518

Modeling the Performance of a NAT/Firewall Network Service for the IXP2400

Tom Verdickt
Department of Information Technology (INTEC)
Ghent University - IMEC
Sint-Pietersnieuwstraat 41
B-9000 Gent, Belgium

Wim Van de Meerssche
Department of Information Technology (INTEC)
Ghent University - IMEC
Sint-Pietersnieuwstraat 41
B-9000 Gent, Belgium

Koert Vlaeminck
Department of Information Technology (INTEC)
Ghent University - IMEC
Sint-Pietersnieuwstraat 41
B-9000 Gent, Belgium

tom.verdickt@intec.UGent.be

ABSTRACT

The evolution towards IP-aware access networks creates the possibility (and, indeed, the desirability) of additional network services, like firewalling or NAT, integrated into the network devices. These new services, however, force the network components to be both flexible (to cope with changing protocols and applications) and powerful. Network processors as a platform on which to implement the network services seem to fit the bill.

System performance should be assured by incorporating performance analysis into the design of the system, by means of performance modeling at the architectural design stage. This paper describes the use of Software Performance Engineering during the design of a firewall/NAT router on the Intel IXP2400 network processor. Several design options were first modeled and analysed, and based on those simulations, a final design was chosen and implemented.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems; C.5 [**Computer Systems Organization**]: Computer System Implementation; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Performance, Design

Keywords

Firewall, network processor

1. INTRODUCTION

Current (DSL-based) access networks (connecting users to the access server of the service provider on the edge of its network and thus to the internet) use the Point to Point Protocol (PPP) over an Asynchronous Transfer Mode (ATM) network. Many providers are, however, considering the possibility to migrate their ATM infrastructure to an Ethernet-based access infrastructure, which is claimed cheaper and easier to install and maintain. [5]

IP-awareness in the access network would allow network components to provide additional services, even at layer 3 (network layer) or higher of the OSI stack. Examples of such value-added services are NAT, firewalling, virus scanning, intrusion detection, etc.

The growing availability of network bandwidth (e.g. more and more homes have a broadband internet connection) has given rise to a whole range of resource-intensive applications, e.g. streaming video, voice over IP, etc. [4] In order to handle these new applications, network components need to be powerful, yet flexible enough to adapt to new and changing protocols and services.

Because performance is still a major concern, it is advantageous to incorporate performance analysis into the design of the network devices, by means of performance engineering and modeling. This paper will describe the use of performance modeling in the design of a firewall/NAT router on an Intel IXP2400 network processor.

1.1 Network Processors and the IXP2400

Network processors seem the right hardware for IP-aware access networks. Being programmable devices, with a design optimized for packet processing, they are more flexible than ASICs (application specific integrated circuits), yet provide a higher "performance to power consumption" ratio than general-purpose processors. An example of a network processor (used in this paper) is the Intel IXP2400 [2].

The IXP2400 contains a.o. 8 programmable, multi-threaded microengines ("processors") with instruction sets optimized for packet processing, interfaces to DRAM and SRAM memory banks, etc. The microengines are interconnected directly, and can also communicate using the shared memory. Figure 1 shows the testing platform used in this paper: the RadiSys ENP-2611, which is an Intel IXP2400 network processor based PCI board [1]. Table 1 lists some of the most

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WOSP'05, July 12-14, 2005, Palma de Mallorca , Spain

Copyright 2005 ACM 1-59593-087-6/05/0007 ...$5.00

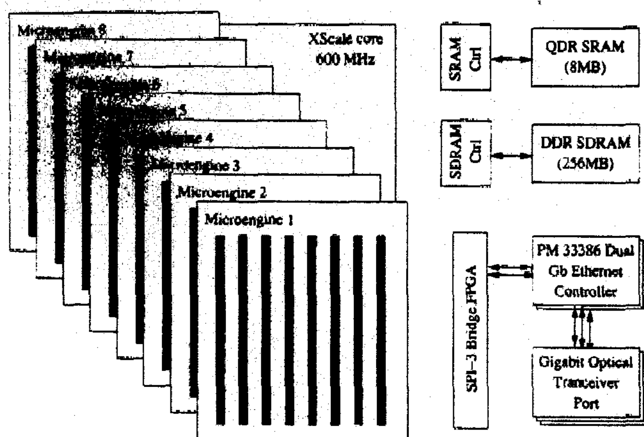


Figure 1: The RadiSys ENP-2611, containing the Intel IXP2400

important performance details of the IXP2400 network processor [2].

1.2 Layered Queueing Networks

The *Layered Queueing Network* (LQN) formalism [6] is an extension to the well-known and widely used queueing network formalism. It adds a concept of nested services to queueing networks, by allowing servers to become clients of another server. This introduces a form of layering into the models. The layering is not strict, however. Clients can use servers in the same layer as the client, calls can skip several layers, etc.

The nodes of the network are either *tasks* or *processors*. They are interconnected by arcs, representing service requests. Tasks represent the "functional parts" of the system. Depending on the granularity of the model, this can range from large software components (a database), to individual functions or even hardware components (e.g. a network, modeled as a delay for sending packets). Processors are used to model the hardware itself, e.g. the actual processors in the system. Every task is associated with a processor.

A task contains one or more *entries*, representing the actual services provided by the task. Entries consist of one or more phases, each with a certain *execution time* (the load on the associated processor), making calls to other tasks, etc. In a server task entry, a response is sent to the client after the first phase, so all further phases represent a form of post-processing.

Service requests can be either synchronous (after sending the request, the client blocks until it receives the reply) or asynchronous. A third form of request is the "forward chain". In that scenario, a server processes (part of) the request, but instead of replying to the client, it forwards the request to another server, which in turn processes (part of) the request, and might also forward the request, etc. In the end, the last server sends the reply to the client (which is different from a scenario with only synchronous and asynchronous requests, where it is always the originally called server that replies). In a forward chain, a server does not block after forwarding a request.

Servers and processors have a single, infinite queue, in which arriving requests can be stored before processing. Tasks can be single-threaded or multi-threaded, indicating the number of requests that can be processed concurrently.

A number of tools exists, including analytic solvers and simulators, to obtain performance estimates from a given LQN model [3]. Results allow to estimate execution times, synchronisation delays for a given request, troughputs of the individual components, processor load, etc.

This paper is structured as follows. Section 2 will provide an overview of the general architecture of the firewall/NAT router application. Section 3 will describe the first design attempt, and the results of the performance modeling. An improved design will be presented in section 4, together with the estimated performance. In section 5, the actual application implementation will be described, and its performance will be compared to the estimates derived from the model. Section 6 will show how modeling can be used to achieve an even better performance, and finally section 7 will provide the conclusions of the modeling and the system design.

2. GENERAL APPLICATION DESIGN

The application to be designed is a combination of a firewall and a Network Address [and Port] Translation (NA[P]T) router.

The firewall part of the application is a stateful firewall, capable of filtering on source and destination IP address, connection status (new or established), L3 protocol, source and destination port, input and output interface and TCP flags. The NAPT part includes masquerading and port forwarding. It uses the firewall code to select the packets on which to apply NAT.

The design of the application is based on the Netfilter[1] architecture, with some modifications and simplifications. The architecture of the application is shown in figure 2. A packet first passes through the connection tracking code. Packets of established connections are matched with the related conntrack structure in memory. Packets of new connections cause such a structure to be created and initialized with reply packet headers and NAT information. Next the packet goes through the "source NAT" code, before being passed to the router code. After the routing the packet is further processed by the firewall code, followed by the "destination NAT" code.

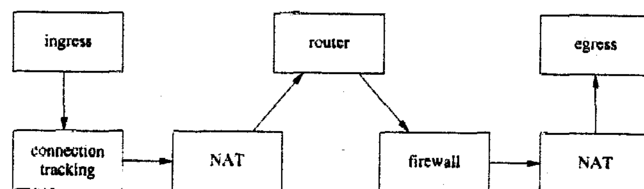


Figure 2: The firewalling netservice architecture

The firewall part of Netfilter filters the incoming packets (accepting some and dropping others), using 3 list of rules, or "chains": INPUT, OUTPUT and FORWARD. If the packet

[1] http://www.netfilter.org/

Table 1: IXP2400 performance details

[illegible] operating frequency	600 MHz 32-bit data path
[illegible] core operating frequency	600 MHz
[illegible]ce (QDR) Two Channels	Peak bandwidth of 1.6 Gbps per channel at frequency of 200 MHz (800 Mbytes/sec read, 800 Mbytes/ sec write); up to 64 Mbytes per channel;
DDR DRAM One Channel	Peak bandwidth of 2.4 Gbytes (19.2 Gbps) per channel at frequency of 150 MHz; up to 2 Gbytes memory; ECC protected; 64-bit wide interface

is destined for the computer running netfilter, the INPUT chain is used. If it must be forwarded on another network interface, FORWARD is used (e.g. in routers). The OUTPUT chain is used for packets which are sent by local processes.

Each chain consists of a set of "firewall rules". A rule specifies what type of packets it applies to, and how packets of that type should be handled (e.g. accept or drop it). Examples of packet characteristics that can be specified are source and destination IP address, protocol (TCP, UDP, ICMP, etc.), input or output network interface, TCP flags, etc. Possible "targets" of a rule (what should be done with a packet that matches the rule) include ACCEPT, DROP, or send to another (user-defined) chain.

3. SINGLE-CHAINED FIREWALL IMPLEMENTATION

After deciding on the architecture, the deployment of the various components on the hardware was still not fixed. It could, however, have a great influence on the performance of the system, by introducing or removing potential bottlenecks. Therefore, a preliminary performance analysis was performed on several possible deployments.

For all candidate-deployments, an LQN performance model was constructed. This model was then solved in order to obtain an estimate for the performance of the system as a whole and of the components individually, including system throughput and average load for the microengines. These performance estimates were used to evaluate the proposed system and its deployment and to locate possible bottlenecks. Once the bottlenecks were located, a new component deployment was created, once again using the performance estimates to find solutions for the previous performance problems.

The first idea was a very simple one (figure 3). Every Netfilter component, as shown in figure 2, is implemented on a different microengine. The eighth microengine is used as a resource manager, allocating and freeing memory, keeping track of used local ports for NAT, and freeing **conntrack** structures if no new packets arrive on their connection for a certain time period.

With this mapping of the application on the microengines, the first microengine receives packets from one of the 3 hardware ports. It stores the packets in the DRAM memory and pushes their handle on the ring to the second microengine. The second microengine does connection tracking, fetching the known connections from memory and matching them to

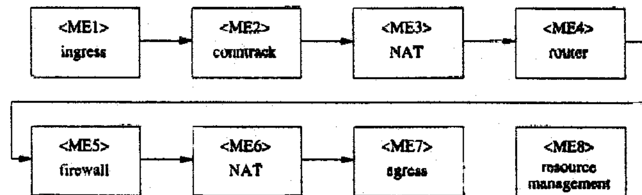


Figure 3: First deployment proposal

the packet. If the packet is from an unknown connection, a new **conntrack** object is created and written to SRAM memory. The packet handle is then pushed onto the ring to microengine 3, which does the pre-routing NAT, reading the necessary parts of the packet header from the DRAM memory and writing back the changes. It forwards the handle to microengine 4. Microengine 4 executes the actual routing code, after which the packet is forwarded to the firewall (microengine 5), which reads the necessary parts of the packet header from the DRAM memory and one by one fetches the firewall rules from the SRAM memory, trying to match the rule to the packet. When a matching rule is found, the firewall accepts or drops the packet (as the rule prescribes), and if it accepts a package, it sends the packet to microengine 6, which executes the post-routing NAT, and pushes the packet handle on the appropriate transmit queue (1 queue for each port). The 7th microengine transmits the packets from these queues (after reading them from the memory).

The firewall rules are stored in SRAM memory, and can be modified while the firewall is running. This means however that they need to be read (from memory) for each arriving packet. This was identified as a potential performance bottleneck.

In order to assess the performance of this system, an LQN performance model was constructed. This was done by a more or less straightforward "translation" of the system design shown in figure 3. LQN tasks were created for all the design components, each running on one of the 8 processors representing the 8 microengines of the IXP2400. Tasks were made multi-threaded, with 8 threads each, representing the fact that each microengine has 8 hardware threads, possibly handling 8 packets concurrently.

The performance of the model components (their execution times) were estimated by estimating the size of the microcode (number of instructions) needed to implement the functionality of that component. An overview of the code size estimates is given in table 2. Additionally, arriv-

[illegible] 2: Estimated microcode sizes

[illegible]ment	Instructions
[illegible]	310
[illegible]n Tracking [illegible]table searches + per new connection	125 +3 * 30 +500
pre-routing NAT	200
routing (bridging)	10[2]
firewall + process firewall rules	220 +100 per firewall rule
post-routing NAT	200
egress	310
SRAM access	3[3]
DRAM access	8[3]

ing packets were modeled to have a size of 64 bytes, with on average one packet arriving every $0.768 \mu s$ (which amounts to 1,302,083 packets per second, the maximum rate for IPv4 packets over gigabit Ethernet).

From the model, the throughput of the system was calculated as a function of the number of rules in the firewall, by using the LQNS analytic solver. Since the firewall rules are retrieved from the memory one by one until a rule is found that matches the packet in the firewall, the performance of the firewall decreases when more rules need to be applied before a matching rule is found.

The result of the performance analysis is given in figure 4. Maximum system throughput (1,302,083 64-byte IP packets per second) can be achieved when only a single firewall rule needs to be fetched from memory, and thus when the first firewall rule always matches the packet. However, in a realistic scenario, more firewall rules will be needed on average, causing the performance of the system to drop dramatically to unacceptable throughput levels. Further examination of the modeling results indicates the firewall component as the bottleneck, quickly reaching maximum load (see figure 5).

4. MULTIPLE CONCURRENT FIREWALL CLONES

In order to relieve the load on the firewall component (trying to remove the bottleneck from the system), another approach was taken in the second attempt to design the system.

Re-examining the modeling results showed that the NAT-code was estimated to be fairly short, compared to the other components (except perhaps the routing code), which was also reflected in the estimated processor load for the microengines running the source and destination NAT components, which was significantly lower than for the other microengines. Therefore, the source NAT code was merged with the routing code and the destination NAT was integrated

[2]The initial implementation was only intended to perform bridging. True routing would take much more instructions

[3]Memory access requires only 1 microcode-instruction, but this is the number of microengine clock ticks, and thus the number of microcode command execution times, 1 memory access takes.

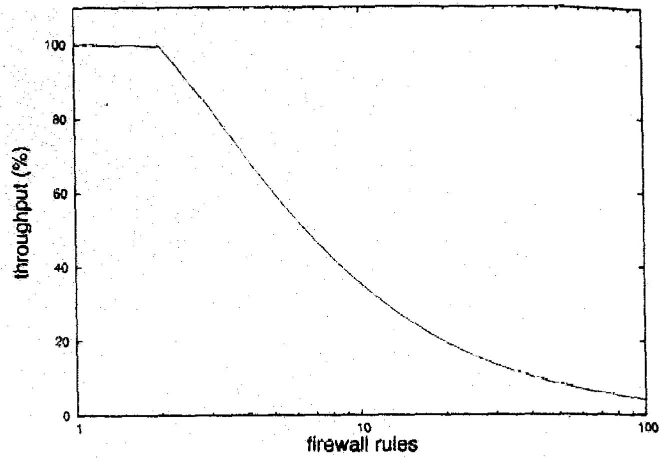


Figure 4: Estimated performance of the first architecture proposal

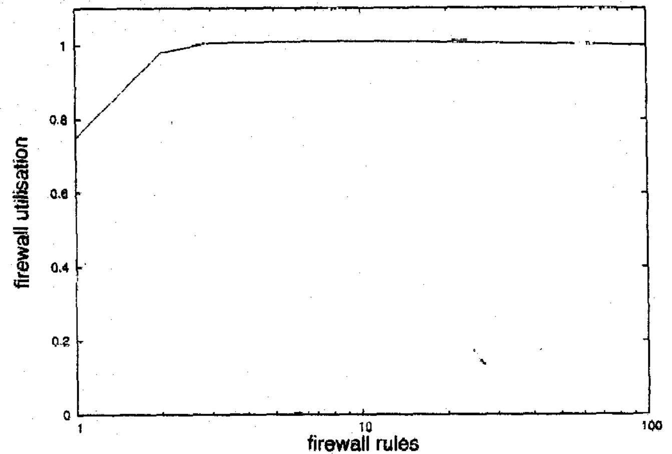


Figure 5: Single firewall utilisation

into the firewall. While this made the firewall component even larger (and thus slower), it also freed up 2 microengines. The next step, then, was to clone the firewall/NAT code on those extra microengines. Thus, the fact that each firewall component was slower than the firewall in the first architecture should be (more than) compensated by the ability to run three firewalls in parallel.

This resulted in the deployment shown in figure 6. Packets leaving routing are sent to an idle firewall component, if one or more firewall components are idle. If all firewalls are busy, the routing component leaves the packet handle on a shared "ring", monitored by the firewall microengines. When a firewall thread has finished processing a packet, it will fetch a new packet from the ring to process next. Only when no more packets are waiting to be processed by the firewall, will the thread go to sleep.

Once again, a performance model of the system was created. This model is presented in figure 7. Processors (microengines) are shown as circles, and tasks are represented as parallelograms, with arrows connecting them to their processors. Solid arrows between tasks represent synchronous calls, while dashed arrows are used for call forwarding.

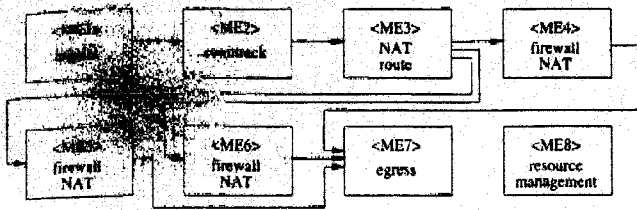


Figure 6: Multiple parallel firewall components

Each task consists of 8 threads, except for the bus and memory tasks, and the `firewall` multi-task. This represents the multiple hardware threads of the microengines. The `firewall` task, on the other hand, has 24 threads and is associated with a multi-processor (consisting of 3 processors), representing the "pool" of firewalls executing on 3 separate microengines, accepting packets from the `routing` task.

As explained earlier, the NAT code is integrated into the `routing` and `firewall` tasks. The tasks `lookup`, `add new` and `handleFWline` are added for modeling convenience. They allow e.g. to easily model the rate at which new connections need to be created (because a packet from a previously unknown connection arrives), or to clearly separate the number of firewall rules to be fetched from memory and the number of memory reads needed to fetch a single rule, instead of just using a single number to represent the number of memory reads to fetch the entire set of rules. This way, changing performance attributes of the model (e.g. the number of firewall rules or the average packet size) is much easier and less error-prone.

The model of the final system architecture (figure 7) was solved for a varying percentage of the arriving packets belonging to a new connection. This could influence the performance, since a new conntrack object needs to be created and stored by the `conntrack` component for every new connection. The estimated throughput of the system is shown in figure 8.

As can be seen in the figure, a bottleneck still causes the throughput to drop, but only when more than 10 firewall rules need to be examined on average, which is a considerable improvement over the first system design. On the other hand, the arrival rate of packets from new connections has no real influence on the system performance.

Examination of the firewall utilisation in the model shows that, once again, the firewalls form the bottleneck (figure 9, firewall load reaches 100% when more than 10 firewall rules need to be evaluated). Since the firewall load is not directly influenced by the arrival of a packet from a new connection, only the load for 10% new connections is shown.

5. ACTUAL SYSTEM PERFORMANCE

The performance of the second design, with 3 firewall components working in parallel, was deemed "good enough", so a proof-of-concept implementation was made. The same performance attributes were measured as were simulated in the model, namely the throughput of the device (the firewall/NAT router) for a rising number of firewall rule evaluations per packet.

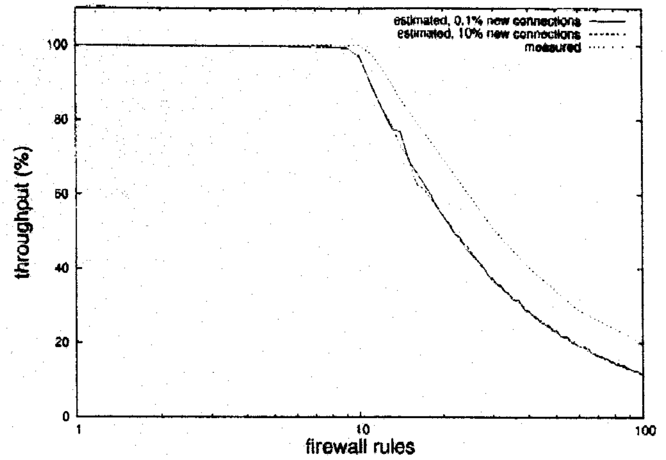


Figure 8: Estimated performance of the final architecture and of the actual implementation

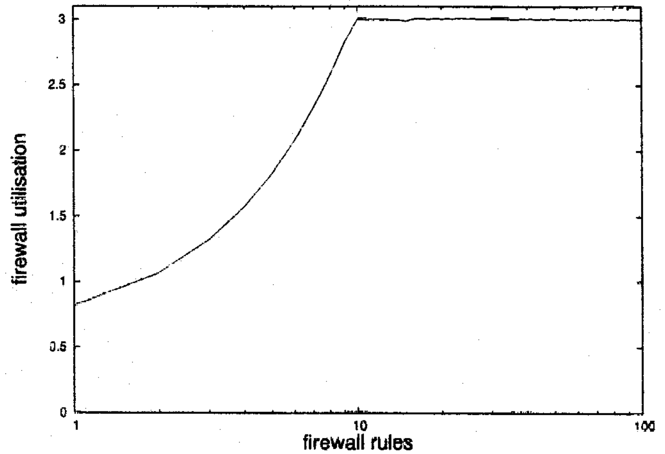


Figure 9: Firewall utilisation with 3 firewalls

Using the Smartbits 6000 networktester, several tests were done to measure latency and throughput. All tests were performed using a 76 byte frame length, which results in a 64 byte IP packet. The result can be found in figure 8.

The performance behaviour of the actual system is very similar to the performance predicted by the model. When few firewall rules need to be examined before a decision is reached on whether to allow a packet or not, the device can function at the maximum bandwidth for IPv4 packets over gigabit Ethernet (1,302,083 packets per second).

The performance estimates stay accurate when more firewall rules are needed, although the model predicted that the system performance would degrade slightly earlier than was actually measured. This is caused by an over-estimation of the necessary processing to handle a single packet in the firewall component.

6. FURTHER IMPROVEMENTS

Further inspection of the performance analysis results of the last model (with 3 firewall components) shows that the NAT/connection tracking and the routing code could be combined on a single microengine, freeing the other micro-

Figure 7: The LQN model of the NAT/firewall on the IXP2400

engine to be used by a fourth firewall clone. This boosts performance even further.

Of course, as mentionned in a footnote to table 2, the microcode size estimates only took bridging into account. True routing would take more microcode instructions, and thus would decrease the performance of the routing component. That might make combining the NAT/connection tracking and routing code less beneficial, or might even decrease overall performance.

This design improves the performance over the 3-firewall design, but the gain is not really significant (especially taking into account the fact the gain would be even further diminished by implementing true routing). Maximum throughput can be sustained until 13 firewall rules are needed (see figure 10), compared to 10 rules in the implemented architecture. Further combining of system components would not be beneficial, as it would generate new performance bottlenecks.

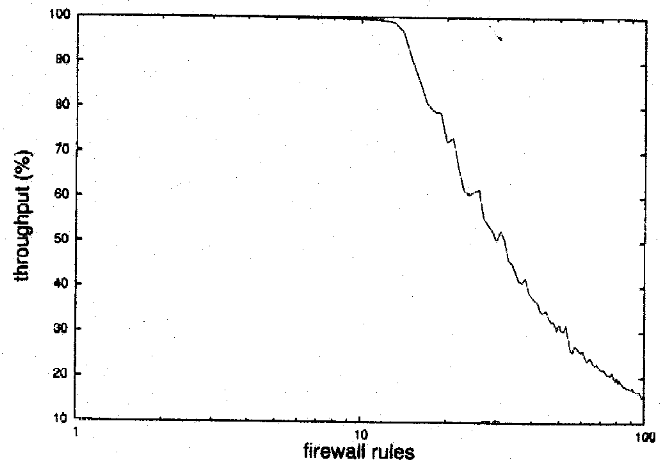


Figure 10: Estimated performance of the architecture with 4 firewalls

7. CONCLUSIONS

This paper shows that performance analysis during the architectural design phase can be used to good effect in order to design a system making maximum use of the available resources, increasing the performance over less efficient designs.

The design and modeling of a firewall NAT router on the IXP2400 network processor was presented. An iterative design methodology was used, by modeling the system, extracting a performance estimate from it (based on rough estimates of the performance of the components), finding the major bottleneck and redesigning the system to improve performance.

When comparing the performance estimates to the measured performance of the actual system, the models proved to provide important design hints, predicting the occurrence of bottlenecks and their influence on the throughput with very high accuracy.

8. [illegible]MENTS

T[illegible] financed by a Ph.D. grant of the I[illegible] of Innovation through Science an[illegible] (IWT-Vlaanderen).

The au[illegible] like to thank the Real-Time and Distributed Systems Group at Carleton University in Ottawa, Canada for their support on LQN modelling and for providing the tools to work with LQN models.

Filip De Turck acknowledges the F.W.O.-V. (Fund for Scientific Research-Flanders) for their support through a post-doctoral fellowship.

9. ADDITIONAL AUTHORS

Additional authors: Tim Stevens, Filip De Turck, Bart Dhoedt and Piet Demeester (Department of Information Technology (INTEC), Ghent University - IMEC)

10. REFERENCES

[1] *ENP-2611 DATA SHEET.* http://www.radisys.com/oem_products/ds-page.cfm?productdatasheetsid=1147.

[2] *Intel IXP2400 Network Processor.* ftp://download.intel.com/design/network/prodbrf/27905302.pdf.

[3] G. Franks, A. Hubbard, S. Majumdar, J. Neilson, C. Petriu, J. Rolia, and M. Woodside. A toolset for performance engineering and software design of client-server systems. *Performance Evaluation*, 24(1–2):117–135, February 1995.

[4] N. Shah. Understanding network processors. Master's thesis, University of California, Berkeley, 2001.

[5] K. Vlaeminck, T. Stevens, S. De Maesschalck, F. De Turck, B. Dhoedt, P. Demeester, T. Vermeiren, M. Pelt, S. Miclea, E. Borghs, and P. Vetter. Deployment of network processors in access networks to provide service enabling functions: evaluation results. In *Proc. of the* 9th *European Conference on Networks & Optical Communications*, pages 70–77, 2004.

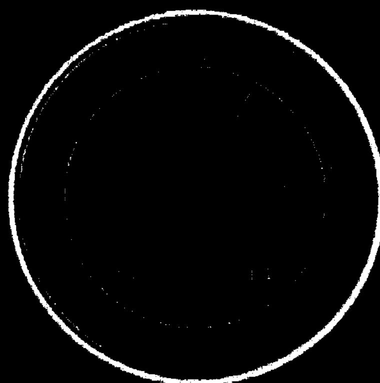
[6] M. Woodside and G. Franks. *Tutorial Introduction to Layered Modeling of Software Performance.* Ottawa, Canada, August 2004.

Proceedings of the

Fifth International Workshop on Software and Performance

WOSP'05

Palma, Illes Balears, Spain
July 12-14, 2005



Sponsored by
Association for Computing Machinery
SIGMETRICS and SIGSOFT
in cooperation with
Computer Measurement Group (CMG)
IFIP WG 6.3 and 7.3

The Association for Computing Machinery
1515 Broadway
New York, New York 10036

Copyright © 2005 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of portions of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permission to republish from: Publications Dept., ACM, Inc. Fax +1 (212) 869-0481 or <permissions@acm.org>.

For other copying of articles that carry a code at the bottom of the first or last page, copying is permitted provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923.

Notice to Past Authors of ACM-Published Articles

ACM intends to create a complete electronic archive of all articles and/or other material previously published by ACM. If you have written a work that has been previously published by ACM in any journal or conference proceedings prior to 1978, or any SIG Newsletter at any time, and you do NOT want this work to appear in the ACM Digital Library, please inform permissions@acm.org, stating the title of the work, the author(s), and where and when published.

ISBN: 1-59593-087-6

Additional copies may be ordered prepaid from:

ACM Order Department
PO Box 11405
New York, NY 10286-1405

Phone: 1-800-342-6626
(US and Canada)
+1-212-626-0500
(all other countries)
Fax: +1-212-944-1318
E-mail: acmhelp@acm.org

ACM Order Number 488053
Printed in the USA