Dynamic Classification in Silicon-Based Forwarding Engine Environments

R. Jaeger[1,2], R. Duncan[2], F. . Travostino[2], T. Lavian[2], J. Hollingsworth[1]
[1]*University of Maryland, College Park, MD 20742*
[2]*Nortel Networks, 4401 Great America Parkway, Santa Clara, CA 9505*
{rfj,hollings}@cs.umd.edu {rduncan, travos, tlavian}@nortelnetworks.com

***Abstract*--Current network devices enable connectivity between end systems with support for routing with a defined set of protocol software bundled with the hardware. These devices do not support user customization or the introduction of new software applications. Programmable network devices allow for the dynamic downloading of customized programs into network devices allowing for the introduction of new protocols and network services. The Oplet Runtime Environment (ORE) is a programmable network architecture built on a Gigabit Ethernet L3 Routing Switch to support downloadable services. Complementing the ORE, we introduce the JFWD API, a uniform, platform-independent portal through which application programmers control the forwarding engines of heterogeneous network nodes (e.g., switches and routers). Using the JFWD API, an ORE service has been implemented to classify and dynamically adjust packet handling on silicon-based network devices.**

***Index terms*-- Programmable Networks, Active Networks, ORE, JFWD, Differentiated Services**

1 INTRODUCTION

Traditional network nodes (e.g. routers on the Internet) enable end-system connectivity and sharing of network resources by supporting a static and well-defined set of protocols. The "virtual machine" defines the service provided by the network to its users at each router, and is simple and fixed. The trend in commercial-grade routers and switches has been to implement ever more functionality of this virtual machine in hardware; hardware implementations have, in turn, enabled ever faster instantiations of traditional network nodes. However, the gain in raw performance due to hardware implementations is, almost by necessity, at a loss of customization options supported by the router on the data path. As more of the router virtual machine is *frozen* in silicon, less are the opportunities to introduce new services inside the network.

1.1 Flexible Forwarding: Active Networks

In contrast, active networks (AN) expose a "programmable" user-network interface that allows network services to be introduced in active networks "on-the-fly". Active networks support per-flow customization of the service provided by the network; flow is defined by the user-network interface. The tenet of active networking is as follows: the utility of the service by the network to individual applications is maximized if applications themselves define the service provided by the network. To provide this level of support, active network implementations incorporate a substantial software component on the data path [12].[1] Thus, implementations of traditional and active networks must address the tradeoff between performance and flexibility.

In this paper, we explore one point in the performance-flexibility space: we describe an implementation of active network techniques on a commercial routing platform--a programmable network service platform implemented on the Nortel Networks Accelar Gigabit Routing-Switch.

1.2 Active Networking applied to Commercial Grade Hardware

This work describes a programmable network service platform implemented on the Nortel Networks Accelar Gigabit Routing-Switch. The primary goal of our work is to build a working platform for implementing programmable services on a commercial-grade router. In doing so, we have tried to (a) preserve the router hardware fast-path for data packets, and (b) leverage existing active networking research as much as possible. Obviously, (a) implies that certain computations that require data plane flexibility are impossible in our implementation. A sub-goal of our work is to identify sets of computations that become possible as additional functionality is placed into hardware. To support goal (b) we have implemented a layer over which existing active network implementations can be ported. In active networking parlance, we have not introduced a new AN execution environment (EE), instead, we added a Java-based run-time environment (the Oplet Run-time Environment) for security and service management over which existing EEs can be run as network services; We ported the ANTS EE to run within the ORE.

This paper details a policy-based packet classification technique that is downloaded to a programmable network device to dynamically adjust the DS-byte in the IP header

[1] In case of both traditional and active networks, it may be possible to provide fast and customizable forwarding by incorporating hardware that is both programmable over a relatively fine time-scale and can forward packets at line-speeds (e.g. fast and programmable FPGAs).

of real-time flows on a silicon-based forwarding engine. The technique is implemented using the Oplet Runtime Environment (ORE), a network services platform which supports dynamically loaded Java services. Crucial to the classification engine is the JFWD network API which provides control of the forwarding plane operations. We demonstrate that we can support Java-based services and implement packet copying to the control plane, packet processing by dynamically downloaded services, and packet forwarding policy adjustment on a silicon-based, Gigabit Layer 3 Routing Switch.

1.3 Roadmap

In Section 2 we provide a brief overview of the DARPA active network architecture, followed by a description of internal architecture of the Accelar router. We discuss the issues that must be resolved before the DARPA AN architecture can be realized on a commercial platform such as the Accelar routing switch and describe our mapping of the DARPA AN architecture on the Accelar. In Section 3 we provide details of each component of our implementation and describe the interfaces supported by each layer. In Section 4 we present a dynamic real-time packet classification application of the ORE/JFWD implementation. In Section 5 we present related work and compare our implementation with existing work both in an architectural context and with respect to supported in-network computations. We present conclusions and in Section 6.

2 IMPLEMENTATION OVERVIEW

In this section, we present an overview of our implementation. We give a synopsis of the DARPA active networking architecture, the internal architecture of the Accelar router, and then describe how we realize parts of the AN architecture on the Accelar.

Figure 1 shows the node architecture for active networks developed by the DARPA active network research community [1].

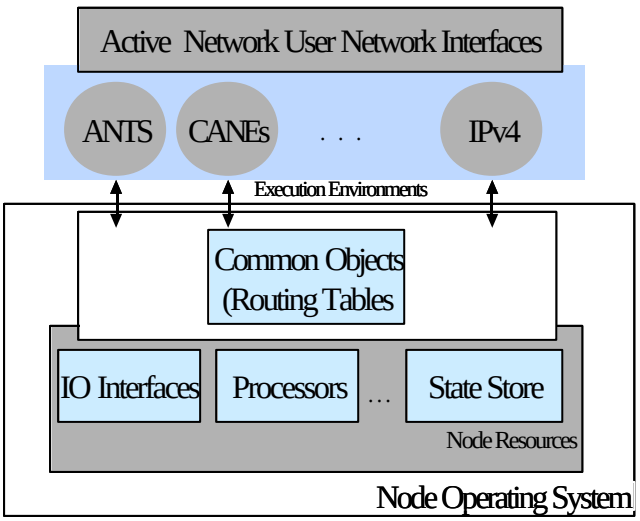

Figure 1: DARPA Active Network Architecture.

2.1 DARPA Active Network Architecture

The DARPA active network architecture must address the issue of the user-network interface supported by active networks, i.e. what is the nature of the "virtual machine" supported by each node in the network and what processing does each packet in the network undergo?

The DARPA architecture supports multiple active network user interfaces called *Execution Environments* (EEs). EEs can implement a wide range of user-network interfaces that exploit different points in the trade-off between performance and flexibility, e.g. IPv4 can be considered a high performance EE that does not provide much flexibility while the ANTS [12] is an EE that provides a Java virtual machine at each node and sacrifices some performance for enhanced flexibility. By supporting multiple EEs, the architecture allows network user applications choose between performance and flexibility.

The computation, communication, and storage resources at an active node are controlled by a *Node Operating System* (Node OS). The node OS interface that exposes resources available at the active node and mediates access. The node OS demultiplexes incoming packets to specific EE(s) and provides support for *common objects* such as routing tables likely to be useful across EEs. Using the node OS interface and abstractions, EEs implement specific user-network interfaces; network services are built using the interface exported by EEs.

2.2 Nortel Accelar Router

The Nortel Networks Accelar family of L3 Routing Switches employs a distributed ASIC-based (Application Specific Integrated Circuit) forwarding architecture with a 5.6-256 Gbps per second backplanes. The switches scale to 384 10/100 ports or 96 Gigabit ports (or some combination of the above). There are up to eight hardware-forwarding priority queues per port and hardware is controlled by VxWorks real-time OS.

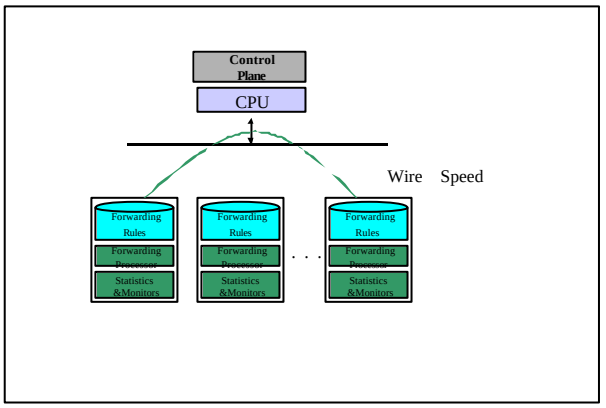

Figure 2: Achitecture of the Accelar Router

Applications can monitor and control the ASIC hardware via a switch-specific API that provides access to hardware MIB instrumentation variables. For example, the switch hardware provides functionality to set certain bits in an IP packet or put a packet in a priority queue based on a

matching a specific filter. This functionality is exposed by an interface to the hardware instrumentation MIB that controls the setting of hardware processing.

2.3 A Programmable Accelar Router

To transform the Accelar routing switch into a programmable network service platform, we implemented a run-time environment over which existing active network EEs could be executed. In general, this would require the implementation of the active network NodeOS API over the Accelar embedded real-time OS. However, the AN NodeOS API [2] was still evolving when we started our work and most EEs are implemented either within a JVM [12, 13] or over legacy OS interface. We chose not to port/implement the NodeOS API but to provide support for Java-based EEs by running an embedded JVM on the Accelar VxWorks OS and by creating a Java-compatible interface to the low-level hardware.

Unlike in traditional operating systems, the service degradation due to a single (possibly malfunctioning or malicious) JVM task on VxWorks is constrained because the JVM runs as just another task in the real-time VxWorks O/S with a fixed and upper-bounded processor share and priority. The Java-compatible interface that provides access to the low-level hardware of the Accelar is the Java Forwarding (JFWD) API described in section 2.5.

Though not technically a necessity, we added separate layer between the JVM and the EE. This layer --the Oplet Runtime Environment (ORE)-- provides security and management services that may eventually be subsumed by the AN NodeOS and was deemed required for the commercial viability of the programmable routers. As mentioned before, the ORE enables a stricter intra-node trust infrastructure allowing for different per-node resource allocation policies without cooperation from EE writers. Thus, the ORE provides mechanisms for node-providers to impose per-EE resource limits without having to trust the EE. A nice side-effect is that the ORE allows multiple EEs (or multiple copies of a single EE) to be instantiated within a single Accelar with different privileges. In the next section, we present details of the ORE and JFWD API.

3 THE ORE AND THE JFWD API

The ORE and the JFWD API are the two major software components added to the hardware router. The ORE provides a secure run-time environment for EE execution while the JFWD API provides a mechanism for EEs to access and control the switch hardware. We start with a description of the ORE.

3.1 The ORE: Oplet Runtime Environment

The ORE is a platform for secure downloading, installation, and safe execution of Java code (called *services*) within a JVM. A service is an atomic piece of code that implements specific functionality. A service may *depend* on other services in order to execute. In order to securely download and impose policy, we define the notion of an "Oplet". Oplets are self-contained downloadable units that encapsulate a non-empty set of services. Along with the service code, an Oplet specifies service attributes, authentication information, and resource requirements. Note that Oplets can encapsulate a service that depends on some other service; in these cases, Oplets also contain dependency information. In general, the ORE must resolve and download the transitive closure of Oplet dependencies before executing a single service.

The ORE provides mechanisms to download Oplets, resolve dependencies, manage the Oplet lifecycle, and maintain a registry of active services. The ORE uses a public-key infrastructure to download "trusted" Oplets. In brief, the security infrastructure provides authentication, integrity, and non-repudiation guarantees on downloaded Oplets. Due to space restrictions, we will not elaborate more on the secure downloading, execution, or resource management features of Oplets.

3.2 Oplet Execution Safety

The ORE must provide safe execution and impose resource limits. As far as possible, the ORE uses the mechanisms provided by the Java language (type safety) and the standard JVM (bytecode verification, sandbox, security manager) to provide execution safety. The ORE controls allocation of system resources by intercepting allocation calls from the service code to the JVM.

To protect itself from denial of service attacks, deadlocks, and unstable states, the ORE implements mechanisms for thread safety and revocation. The ORE controls thread creation by requiring Oplets to request new threads from the ORE. The ORE determines whether to grant the request based upon a node policy that takes into account current thread usage, and the credentials of the requesting Oplet. Once a thread is allocated, however, the current implementation of the ORE has no mechanism in place to account for or limit the consumption of computing resources. In its most general form, the ORE must address denial of service caused by a misbehaving service that consumes CPU resources. To handle these issues, the ORE needs JVM support for CPU accounting [14].

Sharing threads between Oplets presents two main problems: (a) deadlock caused by a callee not returning and (b) caller Oplet killing the shared thread while it is executing in a callee Oplet's critical section. The ORE protects itself from the first problem by never interacting directly with any Oplet that it loads. Instead it creates a trusted proxy which the ORE uses to delegate its commands to the untrusted Oplet. The proxy uses a

separate thread to call a method on the untrusted Oplet and sets a timeout for returning from the call; if the thread call does not return after a conservatively set timeout, a fail-stop situation is assumed and the thread is killed. The second problem is handled by the ORE by revoking Oplet's ability to manipulate a thread's running status.

The ORE uses object revocation to control access to its own resources. If the ORE determines that a specific service is no longer permitted to use a resource reference the reference can be revoked. For example, a service may possess a "handle" to a data structure exported by another Oplet that no longer exists. The ORE can detect these cases and revoke access to "stale" objects. However, for absolute protection, non-standard support is required from the JVM implementation. Significant modification would include the ability to perform accounting for both CPU and memory consumption and support for per-thread heap allocation and garbage collection [14].

The ORE is currently under active development. At present, it supports secure downloading of services, resolves service dependencies, and allows access to native router functionality through the JFWD API. However, the current ORE version is still susceptible to several flavors of denial-of-service attacks. These include spurious triggering of the Java garbage collector, memory fragmentation attacks, and stalling finalization of objects [14]. Several memory related safety hazards confronting the ORE will be resolved as JVMs support multiple heaps, revocation and copy semantics of the JKernel [8].

3.3 JFWD: The Java Forwarding API

The Java Forwarding API specifies interfaces for Java applications to control a generic, platform-neutral forwarding plane. The JFWD API consists of a set of Java classes specifications and has been implemented on the Accelar. Implementations of the JFWD classes is highly platform dependent (recall that on the Accelar, the JFWD API is a wrapper around the hardware MIB instrumentation interface) is not part of the JFWD API.

JFWD is the first Java API package to explicitly address the needs of contemporary forwarding planes of network nodes which are increasingly implemented in silicon, must support multiple protocols, and must realize Quality-of-Service (QoS) low-level functionality. To drive an ever-growing number of options, more diversified control data will have to be fed into control planes.

The JFWD implementation across multiple hardware platforms brings a) the abstraction of a platform-neutral forwarding engine, b) a framework within which new protocols and control data can be described, and c) a new breed of shrink-wrapped Java programs for controlling network nodes. In the rest of this section, we highlight the main mechanisms that are provided by the JFWD API on the Accelar switch.

The JFWD API can be used to instruct the forwarding engine to alter packet processing through the installation of hardware filters. The hardware filters execute "actions" specified by a filter policy. On the Accelar, the filter can be based on combinations of fields in the MAC, IP, and transport headers. The policy can define where the matching packets are delivered and can also be used to alter the packet content. Packet delivery options include discarding matching packets (or conversely, forwarding matching packets if the default behavior was to drop them) and diverting matching packets to the control plane. Diverting packets to the control plane allows applications, such as AN EEs to process packets. Additionally, packets can be "copied" to the control plane or to a mirrored interface. Packets may also be identified as being part of high priority flows; these packets can be placed in a static hardware high priority queue.

The filter policy can also cause packet and header content to be selectively altered (e.g. the Type of Service bits on matching packets can be set). The existing hardware is capable of re-computing IP header checksum information at line speeds when the IP header is changed.

3.4 Network Services supported by the JFWD API

In this section, we explore the set of possible and precluded computations on the ORE/JFWD API. Obviously, these computations define and constrain the set of network services that can be implemented on the ORE/JFWD platform. Note that the ORE does not, a-priori, exclude any computation; instead, it enforces node policy that may cause certain (e.g. processor-intensive) computations to not be started or terminated during execution. Computations are, instead, constrained by the JFWD API since this API defines the capabilities exported by the hardware that can be used to build network services. Thus, some computations, e.g. certain video transcoding techniques that must process every packet, cannot efficiently be implemented in our system regardless of node policy. Not all precluded computations involve data transformation; certain network based anycasting/routing schemes in which a program must be executed to find the outgoing switch port cannot be supported either.

In general, all control-plane only computations, e.g. installing new routing tables or parsing a new ICMP message type, can rather easily be accommodated by the ORE/JFWD API. An important ability provided the JFWD API is to *selectively* route (or copy) packets to the control plane. As we will see, this does significantly enlarge the set of services that can be implemented on the Accelar. In the rest of this section, we identify a specific set of services that can be implemented using the current version of the JFWD API.

The following network services can be implemented using the current implementation (obviously, this list is merely representative and not meant to be exhaustive):

- Filtering firewall - A simple application would be a firewall that allows or denies packets to traverse on specified interfaces depending on whether the packet's header matches a given bit mask.

-Application-specific firewall - It is relatively straightforward to extend the filtering firewall implement certain application-specific firewalls. For example, a FTP gateway that dynamically changes the firewall rules to allow *ftp-data* connections to a ``trusted" host can be implemented. Security functions like stopping TCP segments with no (or all) bits set can also be dynamically programmed on the Accelar.

Almost all modern routers allow for a filtering firewall and application-specific firewall functionality. It is imperative to note that these services can be added, modified, and deleted dynamically on to the Accelar ORE/JFWD platform. The next three services are examples that, in general, are *not* yet available in most commercial routers. (There are isolated instances of routers where some of these services may be built into the hardware).

- Dynamic RTP flow identification - RTP over UDP flows are identified by an ephemeral UDP port number. In general, some host chooses this port number and it is not well known. We have implemented several mechanisms to identify RTP flows traversing the Accelar. Using the JFWD API, control protocol (SIP/RTSP/H.323) messages can be intercepted to identify port numbers for RTP flows.

- DiffServ: Classifier, Marker - The Accelar can be turned into a DiffServ[6] Classifier by suitably programming its hardware filters. Further, the hardware (and in turn, the JFWD API) provides mechanisms to change, at line-speed, selected bits in the IP header. This ability can be used to implement parts of DiffServ ingress/egress marker capabilities on the Accelar. A subtle benefit of this solution is that new firmware or hardware does not have to be shipped each time a new DiffServ scheme/PHB becomes popular. Instead, using existing ORE service instantiation mechanisms, only the service-specific logic has to be uploaded onto the router. This can be accomplished on-line, without interrupting existing flows or services.

- PGM-like Reliable multicast - The packet filtering capabilities of the Accelar allows certain packets to be copied on for inspection by the service code. This mechanism can be used to divert (negative) acknowledgements from multicast sessions to the control plane. The service code can, much like the PGM reliable multicast scheme, send one copy of the NAK upstream and suppress duplicate NAKs. Unlike PGM, modulo resource constrains, it is possible to implement reliable multicast services that keep a small packet cache and immediately re-transmit a lost segment. Other services, such as multicast ancestor discovery, can also be efficiently implemented by providing the service code interfaces to the routing table.

4 EXPERIMENTAL JFWD APPLICATION

To support a new breed of router-based programs that do not delay packet forwarding, we use the JFWD API to instruct the forwarding engine to send packets to both the data plane output queue and also to the control plane. This feature, called CarbonCopy, does not divert packets to the control plane; it performs wire-speed forwarding while copying the packets to the control planes. Packets captured into the control plane can then be processed and decisions about packet processing can be made. By allowing ORE services to modify the packet handling in the ASIC forwarding engine, new types of applications are possible that do not require core router functionality to be altered.

In [5], the authors present a control-on-demand architecture in which flow control is performed on a *best effort* basis. An IPv4 flow is determined by five fields in the IPv4 header: the IP source and destination address, the source and destination port numbers, and the protocol. Packets belonging to a flow are copied to the control plane for processing while simultaneously forwarding the packet. This is our approach as well. It deviates from the store-execute-forward paradigm of Active Networks.

We implemented an ORE service, called the DiffServ Classification service (DSC), to classify flows and modify packet headers based on the policy set for a particular protocol contained in the IP header. A policy determines the action to be taken for a particular flow or class of flows. A policy filter executes the actions specified in the policy.

Our implementation copies all real-time protocol connection signaling messages (e.g. SIP, RTSP, H.323) from the forwarding plane. To accomplish this task, the service sets a filter for packets on the default SIP connection setup port. This filter copies all packets on that port into the control plane. The DSC receives these packets and processes the content to identify the ports on which the RTP communication is to take place. Since the SIP messages use a text format for communicating the port information, identifying the port information is straight forward.
The DSC creates a policy filter for RTP flows identified by the signaling messages. The policy filter created for a particular flow is based on the IP header fields in the packet that determine a flow. We call this the flow signature. In this case, the signature consists of the source and destination IP addresses and the source and destination port numbers. Protocol could also be used as part of the signature (e.g. UDP). The DSC sets a policy

filter for the packet signature that will put those flows into the higher priority queue within the edge router. The Accelar 1100B has 2 priority queues so RTP packets were placed into the higher of the two queues. The DSC also sets the DS-byte in the Ipv4 header on the packets to ensure better processing treatment at downstream processors that implement Differentiated Services as determined by the per-hop behavior policy. A policy filter is stored for each RTP flow identified by the signaling message processing.

4.1 Discussion

The novelty of the experiment described above is that it is done by a service that is downloaded and installed dynamically onto a commercial-grade router and that RTP flows are identified dynamically, not preconfigured. In this section, we discuss an immediate application of this functionality that we are using in our own research facilities.

An immediate benefit of on-line identification of flows and dynamic adjustment of packet priority is to support cluster computing. In cluster systems such as Condor[11], NOW[3], Stealth[10], and Linger-Longer[15] workstations are used to run jobs when the computer's primary user is not using their computers. To make these systems usable, the software that runs guest jobs user's workstations goes to great lengths to ensure that the guest process does not interfere with the primary user. However, until now there has been no clean way to isolate guest use of a workstation from network traffic generated by normal users.

By using active networking at the local area network switch, we can dynamically identify the flows associated with guest jobs. Although these jobs typically have a set of well-known ports, they also can use other network services. To help identify these flows, the cluster scheduler software, can inform the switch when a particular node has started to run a guest process. For some clusters such as Condor and NOW, a node in a cluster is either running guest processes or local process and switches between them on a time-scale measured in minutes. For these types of systems, a simple filter to re-prioritize all traffic from the host running a guest process can be installed by the cluster scheduler. However, system such as Stealth and Linger-Longer allow fine-grain sharing of processors between guest and local processes. To accommodate these systems, the filter needs to be able to identify whether traffic associated from a node is due to a guest process or a local process. To do this a more complete dynamic flow detection that that can now be implemented on the Accelar is required.

5 RELATED WORK

We are not aware of another integrated active networking platform implementation on a (commercial) hardware platform. The active networking work on the Washington University Switch Kit employs locally connected machines as active processors[2]. The Tempest [13] provides a customizable control plane for ATM networks. The basic ideas of high-performance active networking by decoupling the forwarding path from a programmable control plane was introduced, in a software implementation, in the Control-on-Demand (CoD) [9] platform co-developed at AT&T Labs. In this section, we compare our approach to CoD, and discuss how existing active networking research fits within our framework.

The Control-on-Demand platform was developed and implemented over IPv6 as an extension to the Linux kernel [9][3]. Data packets were kept in the kernel in per-flow queues while active control could be applied to the data packets by dynamically loaded "per-flow controllers" that executed in user space. The per-flow controllers affected the data path using the CoD API. A meta-controller loaded each per-flow controller using a signaling protocol. CoD was developed to be specifically mapped onto hardware platforms and its relationship to our work is clear. Services on the Accelar map to per-flow controllers in CoD; the JFWD API on the Accelar maps to the CoD API; the ORE functionality on the Accelar is not completely replicated in CoD, though the meta-controller does provide a subset of the ORE functionality. As CoD was implemented in software; it provides all of the JFWD functionality, and also provides the queue exposure and manipulation facilities on our hardware wish list.

Active networking NodeOS's can potentially be implemented over VxWorks on the Acclear. There is one fundamental problem: the AN NodeOS architecture allows for all packets on specific channels to be delivered to the EE for further processing. This would negate the benefits of the hardware forwarding path available in the Accelar. However, the Accelar provides a perfect platform for implementing fast cut-through paths. The Bowman NodeOS[12] is a particularly good fit as it is specifically supports cut-through paths and is designed as a layer above a host OS that provides low-level hardware access. Thus, Bowman can directly be ported on to the Accelar using VxWorks as its host OS.
For other Node OS efforts, the VxWorks platform already implements much of the required functionality such as memory management. However, it is not obvious if some of the abstractions supported by these systems (e.g. the path abstraction in Scout) can directly be mapped on to the Accelar hardware features.

Java-based EEs can directly be ported on to the ORE. Once a functional AN Node OS has been ported to run over VxWorks, other "native" EEs such as CANEs can be implemented on the Accelar.

6 CONCLUSION

[2] See http://www.ccrc.wustl.edu/gigabitkits/kits.html

[3] Control on Demand was co-developed by S. Bhattacharjee

We presented a summary of the challenges in bringing Active Networking ideas to current high performance hardware-based routers and switches. In addition, we showed that while it is not possible to support active packets for every packet at line speed on these systems (nor any system), it is possible to exploit existing hardware filtering mechanisms to allow a variety of scenarios that require active functionality on routers. To demonstrate the feasibility of our approach, we implemented a dynamic classification service on a commercial-grade Gigabit L3 Routing Switch. The ORE supports the creation of services in Java that are extensible, portable, and easily distributed over the network. It provides a platform for safely running these services without disturbing the existing core functionality of the router. Through the JFWD API, the packet capture, processing, and modification capabilities are realized at the programmable node. Because of the CarbonCopy and filter mapping features provided in the JFWD API, we were able to create an ORE service to perform dynamic DiffServ Classification and DS-byte adjustment in an effort to improve the performance of real-time network flows.

7 REFERENCES

1. "Architectural Framework for Active Networks Version 0.9," . August 31, 1999,.Active Networks Working Group.
2. "NodeOS Interface Specification," . June 15, 1999,.AN Node OS Working Group.
3. R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson, "The Interaction of Parallel and Sequential Workloads on a Network of Workstations," *SIGMETRICS*. May 1995, Ottawa, pp. 267-278.
4. S. Bhattacharjee, *Active Networks: Architectures, Composition, and Applications*,Ph.D., Computer Science Department Georgia Institute of Technology,1999.
5. R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin., *Resource ReSerVation Protocol (RSVP)*, RFC 2205, , September 1997.
6. D. Black, S. Blake, M. Carlson, E. Davies, Z. Wang, and W. Weiss, *An Architecture for Differentiated Services*, RFC2475, , Dec. 1998.
7. S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *IEEE/ACM Transactions on Networking*, **1**(4), 1993, pp. 397-413.
8. C. Hawblitzel, C. Chang, G. Czajkowski, D. Hu, and T. v. Eicken, "Implementing Multiple Protection Domains in Java," *USENIX Technical Conference Proceedings*. June 1998.
9. G. Hja'lmtysson and S. Bhattacharjee, "Control on Demand: An efficient approach to router programmability," . April 1999.
10. P. Kruger and R. Chawla, "The Stealth Distributed Scheduler," *ICDCS*. 1991, pp. 336-343.
11. M. Litzkow, M. Livny, and M. Mutka, "Condor - A Hunter of Idle Workstations," *International Conference on Distributed Computing Systems*. June 1988, pp. 104-111.
12. S. Merugu, S. Bhattacharjee, E. Zegura, and K. Calvert, "Bowman: A Node OS for Active Networks," *to appear INFOCOM'2000*.
13. J. E. v. d. Merwe, S. Rooney, M. Leslie, and S. A. Crosby, "The Tempest - A Practical Framework for Network Programmability," *IEEE Network*, **12**(3), 1998.
14. P.Bernadat, D. Lambright, and F. Travostino, "Towards a Resource-safe Java for Service-Guarantees in Uncooperative Environments," *IEEE Symposium on Programming Languages for Real-time Industrial Applications (PLRTIA)*. Dec. 98, Madrid, Spain.
15. K. D. Ryu and J. K. Hollingsworth, "Linger Longer: Fine-Grain Cycle Stealing for Networks of Workstations," *SC'98*. Nov. 1998, Orlando, ACM Press.
16. D. Wetherall and e. al., "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols," *OPENARACH'98*. 1998.
17. Y. Yemini and S. da Silva, “Towards Programmable Networks,” in IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, L’Aquila, Italy, October, 1996