SANDIA REPORT
SAND2011-7834
Unlimited Release
September, 2011

Investigating the Effectiveness of Many-core Network Processors for High Performance Cyber Protection Systems (PART I – FY2011)

Robert E. Benner, Joshua A. Johnson, John H. Naegle, Uzoma A. Onunkwo, Jay Patel, David Pearson, Jeffrey S. Shelburg, Kyle B. Wheeler, Brian J. Wright, David J. Zage

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

ıssued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd.
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2011-7834
Unlimited Release
September 2011

Investigating the Effectiveness of Many-core Network Processors for High Performance Cyber Protection Systems (PART I – FY 2011)

Robert E. Benner (1422), Joshua A. Johnson (5641), John H. Naegle (9336), Uzoma A. Onunkwo (9336), Jay Patel (5627), David B. Pearson (9336), Jeffrey S. Shelburg (5635), Kyle B. Wheeler (1423), Brian J. Wright (9516), David J. Zage (9516)

Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185

Abstract

This report documents our first year efforts to address the use of many-core processors for high performance cyber protection. As the demands grow for higher bandwidth (beyond 1 Gbits/sec) on network connections, the need to provide faster and more efficient solution to cyber security grows. Fortunately, in recent years, the development of many-core network processors have seen increased interest. Prior working experiences with many-core processors have led us to investigate its effectiveness for cyber protection tools, with particular emphasis on high performance firewalls.

Although advanced algorithms for smarter cyber protection of high-speed network traffic are being developed, these advanced analysis techniques require significantly more computational capabilities than static techniques. Moreover, many locations where cyber protections are deployed have limited power, space and cooling resources. This makes the use of traditionally large computing systems impractical for the front-end systems that process large network streams; hence, the drive for this study which could potentially yield a highly reconfigurable and rapidly scalable solution.

ACKNOWLEDGMENTS

The authors would like to acknowledge that the work, which produced the results presented in this paper, was funded by the U.S. Department of Energy under Sandia's Laboratory-Directed Research and Development (LDRD) Program.

We also acknowledge the contributions of the following individuals, who influenced the direction, focus, and success of this research:

Tim Berg (Project Manager, 9336)

Theresa Keener (10694)

Linda Bonnefoy-Lev (10694)

Keith Vanderveen (8961)

Roger Suppona (9317)

Tan Thai (5630)

We are also grateful for the great laboratory and experimental support provided by Diana Eichert (9338).

CONTENTS

Glossary vii

1. Introduction to many-core processors and cyber protection 1
   1.1. Overview of many-core systems 1
   1.2. Overview of cyber protection tools 1

2. Stateless Firewall 3
   2.1. Design of firewall frontend 3
   2.2. Many-core implementation of stateless firewall processing 5
   2.3. Results 5

3. Stateful Firewall 8
   3.1. Packet distributor using fast hashing and ring buffers 8
   3.2. Extending iptables stateful firewall to many-core solution 10
   3.3. From Juniper NetScreen to Linux iptables 13
      3.3.1. Overview 13
      3.3.2. The code design process 13

4. Conclusion and future work 15

5. References 17

Appendix A: Methods to achieve putting a Linux-based firewall in passive-wire mode 19

Distribution (Electronic Copies): 24

FIGURES

Figure 1. Performance of pinned versus unpinned threads for raw-socket based passive-wire running on two cores of a TilePro (866 MHz) processor. 4
Figure 2. Performance of raw-socket based passive-wire; threads are all pinned to cores. 4
Figure 3. Packet processing rate as a function of number of rules and number of cores in a TilePro (866 MHz) processor. 6
Figure 4. Conceptual model for multithreaded packet processing on a standard Intel architecture 9
Figure 5. A multi-pipeline framework for stateful firewall processing between networks A and B 11

TABLES

No table of figures entries found.

NOMENCLATURE

CPU	Central Processing Unit of a processor
DOE	Department of Energy
FNV	Fowler-Noll-Vo non-cryptographic hash function
FTP	File Transfer Protocol
IP	Internet protocol (a network layer protocol)
NIC	Network Interface Card
NUMA	Non-Uniform Memory Access
SNL	Sandia National Laboratories
TCP	Transmission Control Protocol (a transmission layer protocol)
UMA	Uniform Memory Access

GLOSSARY

rule set	a collection of multiple firewall policies or chains
policy	a grouping of firewall rules to accomplish a specific goal within a Juniper firewall rule set
chain	a grouping of firewall rules to accomplish a specific goal within an iptables rule set
packet 5-tuple	a set of identifying items in a packet's header, namely (i) source IP address, (ii) destination IP address, (iii) source port, (iv) destination port, and (v) protocol

1. INTRODUCTION TO MANY-CORE PROCESSORS AND CYBER PROTECTION

1.1. Overview of many-core systems

Many-core processors have gained prominence ever since the demonstration of higher performance rates using the early dual-core processors on super computers [1]. However, these processors never made special provisions for standard use in network traffic analysis. Today, several teams have started exploring the applicability of new many-core processors for cyber protection with Tilera Corporation making a more direct solution optimized for network environments. In our study, we considered two primary many-core processors – the Intel-based many-core systems and the Tilera-based network-on-a-chip processors. Today, the Intel-based solution have significantly higher clocking rate, but smaller set of available cores partly due to its uniform memory access (UMA) architecture. On the other hand, the Tilera-based solutions have lower clocking rates, but a much larger set (64 cores) of available CPUs with non-uniform memory access (NUMA).

1.2. Overview of cyber protection tools

Although no practical network system is ever fully secure, cyber protection tools are meant to provide means to drastically reduce network security failures – which could include a non-functional network due to denial of service or an uninvited intruder access which can cause unfair compromises. Good network systems typically employ two main cyber protection tools, namely network intrusion detection (NID) systems and firewalls. The particular implementation of these tools can vary vastly based on algorithmic constraints.

In our work, we have focused on the target cyber application being a firewall. In actuality, we have addressed two forms of firewalls – stateless and stateful firewalls. Stateless firewalls are designed to allow or deny traffic based strictly on whether network packets pass a series of matching question on the 5-tuples of a packet. On the other hand, stateful firewalls keep track of states, which allow the firewall to identify "expected" behavior given not just the current condition but on its memory of previous network sessions that are active. Stateless firewalls have a known vulnerability known as spoofing, so one may think of them as not so enticing for strongly protected networks. However, in actuality, the use of stateless firewalls is recommended as long as they are combined with an advanced scheme (stateful firewall) which denies network access to bad actors. As a front-end, the stateless firewall prevents access to already-known bad actors and reduces the amount of input to the more compute-intensive stateful firewall.

In the sections that follow, we provide a description of the aspects of the firewall design we have focused on and a summary of our accomplishments in those areas. The design aspects include the design of an efficient front-end for packet I/O, the distribution of work load to many-core systems and how this is affected by the logistics of stateless versus stateful firewall.

2. STATELESS FIREWALL

2.1. Design of firewall frontend

The frontend of our firewall refers to the section of code that pulls network packets and gets them to their respective processing units for firewall filtering operation. As stateless firewalls do not have a notion of state but simply keeps track of just the current packet, it is important to note that packets can be processed out-of-order and by any core irrespective of whether packets belonging to that session went through a given processing core or not. The performance of a firewall, in terms of processing rate, will always be at most equal to the rate at which the frontend performs its functions.

Another logical decision we made in the frontend design is to set up a mode, we termed "passive-wire". This mode simply means that its existence between a corporate network and the outside network is not detectable by a regular user. In other words, its network interfaces are not viewable over any side of the network and must be managed by logging onto the console of the machine running the firewall.

To get a preliminary set of results on our performance, we initially performed the passive-wire experiments using raw sockets programming. Raw sockets in Linux machines allows a user to put network interfaces of a standard machine in promiscuous mode thereby allowing it to ingest any packet on the network it is connected to. We developed the threads-based program on a Tilera many-core system (TilePro processor) and showed the effects of explicitly pinning threads to cores. A thread is a light-weighted process that can serve as a logical process holding a set of functions. On a typical many-core system, threads can hop from one processing core to another during its execution time; this may be done by the operating system to emphasize fair share of CPU time on each processor. However, during this process of migration, a thread's program context keeps getting switched and can result in unnecessary overhead for high performance operations.

In our design of the stateless firewall, we have chosen a thread-based model with a necessitated pinning semantics. We illustrate the difference pinning makes by showing Figure 1, which contains a comparison of our passive-wire performance running on just two cores of the Tilera processor over its 1 Gbits/sec link. The tests were conducted with a packet generation tool known as *iperf*. The *iperf* program was invoked as follows:

Client: `iperf -c [server_IP_address] -M [packet_size_in_bytes]`
Server: `iperf –s`

Clearly, from Figure 1, we can see close to 60% increase in performance with packet size of one to two kilobytes when intentionally pinning threads to processing cores. In a similar setup, we performed the passive-wire experiment but with increasing number of threads to see how using more cores increase the throughput performance. The results are shown in Figure 2.

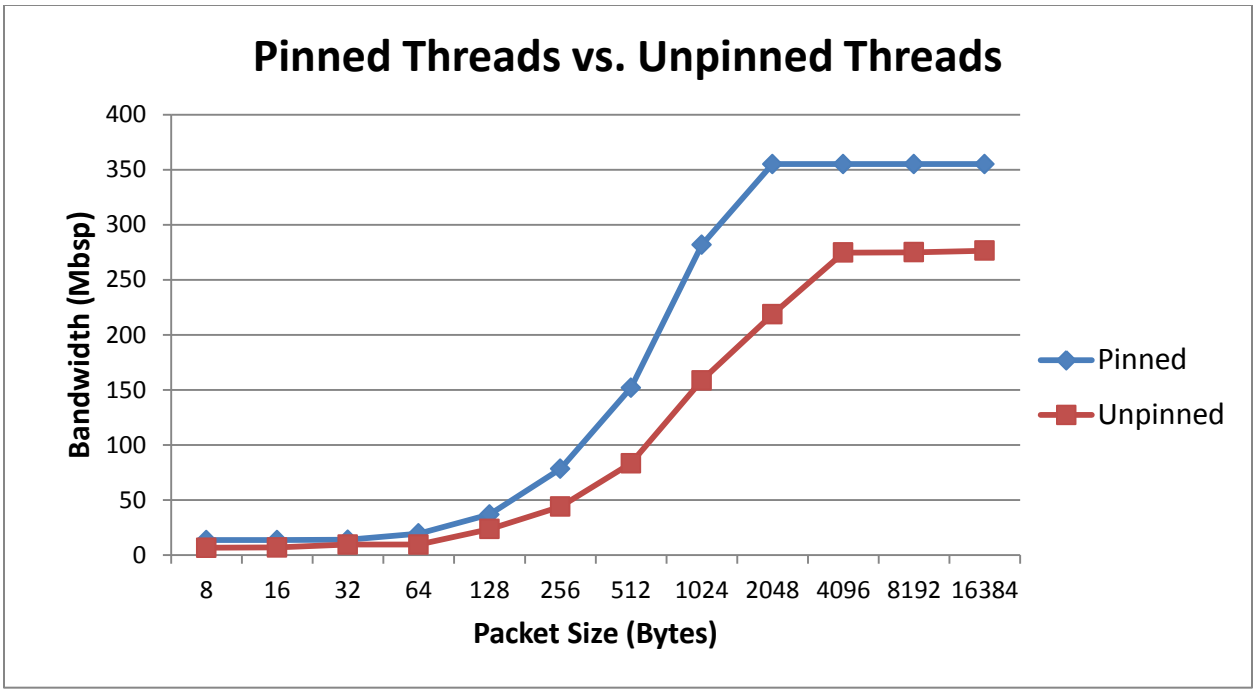

Figure 1. Performance of pinned versus unpinned threads for raw-socket based passive-wire running on two cores of a TilePro (866 MHz) processor.

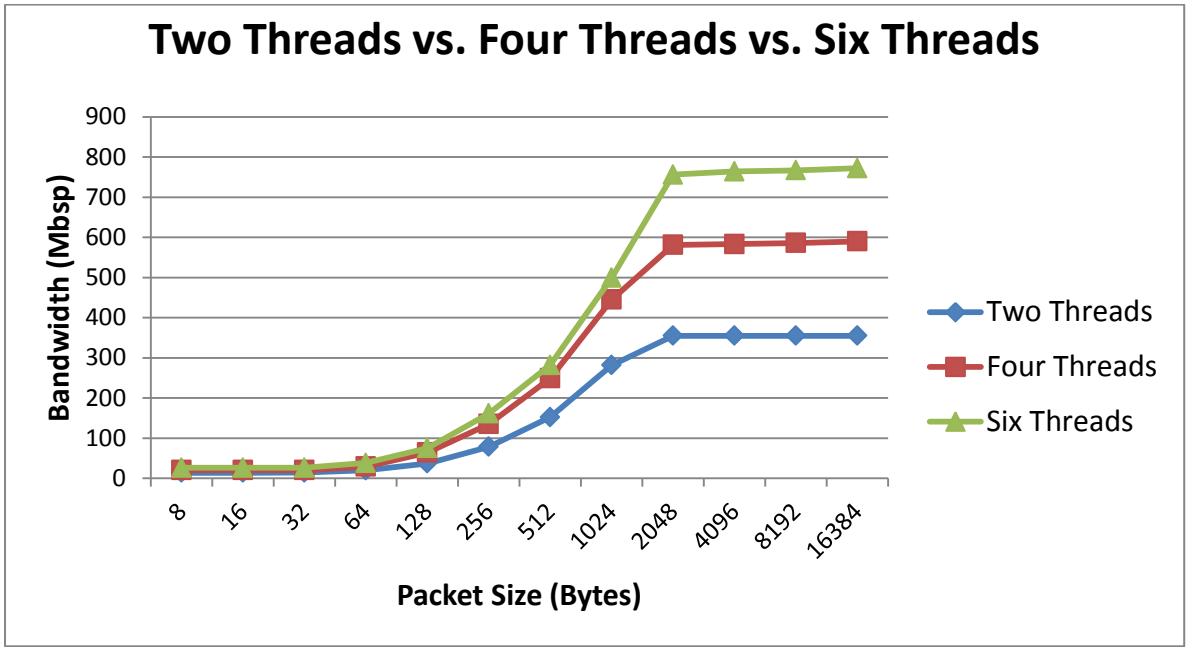

Figure 2. Performance of raw-socket based passive-wire; threads are all pinned to cores.

2.2. Many-core implementation of stateless firewall processing

This aspect covers the design and development used to drive towards a stateless firewall with high network processing rate. Our overall firewall system design is to have the stateless firewall serve as a preprocessor that eliminates known bad actors, thereby reducing processing overhead on the subsequent stateful firewall.

There were two principal design decisions that we made. One was the standard decision of how to arrange the firewall rules so that analyzing incoming traffic is done as quickly as possible. This entails the use of replication of rules on each processing core or the execution of a proper subset of rules on each processing core, with the union of all subsets constituting the full firewall rule sets. The replication method gains in simplicity but may lose in its management of instruction cache; persistent cache misses may result in wasteful trashing operations. The use of subsets typically leans towards a pipelined architecture, which does better for the instruction cache but may suffer latency or unbalanced load distribution. The second decision process was in how to design the many-core traffic handling system, while ensuring data (packet) consistency.

We chose the replication method initially, so the way firewall rules are organized will not depend on how we handled the multi-core traffic handling design. This is because each core on the CPU will get a packet and will have to analyze it against all the rules in a specified configuration file. Thus, every core will need access to the same read-only data structure created as a result of the configuration file. Our configuration file allows for five-tuples packet filtering based on source and destination IP addresses, source and destination ports, and protocol. Anytime the configuration file bases a rule on an individual source or destination IP, our firewall uses hashing to create the rule. Each hash item then contains a linked list based on source/destination ports and protocol. For rules that allow for ranges for the source and destination IPs, there is a default linked list that contains all of those rules.

With regards to the second decision, we chose the following algorithm to ensure data packet consistency: we adopted a front-end system involving a master thread and an arbitrary number of worker threads. The single master thread is responsible for reading in network packets, enqueuing them in a round-robin fashion to each worker thread's queue, while the worker threads process these packets through the given firewall rules. Each worker possesses a queue implemented in a ring buffer data structure, which holds packets read in by the master thread. Due to the nature of ring buffers and the use of round robin scheduling, thread synchronization was not necessary. This was important because the stateless firewall needs to be as lightweight as possible to act as a useful preprocessor to the stateful firewall. By pinning each worker thread to a different CPU core, we know from Figures 1 and 2 that we gain in performance.

2.3. Results

In our tools, we have created two computer programs. The first is a tool that generates random set of firewall rules based on a packet's 5-tuple. Currently, these rules uses a uniform distribution over the entire range of IPv4 address and the entire range of ports. It is this set of

rules that we ask our stateless firewall to use in filtering ingested packet streams and decide whether to pass or drop packets. The second is the actual stateless processing of the firewall rules. The two sub-figures in Figure 3 represent the same data but in different formats. Essentially, we plot the processing rate in bytes per second versus the number of cores used in a TilePro processor.

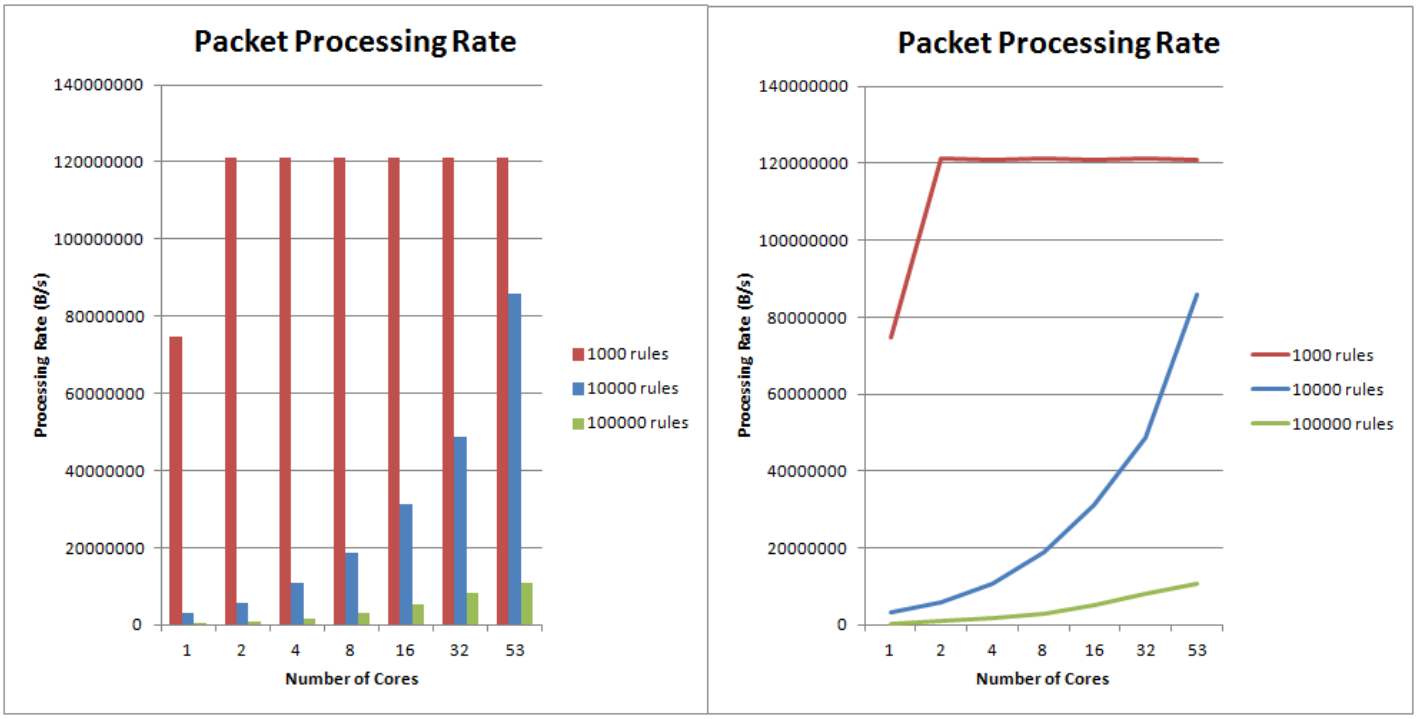

Figure 3. Packet processing rate as a function of number of rules and number of cores in a TilePro (866 MHz) processor.

Currently, our code has not been fully optimized, but it is worth noting the preliminary results we obtained from our experiments. Clearly, as the number of rules increase, the performance drops as the stateless firewalls have to do, on the average, a lot more filtering operations. However, the scaling versus number of cores is almost linear (the horizontal axis is logarithmic) indicating that the NUMA in TilePro did not have significant impact at these rates. The plateau in performance seen beyond 4 cores for the 1000 rule set is due to limited rate of the incoming traffic (about 920 Mbits/sec).

Our next step is to properly profile the performance of our stateless firewall and look for any bottlenecking function in the code, so we can apply any applicable optimization to those areas. Afterwards, we will conduct a correctness check, compare our performance with existing solutions, and test robustness on true networks.

3. STATEFUL FIREWALL

A stateful firewall needs the ability to not only inspect input packets based on current rule snapshot, but on previous inputs particularly those that belong to the same session. A session in this sense can refer to an FTP file transfer session, where port numbers can change as a result of the protocol's design. Another example of a session can be a web traffic involving a single server, multi-client application. Thus, unlike a stateless firewall that can allow a rogue packet to make its way into a protected network, a stateful firewall can deny such a packet because its state dictated that such rogue packet did not belong to an existing session or is not allowed to begin a new session.

This all sounds good in theory, but in practice, there is are a number of difficult issues that must be solved satisfactorily to develop a high performance stateful firewall on a many-core processor. One such challenge is that every computing system has a finite amount of memory, so states need to be managed efficiently keeping track of when to expire existing sessions to make room for newer sessions and prevent dropping otherwise good packet streams. Another challenge is that all cores participating in the firewall processing need to have access to their relevant set of network states to perform the given filtering operations. This access can be granted in a many-core processor by employing a dedicated space of memory for maintaining shared states across all cores or by using a hashing-based packet distributor, which will ensure that packets belonging to the same session are routed through the same cores every time. In the next section, we describe our development effort for the latter on an Intel-based processor.

3.1. Packet distributor using fast hashing and ring buffers

In order to effectively utilize the multi-core nature of the current Intel architecture, we have been investigating the packet processing structure presented in Figure 4. As an incoming packet from an outside network (Internet) is received at the incoming Network Interface Card (NIC), an initial "distribution core" determines which row of the conceptual architecture will process the packet.

This distribution core uses a deterministic hash function to assign each incoming packet a value based on the hash of the numeric XOR of the source and destination IP addresses. The XOR operation ensures the commutativity of the two addresses in the calculation, thereby ensuring that both to- and fro- packets go through the same processing core. Currently, the hash space is divided equally between the different processing rows. For example, on a four core machine, if there is one distribution core and three separate processing rows/cores processing the actual packets, the hash output range will be broken into three equal ranges. Each of these ranges corresponds to a unique row processing queue. In future efforts, ranges may be split unevenly to better distribute load or more than one range may map to a single queue, as incoming packets do not necessarily follow a uniform distribution stochastic process.

Once a packet has assigned a hash value and thus a processing row, the packet is copied into memory in a ring buffer and a pointer to this memory is held in the queues as shown in Figure 4. As mentioned previously, each processing row has its own ring buffer-based thread-safe queue. If there is more than one processing element (core) in a processing row, the packet is passed

logically by passing the pointer to the data held in the ring buffer. Once the packet is processed, it is forwarded on the outbound NIC to the Intranet.

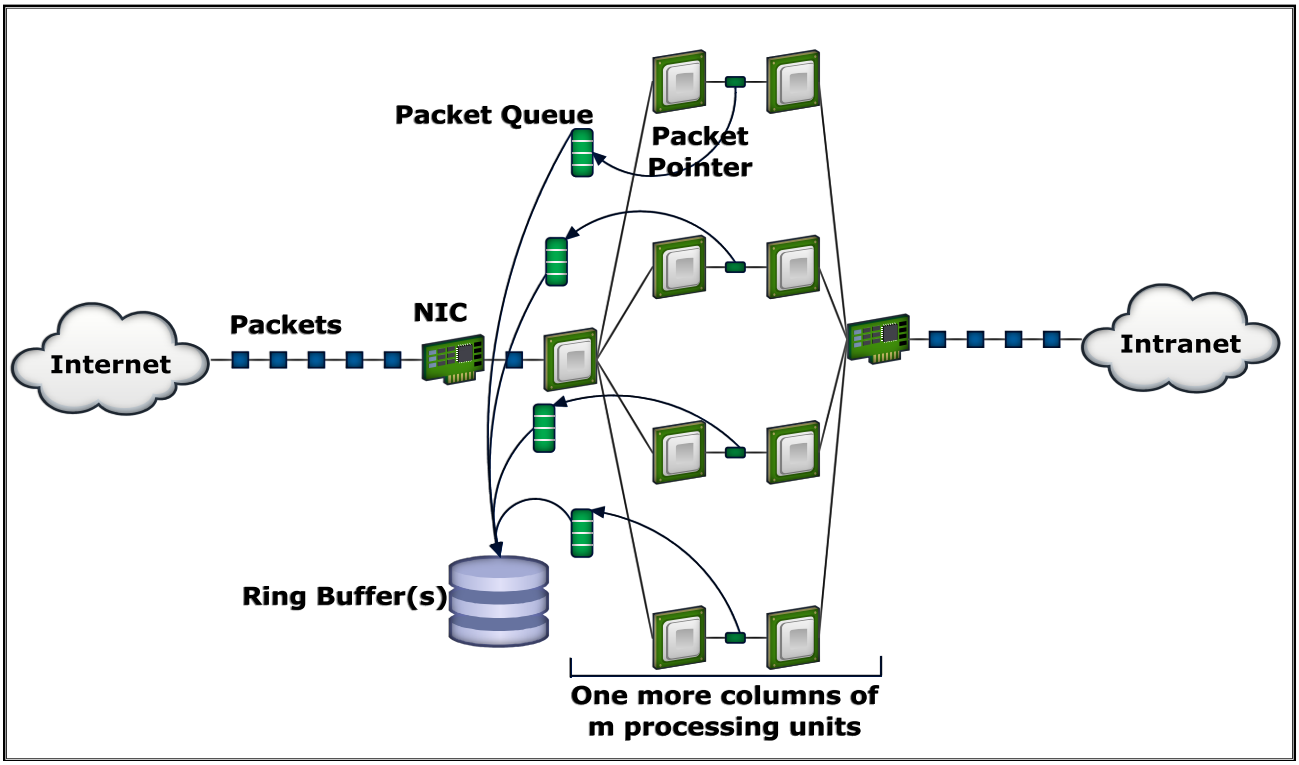

Figure 4. Conceptual model for multithreaded packet processing on a standard Intel architecture

Fast Hashing

In order to deterministically distribute packets between different processing cores, we implemented an augmented version of the Fowler–Noll–Vo (FNV) non-cryptographic hash function. The basic hash algorithm is the following [2]:

```
hash = FNV_offset_basis
foreach octet_of_data to be hashed
    hash = hash × FNV_prime
    hash = hash XOR octet_of_data
return hash
```

As the hash relies solely on XORs and multiplications, it is efficient and lightweight.

Circular Ring Buffer

In order to efficiently store the incoming packets in memory for processing by one or more cores, we have implemented a circular ring buffer. The implementation uses memory mapping of the buffer to two contiguous regions of virtual memory to improve performance. This allows for direct memory access to the buffer since all references to the buffer appear as a single memory block and any calling function does not have to deal with split buffer spaces every time the cycling of data reaches the end of buffer and wraps around to the beginning.

Memory mapping allows the ring buffer to have sizes that are equal to some multiple of the operating system page size (typically 4096 bytes on most Linux-based machines). In our case,

the buffer size is allocated as a multiple of the page size times the estimated size of all of the packets in the queue to the nearest power of two as shown in the code snippet below:

```
int NearestPow2(int n)
{
    int ret = -1;
    __asm__ (
            "dec %1\n\t"
            "movl $2, %0\n\t"
            "bsrl %1, %1\n\t"
            "roll %%cl, %0\n\t"
            : "=q"(ret)
            : "c"(n)
            : "eax"
            );
    return ret;
}

int32_t mmapsize = u_vm_page_sz * NearestPow2(((QUEUE_ENTRIES) *
MAX_PACKET_BUFSIZE)/u_vm_page_sz);
```

This code is written in standard C-Assembly code.

Thread Safe Queue

In conjunction with ring buffer, we have implemented a thread-safe queue in order to manage the order of the incoming packets and potentially enable the sharing of a ring buffer by more than one packet processing thread. As seen in Figure 4, once the core responsible for distributing the packets determines which set of cores is responsible for processing a particular packet, a pointer to the memory location (currently in one of the circular buffers) is placed in the queue. This allows the processing core to access the packet, perform some or all of the necessary processing on the packet, and then either send the packet out to the outbound NIC or allow the next processing core to work on the packet.

3.2. Extending iptables stateful firewall to many-core solution

As our team began exploring potential solutions for porting stateful firewall to many-core systems, it became apparent that Linux iptables was the best starting point. All claims to functional stateful firewall code available all had iptables as their underlying basis. For efficiency and effectiveness, iptables is built into kernel-space but unfortunately, this makes it rather difficult to explore for extension. Regardless, we explored iptables and its functionality on a standard system builds and ported it onto a Tilera with demonstrated effectiveness.

We then started with the next milestone of specifying which processing core(s) we want an iptables chain to run on. We successfully developed and demonstrated this feature on a TilePro processor using modifications in its Linux kernel image. Finally, we integrated the hashing function to allow for multiple spawning of cores to process iptables rules simultaneously. Our setup is shown in Figure 5.

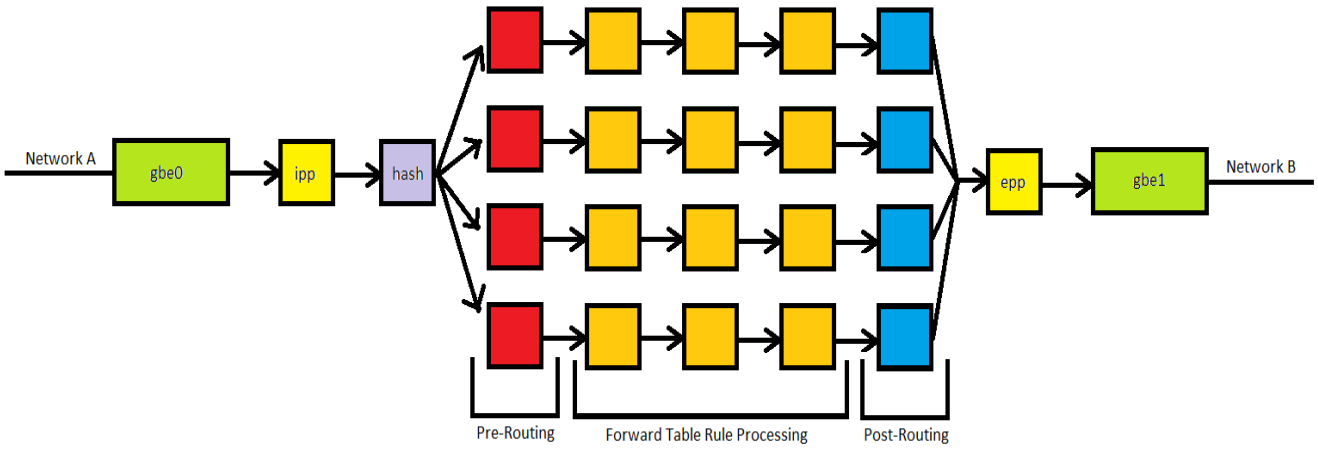

Figure 5. A multi-pipeline framework for stateful firewall processing between networks A and B

In Figure 5, except for the green boxes labeled "gbe0" and "gbe1", each box executes on a single processing core resulting in twenty-three (23) cores being used in all. Essentially, a packet from *Network A* enters the Tilera processor system through a network interface (*gbe0*, in this case). The packet is read by the ingress packet processor (*ipp*) and passed along to the Linux kernel. The kernel then executes a hash function to determine in which CPU's packet queue the packet under investigation should be enqueued. This hashing ensures that packets belonging to the same session always get enqueued in the same CPU always (see description of FNV hashing function above). Once a given packet is dequeued from the chosen CPU's queue, the chosen CPU runs the packet through the pre-routing functions. After passing though the pre-routing functions, the packet is sent from the pre-routing CPU directly to the next CPU in the pipeline. This next CPU will run the packet against a subset of the rules in the Netfilter iptable's Forward Table. If a packet is not dropped by any of its rules, it is forwarded in the same manner as before to the next CPU in the pipeline to check the next subset of rules in the Netfilter's Forward Table. This process is done however many number of times as there are CPUs in the "*Forward Table Rule Processing*" section of the pipeline. In this case, each CPU runs the packet through one third of the rules in the Netfilter's Forward Table because there are three CPUs in the section. Once it is at the last CPU in the "*Forward Table Rule Processing*" section and has not been dropped or stolen, the packet is forwarded to the post-routing CPU where it is run through post-routing functions. Afterwards, the packet is sent to the egress packet processor (*epp*) and out through the other network interface (*gbe1*, in this case) to *Network B*, completing a filtering operation for an okay packet.

Currently, our modified kernel code builds (compiles) but is not fully functional as the kernel "panics" after the hashing function has been performed and we try to spawn the first firewall function on a different CPU core. However, we have been able to split up firewall rules evaluation across multiple CPU cores.

3.3. From Juniper NetScreen to Linux iptables

3.3.1. Overview

The goal desired for this portion of the project is to perform a conversion of a firewall rule set from one router appliance's (Juniper firewall's) language to another (Linux iptables). This has been accomplished by studying both firewall rule languages and compiling all of the rules required for a firewall policy. A mapping of simple (one-to-one) translations served as a foundation for the effort, while a logic-based conversion was facilitated to complete the remainder. Many portions of the rule set have been converted successfully, but there are some documented minor issues. Logging mechanisms were the most difficult to directly convert from NetScreen to iptables, thus causing the majority of the problems.

3.3.2. The code design process

This project section desired a successful conversion between firewall policies written with Juniper Networks' NetScreen [3] language to firewall chains written with the open-source tool iptables [4]. While performing a manual conversion is certainly possible, such a method becomes quickly impractical for large implementations or multiple configuration file conversions. Therefore, a tool created to automate this process as much as possible was the ultimate goal. Because of the requirements and the layout of the data, the Python programming language was chosen.

Undertaking this task required a strong understanding of the layout of each firewall's rule set language, that is, Juniper NetScreen versus Linux iptables. There were many differences between the two – their structures differed syntactically, functionally, and logically. For instance, NetScreen firewall rules can relate groups of addresses (such as ports and Internet Protocol addresses) together into an object which can be referenced from multiple policies, while iptables does not have such functionality. Moreover, this object-oriented capability allows NetScreen policies to reference lines of code which are disconnected from the actual policy to which they are used. In contrast, iptables mandates that a specific order is followed within each firewall chain to achieve the desired effect.

In order to combat these differences, the developed tool parses through the entire NetScreen policies file and creates a list of all lines within the file which relate to a policy. This list is stored as an object in an instance of a class that can be recalled at any point throughout the program's run time. Once the policies are ordered logically, the IP addresses and ports must be associated to their appropriate object label.

Completion of the conversion is accomplished by iterating through each in-order policy line and translating it to its equivalent chain line in iptables. This step requires two general methods: a simple and a complex (logic-based) translation. The simple translation is comprised of a dictionary of words or phrases which directly correlate between the two languages. This helps to speed up the translation of rules, as a dictionary lookup averages $O(1)$ in compute-time complexity. The complex translation requires much more processing of each line, including

cognizance of the structure and purpose of the line. For example, lines can be different lengths, they can explicitly state "ANY" for a protocol/service or IP address object (which must be handled very differently in iptables), or they can specify only an object. Handling these disparities requires a complex decision structure in order to correctly classify each case.

There are some issues that have currently not been remedied in the converted rule set. One of importance is caused by the incomplete translation of NetScreen logging rules to that of iptables. For example, iptables does not have native capabilities to log a session at its initialization. Because of this, it is difficult to correctly translate the initial session logging which is currently being performed by NetScreen.

While the current code is expected to be functional for many of the converted policies, there are more complicated cases which do not correctly translate. Logging (as mentioned above) is an area for improvement, as is the ability to more directly associate objects with their names. Additionally, a more effective strategy for handling chain rule organization is desired.

4. CONCLUSION AND FUTURE WORK

This first year has focused on critical individual aspects needed for a high performance firewall. In this process, our accomplishments are listed below:

- Deployed a stateful firewall code (iptables) on Intel system and verified basic firewall functionalities;
- Ported iptables to a TilePro system, which involved building a Linux kernel module (netfilter) into the default TilePro kernel and building user-space iptables code. This has been demonstrated also to be functional. These first two items are currently demonstrated on single cores only;
- Developed a ring-buffer-based packet distributor with fast packet hashing functions for both the Intel and TilePro systems. This guaranteed that session states were maintained correctly across firewall processing pipelines and is still undergoing optimizations;
- Developed a number of "passive-wire" methods that makes our firewall machines robust. These methods exchange packets between network interfaces on our firewall-hosting machine, where the interfaces are configured without IP addresses. This code serves two purposes: (i) secure communication by ensuring packet exchanges between zones (networks) always pass through our firewall as well as (ii) guaranteeing that the system running the firewall can only be accessed via console, thereby eliminating opportunities for it being hacked over the network. The code is passive because users between zones are ignorant of its existence while being protected by the stateful firewall;
- Created the user-space fast inter-process communication code for Tilera many-core processors. This code sends an arbitrary-sized buffer (typically packets) between firewall tasks without going out to shared memory, thereby reducing contention in non-uniform memory and providing higher performance via cache. This code has been ported to kernel-space, but is still undergoing debugging;
- Created fast pre-processor code for stateless firewall. This setup allows us to mimic conventional firewall setup where known stateless information can be used to minimize unnecessary loading of states in a stateful firewall.
- Developed a translating code for porting rules in Juniper NetScreen language to Linux iptables language. This allows us to run and compare our stateful firewall against typical Juniper firewall for correctness and processing rates;
- Created a virtual testbed network that provides a safe and easy testing of kernel modifications.

In the next year, we anticipate the completion of a parallel pipelined version of stateful firewall processing which we will demonstrate on our many-core processors. In addition, we will complete the optimization of our stateless firewall implementation to increase its processing rate. Finally, we will conduct research into auto load-balancing of tasks within a processing pipeline and deploy our solutions to reduce latency and processing time.

5. REFERENCES

1. Top 500 Supercomputer Sites (http://top500.org)
2. Fowler-Noll-Vo Hash function (http://forum.kalkulators.org/docs/fhash/specs/fnv1.pdf, September 2011)
3. Juniper Networks NetScreen CLI Reference Guide (http://www.juniper.net/techpubs/software/screenos/screenos5x/cli_5_0.pdf, September 2011)
4. Iptables(8) – Linux man page (http://linux.die.net/man/8/iptables, September 2011)

APPENDIX A: METHODS TO ACHIEVE PUTTING A LINUX-BASED FIREWALL IN PASSIVE-WIRE MODE

Description

The passive wire program utilizes two sets of threads. One set of threads is responsible for reading network traffic from a raw socket bound to one network device and relaying the network traffic to another raw socket bound to the another network device. The other set of threads perform the same task except for that they relay network traffic in the opposite direction. All threads are pinned to separate tiles to maximize efficiency. Additionally, spin mutexes are implemented in place of sync mutexes to improve efficiency.

What Worked

- Threading the program
- Pinning threads to specific cores
- Relaying all network traffic using raw sockets
- Increased speed with spin mutexes as opposed to sync mutexes for reading/writing

What Did Not Work

Ability to filter network traffic using standard iptables/netfilter installation: Because raw sockets were being used, copies of all incoming packets are sent to the sockets before passing through the TCP/IP layer of the network stack. Since iptables/netfilter sits in the TCP/IP layer of the network stack, packet filtering is not performed before the packets were sent to the raw sockets.

Network Throughput Testing

Description

In order to test the upper limits of the throughput of various firewall solutions, the program iperf was used on two computers. Each computer running iperf was connected to one of the network interfaces on the firewall machine. The computer running the client version of the iperf program continuously sends packets of a specified size at a very high rate. These packets pass through the firewall machine and to the other computer running the server version of iperf. After a set amount of time, the average throughput of the packets is calculated and output to the screen.

Syntax

Client: `iperf -c [ip_address_of_server] -M [packet_size[A]]`

Server: `iperf -s`

[A] Because extra encapsulation is added to the packet when it is sent, the actual packet size is 28 bytes greater than the number put here. For example, if the option "-M 72" is used, then the actual size of the packet sent is 100 bytes.

Netfilter Hooks and Loadable Kernel Modules

Description

Netfilter hooks are used to grab packets traversing through Netfilter at various points to allow Loadable Kernel Modules (LKMs) to analyze them and tell Netfilter what to do with them.

What Worked

- Grabbing packets from Netfilter hooks
- Analyzing packets (i.e. determine protocol, originating network interface, etc.)
- Tell Netfilter what to do with packet (i.e. accept, drop, steal, etc.)

What Did Not Work

Injecting packets obtained from raw sockets into Netfilter. Hooks can only be used to grab packets already in Netfilter and tell Netfilter what to do with the last grabbed packet.

Comments

Testing of Netfilter hooks and LKM functionalities was done in a Virtual Machine running on a machine with an Intel processor. None of these tests were performed on the Tilera architecture.

Bridging Network Devices

Description

A bridge is a device that connects two or more network segments by forwarding packets between two network interfaces. Because the forwarding is done at Layer 2 (data link layer), the bridge is protocol agnostic and transparent assuming the network interfaces are brought up without an IP address. Traffic flowing between the network interfaces over a bridge can easily be routed through iptables so that packet filtering can be done.

The following sections detail how to create such a bridge on the Tile64 architecture.

Cross Compile brctl Program

The brctl program is a user-space program used for creating and managing bridging devices.

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/shemminger/bridge-utils.git
$ cd bridge-utils
$ autoconf
```

```
$ ./configure --host CC=tile-cc
$ make
```

The `brctl` binary will be located in the `bridge-utils/brctl/` directory.

Modify Linux Kernel to Support Bridging

```
$ mkdir $SRC
$ cp -R TileraMDE-3.0.0.123096/tilepro/src/sys/linux $SRC
$ cd $SRC/linux
$ mkdir $BUILD
$ make ARCH=tile O=$BUILD defconfig
$ cd $BUILD
$ sh $SRC/tile-prepare
$ make menuconfig
```

Select the following:
Networking support
Networking options

Highlight the following and press 'Y' to enable:
802.1d Ethernet Bridging

Select the following:
<Exit>
<Exit>
<Exit>
<Yes>

```
$ make
```

Tilera Startup Script

```
tile-monitor –pci –vmlinux $BUILD/vmlinux –hvc [HVC_FILE] \
--upload bridge-utils/brctl/brctl /usr/bin/brctl \
--quit;
```

Bridging Script (run on Tilera card)

iptables Filtering Enabled

```
echo 1 > /proc/sys/net/bridge/bridge-nf-call-arptables
echo 1 > /proc/sys/net/bridge/bridge-nf-call-iptables
echo 1 > /proc/sys/net/bridge/bridge-nf-call-ip6tables
echo 1 > /proc/sys/net/bridge/bridge-nf-filter-pppoe-tagged
```

```
echo 1 > /proc/sys/net/bridge/bridge-nf-filter-vlan-tagged
echo 1 > /proc/sys/net/ipv4/ip_forward
ifconfig -a [INTERFACE_0] up
ifconfig -a [INTERFACE_1] up
brctl addbr br0
brctl addif br0 [INTERFACE_0]
brctl addif br0 [INTERFACE_1]
ifconfig br0 0.0.0.0 up
```

iptables Filtering Disabled

```
echo 0 > /proc/sys/net/bridge/bridge-nf-call-arptables
echo 0 > /proc/sys/net/bridge/bridge-nf-call-iptables
echo 0 > /proc/sys/net/bridge/bridge-nf-call-ip6tables
echo 0 > /proc/sys/net/bridge/bridge-nf-filter-pppoe-tagged
echo 0 > /proc/sys/net/bridge/bridge-nf-filter-vlan-tagged
echo 0 > /proc/sys/net/ipv4/ip_forward
ifconfig -a [INTERFACE_0] up
ifconfig -a [INTERFACE_1] up
brctl addbr br0
brctl addif br0 [INTERFACE_0]
brctl addif br0 [INTERFACE_1]
ifconfig br0 0.0.0.0 up
```

[INTERFACE_0] can be replaced with gbe0 for 1GbE or xgbe0 for 10GbE
[INTERFACE_1] can be replaced with gbe1 for 1GbE or xgbe1 for 10GbE

DISTRIBUTION (Electronic Copies):

1	MS1319	Jim Ang	1422
1	MS1319	Bob Benner	1422
1	MS1319	Ron Brightwell	1423
1	MS1319	Kyle Wheeler	1423
1	MS0123	D. Chavez, LDRD Office	1911
1	MS0671	Mitch McCrory	5627
1	MS0671	Jay Patel	5627
1	MS1073	Curtis Johnson	5635
1	MS1027	Jeff Shelburg	5635
1	MS1073	Ben Cook	5641
1	MS1073	Joshua Johnson	5641
1	MS0806	Tim Berg	9336
1	MS0806	John Naegle	9336
1	MS0806	Uzoma Onunkwo	9336
1	MS0806	David Pearson	9336
1	MS0933	David Duggan	9516
1	MS0933	Brian Wright	9516
1	MS0933	Dave Zage	9516
1	MS0899	RIM-Reports Management	9532


Sandia National Laboratories