) without compromising the processor's ability toLarge-Scale Network Simulation: How Big? How Fast?

maintain high performance for its ongoing processing tasks.

1. Introduction

Modern commercial distributed applications require the ability (1) to dynamically place service instances at nodes in the distributed infrastructure, at both data sources and intermediate nodes on the application-level data path, so as to increase the shared resource utilization, and (2) to dynamically configure these services to match the current application requirements (i.e., QoS, fault-tolerance), current end-user interests (e.g., in specific data subsets), or current operating conditions (e.g., CPU loads/workloads and networking resources). Changes in such runtime conditions and requirements are difficult to predict, therefore applications require the ability to dynamically react, customize their processing actions, and improve operation. Due to the computation-centric notions of modern machines, application-level implementations of these services are problematic for applications requiring high data transfer rates, for reasons that include the inability of modern architectures to efficiently execute computations with communication. Conversely, network-level implementations of services are limited due to the network's inability to interpret application-level data or execute application-level operations on such data.

This research explores the extent to which programmable network processors, an emergent class of network devices, can be used in conjunction with standard host nodes, to form enhanced computational platforms. The intent is to deliver improved performance and more efficient and flexible service implementations to commercial and scientific distributed applications. We introduce the use of hosts with attached network processors, host-ANP platforms, enable the dynamic configuration of the data path through the host-ANP nodes, and the dynamic creation, deployment, and reconfiguration of the application-level processing applied along this path.

One of the core components of modern commercial applications like web services, operational information systems, and event notification systems [5, 14] is their ability to perform efficient actions on data events exchanged across multiple internal subsystems and/or external 'client' systems or data sources. For instance, the online check-in service supported by operational information systems like the one used by Delta Air Lines [5] processes incoming events (e.g., passenger check-in) to evaluate different check-in options based on specific flights or airports (e.g., international vs. domestic flights), the passenger class (e.g., preferred passengers), baggage information, time to departure, etc. The outcomes of option evaluations differ depending on current state and the combinations of application-level parameters being applied.

The event-based action execution implied by rule processing like passenger check-in is a key determinant of the end-to-end performance of modern business applications. Performance depends on both the ability to efficiently select and execute individual rules and to access and manipulate the state associated with such executions, where state consists of the contents of the business events (local state of a rule) being manipulated, flow-level state, and global state shared across sets of interacting rules. Event-action processing also results in events being consumed, forwarded to subsequently applied rules, or in new events being created. In the airline 'check-in' example, an event requesting online check-in on international flights is 'consumed' and no other application-level processing is applied to it (except for generating a client notification that check-in is not allowed). In contrast, if check-in is allowed, the requesting event is forwarded to the OIS's subsystem (another rule engine) that implements the actual check-in procedures.

The actions executed and conditions checked in procedures like 'passenger check-in' capture the basic business logic of a company's operation, reflecting company rules and regulations as well as higher level policies (e.g., never allow online check-in for international flights). Actions are implemented by so called *rule engines* and are commonly referred to as *business rules*. While past work in Artificial Intelligence has done extensive research on the use of rules and rule engines to implement complex cognitive or reasoning processes, the event-action structures used in applications like publish/subscribe [14] and/or in the business rule engines used by companies are more concerned with making it easy to add and/or remove rules or rule sets, in order to permit organizations to rapidly adapt their rules to new customer requirements, business environments, or regulatory changes. In publish/subscribe, rule engines apply SQL-like actions on incoming events [14], or they execute actions formally described by first order predicate logic. More complex functionality is offered in software products like ILOG and Blaze [7, 2], which address usability needs with convenient user interfaces, embedded procedures for rule optimization, and similar functionality.

The rule engines addressed by our work are focused on Operational Information Systems (OISs), where scalability is a dominant concern, both with respect to sustained data rates and number of clients. As with the rule engines used by companies like Delta Air Lines or Worldspan, they are focused on high performance (i.e., high data rates) and low, predictable delays in rule processing. These engines are typically comprised of rules coded in a standard language like C++, consuming and/or producing well-structured events, like those in TPF [6], and inspecting and updating state represented by binary, in-core data structures. Despite their relative 'simplicity', however, these coded rule engines share, with software products like ILOG, the need to support dynamic rule changes and updates, thereby making it difficult to accelerate rule engine execution with custom ASICs or FPGAs.

This paper investigates the extent to which programmable network processors (NPs) can be used to attain efficient and flexible implementations of dynamically configurable rule (i.e., event-action) engines. NPs have hardware optimized for high-performance data transfers, at rates exceeding Gbps, and their programmability makes them attractive vehicles for deploying new functionality 'into' the network infrastructure. We leverage the fact that previous work has already shown that network-centric functionality can be efficiently implemented with the excess of cycles ('headroom') available on the packets' fast path between the NP's incoming and outgoing ports [13, 9]. Examples include instrusion detection, firewalls, service differentiation, software routing, etc. [13, 10, 9, 1]. Our own previous work [4] has created the ANP (Attached Network Processor)-resident abstraction of *stream handlers* to efficiently carry out a variety of application-specific processing tasks on NPs by judiciously 'splitting' application functionality across the combined host-NP pairs, thereby improving application performance compared to host-resident solutions. User application performance can be further improved by judiciously 'splitting' application functionality across the combined host-NP pairs. Improvements are derived both from the fact that the ANP-resident processing removes load from the host' CPU and more importantly, I/O infrastructure, and because ANPs are well-optimized to carry out tasks like traffic replication, data forwarding, protocol processing, and 'lightweight' operations on packet contents.

The paper extends our previous work by designing a flexible, gigabit-capable, NP-resident rule engine. *Rule handlers* implement application-level actions on all, or a subset of the application's data. The rule engine itself supports dynamic rule installation and removal (i.e., hotswapping), and it offers efficient functions for dynamic rule selection in response to the receipt of business events. State is maintained in data structures residing 'close to' the rule handlers that use it, leveraging the network processor's internal, hierarchical memory architecture. Our design is based on Intel's IXP2400 network processor [8], and it can sustain cumulative traffic rates close to 3Gpbs. Our intent is to evaluate this NP's ability to execute condition-based rules and analyze the performance tradeoffs with respect to event consumption, rule complexity, and state accesses required for rule execution. We also demonstrate the flexibility and performance of dynamically updating the event-action rule sets and rule flows resident on the NP, using hot swapping of microcode on the IXP's microengines.

Experimental results demonstrate the viability of efficient NP-resident event-action processing on the IXP. We

show that the NP has sufficient headroom for running rules of different complexity and even for chaining multiple rules applied to an incoming event. The rule execution times depend not only on the complexity of the rules themselves, but also on the location and amounts of state being accessed. Finally, dynamic rule deployment (i.e., hot swapping) can be performed with low overheads and so that ongoing rule flows are not perturbed.

The remainder of this paper first presents the desing of an NP-based rules engine. Next, we describe the NP firmware architecture and discuss certain design tradeoffs for dynamically configurabale rule engine implementation. This is followed by a report of performance measurements, in which we estimate the IXP2400's headroom available for the various processing tasks implied by business rule execution, and evaluate the cost of state accesses required by rules. Conclusions and future work appear at the end.

2. Rules and Rules Handlers

Business rules are the set of conditional checks performed for each of the myriad of data events exchanged in modern commercial systems, in order to determine the actions that must be applied to the event and the application subsystems involved in that process. Multiple rules can be chained to form *ruleflows*. In the Delta OIS, an example of a ruleflow is the following series of decisions performed on a single data event requesting passengers P's check-in on flight F:

1. 'Is F open for check-in?', and
2. 'Is P allowed to check-in on F?'.

In addition, as a result of business rules, events can be 'consumed', - inelligible to further advance in the company's infrastructure, or they can trigger state updates solely. The decision making process, i.e. the application of a particular business rule, requires access to the event's content and type information (local state), as well as flow- and system-level (global) state. These parameters are matched against sets of conditions, stored in *decision tables*. For specific event types, sample entries in the decision tables consist of pairs of target_value and corresponding action, where basic actions include consume, modify, and modify and forward. We note that the set of all valid conditions in a specific application can be quite large, extending to gigabytes of data jointly with the state constructed from repeated rule application. We do not suggest to maintain such large data sets on NPs. Instead, we rely on application developers to identify the state-constrained rule subsets suitable for mapping to NPs, where each such subsystem requires access to only a small subset of these conditions, relevant for the parameters and actions that are being evaluated. Suitable subsets include rules that perform application-level data routing, i.e., that determine the subsystems to which data events should be routed, actions that pre-screen events to determine their validity or well-structuredness (e.g., to avoid certain failure modes [5]), and actions that simplify events in order to reduce their processing overheads (e.g., eliminating event fields not needed by certain subsystems).

In order to analyze the extent to which a network processor's built-in parallelism and multi-level memory hierarchy can be exploited to efficiently perform event-action processing, a simple rule engine is used to evaluate the NP's basic capabilities for rule processing, with future work concerning the detailed design and implementation of decision tables and/or efficient matching algorithms. The rule engine's design is based on the SPLITS software architecture, developed by our group for the first generation IXP1200 network processors [3]. SPLITS (*Software architecture for Programmable LIghtweighT Stream handling*) enables applications to dynamically create, deploy, and configure application-level computational units, termed stream handlers, at well-defined *activation points* along the stream data path on host-ANP pairs. On each NP, dedicated Receive (Rx) and Transmit (Tx) contexts receive and transmit data events and implement basic protocol-related functionality. The application-level data path 'through' each SPLITS node can be configured to span multiple ANP and/or host contexts. Simple application-level rules are represented by single *rule handlers*, which are then composed into rule flows. Each rule flow consists of multiple processing contexts 'along' the data path 'through' the NP-based rule engine. (see Figure 1). These contexts coincide with the ANP hardware-supported contexts (i.e., microengines and threads), and as a result, each ruleflow is implemented as a multi-stage execution pipeline.

Event processing proceeds as follows. After an event is processed by a rule handler, it can (1) cause state updates, (2) be consumed, (3) an event (the one used by the rule or a newly produced one) can be forwarded to other application components, and/or (4) it can trigger the next rule in the ruleflow, executed at the next pipeline stage. Data events are passed from one pipeline stage to the next directly (i.e., by passing on a data buffer handle directly to another context) or through controlled access to shared ring-buffers, the latter being used when the different processing blocks of the pipeline are programmed separately. Classification is performed based on application-level information contained in each data event (e.g., Delta or FAA event type), so as to dispatch it to the appropriate rule handler, or to forward it to the host or other application subsystems.

Rule handlers are implemented by application programmers, whereas we provide the ability to dynamically embed and configure rule handlers in the data processing pipeline on NPs, to access shared global and local handler state, as well as memory regions for storing decision tables, and to create and manipulate ruleflows in application-specific

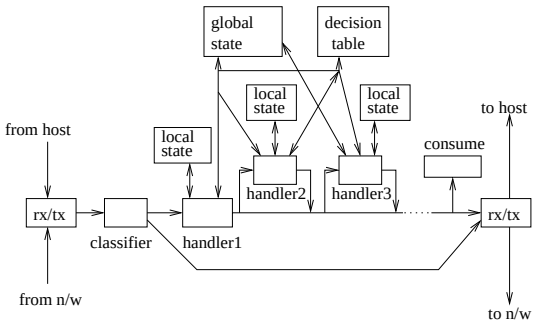

Figure 1. Rule Engine. Rule handlers have access to local state, shared global state, and to decision tables. Events are 'consumed' or forwarded to the attached host or the network.

manner.

A benefit of an NP implementation of a rule engine is that in addition to carrying out decision making, actions associated with each rule and implemented as NP-resident stream handlers can perform certain simple processing steps. An example is event re-formatting for simplification prior to forwarding it to a different subsystem. For instance, if the rule handler decides that the flight event is to be shared with an external caterer system, because a gate change occured for instance, the event would have to be reformated only to include those fields which the company permits its external clients to access. We have previously shown that such format translations such NP-resident application-level processing can improve the performance of end user applications [4]. Improvements are due to offloading the host's CPU and I/O infrastructure and due to certain hardware accelerators present in the NP (e.g, hardware support for queuing, CRC calculation, asynchronous memory read/write operations).

3. Baseline Architecure

Our work evaluates the ability of performing the condition-based execution of business rules on standard network processors, exemplified by the Intel IXP2400 programmable NP. We have implemented a baseline network interface on an IXP2400-based PCI card as part of a larger effort that is exploring the idea of attached network processors that enhance the capabilities of the hosts to which they are attached. The IXP2400 chip [8] used in that research includes 8 8-way multithreaded microengines for data movement and processing, local SRAM and DRAM controllers and an integral PCI interface with three DMA channels. The Radisys ENP2611 board [12] on which the IXP2400 resides includes a 600MHz IXP2400, 256MB DRAM, 8MB SRAM, a POS-PHY Level 3 FPGA which connects to 3 Gigabit interfaces and a PCI interface. An XScale core runs Linux and is primarily used for initialization, management and debugging. Our design uses host-side drivers with NP firmware to implement the host-ANP interface. The IXP2400 is attached to hosts running standard Linux kernels over a PCI interface. Data is delivered to and from the host-resident application components through the ANP's network interfaces.

The rule engine's processing pipeline from Figure 1 can be fully contained on the ANP, or it can be 'split' across host-ANP boundaries. The results presented in this paper deal with ANP-resident rule handlers. The analysis of 'split' rule engines will be addressed by our future work. The software architecture developed in our work differs from other network interfaces. First, many optimizations found in other network interfaces are based on the assumption that the network driver will be working on the header part of the packet. Our goal is to enable access and processing on entire application-level messages. Second, in order to enable the dynamic reconfiguration of the ANP runtime, we provide methods to efficiently and dynamically insert application-specific processing code into the network driver. The remainder of this section discusses the main features of our architecture.

Host-ANP Communication. We model the Host-ANP interface after the host-IXP1200 PCI-based interface presented in [11]. The IXP2400 and host communicate through two shared circular buffers, one in each direction. For each direction, three threads from a single IXP microengine are involved in the coordination and synchronization of the DMA-based data transfers. Messages are kept contiguous despite the circular buffers, and the firmware also includes optimizations that result in improved PCI utilization.

ANP-MAC Communication. The firmware uses two microengines (one each for transmit and receive) for the high speed data transfers between DRAM and the underlying physical interface. Two threads per port, all on one microengine, are used to receive packets from the on-chip media switch fabric (MSF), which are assembled into messages. Similarly, two threads per port, from one microengine, are used to transmit messages from DRAM to the MAC device. A list of buffers is preallocated to reduce buffer allocation overheads. Once assembled, a message is enqueued for further processing by other processing blocks. Similarly, transmit threads dequeue messages from the corresponding transmit queue, fragment them and send to MSF, which delivers the fragments on the wire.

Resources. Our baseline design uses only 3 microengines for basic data transfer actions. This leaves 5 microengines free for business-rule processing and for the execution of rule handlers. The shared ring buffers used to pass data between the ruleflow stages are implemented in scratch memory. Dedicating the low-latency scratch memory for this task is necessary in order to meet the high

performance needs of OIS processing and to exploit the hardware-supported queue management functionality resident on the IXP. The firmware also leaves substantial memory in SRAM (7 MB) and DRAM (200 MB) for storing application-level state, as well as local memory private to each microengine. Size limitations motivate the joint use of the combined memory resources. We provision state information to be stored in local memory and/or SRAM and decision tables to be stored in SRAM or DRAM. In Section 4, we evaluate the tradeoffs of accessing each of these memories. Our future work will include the development of a state/memory management utility, in order to dynamically match rule performance requirements to available storage resources. We will also analyze the efficient partition, sharing, and access to decision tables. Currently, our design assumes a static, hash-table based implementation of decision tables. Our future work will incorporate results about the dynamic addition and deletion of rules.

Dynamic Reconfiguration. Our architecture currently supports the dynamic reconfiguration of rule handlers as well as ruleflows. This is enabled through the exchange of control messages between the host and the ANP-resident contexts via shared mailboxes. On the IXP, it involves a designated control thread to perform mailbox polling. Control messages can be used to select and activate a new rule handler from a set of pre-existing ones, as well as to pass new parameters to currently active rules. Parameters can be used to enable application-level modifications in the rule processing, and to configure ruleflows, i.e. map the processing decision reached at one stage in the rules engine to the hardware context, i.e. microengine, currently executing the next rule in the chain.

To enable the dynamic deployment of new rules or of new implementations of existing ones, our architecture supports dynamic hot-swapping of pipeline contexts with negligible downtime. The implementation of hot-swapping on the IXP NPs requires that we reserve one of the available microengines and maintain it idle. The new rule handler is loaded into this microengine and is activated after the appropriate ruleflow reconfigurations are made (i.e, to ensure that the new context is now part of the ruleflow through the engine). The newly deployed handler can replace a currently active one, which also requires that local state information (if any) is correctly passed from the old handler before its execution is stopped. As a result, total actual downtime for handler processing is simply the sum of the times involved in stopping one microengine and starting another. Measurements on IXP1200s showed that this can be done in a few (28-30) microseconds, rendering this downtime practically unnoticeable. Note that hot-swapping reduces the available resources for business rule processing on the IXP, but that it enables us to increase the options for dynamic reconfigurability, required in many commercial applications.

packet size (B)	128	256	576	1024	1500
Throughput (Gbps)	2.92	2.64	2.81	2.83	2.84
Latency (usec)	3.58	4.16	5.16	7.00	8.33

Table 1. Throughput and latency calculation for various packet sizes with the baseline network driver.

4. Evaluation

The experiments reported in this section use the Intel Workbench Simulator SDK3.1, claimed to be cycle-accurate within 2%, and a cluster of 8 dual-processor nodes interconnected via a Ciscoxxx switch, each with a 2400-based Radysis ENP2611 board attached via the PCI bus. The data streams used in these experiments include streams generated from the Workbench' Traffic Simulator, as well as binary representations of an original 30MB XML data stream acquired from a large corporation. All data sinks and sources in the experimental setup execute the same protocol used for efficient message fragmentation and reassembly as the one used on the ANP.

The first set of experiments compares the sustained throughput levels and incurred latencies for varying data sizes for the baseline architecture. The throughput measurements reported are for the cumulative incoming rates for all 3 Gigabit interfaces. Results in Table 1. show that the network driver is capable of sustaining the near 3 Gbps level of throughput for various packet sizes. The throughput is worst when the packet size is around 256 bytes because of the manner in which IXP2400 transmits packets.

Next, for a single message size (576 bytes), we evaluate the effect of chaining rule handler stages into ruleflows. The vertical bars in Figure 2 show the additional latencies incurred as a result of adding rule handler stages to the ANP pipeline. Measured throughput is represented with the dark horizontal line. In each of the four handlers evaluated we vary the memory location where we store the state and decision tables accessed by the handler code. The results demonstrate that with our architecture throughput levels can be maintained even for chained rule handlers accessing slower memories, and that the increase in processing time as a result of the rule processing can result in acceptable delays.

The results in Figure 3 further describe the relationship between amount on memory accessed by the rule handler, its location in the IXP's memory hierarch, and the sustainable throughput levels. The size of the memory accessed by the handler is denoted on the x axis, while The numbers associated with the graph tickers denote the actual number of

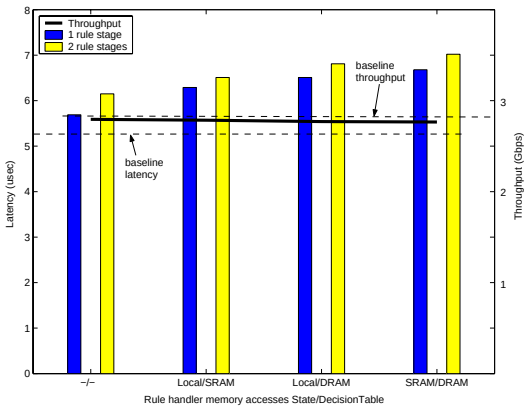

Figure 2. Throughput and latency for ruleflows accessing different IXP memories. 8 threads execute the same rule handler in each stage.

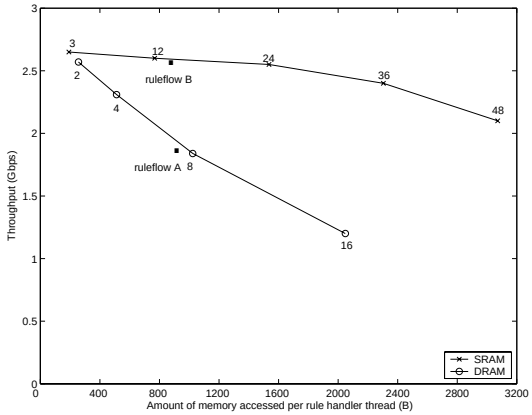

Figure 3. Throughput as a function of size of state accessed. Total memory accessed at each stage is 8 times the amount shown on x-axis.

memory accesses performed. The results are gathered for a worst case scenario, where at each stage, each of the eight threads executing the rule handler access the same memory locations for every data item, which is very unrealistic. No events are consumed at either of the stages. The results indicate that while state-constrained handlers can be efficiently supported by storing state in SRAM, extra care needs to be taken to ensure that the number of required DRAM accesses is kept to a minimum.

This is further demonstrated by measurements made for two actual ruleflows from the Delta's OIS, applied to a flight event data stream derived from this application (points A and B in Figure 3). The business rule processing performed here determines whether a change in the departure gate occurred for the specific flight, in which case an update event needs to be produced for the flight caterers. The rule handler in A accesses DRAM-resident state for all Delta flights in order to determine the gate change. In the ruleflow presented in B, it is first determined if a departure airport matches any of airports for which flight information is stored in SRAM, and DRAM accesses are performed only if necessary. In doing so, in spite of the increased amount of SRAM state accessed, the average number of costly DRAM accesses is reduced by more than 60%. This translates in increased throughput levels close to 20%.

The last set of experiments evaluates the ability of the NP rules engine to execute different actions for different event types. We compare the performance of a general handler, which is invoked for all events in the data stream, to the performance of a collection of specialized handlers, each corresponding to specific event types. The general handler uses state information not only for accessing decission tables, i.e., target_value-action pairs, but also for interpreting the event type, determining offsets to relevant event fields, and encoding the required operations. These additional operations and field offsets are hardcoded in the case of the specialized handlers. For the experiments, we use the Delta data stream as input, with the flight number identifying the event type, so as to simulate larger number of distinct event types. For the Delta event handlers used in these experiments, the rule engine could store up to 130 simple target_value-action rules, where action is consume or forward.

First we observed that for small number of accesses into the event data structure, both configurations result in comparable throughput levels. As the number of data accesses increases, the specific handlers increase in size, and the general handler in its state requirements. In the first case, the number of rules which can be sustained is linearly decreased, which in the second case, the measured throughput levels start degrading, and in the test performed decreased to 47%. These results indicate that (1) we can implement efficient and scalable NP-base rule engines, which can sustain a large number of relatively simple rules efficiently, and (2) that the built-in parallelism on the NP permits us to execute efficiently more complex rules, which require repeated accesses to the event data structure.

In summary, the results demonstrate the feasibility of executing application-level event-action rules on high-data rate networking devices, and the ability to exploit the programmability of such devices to dynamically and efficiently, with pracitically negligible overheads (the overhead of 28-30 microseconds is ammortized over multiple data events), customize the operations of the rules engine and tune it to match the current operating conditions or application requirements.

5. Conclusions and Future Work

Event-action processing is a commonly used paradigm in business applications, ranging from relatively simple actions in publish/subscribe systems to the complex rules executed in complex business codes. This paper explores the ability of network processors (NPs) to participate in event-action applications. The software architecture used assumes that network processors are attached to host machines (ANPs), thereby enabling developers to 'split' rules and rule state across the combined host-ANP nodes. Moreover, since ANP resources are limited, such splitting can be done at runtime (i.e., hot-swapping) such that only the most important rules and their state working set reside on the ANP.

The basic contribution of this paper is the design and exploration, in terms of performance, of a simple ANP-resident rule engine. Performance not only depends on rule complexity but also on the amount and location of state accessed by ANP-resident rules. Experimental measurements, attained on the Intel's IXP2400 network processor and on a cycle-accurate simulator for the gigabit-based boards, detail the dependence of rule performance on the memory hierarchy and represent the overheads of rule hot-swapping.

Future work will establish that: (1) our IXP-resident rule engine is sufficiently rich to encode a wide variety of actual business rules, (2) the ability to split rule processing across hosts and ANPs is key to the use of rule engines in realistic business applications, (3) there are other applications for ANP-resident rule engines beyond the business codes explored in our work, and (4) that the approach presented here can be made accessible to non-expert end users by integrating the SPLITS software architecture with the middleware used by OIS applications.

References

[1] 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.

[2] Fair Isaac. *The Business Case for Blaze Advisor. White Paper*, 2002.

[3] A. Gavrilovska. *SPLITS Stream Handlers: Deploying Application-level Services to Attached Network Processors*. PhD thesis, Georgia Institute of Technology, 2004.

[4] 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.

[5] A. Gavrilovska, K. Schwan, and V. Oleson. Practical Approach for Zero Downtime in an Operational Information System. In *Proc. of ICDCS'02*, Vienna, Austria, July 2002.

[6] IBM Corporation. *IBM Transaction Processing Facility*. http://www.s390.ibm.com/products/tpf.

[7] ILOG. *ILOG Business Rules*. http://www.ilog.com/-products/businessrules/.

[8] Intel IXP Network Processors. http://developer.intel.com/design/network/producs/npfamily/.

[9] C. Liao, M. Martinosi, and D. W. Clark. Performance Monitoring in a Myrinet-Connected Shrimp Cluster. In *ACM Sigmetrics Symposium on Parallel and Distributed Tools (SPDT)*, Aug. 1998.

[10] Y.-D. Lin, Y.-N. Lin, S.-C. Yang, and Y.-S. Lin. DiffServ over Network Processors: Implementation and Evaluation. In *Proc. of Hot Interconnects 10*, Stanford, CA, Aug. 2002.

[11] K. Mackenzie, W. Shi, A. McDonald, and I. Ganev. An Intel IXP1200-based Network Interface. In *Proceedings of the Workshop on Novel Uses of System Area Networks at HPCA (SAN-2 2003)*, Anaheim, CA, Feb. 2003.

[12] Radisys ENP-2611 Data Sheet. http://www.radisys.com/files/ENP-2611_07-1236-02_0803.pdf.

[13] 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.

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