Design and Implementation of A Content-aware Switch using A Network Processor

Li Zhao, Yan Luo and Laxmi Bhuyan
Department of Computer Science and Engineering
University of California, Riverside, CA 92521
{zhao, yluo, bhuyan}@cs.ucr.edu

Ravi Iyer
Communications Technology Lab
Intel Corporation, Hillsboro, OR 97124
ravishankar.iyer@intel.com

Abstract

Cluster based server architectures have been widely used as a solution to overloading in web servers because of their cost effectiveness, scalability and reliability. A content aware switch can be used to examine the web requests and distribute them to the servers based on application level information. In this paper, we present the analysis, design and implementation of such a content aware switch based on an IXP2400 network processor (NP). We first analyze the mechanisms for implementing a content-aware switch and present the necessity for an NP-based solution. We then present various possibilities of workload allocation among different computation resources in an NP and discuss the design tradeoffs. Measurement results based on an IXP 2400 NP demonstrate that our NP-based switch can reduce the http processing latency by an average of 83.3% for a 1K byte web page, compared to a Linux-based switch. The amount of reduction increases with larger file sizes. It is also shown that the packet throughput can be improved by up to 5.7x across a range of files by taking advantage of multithreading and multiprocessing, available in the NP.

1 Introduction

With the explosive growth in Internet traffic, web servers have been overloaded with high volume of requests. Many ISPs and search engines employ a server cluster to build a cost effective, scalable and reliable server system. To make such a distributed server system transparent to the clients, a switching device is usually placed in front of the server cluster as a common interface. The switch is connected to the external world and has one virtual IP (VIP) address, whereas servers inside the cluster can have their private IP addresses.

As shown in Figure 1, layer 5 or content-aware switches [3] [6] [11], which route packets based on layer 5 information (request content or application data) are gaining wide popularity. Compared to traditional layer 4 switches, they

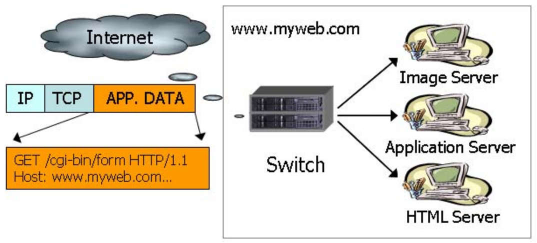

Figure 1. A cluster of servers front-ended by a switch

can provide: (1) better load balancing by distributing requests based on the content type, such as static or dynamic, to servers optimized for a particular type; (2) faster response by sending the request to servers that have already serviced the same request recently to exploit cache affinity. (3) better resource utilization by partitioning the servers' database among servers instead of replication.

Currently content-aware switches are built based on ASICs [3] [6] [11] and general purpose processors [4] [7] [20]. However, ASIC based switches have no flexibility or programmability although they can achieve very high processing capacity. Switches based on general purpose processors, on the other hand, cannot provide satisfactory performance due to (1) interrupt, (2) moving packets through PCI bus and (3) large protocol stack overhead in the operating system. The third problem is introduced because of the software implementation of the content aware switch. Usually an HTTP proxy running on the application level maintains connections with the client and the server separately, and forwards data between these two connections. Although this approach is easy to implement, the overhead to copy data between these two connections is very high. This data copy problem can be solved by TCP splicing technique [10] [17], which splices the two connections after both of them are established. The switch then forwards subsequent data packets on the spliced connection by modifying particular fields (e.g. sequence numbers) in their TCP and IP headers. Since this data forwarding is performed at the IP level, the overhead of copying data between the user space and the kernel space is avoided.

The above problems can be solved by using Network processors (NPs). NPs operate at the link layer of the protocol like ASICs, thus avoiding the large overhead of interrupt and moving data through PCI bus in general purpose processors. At the same time, they are programmable so that they can achieve the same flexibility as general purpose processors. In addition, NPs have an instruction set architecture optimized for packet processing, and are usually equipped with multiprocessing and multithreading in hardware, which can provide a good throughput. To the best of our knowledge, no study on design of content-aware switches has been done using NPs. The aim of this paper is to design and implement such an NP-based switch using TCP splicing technique and demonstrate its high performance.

Implementing a content aware switch using an NP is not simple. First, the NPs are programmed at a low level language (microC or microcode) without the availability of a compiler that can directly translate C code to this language. Second, the instruction memory available in an NP is limited, so an incredible amount of effort is needed to reduce and optimize the existing code. In this paper, we first analyze several design options for a content-aware switch built upon an NP. Then we carefully allocate the workload among various resources in the NP for optimal performance. In the process, we have to ensure that the computational power of the NP is fully utilized for maximum throughput. We implement a content aware switch on an ENP2611 board that contains an Intel's IXP2400 NP. Our performance evaluation results show that this switch can significantly improve the processing latency as well as the throughput.

The rest of the paper is organized as follows. Section 2 gives a background of content aware switches and presents the design options to build an NP-based content-aware switch. Section 3 describes details of our design and implementation based on an IXP2400 NP. The experimental results are presented in Section 4. Section 5 describes the related work. Section 6 summarizes and concludes this paper.

2 Design Options

Content-aware switches have been built in Linux machines by inserting loadable kernel modules into the operating system [20] [15]. As shown in Figure 2(a), although data copy between two connections is avoided, packets have to be moved between the host DRAM and the NIC over the PCI bus. This imposes heavy bandwidth pressure on PCI bus when the number of connections is large. It also introduces interrupt overhead to the host CPU.

Figures 2(b) and (c) utilize NP-based network interfaces. An NP usually has a control processor (CP) and multiple data processors (DPs). CPs are typically embedded general purpose processors, which enable us to easily develop complex software. Therefore they are used to maintain control information. DPs are tuned specifically for processing network packets in fast path, but need to be developed in low level languages. CPs and DPs communicate through shared DRAMs.

In Figure 2(b), the CP can be used to create connections to clients and servers, and splice these two connections. Then packets sent after splicing can be processed on the DPs. However, since DPs are responsible to receive and transmit packets through NICs, packets sent before splicing need to be passed up through DRAM queues to the CP and passed down to the DP after they are processed. The packet en-queueing, de-queueing and polling time taken together increase the processing latency on these packets. Notice that these packets fall in the critical path for TCP splicing. This delay is detrimental to the overall performance because longer delay may cause timeout on clients and lead to packet retransmissions.

Given a large number of DPs and threads in an NP, Figure 2(c) is a natural evolution over (b). After receiving packets from NICs, the data processors handle the connection creation, splicing and data forwarding, without the need to communicate with the CP or the host CPU. The large number of hardware threads in data processors are capable of fast packet processing and eliminating data copying. However, developing optimized code on the DPs without compiler support and with limited control memory is a challenge. We use this architecture to design and implement a content-aware switch, which is described in detail in the next section.

3 Design and Implementation

In this section, we first describe the architecture of the Intel IXP2400 network processor. Then, we study how to efficiently distribute the workload among various resources in such a hardware environment. After that, detailed design and implementation are presented.

3.1 Hardware

Figure 3 shows an ENP2611 board with an embedded IXP2400 network processor, which is connected to the host machine through a PCI bus. The IXP2400 network processor contains a general-purpose XScale core, and eight microengines (MEs), which have instruction sets tuned specifically for processing network packets. Each microengine has a 16KB instruction memory preloaded by the XScale processor core. Up to eight threads can run in parallel on each microengine. The XScale runs an embedded Linux. All processors share an SRAM and a DRAM.

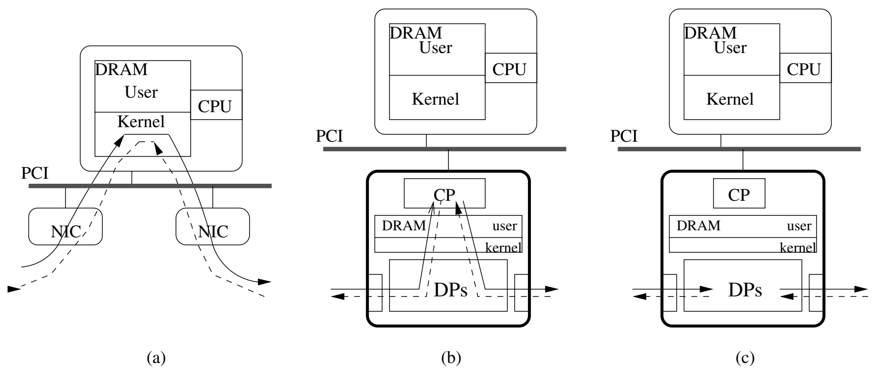

Figure 2. Three architecture candidates for web switches

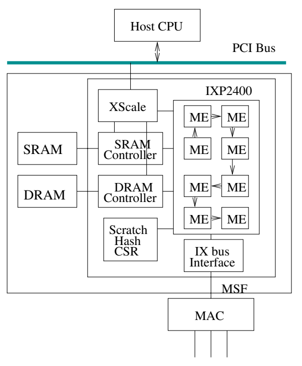

Figure 3. A high level architecture of the ENP2611

When a packet arrives at the Ethernet interface, it is received by one of the Media Access Controller (MAC) devices attached to the Media Switching Fabric (MSF). Threads in Microengines are programmed to move packets into a Receive FIFO, do some processing, and put outgoing packets in a Transmit FIFO, where they are transmitted to the line.

3.2 Resource Allocation

Given the hardware environment of the IXP2400 network processor, consisting of multiple processors/threads and various memory modules, it is a challenge to allocate these resources for minimum packet processing time. We also have to carefully allocate data in the memory. The two off-chip memory modules, SRAM and DRAM, not only have different sizes, but also have different access latencies. When unloaded, their access latencies are about 90 and 120 cycles, respectively [5]. Because SRAM is faster than DRAM, we use it to maintain all the control data structures. DRAM is relatively large and used for buffering packets.

Figure 4 shows the resource allocation for our content-aware switch. We first differentiate client ports from server ports. Client ports connect with the external world (clients). Server ports connect to servers in the cluster and are responsible for receiving packets from servers. Microengines are divided into four groups: receiving microengines (RX_ME), transmitting microengines (TX_ME), microengines that process packets from the client ports (ClientME) and from the server ports (ServerME). These microengines form a pipeline for processing packets. RX_MEs receive packets from the input ports and put them into the input queue. ClientME or ServerME process packets from these queues and put them into the next output queue. Finally TX_MEs are responsible to transmit those packets out onto the line.

The input and output queues are used to convey packet information between microengines. These queues are implemented in SRAM. They store packet descriptors, which

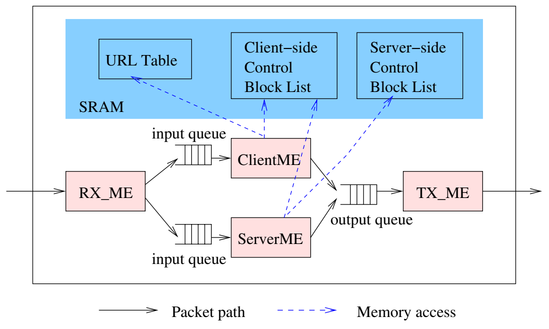

Figure 4. Microengines workload partition

contain the DRAM address, length of packets, input and output ports, etc. TX_MEs send these packets out based on the output port number.

Three major data structures are used in our switch: a client-side control block list (C-list), a server-side control block list (S-list) and a URL table. The C-list records the state for the connection between the client and the switch, and the state for forwarding data packets after connections are spliced. The S-list records the state for the connection between the switch and the selected server. The URL table is used to select a back-end server for an incoming HTTP request. This table contains a set of pre-defined mappings from URL suffixes to back-end servers. We left the implementation of more advanced algorithms for future work. All these data structures are maintained in SRAM. In addition, since the control blocks might be accessed by multiple threads/microengines simultaneously, updating these control blocks must be performed atomically. We exploit the SRAM locks supported in IXP2400 for this purpose.

3.3 Processing on Microengines

We classify packets into two types: control packets and data packets. Control packets are those sent before the two connections are spliced. These packets, such as SYN packets, are used to set up connections. The HTTP request packet is also treated as a control packet as it causes the second connection to be setup. Data packets are those sent after the two connections are spliced. They are response packets from the server, ACK packets from the client, and FIN packets from both sides.

When a packet arrives, a clientME/serverME extracts its IP and TCP headers and does a lookup of a control block in the control block list. The processing on this packet is based on the state in the control block. The detailed operations on clientMEs/serverMEs are described below.

3.3.1 ClientMEs

Figure 5 shows the data flow on a clientME starting when the clientME de-queues a packet from the input queue. This packet is first checked to make sure it is a valid IP packet. The IP validation includes checking its version, length and header checksum. Corrupted packets or packets other than IP or TCP are dropped. IP options are not handled as they are rarely used.

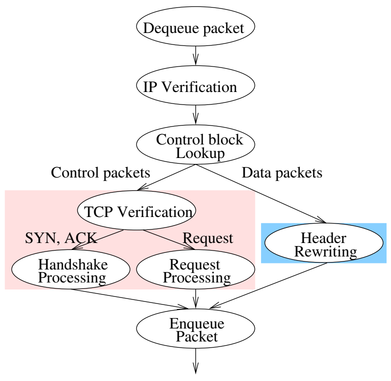

Figure 5. Data flow on the clientME/serverME

The C-list is searched to identify whether this a control packet or data packet based on a hash value calculated from the source port and IP address of this packet. Control packet processing and data packet processing are shown in the two shaded boxes respectively in Figure 5. For a control packet, the clientME first validates the TCP checksum and sequence number of this packet. Then it checks whether this packet is a SYN, an ACK, or an ACK/request packet. These three types of packets are the only control packets that need to be processed at the clientME. The handshake processing part processes SYN or ACK for connection establishment. For a SYN packet (with CSEQ as its initial sequence number), a control block is inserted into the C-list for the new connection. The ACK packet finishes the establishment of the connection. The request packet is parsed in the request processing module, and a back-end server is chosen based on the URL table. Then the clientME set up the second connection with the selected server by sending a SYN packet with the client's IP and port as its source IP address and port number. The initial sequence number of this SYN packet is set as CSEQ. The effect is that the switch masquerades as the client to send this SYN packet, so that only minimum changes are required in the subsequent forwarding part. For this second connection, the clientME inserts a control block in the S-list.

If the incoming packet is found to be a data packet (it is an ACK packet to acknowledge server's response in most cases or a FIN packet used to close the connection), it is directly forwarded with its IP and TCP header updated. Its destination IP address is changed to the server IP. The acknowledge number is updated with the following formula: $new\_acknowledge\_number = old\_acknowledge\_number - DSEQ + SSEQ$, where $DSEQ$ and $SSEQ$ are initial sequence numbers in the SYN packet sent from the switch and the server respectively. The checksum in both the IP and TCP header are recalculated with the incremental checksum calculation method [14].

3.3.2 ServerMEs

Processing on ServerMEs has a similar data flow in Figure 5 with minor differences. One difference is that the serverME accesses the C-list using the hash value based on the destination IP address and port number. For the control packet, the serverME only needs to handle the SYN/ACK packet from the server because the SYN packet to initialize a connection with the chosen server is sent by the clientME. In response to this SYN/ACK packet, the serverME can send an ACK packet, and then the HTTP request. Since the data can be piggybacked with the ACK packet, we send the saved request along with the ACK. The state of the control block in the C-list is changed to SPLICED thereafter. The corresponding entry in the S-list is deleted.

The data packet processing is also similar to that on the clientME, with the difference on updated fields. The source IP is set to the switch IP address VIP. The sequence number is updated with the following formula: $new\_sequence\_number = old\_sequence\_number - SSEQ + DSEQ$.

3.4 Other Implementation Issues

When the connection between the server and the client is terminated, the corresponding control block needs to be deleted after 2MSL (120 seconds). To implement this time control, we maintain a timeout table in SRAM, with each entry containing a pointer to a control block and a timestamp that records the time when the control block should be deleted. As the deletion of the control block is not on the critical path for a connection, we run a timeout-table checking program on the XScale. Its main functionality is to check the timeout table regularly and delete the control block if it expires.

TCP options such as Maximum Segment Size (MSS), timestamp, and SACK are negotiated between the two end points in a three-way handshake. Since the cluster of servers may have various options, the switch may either reject all TCP options, or maintain a minimum set of options for the web servers. Currently we implement MSS option processing in the switch (1460 bytes in Ethernet). Other options like timestamp and SACK are left as the future work.

4 Performance Evaluation

In this section, we describe the experimental environment and present the performance results on our NP-based content-aware switch in terms of latency and throughput. The data are compared with a Linux based switch.

4.1 Experimental Setup

We implement a content aware switch using an ENP2611 board that contains an Intel IXP2400 processor. The XScale and microengines run at 600MHz. This board has 8MB SRAM and 128MB DRAM. It also has three 1Gbps Ethernet ports. We use one port as the client port and the other as a server port. The server port is connected with an Apache [1] web server running on an Intel 3.0GHz Xeon processor. The client port is connected to a layer 2 switch that connects two clients. Each client runs *httperf* [9] on a 2.5GHz Intel Pentium 4 processor. All PCs are running Linux 2.4.20. To compare its performance with that of Linux-based switch, we also build a Linux-based switch by inserting a loadable kernel module [8] into its operating system. This switch runs a Linux 2.4.20 kernel on a 2.5GHz Pentium 4 system with two 1Gbps Ethernet NICs.

The following results are obtained with one ME for RX_ME and one ME for TX_ME. In our experiments, we did vary the number of ClientMEs and ServerMEs. However, the result using one ClientME and one ServerME is the same as that using 2 ClientME and 2 ServerME, which implies that a pool of processors may not be helpful because all the MEs compete for the shared SRAM and DRAM.

4.2 Latency

First we conduct experiments to obtain the latency of packet processing for an HTTP session. Figure 6 shows the latency spent on the switch when we vary the request file size. We can see that the latency is reduced significantly by using IXP2400. Compared to the Linux-based switch, the latency on the NP-based switch is reduced by 83.3% (0.6 ms to 0.1 ms) with a small file size as 1KB, and the larger the file size, the reduction is higher. At a very large file size as 1024KB, the latency is reduced by 89.5%. We also measure the processing latency for data and control packets separately. It is shown that the latency reduction for control packets is larger than that of data packets (80% vs. 50%, detailed results are not shown due to space limitations).

The latency reduction by using NP comes from three factors: (1) Interrupt vs. polling: When NIC in the Linux machine receives packets, it raises an interrupt to the CPU. Although current NICs have the ability to accumulate multiple packets and then notify the processor using a single interrupt, the overhead of interrupt is still very high. NPs use polling instead of interrupt to reduce this overhead.

(2) NIC-to-memory copy vs. no copy: In the Linux-based switch, the NIC has to copy the received packets to the main memory, which is a DMA transfer through the PCI bus. Similarly, when the packets are sent out, they are transferred from the memory to the NIC buffer by DMA again. In NP-based switch, however, packets are processed inside the NIC without these two copying (transfers). We had an experiment on Xeon 3.0 GHz Dual processor with 1Gbps Intel 88544GC NIC (Intel Pro 1000). The result showed that about three microseconds is spent on DMA receiving for a 64-byte packet. We do not have this latency in IXP implementation.

(3) Linux processing vs. IXP processing: Even if the whole functionality of the content aware switch is implemented in the Linux kernel, the OS overhead like context switch would happen, whereas NPs do not have such overheads. In addition, optimized instruction set enables us to process packets much more efficiently. We can reduce the number of instructions executed. e.g., we can load an IP or TCP header in one instruction, and the memory latency can be hidden by switching to other threads. We observed that processing a data packet (only the packet rewriting part, not including receiving and transmitting) in splicing state is about 6.5 μs for IXP compared to 13.6 μs for Linux.

Note that the above advantages are obtained by comparing with commonly used protocol stack. Although there exist optimized network protocol implementations which use polling instead of interrupt, our design still has the latter two advantages.

4.3 Throughput

We measure the throughput achieved by these two switches by sending requests of a uniform size as fast as possible from the clients. Figure 7 shows the results. We can see that the throughput is increased by 5.7x for small size requests like 1KB (8.2 Mbps to 46.4 Mbps). For a much larger file size like 1024KB, the improvement is 2.2x. Requests for small files have higher improvement because control packets take a larger portion in HTTP session for small files. Since the latency reduction for control packets is larger than that of data packets, the improvement is more apparent for small requests. As we increase the request file size, data packet processing becomes dominant, thus we see

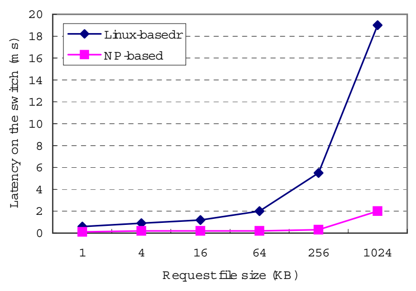

Figure 6. Latency comparison for an HTTP session

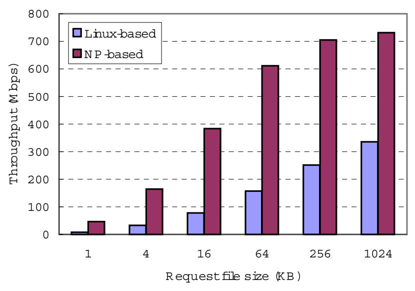

Figure 7. Throughput comparison for HTTP sessions

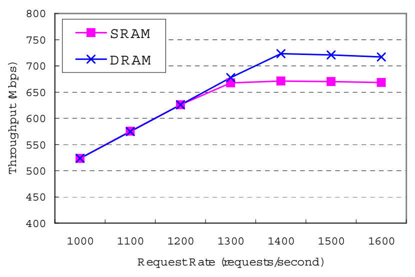

Figure 8. Throughput comparison for control blocks implemented in SRAM and DRAM

relatively smaller improvement on SpliceNP. It is noticed here that we use only one clientME and one serverME to process the packets, as shown in Figure 4. The throughput may be further improved by using more microengines in the IXP2400.

4.4 SRAM vs. DRAM

The previous results are obtained by maintaining the control blocks in SRAM. Hash tables that help fast table lookup for control blocks are also maintained in SRAM. In addition, locks are implemented in SRAM too. Therefore, for each packet to access its control block, there are at least three contiguous SRAM accesses: one for the lock, one for the hash table and another for the control block. When thousands of connections are processed simultaneously, these SRAM accesses can become a bottleneck. Also, maintaining a large number of control blocks in SRAM is not possible due to its size limitation. Therefore we measure the performance when maintaining the control blocks in the DRAM but keeping the locks and hash tables in SRAM. In this way, the memory accesses can also be distributed more evenly into the SRAM and DRAM modules, and their accesses can be pipelined. Further, since DRAM is much larger than SRAM, this allows us to increase the number of control blocks so that the switch can support many more connections simultaneously. It is found that the latency obtained on the client side is the same when control blocks are implemented in the SRAM or DRAM. Although DRAM latency is longer than that of SRAM (120 cycles vs. 90 cycles), the difference in http latency is negligible because of the waiting time for SRAM when all the tables are implemented in SRAM.

We then measure the throughput when the control blocks are implemented in DRAM and SRAM as a function of the request rate. We also increase the number of servers to 2 so that more requests are satisfied. Figure 8 shows the results when we fix the request file size at 64 Kbytes. The x-axis is the request rate in the unit of requests per second (or connections per second as we send one request in one connection). As we increase the request rate, the inter-arrival time between packets is reduced accordingly. When the request rate is increased to 1300, the throughput saturates at 665.6Mbps when control blocks are implemented in SRAM. However, with control blocks in DRAM, the throughput keeps increasing until 720.9Mbps. This verifies that the throughput can be increased by distributing memory requests to as many modules as possible.

5 Related Work

Content aware switches have been studied extensively. Cohen et al. [4] implement a content-aware switch in Linux using TCP splicing. They use an application level proxy to determine the destination server based on the clients' requests. Yang et al. [20] further moves all the processing down to the Linux kernel, so that the data forwarding as well as the routing decision are all performed in the kernel level. This can avoid the overhead of passing the HTTP request packet through the protocol stack to the user level proxy as in [4]. Our approach, implemented on NPs, moves the whole processing further down to the NIC level, thus reduces the end-to-end latency as much as possible. G. Apostolopoulos et al. [2] build a content-aware switch based on a switch core with custom built intelligent port controllers and a PowerPC processor. As an ASIC design, this switch can achieve very high throughput. However, it can hardly be extended to incorporate new services such as QoS scheduling. In addition to ASICs, FPGAs can be used to speed up pattern matching process in the content-aware switches [18]. However, they have higher power consumption compared to NPs.

Tammo Spalink et al. [16] suggest that TCP splicing processing be separated on a data forwarder and a control forwarder, which run on the IXP2400 microengines and the host processor (a Pentium), respectively. However, our analysis shows that performing all the processing on the microengines gives better performance. Therefore, not only data forwarder, but also the control forwarder are put on the microengines.

Besides TCP splicing, TCP handoff [12] is another mechanism to build a content-aware switch. It allows the response from the server to reach the client directly without going through the switch, so that the switch's load can be reduced. However this approach requires that the TCP state machine in servers' operating system be modified. This would be impractical to large scale server clusters. Papathanasiou et al. [13] exploit both the TCP splicing and hand-off techniques on a web switch. The switch performs TCP splicing whereas back-end servers perform the handoff operation. Their approach requires that a proxy application runs on each of the back-end servers, though no modification is required to the operating system.

6 Conclusions and Future Work

In this paper, we designed and implemented a content aware switch on a network processor - Intel's IXP2400. We analyzed various tradeoffs in implementation and compared the performance of this NP-based switch with the Linux-based one. Our experimental results showed that the processing latency of the NP-based switch is reduced by about 83.3% for a 1K byte web page. It also showed that the throughput can be improved by up to 5.7x.

Our future work includes processing all the TCP options in the network processor since they can affect the performance. We plan to further breakdown the functionality of the ClientME/ServerME and assign them to more MEs, so that the processing can be parallelized and pipelined to improve the throughput. In addition, we plan to incorporate other functionalities such as Quality of Service (QoS) by identifying the packet flows and providing differentiated service to an individual flow.

Acknowledgement

This work is supported by NSF grant and Intel Corp.

References

[1] Apache Software Foundation, http://www.apache.org.

[2] G. Apostolopoulos, et.al, Design, Implementation and Performance of a Content-Based Switch, in Proc. IEEE INFOCOM'00.

[3] Cisco Systems, Cisco Content ServicesSwitch, http://www.cisco.com/en/US/products/hw/contnetw/ps789/prod_models_home.html

[4] A. Cohe, S. Rangarajan, H. Slye, On the Performance of TCP Splicing for URL-Aware Redirection. In Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems, Boulder, CO, Oct. 1999

[5] Erik J. Johnson and Aaron R. Kunze, IXP 1200 Programming The Microengine Coding Guide for the Intel IXP2400 network Processor Family, Intel Press

[6] Foundry Systems, Foundry ServerIron XL/G, http://www.b2net.co.uk/foundry/foundry_serveriron_xlg_web_switch.htm

[7] IBM, IBM WebSphere Edge Server, http://www.ibm.com/software/webservers/edgeserver/

[8] Linux Virtual Server Project, http://www.linuxvirtualserver.org

[9] D. Mosberger and T. Jin, HP Research Labs A Tool for Measuring Web Server Performance, 1998

[10] D. Maltz, P. Bhagwat, TCP Splicing for Application Layer Proxy Performance, IBM Research Report RC 21139, 1998

[11] Nortel Networks, Alteon Web Switches, http://www.nortelnetworks.com/products/01/alteon/webswitch/index.html

[12] V. Pai, et. al, Locality-Aware Request Distribution in Cluster-based Network Servers. In Proc. ASPLOS'98

[13] A. Papathanasiou, E. Hensbergen, KNITS: Switch-based Connection Hand-off, 21st Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, Volume 1, 2002

[14] RFC1624: Computation of the Internet Checksum via Incremental Update, May 1994

[15] M. Rosu, D. Rosu, Kernel Support for Faster Web Proxies, USENIX Annual Technical Conference, June 2003

[16] T. Spalink, S. Karlin, L. Peterson, Y. Gottlieb, Building a Robust Software-Based Router Using Network Processors, Proceedings of the eighteenth ACM symposium on Operating systems principles, pages 216 - 229, 2001

[17] Oliver Spatscheck, et. al, Optimizing TCP Forwarder Performance, IEEE/ACM Transactions on Networking, 2000

[18] Tarari Inc., Regular Expression Content Processor, http://www.tarari.com/regexEAP/index.html

[19] The Linux Kernel Archives, http://www.kernel.org

[20] C. Yang and M. Luo, Efficient Support for Content-Based Routing in Web Server Clusters. In Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems, Boulder, CO, Oct. 1999