Scalable High-Performance Parallel Design for Network Intrusion Detection Systems on Many-Core Processors

Haiyang Jiang, Guangxing Zhang, Gaogang Xie
Institute of Computing Technology
Chinese Academy of Sciences, China
{jianghaiyang,guangxing,xie}@ict.ac.cn

Kavé Salamatian
University of Savoie, France
kave.salamatian@univ-savoie.fr

Laurent Mathy
University of Liège, Belgium
laurent.mathy@ulg.ac.be

ABSTRACT

Network Intrusion Detection Systems (NIDSes) face significant challenges coming from the relentless network link speed growth and increasing complexity of threats. Both hardware accelerated and parallel software-based NIDS solutions, based on commodity multi-core and GPU processors, have been proposed to overcome these challenges. This work explores new parallel opportunities afforded by many-core processors for high performance, scalable and inexpensive NIDS. We exploit the huge many-core computational power by adopting a hybrid parallel architecture combining data and pipeline parallelism. We also design a hybrid load balancing scheme, using both ruleset and flow space partitioning. Furthermore, the proposed design leverages particular features of the processor to break the bottlenecks. We have integrated the open source NIDS Suricata into our proposed design and evaluated its performance with synthetic traffic. The prototype exhibits almost linear speedup and can handle up to 7.2 Gbps traffic with 100-bytes packets.

Keywords

many-core, network intrusion detection system, parallel, load balancing

1. INTRODUCTION

Network Intrusion Detection Systems (NIDSes) have become important components of modern network security infrastructures. Signature based NIDSes [1] that parse packet headers and inspect payloads, checking against a large ruleset, *i.e.*, a collection of signatures of known attacks, viruses, worms, spyware or malicious code [2], are the de-facto NIDS standard. With increasing line speed and traffic complexity, growing number of exploits and attacks that result in inflation of the NIDS rulesets, signature based NIDS is becoming more challenging [3].

Dedicated hardware-based ASIC/FPGA implementations [4, 5] and software implementations on commodity multicore processors [6, 7] or GPU based solutions [8, 9], have been proposed for high speed NIDS. Among all stages in NIDSes' packet processing, the payload signature matching is the main performance bottleneck: *e.g.*, in Snort [10] approximately 70% to 80% of the processing time is consumed by the signature matching engine [11]. Custom ASIC/FPGA hardware and GPU based solutions target this particular stage for acceleration. However, both solutions generally leave flow level tasks like protocol analysis, IP defragmentation, TCP stream reassembly and application-level parsing, to classical CPUs. While hardware component or GPU can achieve signature-matching engine of tens of gigabits per second, massive payload transfer between components, flow level coordination and CPU-bound flow management processes reduce overall system performance. Besides, the low flexibility, low scalability and high cost of FPGA/ASIC based hardware solutions are serious issues that hinder large-scale deployment of NIDSes. GPU solutions, in addition to performance limitations, suffer also from high power consumption.

The availability of generic many-core architectures with tens to hundreds of cores per processor is offering new opportunities for high-speed NIDS. In this paper, we explore the design and performance of NIDS systems on such processors. More precisely, we present a parallel and modular design for implementing NIDS on the TILERAGX36, a many-core processor with 36 cores, developed by Tilera Inc.[12]. The design follows two strategies. We first designed a hybrid parallel architecture, combining data and pipeline parallelism. We also designed a hybrid load-balancing scheme, using both ruleset and flow space partitioning. Furthermore, the proposed design leverages particular features of the TILERAGX36 hardware to break the bottlenecks. By using the mPIPE (multicore Programmable Intelligent Packet Engine) [12] available on TILERA platforms, the lock contention is reduced by 75%. In order to evaluate scalability and performance, we have integrated the Suricata [14, 15] NIDS into our proposed design. Our experimental results show that the system achieves an almost linear speedup with the number of core dedicated to NIDS. As an integrated NIDS solution, the prototype implementation can handle up to 7.2 Gbps synthetic traffic with 100-bytes packets.

In section 2 we introduce background information on many-core processors and describe related work on parallel NIDS. Section 3 develops the proposed design on TILERAGX36 and discusses its strong and weak points. Section 4 introduces optimizations applied on the initial design and describes in detail the proposed hybrid load-balancing scheme

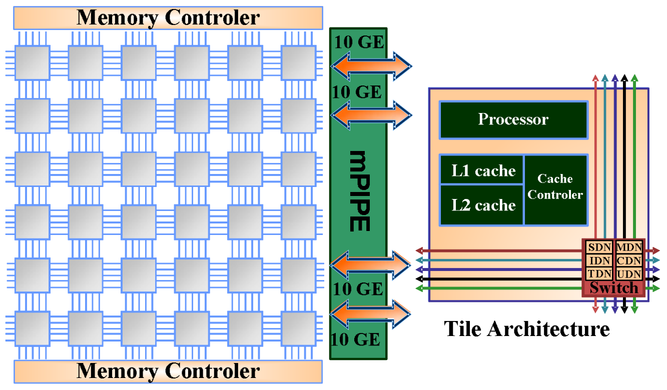

Figure 1: High-level overview of the TILERAGX36 architecture [12]

that is used in the system. Section 5 describes the experimental environment and presents the experimental results. The paper is concluded in section 6.

2. BACKGROUND AND RELATED WORK

2.1 Overview of TILERAGX36 Processor

Many-core processors, which are usually equipped with a larger number of cores per processor (tens to hundreds of cores per processor compared to up to 6 or 8 in existing commodity multi-core processors), have emerged in the past years. In particular, each core in these processors can run a full operating system, which provides high flexibility and simplicity for software development.

The TILEGX36 processor is a typical many-core processor with 36 homogeneous, general-purpose computational elements, named tiles, organised in a 6×6 grid interconnected through a *iMesh* on-chip network [12, 16](see Fig.1). Each tile element consists of a full-featured processor core with both L1 and L2 cache and a non-blocking switch that connects the core to five 2-dimensional mesh networks connecting all titles. Among these five Network-on-Chip (NoC) interconnects, only one, the User Dynamic Network (UDN), is dedicated to the applications for communication among tiles, while the processor itself uses other networks to improve efficiency and speed up data transfers among the tiles, the I/O devices and the memory. The UDN is connected directly to the Arithmetic Logical Unit (ALU) in the tile, which allows very low communication latency between tiles. Table.1 shows a comparison of communication among tiles via UDN and shared memory: The UDN network exhibits a very low transmission latency and meanwhile very high bandwidth up to 60 Tbps, which provides a much better performance than traditional shared memory. Each tile of the TILERAGX36 chip has a 256 KB L2 cache. Each of these L2 caches, in addition to be used locally, can be used as a remote L3 cache for other tiles. This results in a big distributed L3 cache accessible by all the tiles. This strategy allows a page of virtual memory to be homed on a specific tile, then cached remotely by others. The processor also integrates a packet capture engine (mPIPE) on the die [12].

2.2 Packet processing in NIDS

Fig.2 depicts the major packet processing steps in a NIDS. Packets are captured from the wire in the *Packet Capture*

Table 1: Comparison of UDN and shared memory based communication among tiles

$Medium$	$Bandwidth(bps)$	$latency(cycles)$
UDN	60T	1 per tile hop
Shared Memory	170G	L1 cache hit: 2 L2 cache hit: 11 L2 cache miss: 80

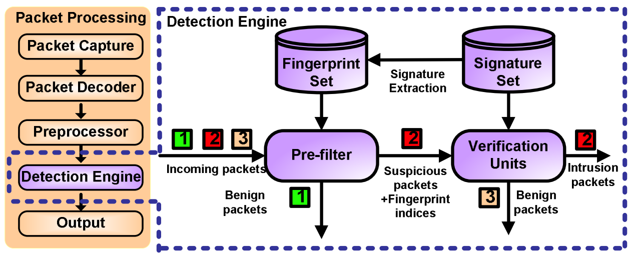

Figure 2: Packet processing in NIDS and detection engine

module and decoded in the *Packet Decoder*; the *Preprocessor* module processes the TCP/IP protocol, dealing with IP fragments and TCP reassembly; the *Detection Engine* module scans the payload for possible patterns and generates the logs to be recorded in the *Output* module. Among all these stages, the *Detection Engine* module is the most demanding computationally. The architecture of this stage is represented on the right-hand side of the figure. It contains two-stages: a pre-filtering stage [17] that searches in the packet payload for occurrences of a set of fingerprints extracted from the whole signature set, and a verification stage that only receives packets that match at least one fingerprint. The verification stage uses the indices of the matched fingerprints in the pre-filtering and carries, over the packets, in-depth checks for the rules associated with these fingerprints. A match is returned if at least one of the rules is matched. This process eliminates any potential false positive in the pre-filtering stage. When the fingerprints are wisely chosen to ensure that it has a full coverage of the rule-set, *i.e.*, each rule is at least represented by one fingerprint, the above two-step architecture is very efficient.

2.3 Parallel Architectures in NIDS

Both data and pipeline parallelism have been used in the context of NIDS. In pipeline parallelism, the whole packet processing is divided into several sequential stages that each run on a dedicated execution unit. A packet is transferred sequentially from one execution unit in the pipeline to the next. In addition to the parallelism gained, pipelining improves reference locality and potentially increases the cache hit ratio since each execution unit only deals with a subset of the entire application memory [18]. In contrast, data parallelism replicates the entire packet-processing loop on separate cores, such that several packets can be processed independently in parallel. Both forms of parallelism can only achieve a linear speedup when the memory space of each execution unit is independent and isolated. When there is an overlap in memory, costly locking mechanisms – resulting in performance loss caused by contention – are needed to ensure consistency. The alternative of replicating the data-structures in each execution unit results in linear increase of the memory footprint. Load balancing is also critical in order to achieve full parallel performance through full compute

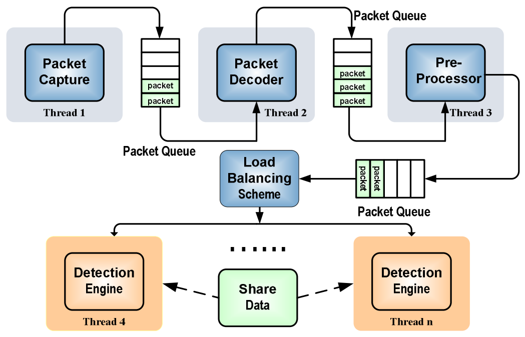

Figure 3: Suricata's pipeline based architecture

resource usage. In the rest of this section, we will describe how different software NIDS architectures have specifically dealt with these issues.

Snort is often regarded as the reference for the evaluation of parallel NIDS designs. As snort is essentially single-threaded, data parallelism is the most direct applicable parallel strategy. A Round-Robin scheduling scheme is adopted for data parallelism in [19] and achieves a throughput of 29 Mbps with 4 threads concurrently running on a server platform equipped with 2 Dual-Core Intel Xeon Processors. The pipeline parallelism is implemented in this work too by dividing the program into a two-stage pipeline where one thread is responsible for packet capturing and dispatching to the second stage where several packet-processing threads process the traffic concurrently. To share the load among the packet processing threads, a Flow Static Hash (FSH) is used, *i.e.*, the 5-tuples identifying a TCP/UDP flow in the IP packet header is hashed modulo the number of parallel packet processing threads. Experimental results showed that the pipeline provides about six times better performance than data parallelism alone, achieving a throughput of 188 Mbps, by improving the cache hit ratio.

A follow-up of this work [20] proposed lock optimization and attained 10% to 75% performance improvement. Nonetheless the proposed techniques still suffered from two issues: a huge memory footprint due to the replication of the Snort data structures in each concurrent thread, and residual imbalance between packet processing threads. While FSH is very easy to implement, it suffers from the fact that the load induced by a flow depends on its duration and size and these values are very difficult to predict *a priori*. This results in poor load balancing performance [18] and further impacts the overall performance. To deal with these issues [21] implemented a shared memory into the Intel pipeline and achieved a throughput of 3.1 Gbps on a commodity 8-core server platform.

Further, a dynamic flow based load balancing scheme, called Join-the-Shortest-Queue(JSQ), was used in the Para-Snort system in [22]. The reported highest performance in this system is about 800Mbps with two quad-cores Xeons E5335 at 2.00GHz. While adaptive flow-based load balancing schemes such as JSQ achieve a better equilibrium between threads [23], it still cannot fully resolve the load imbalance problem as the flows exhibit intrinsic large imbalance [21] and short-term packet bursts which are common in network traffic [24] are not managed well.

Suricata is another open source NIDS that adopts a pipeline model. As shown in Fig.3, Suricata divides the overall process of a NIDS into a larger number of fine-grained stages. Each stage of the pipeline runs as a separate thread that is connected to the next stage through a buffer. The traffic is scanned in several parallel *Detection Engine* threads to break the performance bottleneck, *i.e.* packets belonging to different flows are dispatched by the load balancer among the parallel detection engines. Suricata has a very modular architecture that makes it very flexible, so that the modules and their interconnection can be easily changed. This is the reason why we adopted Suricata as our main evaluation platform.

The latest version of Suricata uses FSH for the sake of ease of implementation. In our previous work [25], we have proposed Ruleset Partition Balancing (RPB) to optimize performance. The approach consists of partitioning the NIDS ruleset, rather than the traffic, across the execution units. Experiment results showed that the system performance was improved by 42%, comparing with FSH. In this current work, we have adopted RPB in conjunction with flow-based load balancing in the hybrid load-balancing approach that will be described later.

MIDeA [8] and Kargus [9] are two recent NIDSes that are designed using Graphics Processing Unit (GPU). GPUs are specialized and highly parallel hardware units designed for graphical computation-intensive purpose. While the raw performance of GPUs is much higher than multi-core based solutions CPUs, they are dedicated to Single-Instruction, Multiple-Thread (SIMT) operation mode where a same process is applied in parallel to a large number of data instances. In particular they are not well suited to implementing algorithms with several branches, which are unfortunately common in protocol processing in NIDS. For this reason, in both MIDeA and Kargus, the protocol processing is still done on classical CPUs and the pattern recognition is deported to GPUs. This means that large amount of data (packet payloads) have to be exchanged between CPU and GPU over the video card bus and in order to reduce the contention over this bus, batching mechanisms must be implemented that add relatively large delays in the packet processing pipeline. Moreover, optimizing a GPU application requires leveraging particular hardware primitives/structures meant for graphics and this is more challenging than a many-core implementation [26]. And last but not least, GPUs often have very large power consumption, *e.g.*, Kargus uses two NVIDIA GTX580 GPU card that consume up to 720W when processing 10Gbps traffic.

3. MANY-CORE NIDS ARCHITECTURE

In this section, we present our many-core NIDS parallel design. A high level description of the proposed architecture is shown in Fig.4. Our approach consists of several *Packet Processing* modules running in parallel (data parallelism), each containing a fully functional NIDS running on several cores (pipeline parallelism). Internally a *Packet Processing* module is structured as three stages. The first stage is the *Packet Capture* that receives packets and fills a message data structure (MSG) obtained from a message pool shared among all *Packet Processing* modules. This MSG will be propagated through the packet processing pipeline. The *Packet Capture* feeds several parallel *Protocol Processing* components (a component is a thread exclusively assigned to a core) that parse protocol information and update the MSG accordingly. Each *Protocol Processing* stage is followed by several parallel *Detection Engines* that implement

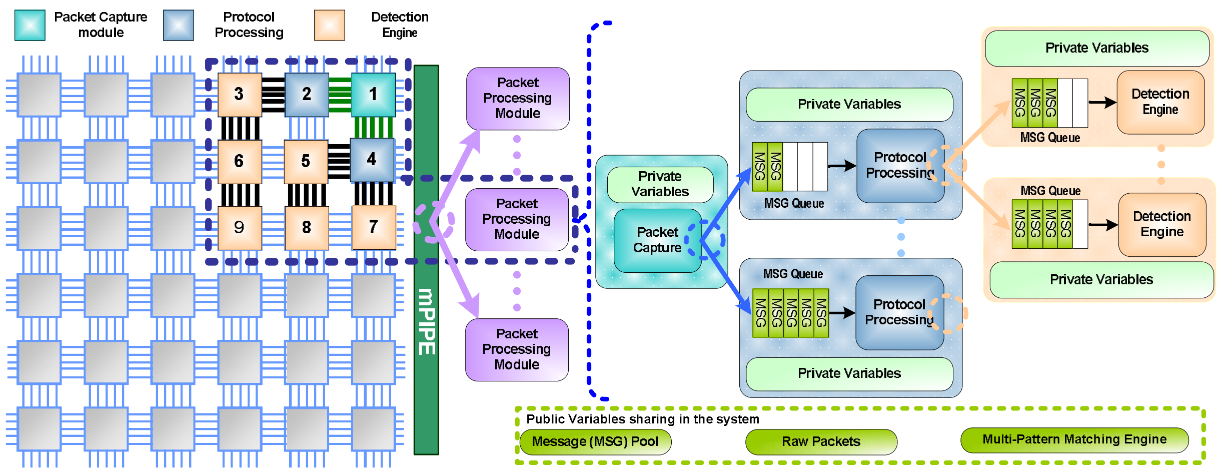

Figure 4: Proposed NIDS parallel design on TILERAGX36

the pattern matching algorithms, possibly generating alerts, and release the MSG data structure when all scanning tasks are finished.

The message propagation is a performance issue for each *Packet Processing* module. As it shown in Fig.4, each *Protocol Processing* and *Detection Engine* component in the module contains an MSG buffer to absorb temporary traffic bursts. In order to eliminate overheads such as synchronization, cache missing and false sharing in shared memory based solution, we adopted UDN for the message propagation among stages. To use the UDN, the MSG descriptor is written at the source (core) in special-purpose registers accessible by the ALU and retrieved at the destination (core) from registers too. The overhead of reading and writing to these special-purpose registers is much lower than writing to memory. In addition MSGs are transferred between cores without polluting local caches, so that processing steps can be performed at the remote core immediately without checking any condition variables.

The load balancing in the proposed architecture consists of dispatching MSG data structures at each bifurcation point in Fig.4(the dashed circles in the figure). We used different load balancing schemes for each one of these points. First, load balancing must decide which *Packet Processing* module should process the incoming packet. For this step, we propose to use a dynamic load balancing scheme such as JSQ [22], in order to reduce the impact of non uniform flow sizes. In between the *Packet Capture* and *Protocol Processing* threads, a simple FSH load balancer is used. For the last stage of load balancing, towards *Detection Engines*, a rule splitting load balancer [25] is used. With this load balancer each *Detection Engines* only checks a subset of the rules. Meanwhile if any signature matching is found in a packet in one of *Detection Engines*, the others can skip over the packet.

3.1 Discussions about the design

The above architecture exploits both data and pipeline parallelism. However, the fine granularity of the internal stages of each *Packet Processing* module enables a more flexible core allocation. Moreover as the *Packet Processing* modules are independent and identical, a linearly increase in performance can be achieved by replicating them as long as processing cores are available. Another benefit of fine modules granularity is improved reference locality and cache hit ratio. This is because each execution unit only deals with a subset of the entire application memory. In particular, the flow management in *Protocol Processing* threads benefits from caching recent flow data for subsequent packets belonging to the same flow. Finally, the design benefits from reduced memory footprint by sharing, between all modules, common data structures like MSG pool, packets buffers and memory containing intrusion/attacks signatures that are used by the *Detection Engine* threads.

This memory sharing has a cost: contention resulting from shared data structures. More precisely, there are three types of public variables that are shared among all threads and are encircled by a green dashed rectangle in the lower-right corner of Fig.4. The intrusion/attack signatures used by the signature matching engine, and the memory used for storing raw packets can be freely shared among all threads because they are read-only. However, the shared MSG pool can cause frequent contention, as costly lock mechanisms are needed to manage concurrent accesses. In addition when contention builds up, the lock variables in the caches are invalidated frequently, and the cache miss rate increases. This indeed has a negative effect on scalability, since the contention will increase with the number of cores [27]. The load balancing is another source of performance loss. As mentioned, we have used, for each *Protocol Processing* thread, an FSH balancing scheme that is simple to implement, but can result in load imbalance because of non-uniform flow size distribution. In order to overcome the above issues, we have optimized the proposed architecture with some features of the TILERAGX36 hardware.

4. NIDS OPTIMIZATION ON TILERAGX36

4.1 Contention reduction

As explained above, the contention in our proposed design comes mainly from concurrent accesses to the MSG pool containing MSG data structures that are propagated through the whole pipeline. To avoid the per-packet allocation and deallocation costs of MSG data structures, we preallocate a number of these data structures in the initialisation phase of the NIDS and store them into the MSG pool. For each incoming packet, *Packet Capture* thread gets an unoccupied MSG data structure from the pool and erases

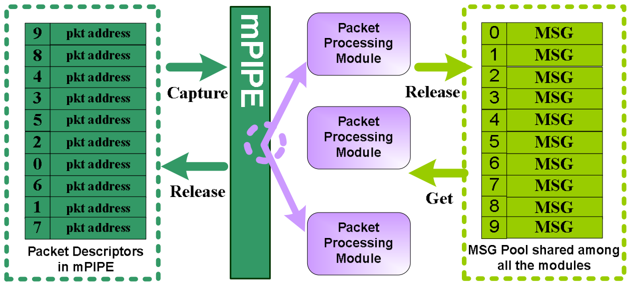

Figure 5: Elimination of the contention for MSG Pool

the previous information recorded in it, then the last *Detection Engine* thread to finish processing the packet needs to return the MSG to the pool. And sometimes the *Protocol Processing* threads also need to fetch and release MSG data structures from the pool to do fragment reassembly of TCP/IP packets. The MSG pool is thus a multi-reader, multi-writer data structure, and any operation on it should be protected by locking mechanisms.

In order to keep the parallel system busy and avoid pipeline stalls, the MSG pool must contain thousands of MSG data structures. But protecting the whole MSG pool with a single lock would result in coarse-grained locking that significantly increases access contention and reduces performance. Fortunately, the mPIPE hardware packet capture component in the TILERAGX36 provides an alternative solution to avoid this coarse-grain lock. The mPIPE preallocates fixed size buffers, in a packet capture ring, that record packet metadata and are accessed through DMA (Direct Memory Access). Each buffer position in the ring is indexed, tracked and managed by mPIPE. This capture buffer index is exposed by the mPIPE and sent along with the packet descriptors to a thread managing the packet. To reduce contention, we allocate an MSG data structure for each buffer in the packet capture ring. As shown in Fig.5, each MSG data structure in the MSG pool is associated with a buffer in mPIPE. When a packet is captured, the mPIPE provides a packet descriptor as well as the index of the packet buffer to one of our *Packet Capture* threads which uses this index to get the corresponding MSG data structure from the MSG pool. Therefore, locking is no longer required for the management of the MSG pool as each MSG data structure is associated with a specific packet buffer. However, as several threads can scan a packet simultaneously, a fine grain lock is still needed to detect when all *Detection Engine* have finished their work. For this purpose, two atomic variables are added in each MSG data structure rather than the whole MSG pool. The first variable represents the number of *Detection Engines* that have not yet finished their activity. The counter is initialized to the number of concurrent *Detection Engines* in the third stage, which is known at initialization time. When a *Detection Engine* finishes its processing, it calls an atomic *__sync_sub_and_fetch* (provided by the hardware) to decrease the counter and check its value. If the operation returns zero, the corresponding *Detection Engine* is the last to finish the scanning and the MSG data structure can be released to the pool. The second variable represents if any signature matching is found in a packet. A *Detection Engine* will check the variable before processing a packet. If the variable has been set yet, the thread will skip over the packet. This highly reduces lock contention as getting the MSG will not cause any contention, and the contention to release a MSG is simply some fine grained atomic operations.

4.2 Hybrid load balancing scheme

Load balancing is a particularly important element controlling the performance of parallel systems. As explained in the architecture description, there are three levels of load balancing. A first level is load balancing between *Packet Processing* modules. For this level we benefit from the dynamic load balancing scheme similar to the JSQ that is implemented in the mPIPE hardware. This dynamic load balancing alleviates partially the issues of imbalance resulting from different flow sizes.

The second level of load balancing dispatches incoming flows in a *Packet Processing* module (actually in the *Packet Capture* thread of the *Packet Processing* module) among the attached concurrent *Protocol Processing* threads. We used a simplistic FSH scheme to ensure that all packet of the same flow are managed by the same *Protocol Processing* thread. This is likely to generate some imbalance between *Protocol Processing* engines. However the impact should be reduced by the mPIPE dynamic load balancing scheme used in level 1 that already strongly reduces the imbalance between *Packet Processing* module.

The third level of load balancing is among *Detection Engines* after a *Protocol Processing* thread. As explained earlier, a Ruleset Partition Balancing (RPB) scheme is used in this level, rather than a Flow Balancing (FB) one. The main idea consists of dividing offline the whole ruleset into several smaller signature subsets that have approximately the same computational load and to assign each subset to a particular *Detection Engine*. Thereafter packets coming out of a *Protocol Processing* thread are forwarded to all concurrent *Detection Engines* attached to it. If one signature matching is found in a packet in one *Detection Engine*, the packet will be labeled and the other *Detection Engines* will skip over the packet then. As reported in [25], RPB achieves very good load balancing, better cache locality and decreased the cache miss ratio, thanks to the smaller memory footprint of the smaller rule subsets. Indeed, the balancing performance depends on the way the rules are split among the threads. In [25], an algorithm to achieve this aim is proposed. The main rational of this algorithm is that it evenly distributes rules with the same protocol information in different subsets, *i.e.*, a packet is likely to either make several parallel *Detection Engines* to check or to not need any processing at all.

5. PERFORMANCE EVALUATION

In this section, we give details of the evaluation platform we have built to evaluate the performance of the proposed parallel NIDS design. We will evaluate the performance using a theoretical optimistic model. First we analyse the performance of a single *Packet Processing* module and optimize its resource allocation. Thereafter, we extend the analysis to several *Packet Processing* modules, each with the optimal resource allocation, and get the overall throughput for our parallel design. We also compare the design with other proposed parallel NIDS platforms.

5.1 Experimental Setup

We build an evaluation platform using a TILERAGX36 PCIe card installed in a x86 server platform. Each tile processor runs at 1.2GHZ clock rate, with 64 KBytes L1-cache and 256 KBytes L2-cache. The Tilera processor accesses 8 GBytes DDR3 memory and four 10GE ports, all installed on the processor card. The TILERAGX36 card is managed by the Tile-monitor [12] program running on the server.

The TILERAGX36 processor runs the Linux operating system. We use the Multicore Development Environment (MDE) provided by Tilera corporation [12] to compile, upload and manage our prototype programmes on the PCIe card. The components running in each *Packet Processing* module are obtained from Suricata 1.3's architecture thanks to its modular design. In the evaluation experiments, we have used the Snort official ruleset containing 7571 rules. We use a version of OProfile [28] utility customized by Tilera that uses the hardware performance counters built in the processor to collect performance data.

We evaluate the performance of the NIDS under a variety of workloads generated by a custom synthetic traffic generator that can generate variable-size packets with speed up to 10 Gbps.

5.2 Performance evaluation model

Assuming a perfect data level parallelism with complete isolation and perfect load balancing among the *Packet Processing* modules, the cumulative processing performance of the system is

$$P_{\text{sys}} = N_{\text{mod}} * P_{\text{mod}} \tag{1}$$

where N_{mod} is the number of parallel *Packet Processing* modules in the system, and P_{mod} is the throughput performance of each *Packet Processing* module. As the *Packet Processing* module is pipelined, the throughput of the module is at most the throughput in the slowest stage. The highest throughput of each stage is attained when it reaches saturation. We represent this by stating that

$$P_{\text{mod}} = \min\{P_1, N_{\text{Prot}} * P_2, N_{\text{Prot}} * N_{\text{Det}} * P_3\}, \tag{2}$$

where P_i is the throughput of a thread in the i^{th} stage of the pipeline, N_{Prot} is the number of parallel *Protocol Processing* threads and N_{Det} is the number of *Detection Engine* threads attached to each *Protocol Processing* thread. However, the throughput can even be less because of contention and cache misses in downstream levels.

The performance of a thread in the last stage, the *Detection Engine* stage, can be evaluated as $P_3 = \alpha P_{\text{Det}}$, where P_{Det} is the throughput performance of a single *Detection Engine* thread in isolation, and $0 < \alpha < 1$ is a coefficient representing the negative effects of residual contention and cache misses induced by the parallel execution of *Detection Engine* threads. α decreases as the number of *Detection Engine* threads increases.

The performance of a *Protocol Processing* thread can be estimated as $P_2 = \min\{\beta P_{\text{Prot}}, N_{\text{Det}} * P_3\}$, where P_{Prot} is the throughput performance of a single *Protocol Processing* thread in isolation, and $0 < \beta < 1$ represents the negative effects of residual contention and cache misses on the *Protocol Processing* thread from downstream *Detection Engines*. Saturation can occur when $\beta P_{\text{Prot}} < N_{\text{Det}} * P_3$, *i.e.*, the *Protocol Processing* becomes the bottleneck. β keeps decreasing with more *Detection Engine* threads. In other words, after saturation of *Protocol Processing*, one can expect the overall throughput to decrease with the addition of more *Detection Engine* threads.

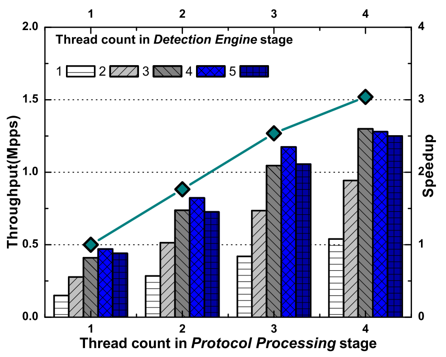

Figure 6: Performance of a single non-optimized *Packet Processing* module. The speedup curve shows the improvement when increasing the number of *Protocol Processing* elements and choosing the best thread number in each pipeline stage.

The performance of the *Packet Capture* stage can be estimated as $P_1 = \min\{\gamma P_{\text{Cap}}, N_{\text{Prot}} * P_2\}$, where P_{Cap} is the throughput performance of a single *Packet Capture* thread in isolation, N_{Prot} is the number of attached parallel *Protocol Processing* threads and $0 < \gamma < 1$ represents the negative effects of residual contention and cache misses on the *Packet Capture* engine from downstream *Protocol Processing* threads and attached *Detection Engines* threads. Increasing the number of *Protocol Processing* threads and the attached *Detection Engines* threads will also decrease the value of γ, decreasing further the overall performance when the *Packet Capture* module is saturated. The above analysis shows that one has to choose carefully the number of cores dedicated to a *Packet Processing* module in order to ensure that maximal throughput is achieved without wasting downstream resources because of saturation of upstream levels.

5.3 Performance of the *Packet Processing* pipeline

Backed by the analysis in previous section we analyse the throughput performance of a *Packet Processing* module and find the optimal number of core to dedicate to each level of the pipeline. We have implemented each thread on a single and dedicated tile core, *i.e.*, the number of core used for a *Packet Processing* module is equal to the number of threads. In this evaluation we use the traffic generator to send 256 Bytes packets at a 10 Gbps rate.

5.3.1 Performance without optimization

In Fig.6, we plot the measured throughput for a single *Packet Processing* module when we increase the number of tile cores allocated to the *Protocol Processing* and the *Detection Engines* stages. We see that the simplest case with a single *Protocol Processing* and a single *Detection Engines* achieves a throughput around 0.15 Mpps. By increasing the number of attached *Detection Engine* threads, we observe a gradual increase in throughput and saturation, at 0.47 Mpps, when the *Protocol Processing* component has four attached *Detection Engine* threads. As predicted by the model, increasing the number of *Detection Engine* threads after saturation only decreases global throughput (for 5 *Detection Engine* threads the throughput drops to 0.44 Mpps). The throughput in the module increases when augmenting the number of *Protocol Processing* threads. We achieve a quasi-linear improvement that, as predicted by the performance model flatten when the number of parallel threads

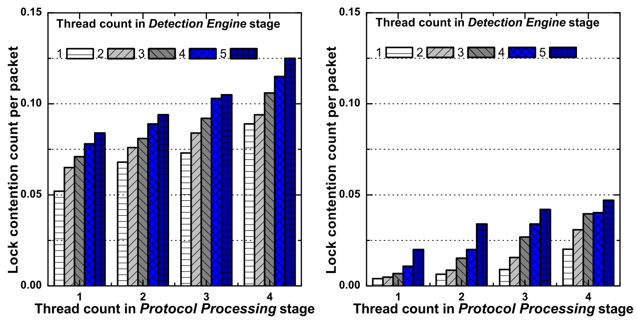

Figure 7: lock contention count per packet during the processing before and after the lock contention optimization

increases the overhead due to threads interaction. Fig.6 enables us to derive the saturation throughput of each level of the pipeline. With a single *Protocol Processing* thread we observe saturation at 0.47 Mpps corresponding to 4 attached *Detection Engine* threads. The highest performance in the figure is achieved on the rightmost part, with 1 *Packet Capture* thread, 4 *Protocol Processing* threads and 3 *Detection Engine* threads for each, *i.e.*, 17 cores dedicated to the *Packet Processing* module, and is 1.3 Mpps.

5.3.2 Lock contention avoidance

The previous evaluation was done without applying any of the optimizations we described in section 4. In this section we will add the first optimization trick and see its impact on the performance. A first level of optimization used the mPIPE capture buffer to alleviate the need for locking the MSG data structure until the *Detection Engine* phase. In the original Suricata implementation, a coarse-grained lock is used for protecting the MSG pool, causing a lot of contention. We proposed to replace the lock by atomic operations with much smaller overhead. We show in Fig.7 the average number of locking attemps per packet, with and without the optimization, and observe a major drop of the contention probability by up to 75%.

We now show in Fig. 8 the throughput of a single *Packet Processing* module after adding the lock contention avoidance optimization. A comparison with Fig.6 shows that the performance of the optimized system improves. The processing ability of threads in the three stages are enhanced correspondingly. The highest performance obtained in the module with 3 *Protocol Processing* threads and 4 *Detection Engine* threads for each, improving from 1.3 Mpps to 1.6 Mpps (a 23% improvement). Moreover the saturation throughput of the *Protocol Processing* thread is now 0.78 Mpps that is an improvement of 66% over the 0.47 Mpps with optimisation.

5.3.3 RPB scheme optimization

The second part of the optimization was related to load balancing. The original Suricata implementation was using a static flow based load balancing scheme that can be unbalanced due to the non-uniform distribution of flow sizes. We proposed earlier to use the hybrid load balancing scheme consisting of the mPIPE provided dynamic adaptive load balancing for the first stage, a simple FSH at the second stage and the RPB scheme for the last stage. We argued that this combination achieves a better load balancing and has a better space locality.

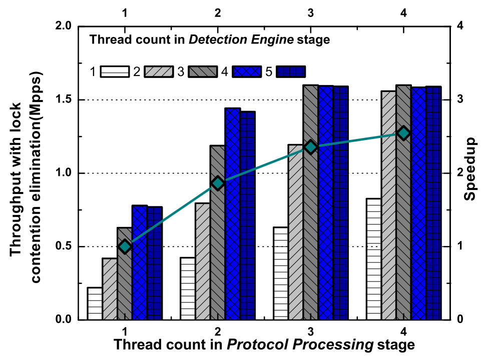

Figure 8: Performance in one *Packet Processing* module with lock contention elimination

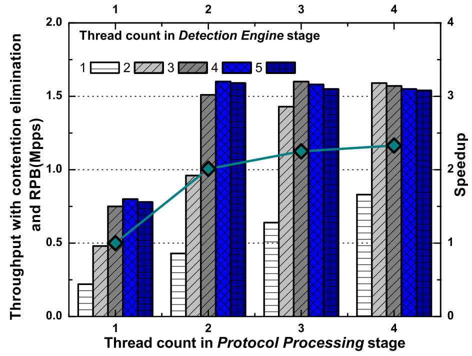

Figure 9: Performance in one *Packet Processing* module with lock contention elimination and RPB scheme

We show in Fig.9 the attained throughput when using this hybrid load balancing. As we observed in previous sections, the performance is now governed by the *Packet Capture* saturation, meaning that one cannot expect that the load balancing will improve a lot the overall performance. Nonetheless, when there are several concurrent *Detection Engines* in the module, the load balancing improves the throughput in the third stage. Take the module with one *Protocol Processing* and two *Detection Engines* for example, the performance in the third stage is improved from 0.42 Mpps (see Fig. 8)to 0.48 Mpps (a 14% improvement). The improvement enables to achieve almost the best throughput of 1.6 Mpps with only 2 *Protocol Processing* threads, each attached with 3 *Detection Engine* threads. This similar performance was happening with 2 *Protocol Processing* threads, each is attached with 4 *Detection Engine* threads without the optimization *i.e.*, best performance can almost be achieved with 9 cores in place of 11 cores. This last point will become very valuable in next section.

5.4 Full system Performance

In previous sections, we described the performance of a single *Packet Processing* module and saw that the saturation throughput of the *Packet Capture* module happens at around 1.6 Mpps, meaning that we cannot go beyond this throughput for a single *Packet Processing* module. Fig.9 shows that this performance can be attained with 11 cores (2 *Protocol Processing*, each attached to 4 *Detection Engine* threads), but 9 cores almost achieve a throughput of 1.5 Mpps (2 *Protocol Processing*, each attached to 3 *Detection Engine* threads, when load-balancing is optimized). Because the TILERAGX36 contains 36 cores, choosing the second

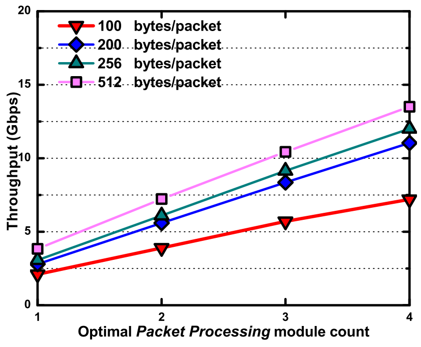

Figure 10: Overall performance of an NIDS with 4 *Packet Processing* optimized engines

configuration enables to run 4 *Packet Processing* modules concurrently, instead of 3 with the first configuration. As we can expect that the throughput of parallel *Packet Processing* modules increases linearly, a more frugal *Packet Processing* in terms of cores is a more attractive option. Also, as almost all data-structures are shared, each additional *Packet Processing* module results in relatively small memory consumption increase.

We show in Fig.10 the performance obtained for a synthetic traffic with different packet sizes ranging from 100 to 512 Bytes over a 10 Gbps interface. As can be seen the overall system can handle traffic with 100 Bytes packet size at 7.2 Gbps (9 Mpps) speed rate. This is almost the maximum rate that can be send over a 10 GE interface considering the padding added by MAC layer. The NIDS was able to handle more than 10 GE bandwidth when connected to two 10 GE interfaces for tests with larger packet sizes. By increasing the packet size to respectively 200 Bytes, 256 Bytes and 512 Bytes, the throughput grows to resp. 11.04 Gbps (6.9 Mpps), 12 Gbps (6 Mpps), 13.5 Gbps (3.3 Mpps). The decreasing gain in packets per second is due to the larger overhead associated to larger packets. In particular the achieved throughput for 256-bytes packets validates the linear improvement with concurrent execution of optimized *Packet Processing* modules. For example, the system achieves a speedup almost 4 with 4 concurrent modules. As the optimal configuration for the module is gained under traffic with 256-bytes packet, the speedups for traffic with 100-bytes and 512-bytes packet are less linear in Fig.10, comparing with the other two curves. For example, the system achieves a speedup about 3.5 with 4 concurrent modules, when processing traffic with 512-bytes packet. Thus we need to get different optimal configurations for different traffic scenario in practice. As it shown in the last paragraph, the optimal configuration for a scenario should be selected carefully and may not occur at the point in one module with the best performance.

5.5 Comparison with existing systems

In order to show the gain obtained with our architecture, we have also implemented raw Suricata and Snort on the TILEARGX36 processor. Porting Suricata on Tilera is easy as it is designed as a pipeline and can be compiled and run directly on the TILERAGX36 processor. For Snort, which is designed as a single thread, we ran several instances and bound each instance through the load balancing offered in hardware by mPIPE. The evaluation is done with 256-bytes packets generated at 10 Gbps.

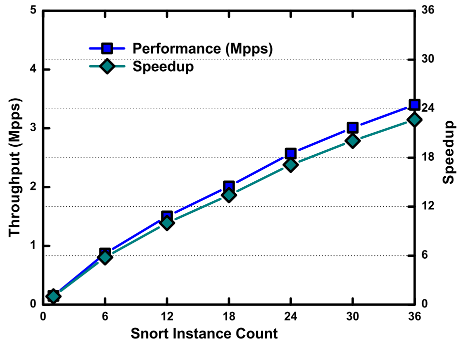

Figure 11: Performance of non-modified Snort on TILERAGX36 processor with data parallelism

The performance of a single instance of non-modified Suricata running a pipeline has already be shown in the leftmost side of Fig.6. We see that with a single *Packet Processing* module, the pipeline saturates with around 4 *Detection Engine* threads and the performance decreases with more threads. The highest throughput achieved is only around 0.47 Mpps when using 6 tile-cores.

We then evaluated the non-modified Snort running as parallel independent threads. We bound each instance of non-modified Snort on a single tile in TILERAGX36. The performance results are shown in Fig.11. As all the instances are isolated, the performance exhibits an almost linear speedup up to 6 instances and saturates thereafter. This is due to the huge memory consumption resulting from independent data. The highest achieved throughput in Fig.11 is around 3.4 Mpps.

We wanted to compare our design with GPU based solutions [8, 9], however we do not have the access to their source code to be able to execute it on the same ruleset and experimental setting we used. However we compared our obtained performance with the reported performance of these two systems with 200 bytes packets. We show the comparison in Table 2. Our achieved throughput is about 6 times larger than that of MIDeA's[8]. Comparing with Kargus [9] that was proposed in 2012, we achieve 60% of their performance. Nonetheless, the Kargus NIDS is using two GPU's and it will be fair to compare its performance with an implementation using two TILERAGX36 processor that would be able to achieve 15% more throughput that the one achieved by Kargus. We added two other elements of comparison the price of the processor and the achieved throughput per dollar spend. We observe that TILERAGX36 based design achieves a throughput per dollar cost of 17.40 Mbps/$ that is 8 times larger than MIDeA's, and almost 3 times larger than Kargus's. As our proposed design is more cost effective, we can easily consider improving the performance by just adding more TILLERAGX36 processors and running them in parallel.

6. CONCLUSION

In this paper, we have proposed a parallel design for high performance, flexible, scalable and inexpensively NIDS on TILERAGX36 many-core processor. We optimized the architecture by exploiting existing features of TILERAGX36 to break the bottlenecks in the parallel design. The system with optimal configuration exhibits linear speedup on TILERAGX36 and can deal with traffic over 13.5 Gbps for

Table 2: Comparison of our parallel design with existing systems(200 Bytes/packet)

name	*publication*	*ruleset size*	*throughput(Gbps)*	*processor cost($)*	*through per dollar(Mbps/$)*
MIDeA	$CCS'2011$	8192	3.2	1138	2.81
Kargus	$CCS'2012$	~ 3000	19	3164	6.01
Proposed design	$\sim$	7571	11.04	650	17.4

512-bytes packets and 7.2 Gbps for 100-bytes packets. Comparing with current proposed NIDS systems on GPU, our parallel design on TILERAGX36 achieve a throughput per dollar that is three fold. The obtained performance looks very promising. As it is possible to have on the same board up to 2 TILERAGX36 many-core processor, we can expect to almost double the attained performance in practice. Moreover as the power consumption of TILERA processor is 50 W at 1.2 Ghz (compare with 750 W for GPU based design), one can even think of several processor cards. In the context of the evolution of the PEARL programmable router platform presented in [29], we are precisely working on that. Another promising aspect of our design is related to the usage of rule based partition that enable our design to easily absorb greater attack complexity and a larger ruleset, especially as future processors will have increasing number of cores. This means that basically we can "scale" our design along two axes: more traffic with more TILERA processor implementing more packet processing modules a more rules with more Detection Engines per packet processing. In our future work, we also plan to mitigate load variation resulting from different rulesets and traffic variation by doing on the fly dynamic tile allocation to accommodate more realistic real scenario.

7. ACKNOWLEDGE

This work was supported in part by the National High-Tech R&D Program of China under Grant No.2013 AA013501, the National Science and Technology Major Project of China under Grant No.2012ZX03002016, the Instrument Developing Project of CAS under Grant No.YZ201229, the National Natural Science Foundation of China (NSFC) with Grants 61202411 and the NSFC-ANR pFlower project with grant No.61061130562.

8. REFERENCES

[1] V. Paxson. Bro: A System for Detecting Network Intruders in Real-time. In Proceedings of the 7th Conference on USENIX Security Symposium, 1998

[2] Zachary K. Baker and Viktor K. Prasanna, "High-throughput Linked-Pattern Matching for Intrusion Detection Systems", In ANCS 2005, Oct 26-28, 2005, Princeton, New Jersey, USA

[3] V. Paxson, R. Sommer, and N. Weaver, "An architecture for exploiting multi-core processors to parallelize network intrusion prevention" In Proceedings IEEE Sarnoff Symposium, May 2007

[4] J. Lee, S. H. Hwang, N. Park, S.-W. Lee, S. Jun, and Y. S. Kim. A High Performance NIDS Using FPGA-based Regular Expression Matching. In Proceedings of the 22nd ACM Symposium on Applied computing (SAC), 2007

[5] Mitra, W. Najjar, and L. Bhuyan. Compiling PCRE to FPGA for accelerating SNORT IDS. In Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems, ANCS, 2007

[6] M. Colajanni and M. Marchetti, "A parallel architecture for stateful intrusion detection in high traffic networks", IEEE IST Workshop on Monitoring, Attack Detection and Mitigation, Tuebingen, Germany, Sept. 2006

[7] Kim, Sunil, and Jun-yong Lee. "A system architecture for high-speed deep packet inspection in signature-based network intrusion prevention." Journal of Systems Architecture 53.5 (2007): 310-320.

[8] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. "MIDeA: A Multi-Parallel Intrusion Detection Architecture", In Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2011.

[9] M. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park. "Kargus: a highly-scalable software-based intrusion detection system", In Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2012

[10] http://vrt-blog.snort.org/

[11] J. Cabrera, J. Gosar, W. Lee, and R. Mehra, ąřOn the statistical distribution of processing times in network intrusion detectionąś In 43rd IEEE Conference on Decision and Control, Dec 2004, pp. 75ĺC80.

[12] TILE-Gx8036 product brief http://www.tilera.com/sites/default/files/productbriefs/TILE-Gx8036_PB033-02_web.pdf

[13] Salminen, Erno, Ari Kulmala, and Timo D. Hamalainen. "Survey of network-on-chip proposals." white paper, OCP-IP (2008): 1-13.

[14] www.suricata-ids.org

[15] www.openinfosecfoundation.org

[16] Wentzlaff, David, et al. "On-chip interconnection architecture of the tile processor." Micro, IEEE 27.5 (2007): 15-31.

[17] Sourdis, Ioannis, et al. "Packet pre-filtering for network intrusion detection." ACM/IEEE Symposium on Architecture for Networking and Communications systems, 2006

[18] Suleman, M. Aater, Moinuddin K. Qureshi, and Yale N. Patt. "Feedback-directed pipeline parallelism." Proceedings of the 19th international conference on Parallel architectures and compilation techniques. ACM, 2010.

[19] Supra-linear packet processing performance with intel multi-core processors white paper. Intel Corporation, 2006.

[20] Removing System Bottlenecks in Multi-threaded Applications white paper. Intel Corporation, 2008.

[21] Schuff, Derek L., Yung Ryn Choe, and Vijay S. Pai. "Conservative vs. optimistic parallelization of stateful network intrusion detection." IEEE International

Symposium on Performance Analysis of Systems and software, 2008

[22] Chen, Xinming, et al. ”Para-snort: A multi-thread snort on multi-core ia platform.” Proceedings of Parallel and Distributed Computing and Systems (PDCS) (2009).

[23] Martin, Ruediger, Michael Menth, and Michael Hemmkeppler. ”Accuracy and dynamics of hash-based load balancing algorithms for multipath Internet routing.” Broadband Communications, IEEE International Conference on Networks and Systems, 2006

[24] Weiguang Shi, Lukas Kencl, Sequence-preserving adaptive load balancers, Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems, December 03-05, 2006, San Jose, California, USA

[25] Haiyang Jiang, Gaogang Xie, Kavé Salamatian, Load Balancing by Ruleset Partition for Parallel IDS on Multi-Core Processors, International Conference on Computer Communications and Networks, ICCCN 2013

[26] C. Grozea, Z. Bankovic, and P. Laskov, FPGA vs. multi-core cpus vs. gpus: Hands-on experience with a sorting application, In Conference Facing the Multicore-Challenge, pp.105-117, 2010

[27] P. P. C. Lee, T. Bu, and G. P. Chandranmenon, A Lock-free, Cache efficient Multi-core Synchronization Mechanism for Line-rate Network Traffic Monitoring, in IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2010), Atlanta, GA, April 2010, pp.1012

[28] http://oprofile.sourceforge.net/download/

[29] Xie, Gaogang, et al. ”PEARL: a programmable virtual router platform.” Communications Magazine, IEEE 49.7 (2011): 71-77.