Haetae: Scaling the Performance of Network Intrusion Detection with Many-core Processors

Jaehyun Nam*, Muhammad Jamshed, Byungkwon Choi,
Dongsu Han, and KyoungSoo Park

School of Computing*, Department of Electrical Engineering
KAIST
{namjh,ajamshed,cbkbrad,dongsu_han,kyoungsoo}@kaist.ac.kr

Abstract. In this paper, we present the design and implementation of Haetae, a high-performance Suricata-based NIDS on many-core processors (MCPs). Haetae achieves high performance with three design choices. First, Haetae extensively exploits high parallelism by launching NIDS engines that independently analyze the incoming flows at high speed as much as possible. Second, Haetae fully leverages programmable network interface cards to offload common packet processing tasks from regular cores. Also, Haetae minimizes redundant memory access by maintaining the packet metadata structure as small as possible. Third, Haetae dynamically offloads flows to the host-side CPU when the system experiences a high load. This dynamic flow offloading utilizes all processing power on a given system regardless of processor types. Our evaluation shows that Haetae achieves up to 79.3 Gbps for synthetic traffic or 48.5 Gbps for real packet traces. Our system outperforms the best-known GPU-based NIDS by 2.4 times and the best-performing MCP-based system by 1.7 times. In addition, Haetae is 5.8 times more power efficient than the state-of-the-art GPU-based NIDS.

Keywords: Many-core processor, Network intrusion detection system, Parallelism, Offloading

1 Introduction

High-performance network intrusion detection systems (NIDSes) are gaining more popularity as network bandwidth is rapidly increasing. As traditional perimeter defense, NIDSes oversee all the network activity on a given network, and alarm the network administrators if suspicious intrusion attempts are detected. As the edge network bandwidth of large enterprises and campuses expands to 10+ Gbps over time, the demand for high-throughput intrusion detection keeps on increasing. In fact, NIDSes are often deployed at traffic aggregation points, such as cellular core network gateways or near large ISP's access networks, whose aggregate bandwidth easily exceeds a multiple of 10 Gbps.

Many existing NIDSes adopt customized FPGA/ASIC hardware to meet the high performance requirements [4,13]. While these systems offer monitoring throughputs of 10+ Gbps, it is often very challenging to configure and adapt such systems to varying network conditions. For example, moving an FPGA application to a new device requires non-trivial modification of the hardware logic even if we retain the same application semantics [25]. In addition, specialized hardware often entails high costs and a long development cycle.

On the other hand, commodity computing hardware, such as multi-core processors [3, 15] and many-core GPU devices [2, 9], offers high flexibility and low cost because of its mass production advantage. In addition, recent GPU-based NIDSes [23,34] enable high performance, comparable to that of hardware-based approaches. However, adopting GPUs leads to a few undesirable constraints. First, it is difficult to program GPU to extract the peak performance. Since GPU operates in a single-instruction-multiple-data (SIMD) fashion, the peak performance is obtained only when all computing elements follow the same instruction stream. Satisfying this constraint is very challenging and often limits the GPU applicability to relatively simple tasks. Second, large number of GPU cores consume a significant amount of power. Even with recent power optimization, GPUs still use a significant portion of the overall system power. Finally, discrete GPUs incur high latency since packets (and their metadata) need to be copied to GPU memory across the PCIe interface for analysis. These extra PCIe transactions often exacerbate the lack of CPU-side memory bandwidth, which degrades the performance of other NIDS tasks.

Recent development of system-on-chip many-core processors [8,16] has bridged the technology gap between hardware- and software-based systems. The processors typically employ tens to hundreds of processing cores, allowing highly-flexible general-purpose computation at a low power budget without the SIMD constraint. For example, EZchip TILE-Gx72 [16], the platform that we employ in this paper, has 72 processing cores where each core runs at 1 GHz but consumes only 1.3 watts even at full speed (95 watts in total). With massively parallel computation capacity, a TILE platform could significantly upgrade the performance of NIDS.

In this paper, we explore the high-performance NIDS design space on a TILE platform. Our guiding design principle is to balance the load across many cores for high parallelism while taking advantage of the underlying hardware to minimize the per-packet overhead. Under this principle, we design and implement Haetae, our high-performance NIDS on TILE-Gx72, with the following design choices. First, we run a full NIDS engine independently on each core for high performance scalability. Unlike the existing approach that adopts the pipelining architecture [24], our system removes all the inter-core dependency and minimizes CPU cycle wastes on inter-core communication. Second, we leverage the programmable network interface cards (NICs) to offload per-packet metadata operations from regular processing cores. We also minimize the size of packet metadata to eliminate redundant memory access. This results in significant savings in processing cycles. Finally, Haetae dynamically offloads the network flows

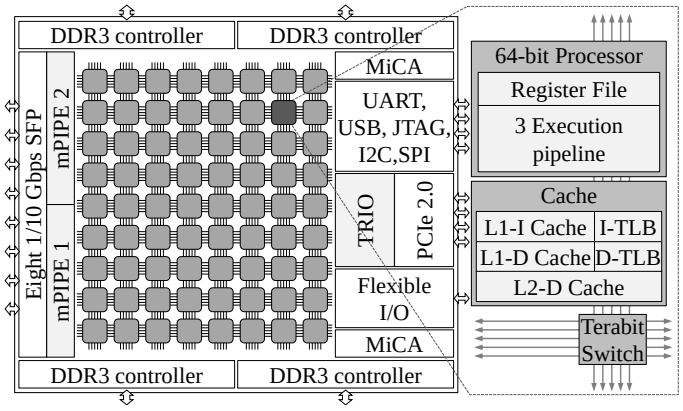

Fig. 1. Overall architecture of TILE-Gx72 processor

to host-side CPU for analysis when the system experiences a high load. We find that the host offloading greatly improves the performance by exploiting available computing cycles of different processor types.

We implement Haetae by extending open-source Suricata [14] optimized for TILE-Gx72 processors. Our evaluation shows that Haetae achieves 79.3 Gbps for large synthetic packets, a factor of 1.7 improvement over the MCP-based Suricata. Our system outperforms Kargus [23], the best-known GPU-based NIDS, by a factor of 2.4 with 2,435 HTTP rules given by Snort 2.9.2.1 [29] that Kargus used. With real traffic traces, the performance of Haetae reaches 48.5 Gbps, which is 1.9 times higher throughput than that of the state-of-the-art GPU-based NIDS. In terms of power efficiency, Haetae consumes 5.8 times less power than the GPU-based NIDS.

While we focus on the development of Haetae on TILE-Gx72 in this paper, we believe that our design principles can be easily ported to other programmable NICs and many-core processors as well.

2 Background

In this section, we provide a brief overview of many-core processors using EZchip TILE-Gx72 as a reference processor. We then describe the operation of a typical signature-based NIDS.

2.1 Overview of EZchip TILE-Gx

Figure 1 shows the architecture of the EZchip TILE-Gx72 processor with 72 processing cores (called tiles in the TILE architecture). Each tile consists of a 64-bit, 5-stage very-long-instruction-word (VLIW) pipeline with 64 registers, 32 KB L1 instruction and data caches, and a 256 KB L2 set-associative cache. TILE-Gx72 does not provide a local L3 cache, but the collection of all L2 caches

serves as a distributed L3 cache, resulting in a shared L3 cache of 18 MB. Fast L3 cache access is realized by a high-speed mesh network (called iMesh), which provides lossless routing of data and ensures cache coherency among different tiles. The power efficiency comes from relatively low clock speed (1 to 1.2 GHz), while a large number of tiles provide ample computation cycles.

The TILE-Gx72 processor contains special hardware modules for network and PCIe interfaces as well. mPIPE is a programmable packet I/O engine that consists of ten 16-bit general-purpose processors dedicated for packet processing. mPIPE acts as a programmable NIC by directly interacting with the Ethernet hardware with a small set of API written in C. mPIPE is capable of performing packet I/O at line speed (up to 80 Gbps), and its API allows to perform direct memory access (DMA) transactions of packets into the tile memory, inspect packet contents, and perform load-balancing. The primary goal of the mPIPE module is to evenly distribute incoming packets to tiles. Its packet processors help parse packet headers and balance the traffic load across all tiles: a feature that closely resembles the receive-side scaling (RSS) algorithm available in modern NICs. The mPIPE processors can be programmed to check the 5-tuples of each packet header (*i.e.*, source and destination IP addresses, source and destination ports, and protocol ID) and to consistently redirect the packets of the same TCP connection to the same tile.

Besides the mPIPE module, the TILE-Gx72 processor also has the TRIO hardware module, which performs bidirectional PCIe transactions with the host system over an 8-lane PCIev2 interface. The TRIO module maps its memory region to the host side after which it handles DMA data transfers and buffer management tasks between the tile and host memory. TRIO is typically used by the host system to manage applications running in a TILE platform. Since the TILE platform does not have direct access to block storage devices, some TILE applications also use TRIO to access host-side storage using FUSE. In this work, we extend the stock TRIO module to offload flow analyzing tasks to the host machine for Haetae.

The TILE processors are commonly employed as PCIe-based co-processors. TILEncore-Gx72 is a PCIe device that has the TILE-Gx72 processor and eight 10 GbE interfaces [5], and we call it TILE platform (or simply TILE-Gx72) in this paper.

2.2 Overview of the Suricata NIDS

We use a TILE-optimized version of Suricata v1.4.0 [14] provided by EZchip. We refer to it as baseline Suricata (or simply Suricata) in this paper. Baseline Suricata uses a *stacked* multi-threaded model where each thread is affinitized to a tile, and it runs a mostly independent NIDS engine except for flow table management and TRIO-based communication. It follows a semi-pipelining architecture where a portion of NIDS tasks are split across multiple tiles. The incoming traffic is distributed to the tiles, and each tile has the ownership of its share of the traffic. In this work, we extend baseline Suricata to support the design choices we make for high NIDS performance.

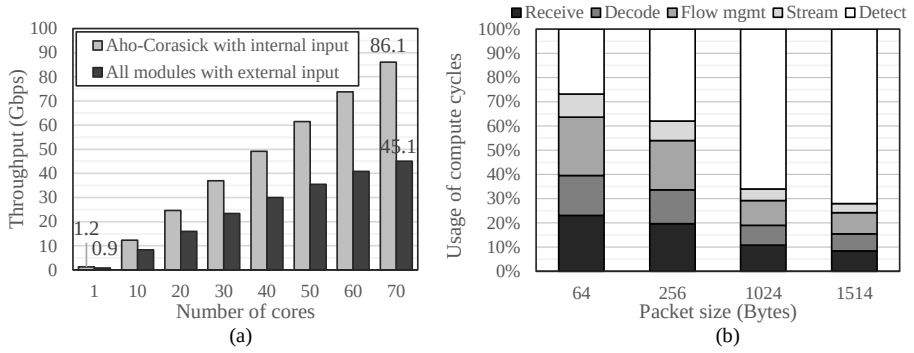

Fig. 2. Performance bottleneck analysis of Baseline Suricata: (a) Throughputs of the Aho-Corasick algorithm over varying numbers of TILE-Gx72 cores, (b) CPU usage breakdown of Suricata modules over various packet size

Incoming packets to Suricata go through the following five NIDS modules.

1. The **receive** module reads packets through packet I/O engines. In commodity desktop and server machines, such packet I/O engines may include PF_RING [11], PSIO [20], and DPDK [7]. Haetae, on the other hand, uses EZchip's mPIPE module for network I/O communication. After receiving a batch of packets from the mPIPE module, the NIDS allocates memory for each ingress packet and initializes the corresponding packet data structure.
2. The **decode** module parses packet headers and fills the relevant packet substructures with protocol-specific metadata. As a last step, it registers the incoming packets with the corresponding flows.
3. The **stream** module handles IP defragmentation and TCP segment reassembly. It also monitors IP-fragmented and TCP-segmented evasion attacks as mentioned in [21].
4. The **detect** module inspects the packet contents against attack signatures (also known as rules). This phase performs deep packet inspection by scanning each byte in the packet payloads. It first checks if a packet contains possible attack strings (*e.g.*, multi-string matching) and if so, more rigorous regular expression matching is performed to confirm an intrusion attempt. This two-stage pattern matching allows efficient content scanning by avoiding regular expression matching on the innocent traffic.
5. Finally, the **output** module logs the detection of possible intrusions based on the information from the matched signatures.

3 Approach to High Performance

In this section, we identify the performance bottlenecks of baseline Suricata on the TILE platform and describe our basic approach to addressing them.

3.1 Performance Bottlenecks of Suricata

A typical performance bottleneck of a signature-based NIDS is its pattern matching. However, for TILE-Gx72, we find that parallel execution of pattern matching may provide enough performance while per-packet overhead related to metadata processing takes up a large fraction of processing cycles.

To demonstrate this, we measure the performance of a multi-pattern matching (MPM) algorithm (Aho-Corasick algorithm [17], which is the de-facto multi-string matching scheme adopted by many software-based NIDSes [14,23,29,34]). Figure 2 (a) shows the performance of the MPM algorithm on the TILE platform without packet I/O and its related NIDS tasks. For the experiment, we feed in newly-created 1514B TCP packets with random payloads from the memory to the pattern matching module with 2,435 HTTP rules from the Snort 2.9.2.1 ruleset. We observe that the performance scales up linearly as the number of cores grows, peaking at 86.1 Gbps with 70 cores. The pattern matching performance is reasonable for TILE-Gx72 that has eight 10G network interfaces.

However, if we generate packets over the network, the overall performance drops by more than 40 Gbps. This means that modules other than pattern matching must be optimized for overall performance improvement. To reveal a detailed use of processing cycles, we measure the fraction of compute cycles spent on each NIDS module. The results in Figure 2 (b) show that tasks other than pattern matching (i.e., the detect module) take up 28 to 72% of total processing cycles, depending on the packet size. The tile usage for the non-pattern matching portion is a fixed overhead per packet as the fraction gets higher for smaller packets.

Our detailed code-level analysis reveals that these cycles are mostly used to process packet metadata. They include the operations, such as decoding the protocol of each packet, managing concurrent flows, and reassembling TCP streams for each incoming packet. In this work, we focus on improving the performance of these operations, since the overall NIDS performance often depends on the performance of these operations while leveraging the unique hardware-level features of TILE-Gx72.

3.2 Our Approach

Our strategy for a high-performance NIDS is two folds. First, we need to parallelize pattern matching as much as possible to give the most compute cycles to the performance-critical operation. This affects the basic architecture of the NIDS, which will be discussed in more detail in the next section. Second, we need to reduce the overhead of the per-packet operation as much as possible. For the latter, we exploit the special hardware provided by the TILE-Gx72 platform. More specifically, our system leverages mPIPE and TRIO for offloading some of the heavy operations from regular tiles. mPIPE is originally designed to evenly distribute the incoming packets to tiles by their flows, but we extend it to perform per-packet metadata operations to reduce the overhead on regular tiles. The key challenge here is that the offloaded features need to be carefully chosen

because the mPIPE processors provide limited compute power and memory access privilege. TRIO is mostly used to communicate with the host-side CPU for monitoring the application behavior. We extend the TRIO module to pass the analyzing workload to the host side when the TILE platform experiences a high load. That is, we run a host-side NIDS for extra flow analysis. The challenge here is to make efficient PCIe transfers to pass the flows and to dynamically determine when to deliver the flows to the host side. We explain the design in more detail in the next section.

4 Design

In this section, we provide the base design of Haetae, and describe three optimizations: mPIPE computation offloading, lightweight metadata structure, and dynamic host-side flow analysis.

4.1 Parallel NIDS Engine Architecture

Haetae adopts the multi-threaded parallel architecture where each thread is running a separate NIDS engine, similar to [23]. Each NIDS engine is pinned to a tile, and repeats running all NIDS tasks in sequence from receive to output modules. This is in contrast to the pipelining architecture used by earlier TILE-based Suricata [24] where each core is dedicated to perform one or a few modules and the input packets go through multiple cores for analysis. Pipelining is adopted by earlier versions of open-source Suricata, but it suffers from a few fundamental limitations. First, it is difficult to determine the number of cores that should be assigned for each module. Since the computation need of each module varies for different traffic patterns, it is hard to balance the load across cores. Even when one module becomes a bottleneck, processing cores allocated for other modules cannot help alleviate the load of the busy module. This leads to load imbalance and inefficient usage of computation cycles. Second, pipelining tends to increase inter-core communication and lock contention, which is costly in a high-speed NIDS. Since an NIDS is heavily memory-bound, effective cache usage is critical for good performance. In pipelining, however, packet metadata and payload have to be accessed by multiple cores, which would increase CPU cache bouncing and reduce the cache hits. Also, concurrent access to the shared packet metadata would require expensive locks, which could waste processing cycles.

To support our design, we modify baseline Suricata to eliminate any shared data structures, such as the flow table. Each thread maintains its own flow table while it removes all locks needed to access the shared table entry. Incoming packets are distributed to one of the tiles by their flows, and a thread on each tile analyzes the forwarded flows without any intervention by other threads. Since each thread only needs to maintain a small amount of flow ranges, dividing the huge flow table into multiple pieces for each thread is not a big trade-off. Thus, this *shared-nothing architecture* ensures high scalability while it simplifies the implementation, debugging, and configuration of an NIDS.

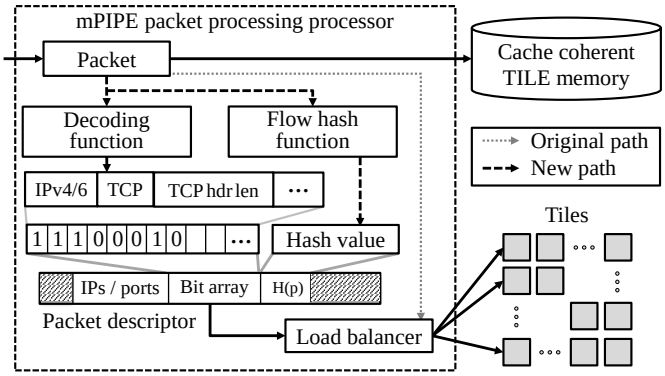

Fig. 3. Design of the mPIPE engine with decoding and hash computations

One potential concern with this design is that each core may not receive the equal amount of packets or flows from the NICs. However, recent measurements in a real ISP show that a simple flow-based load balancing scheme like RSS more or less evenly distributes the flows among the processing cores [35]. According to the study, the maximum difference in the number of processed flows per each core on a 16-core server is within 0.2% of all flows at any given time with real traffic. This implies that the randomness of IP addresses and port numbers used in real traffic is sufficient to distribute the packet load evenly among the tiles.

4.2 mPIPE Computation Offloading

With the highly-scalable system architecture in place, we now focus on optimizing per-tile NIDS operations. Specifically, we reduce the packet processing overhead on a tile by offloading some common computations to the mPIPE programmable hardware module. When a packet arrives at a network interface, mPIPE allocates a packet descriptor and a buffer for the packet content. The packet descriptor has packet metadata such as timestamps, size, pointer to the packet content as well as some reserved space for custom processing. After packet reception, the software packet classifier in mPIPE distributes the packet descriptors to one of the tile queues, and the tile accesses the packet content with the packet descriptor. mPIPE allows the developers to replace the packet classifier with their custom code to change the default module behavior.

Programming in mPIPE, however, is not straightforward due to a number of hardware restrictions. First, in the case of mPIPE, it allows only 100 compute cycles per packet to execute the custom code at line rate. Second, the reserved space in the packet descriptor is limited to 28 bytes, which could be too small to perform intensive computations. Third, mPIPE embedded processors are designed mainly for packet classification with a limited instruction set and programming libraries. They consist of 10 low-powered 16-bit processors, which do not allow flexible operations such as subroutines, non-scalar data types (*e.g.*, structs and pointers), and division (remainder) operations.

Given these constraints, Haetae offloads two common packet processing tasks of an NIDS: packet protocol decoding and hash computation for flow table lookup. We choose these two functions for mPIPE offloading since they should run for every packet but do not maintain any state. Also, they are relatively simple to implement in mPIPE while they save a large number of compute cycles on each tile.

Figure 3 shows how the customized mPIPE module executes protocol decoding and flow hash computation. A newly-arriving packet goes through packet decoding and flow hash functions, saving results to the reserved area of an mPIPE packet descriptor. Out of 28 bytes of total output space, 12 bytes are used for holding the packet address information (*e.g.*, source and destination addresses and port numbers) and 4 bytes are used to save a 32-bit flow hash result. The remaining 12 bytes are employed as a bit array to encode various information: whether it is an IPv4 or IPv6 packet, whether it is a TCP or UDP packet, the length of a TCP header in the case of the TCP packet, etc. Each bit can indicate multiple meanings depending on protocols. After these functions, a load balancer determines which tile should handle the packet, and the packet descriptor along with the packet is directly passed onto the L2 cache of the tile that handles the packet, a feature similar to Intel data direct I/O [6]. As a result, each NIDS thread can proceed with the pre-processed packets and avoids memory access latencies.

Our micro-benchmarks show that mPIPE offloading improves the performance of the decode and flow management modules by 15 to 128% (in section 6). Since these are per-packet operations, the cycle savings are more significant with smaller packets.

4.3 Lightweight Metadata Structure

mPIPE computation offloading confirms that reducing the overhead of per-packet operation greatly improves the performance of the overall NIDS. The root cause for performance improvement is reduced memory access and enhanced cache access efficiency. More efficient cache utilization leads to a smaller number of memory accesses, which minimizes the wasted cycles due to memory stalls. If the reduced memory access is a part of per-packet operation, the overall savings could be significant since a high-speed NIDS has to handle a large number of packets in a unit time.

To further reduce the overhead of per-packet memory operation, we simplify the packet metadata structure of baseline Suricata. Suricata's packet metadata structure is bloated since it has added support for many network and transport-layer protocols over time. For example, the current data structure includes packet I/O information (*e.g.*, PCAP [10], PF_RING [11], mPIPE), network-layer metadata (*e.g.*, IPv4, IPv6, ICMP, IGMP) and transport-layer metadata (*e.g.*, TCP, UDP, SCTP). The resulting packet metadata structure is huge (1,920 bytes), which is not only overkill for small packets but also severely degrades the cache utilization due to redundant memory access. Also, the initialization cost for metadata structure (*e.g.*, memset() function calls) would be expensive.

To address these concerns, we modify the packet metadata structure. First, we remove the data fields for unused packet I/O engines. Second, we separate the data fields for protocols into two groups: those that belong to frequently-used protocols such as TCP, UDP, and ICMP and the rest that belong to rarely-used protocols such as SCTP, PPP, and GRE. We move the data fields for the latter into a separate data structure, and adds a pointer to it to the original structure. If an arriving packet belongs to one of rarely-used protocols, we dynamically allocate a structure and populate the data fields for the protocol. With these optimizations, the packet metadata structure is reduced to 384 bytes, five times smaller than the original size. Our profiling results find that the overall number of cache misses is reduced by 54% due to lightweight metadata structures.

4.4 Flow Offloading to Host-side CPU

Since TILE-Gx72 is typically attached to a commodity server machine, we could improve the NIDS performance further if we harness the host-side CPU for intrusion detection. The TILE-Gx72 platform provides a TRIO module that allows communication with the host machine. We exploit this hardware feature to offload extra flows beyond the capacity of the TILE processors to the host-side CPU.

The net performance increase by host-side flow offloading largely depends on two factors: *(i)* how fast the TILE platform transfers the packets to the host machine over its PCIe interface, and *(ii)* the pattern matching performance of the host-side NIDS. In our case, we use a machine containing two Intel E5-2690 CPUs (2.90 GHz, 16 cores in total) that run Kargus with only CPUs [23]. Since the performance of the Aho-Corasick algorithm in Kargus is about 2 Gbps per CPU core [23], the host-side NIDS performance would not be an issue given the 8-lane PCIev2 interface (with 32 Gbps maximum bandwidth in theory) employed by the TILE platform.

We first describe how we optimize the TRIO module to efficiently transfer packets to the host side, and explain which packets should be selected for offloading. Also, we determine when the packets should be offloaded to the host machine to maximize the performance of both sides.

Efficient PCIe Communication Baseline Suricata provides only rudimentary host offloading support mainly used for remote message logging; since the TILE platform does not have built-in secondary storage, it periodically dispatches the batched log messages from its output module to the host-side storage via its TRIO module. Since log transmission does not require a high bandwidth, the stock TRIO module in the baseline Suricata code is not optimized for high-speed data transfer. First, the module does not exploit zero-copy DMA support. Second, it does not exercise parallel I/O in PCIe transactions, incurring a heavy contention in the shared ring buffer. Our measurement shows that the stock TRIO module achieves only 5.7 Gbps of PCIe data transfer throughput at best out of the theoretical maximum of 32 Gbps.

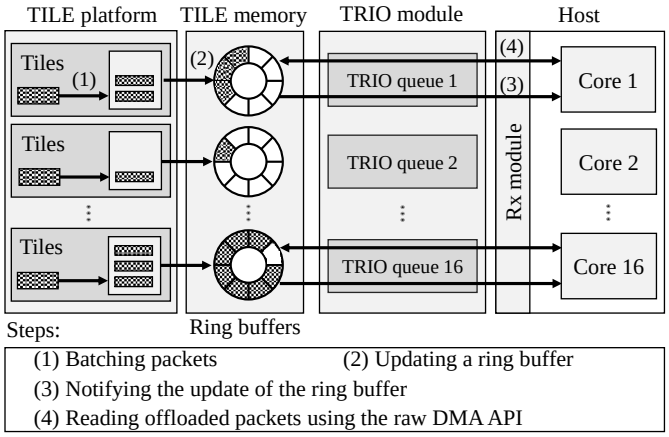

Fig. 4. Design of the offloading module with the TRIO engine

We exploit three features in the TRIO module to maximize the PCIe transfer performance for high host-side NIDS throughput. First, we develop a zero-copying offloading module with the raw-DMA API provided by the TRIO engine. The raw-DMA API ensures low-latency data transfer between the TILE platform and the host. It requires physically-contiguous buffers to map the TILE memory to the host-side address space. For zero-copy data transfer, we pre-allocate shared packet buffers at initialization of Suricata, which are later used by mPIPE for packet reception. Packets that need to be offloaded are then transferred via the TRIO module without additional memory copying, which greatly saves compute cycles. Second, we reduce the contention to the shared ring buffer by increasing the number of TRIO queues. The baseline version uses a single ring buffer, which produces severe contention among the tiles. We increase the number to 16, which is the maximum supported by our TILE platform. This allows parallel queue access both from tiles and CPU cores. Finally, we offload multiple packets in a batch to amortize the cost incurred due to per-packet PCIe transfer. Our packet offloading scheme is shown in Figure 4. We find that these optimizations are very effective, improving the performance of PCIe transfer by 5 to 28 times over the stock version.

Dynamic Flow Offloading We design the TRIO offloading module to fully benefit from the hardware advantage of the TILE platform. We make the TILE platform handle as much traffic as possible to minimize the power consumption and the analyzing latency. To determine when to offload the packets to the host side, each tile monitors whether it is being under pressure by checking the queue size in mPIPE. A large build-up in the queue indicates that the incoming load may be too large for the tile to catch up. Figure 5 shows the design of the dynamic offloading algorithm in Haetae. The basic idea is similar to opportunistic packet offloading to GPU in [23], but the unit of offloading is a flow in our case, and the task for offloading is the entire flow analysis instead of only pattern matching.

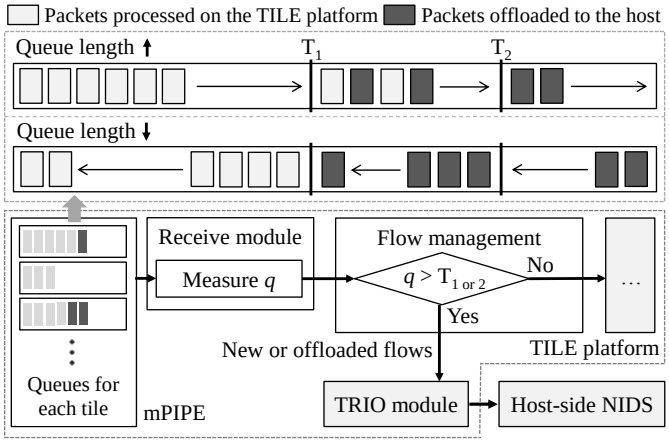

Fig. 5. Dynamic flow offloading

In our algorithm, we use two thresholds to determine whether a new flow should be offloaded or not. If the queue length (q) exceeds the first threshold (T_1), a small portion (L_1) of new flows are chosen to be offloaded to the host machine. If it successfully curbs the queue size blowup, Haetae reverts to TILE-only flow analysis and stops offloading to the host side. However, if the queue size increases beyond the second threshold (T_2), a larger portion (L_2, typically, $L_2 = 1$) of new flows is offloaded to the host machine, which helps drain the queue more quickly. When the queue length exceeds the second threshold, the system keeps the offloading rate to L_2 until the queue length goes below the first threshold (T_1). This two-level offloading scheme prevents rapid fluctuation of the queue length, which would stabilize flow processing in either mode.

The unit of offloading should be a flow since the host-side NIDS is independent of the TILE-side NIDS. The host-side NIDS should receive all packets in a flow to analyze the protocol as well as reassembled payload in the same flow. To support flow-level offloading, we add a bit flag to each flow table entry to mark if a new packet belongs to a flow being offloaded or not. This extra bookkeeping, however, slightly reduces the per-tile analyzing performance since it is rather heavy per-packet operation.

5 Implementation

We implement Haetae by extending a TILE-optimized Suricata version from EZchip. This version optimizes the Aho-Corasick algorithm with special TILE memory instructions, and uses a default mPIPE packet classifier to distribute incoming packets to tiles. To support the design features in section 4, we implement per-tile NIDS engine, mPIPE computation offloading, lightweight packet metadata structure, and dynamic host-side flow offloading. This requires a total of 3,920 lines of code modification of the baseline Suricata code.

For shared-nothing, parallel NIDS engine, we implement a lock-free flow table per each tile. By assigning a dedicated flow table to each NIDS engine, we eliminate access locks per flow entry and improve the core scalability. The flow table is implemented as a hash table with separate chaining, and the table entries are pre-allocated at initialization. While the baseline version removes idle flow entries periodically, we adopt lazy deletion of such entries to reduce the overhead of per-flow timeouts. Idle flow entries are rare, so it suffices to delete them in chain traversal for other activities only when there is memory pressure. To maximize the parallelism, we run an NIDS engine on 71 tiles out of 72 tiles. The remaining tile handles shell commands from the host machine.

Supporting lightweight packet metadata structure is the most invasive update since the structure is used by all modules. To minimize code modification and to hide the implementation detail, we provide access functions for each metadata field. This requires only 360 lines of code modification, but it touches 32 source code files.

Implementing mPIPE computation offloading is mostly straightforward except for flow hash calculation. Baseline Suricata uses Jenkin's hash function [1] that produces a 32-bit result, but implementing it with a 16-bit mPIPE processor requires us to emulate 32-bit integer operations with 16-bit and 8-bit native instructions. Also, we needed to test whether protocol decoding and hash calculation is within the 100-cycle budget so as not to degrade the packet reception performance. mPIPE offloading modifies both the existing mPIPE module and Suricata's decode and flow management modules, which requires 130 and 100 lines of new code, respectively.

For dynamic host-side flow offloading, we implement 1,700 lines of code on the tile side and 1,040 lines of code on the host side. First, we modify the receive module to measure the load of each tile and to keep track of the flows that are being offloaded to the host. Second, we implement the tile-to-host packet transfer interface with a raw DMA API provided by TRIO. Finally, we modify the CPU-only version of Kargus to accept and handle the traffic passed by the TILE platform.

6 Evaluation

Our evaluation answers three aspects of Haetae:

1. We quantify the performance improvement and overhead of mPIPE and host-side CPU offloading. Our evaluation shows that the mPIPE offloading improves the performance of the decode and flow management modules by up to 128% and the host-side CPU offloading improves the overall performance by up to 34%.
2. Using synthetic HTTP workloads, we show the breakdown of performance improvement for each of our three techniques and compare its overall performance with Kargus with GPU and baseline Suricata on the TILE platform. The result shows that Haetae achieves up to 2.4x improvements, over Kargus and baseline Suricata.

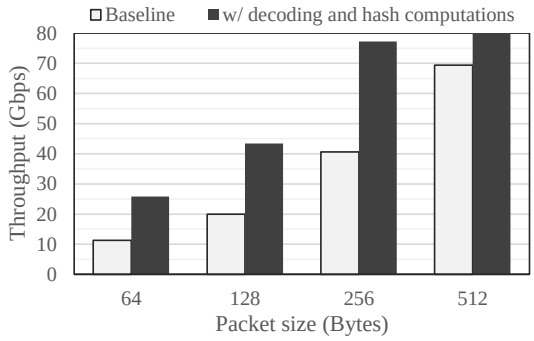

Fig. 6. Throughputs of the decoding and flow management modules with mPIPE offloading. The throughputs are line rate (80 Gbps) for 1024 and 1514B packets.

3. Finally, we evaluate the NIDS performance using real traffic traces obtained from the core network of one of the nation-wide cellular ISPs in South Korea. Haetae achieves a throughput of 48.5 Gbps, which is a 92% and 327% improvement respectively over Kargus and baseline Suricata.

6.1 Experimental setup

We install a TILE-Gx72 board on a machine with dual Intel E5-2690 CPUs (octacore, 2.90 GHz, 20 MB L3 cache) with 32 GB of RAM. We run Haetae on the TILE platform and CPU-based Kargus on the host side. Each NIDS is configured with 2,435 HTTP rules from the Snort 2.9.2.1 ruleset. For packet generator, we employ two machines that individually have dual Intel X5680 CPUs (hexacore, 3.33 GHz, 12 MB L3 cache) and dual-port 10 Gbps Intel NICs with the 82599 chipset. Our packet generator is based on PSIO [20] that can transmit packets at line rate (40 Gbps each) regardless of packet size. For real traffic evaluation, we replay 65GB of packet traces obtained from one of the largest cellular ISPs in South Korea [35]. We take the Ethernet overhead (such as preamble (8B), interframe gap (12B), and checksum (4B)) into consideration when we calculate a throughput.

6.2 Computation Offloading Overhead

This section quantifies the performance benefit and overhead of mPIPE and TRIO offloading.

mPIPE offloading overhead We first verify whether offloaded computations adversely affect mPIPE's packet I/O throughput. For this, we disable all NIDS modules other than the receive module, and compare the packet acquisition throughputs with and without mPIPE computation offloading. We generate TCP packets of varying size from 64 to 1514 bytes and measure the throughput for each packet size. Our result shows that even with mPIPE computation offloading

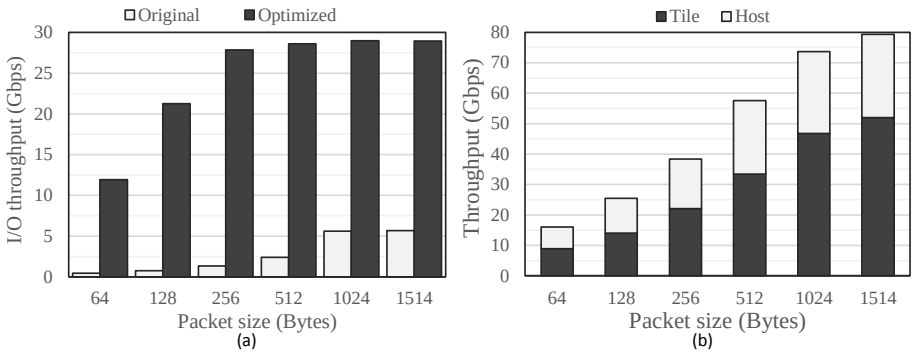

Fig. 7. TRIO performance benchmarks: (a) TRIO throughputs with and without our optimizations, (b) Throughputs with flow offloading

packet I/O achieves line rates (80Gbps) regardless of packet size. This confirms that the offloaded computations are within the cycle budget of the mPIPE processors, and offloading does not adversely affect the packet I/O performance.

We then evaluate the performance improvement achieved by mPIPE offloading. Figure 6 compares the performances with and without offloading. To focus on the performance improvement by packet reception and flow management, we enable the receive, decode, and flow management modules and disable other modules (*e.g.*, stream and detect modules) for the experiments.

The mPIPE offloading shows 15 to 128% improvement over baseline Suricata depending on the packet size. Because mPIPE offloading alleviates per-packet overhead, improvement with small packets is more noticeable than with large packets. In sum, the results show that computation offloading to mPIPE brings significant performance benefits in the NIDS subtasks.

TRIO offloading overhead We now measure TRIO's throughput in sending and receiving packets over the PCIe interface. Note this corresponds to the maximum performance improvement gain achievable using host-side flow offloading. We compare the throughputs of our optimized TRIO module and the existing one. Figure 7 (a) shows the throughputs by varying packet sizes. The original TRIO module cannot achieve more than 5.7 Gbps of throughput because it first copies data into its buffer to send data across the PCIe bus. Such additional memory operations (i.e., memcpy()) significantly decrease the throughputs. Our optimized TRIO is up to 28 times faster. The relative improvement increases as the packet size increases because the overhead of DMA operation is amortized. The throughput saturates at 29 Gbps over for packets larger than 512B, which is comparable to the theoretical peak throughput of 32 Gbps for an 8-lane PCIe-v2 interface. Note that the raw channel rate of a PCIe-v2 lane is 5 Gbps, and the use of the 8B/10B encoding scheme limits the peak effective bandwidth to 4 Gbps per lane. Figure 7 (b) shows end-to-end throughputs of Haetae with the CPU-side flow offloading by varying packet size. By exploiting both the TILE

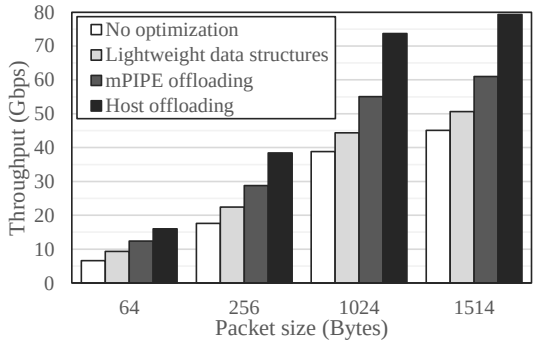

Fig. 8. Breakdown of performance improvement by each technique

processors and host-side CPUs, we improve the overall NIDS performance by 18 Gbps, from 61 to 79.3 Gbps, for 1514B packets. While the overall performance is improved, we notice that the TILE-side performance degrades by 9 Gbps (to 52 Gbps in Figure 7 (b)) when TRIO offloading is used. This is because extra processing cycles are spent on PCIe transactions for packet transfers. We also note that the improvement with larger packets is more significant. This is because the PCIe overhead is relatively high for small-sized packets and the CPU-side IDS throughput with small packets is much lower compared to its peak throughput obtained for large packets. Despite the fact, the flow offloading improves the performance by 79% for 64B packets. Given that the average packet size in real traffic is much larger than 100B [35], we believe that the actual performance improvement would be more significant in practice.

6.3 Overall NIDS Performance

Figure 8 shows the performance breakdown of the three key techniques under synthetic HTTP traffic. The overall performance ranges from 16 to 79 Gbps depending on the packet size. mPIPE offloading and metadata reduction achieve 33% (1514B packets) to 88% (64B packets) improvements and CPU-side flow offloading achieves 32% additional improvement on average. Through the results, we find that reducing the per-packet operations significantly improves the overall NIDS performance, and we gain noticeable performance benefits by utilizing the host resources. Figure 9 (a) shows the performances of Haetae compared to other systems under the synthetic HTTP traffic. We compare with the baseline Suricata, customized for Tilera TILE-Gx processors, and Kargus with two NVIDIA GTX580 GPUs. In comparison with baseline Suricata, Haetae shows 1.7x to 2.4x performance improvement. We also see 1.8x to 2.4x improvement over Kargus in throughput (except for 64B packets). The relatively high performance of Kargus for 64B packets mainly comes from its batched packet I/O and batched function calls, which significantly reduces the overhead for small packets. In case of Haetae, we find that batch processing in mPIPE is ineffective in packet reception due to different hardware structure.

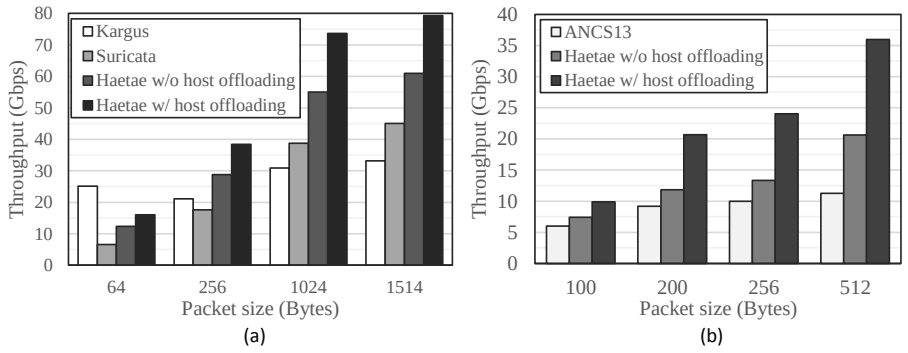

Fig. 9. Performance comparison with (a) synthetic HTTP workloads, (b) the NIDS proposed in ANCS '13 [24] (36 tiles)

Here, we compare Haetae with a pipelined NIDS design in [24]. Because the source code is not available, we resort to indirect comparison by taking the performance number measured using a TILE-Gx36 processor from [24]. Since the clock speeds of the TILE-Gx36 (1.2 GHz) and TILE-Gx72 (1.0 GHz) processors are different, we scale down the performance numbers in the paper. For fair comparison, we use only 36 tiles for Haetae but increase the number of rules (7,867 rules), similar to [24]. Figure 9 (b) shows the final results. While previous work achieves 6 to 11.3 Gbps for 100 to 512B packets, Haetae without host offloading achieves 7.4 to 20.6 Gbps for the same size, which is 1.2 to 1.8x more efficient. Moreover, Haetae with host offloading achieves 1.7 to 3.2x improvements over the previous work. The improvements come from two main reasons. First, unlike pipelining, Haetae's parallel architecture reduces load imbalance and inefficient usage of the tiles. We observe that the performance of [24] flattens at 512B packets, presumably due to the overheads of pipelining. Second, Haetae saves the computation cycles by applying the mPIPE offloading and the lightweight metadata structures.

In terms of power consumption, Haetae is much more efficient: Haetae with host offloading (TILE-Gx72 and two Intel E5-2690 CPUs) shows 0.23 Gbps per watt while Kargus (two Intel X5680 CPUs and two NVIDIA GTX580 GPUs) achieves only 0.04 Gbps per watt, spending 5.8x more power than Haetae.

6.4 Real Traffic Performance

We evaluate the performance with real traffic traces obtained from a 10 Gbps LTE backbone link at one of the largest mobile ISPs in South Korea. We remove unterminated flows from the real traffic traces and shape them to increase the overall transmission rate (up to 53 Gbps). The real traffic trace files are first loaded into RAM before packets are replayed. The files take up 65 GB of physical memory (2M TCP flows, 89M packets). To increase the replay time, we replay the files 10 times repeatedly. Like the previous measurements, we use the same ruleset (2,435 HTTP rules) as well.

IDS	Baseline Suricata	Kargus	Haetae
Throughput	11.6 Gbps	25.2 Gbps	48.5 Gbps

Table 1. Performance comparison with the real traffic

Table 1 shows the throughputs of Haetae and other NIDSes. With the real traces, Haetae is able to analyze 4.2x and 1.9x more packets than Baseline Suricata and Karugs respectively. While Haetae achieves up to 79.3 Gbps with the synthetic workload, the throughput with the real workload decreases due to two major reasons. First, the modules related to flows are fully activated. Unlike the synthetic workload, the real workload has actual flows. The flow management module needs to keep updating flow states and the stream module also needs to reassemble flow streams. Thus, these modules consume much more cycles with the real workload than with the synthetic workload. Second, while the synthetic workload consists of packets of the same size, the real traffic has various data and control packets of different sizes. The average packet size of the real traffic traces is 780 bytes, and the throughput is 16% lower than that of 512B packets in the synthetic workload.

7 Related Work

We briefly discuss related works. We categorize the previous NIDS works into three groups by their hardware platforms: dedicated-hardware, general-purpose multi-core CPU, and many-core processors.

NIDS on dedicated-hardware: Many works have attempted to scale the performance of pattern matching with dedicated computing hardware, such as FPGA, ASIC, TCAM, and network processors. Barker et al. implement the Knuth-Morris-Pratt string matching algorithm on an FPGA [18]. Mitra et al. develops a compiler that converts Perl-compatible regular expression (PCRE) rules into VHDL code to accelerate the Snort NIDS [27]. Their VHDL code running on an FPGA achieves 12.9 Gbps of PCRE matching performance. Tan et al. implement the Aho-Corasick algorithm on an ASIC [32]. Yu et al. employ TCAMs for string matching [36] while Meiners et al. optimize regular expression matching with TCAMs [26]. While these approaches ensure high performance, a long development cycle and a lack of flexibility limit its applicability.

NIDS on multi-core CPU: Snort [29] is one of the most popular software NIDSes that run on commodity servers. It is initially single-threaded, but more recent versions like SnortSP [12] and Para-Snort [19] support multi-threading to exploit the parallelism of multi-core CPU. Suricata [14] has the similar architecture as Snort and it allows multiple worker threads to perform parallel pattern matching on multi-core CPU.

Most of multi-threaded NIDSes adopt pipelining as their parallel execution model: they separate the packet receiving and pattern matching modules to a different set of threads affinitized to run on different CPU cores so that the

incoming packets have to traverse these threads for analysis. As discussed earlier, however, pipelining often suffers from load imbalance among the cores as well as inefficient CPU cache usage.

One reason for the prevalence of pipelining in early versions of multi-threaded software NIDSes is that popular packet capture libraries like pcap [10] and network cards at that time did not support multiple RX queues. For high performance packet acquisition, a CPU core had to be dedicated to packet capture while other CPU cores were employed for parallel pattern matching. However, recent development of multi-queue NICs and multi-core packet I/O libraries such as PF_RING [11], PSIO [20], netmap [28] allows even distribution of incoming packets to multiple CPU cores, which makes it much easier to run an independent NIDS engine on each core. Haetae takes the latter approach, benefiting from the mPIPE packet distribution module while it avoids the inefficiencies from pipelining.

NIDS on many-core processors: Many-core GPUs have recently been employed for parallel pattern matching. Gnort [33] is the seminal work that accelerates multi-string and regular expression pattern matching using GPUs. Smith et al. confirm the benefit of the SIMD architecture for pattern matching, and compare the performance of deterministic finite automata (DFA) and extended finite automata (XFA) [30] on G80 [31]. Huang et al. develop the Wu-Manber algorithm for GPU, which outperforms the CPU version by two times [22]. More recently, Snort-based NIDSes like MIDeA [34] and Kargus [23] demonstrate that the performance of software engines can be significantly improved by hybrid usage of multi-core CPU and many-core GPU. For example, Kargus accepts incoming packets at 40 Gbps with PSIO [20], a high-performance packet capture library that exploits multiple CPU cores. It also offloads the Aho-Corasick and PCRE pattern matching to two NVIDIA GPUs while it performs function call batching and NUMA-aware packet processing. With these optimizations, Kargus achieves an NIDS throughput over 30 Gbps on a single commodity server.

Jiang et al. have proposed a Suricata-based NIDS on a TILE-Gx36 platform with 36 tiles [24]. While their hardware platform is very similar to ours, their NIDS architecture is completely different from Haetae. Their system adopts pipelining from Suricata and mostly focuses on optimal partitioning of tiles for tasks. In contrast, Haetae adopts per-tile NIDS engine and focuses on reducing per-packet operations and offloading flows to host machine. We find that our design choices provide performance benefits over their system: 20 to 80% performance improvement in a similar setting.

8 Conclusion

In this paper, we have presented Haetae, a highly scalable network intrusion detection system on the Tilera TILE-Gx72 many-core processor. To exploit high core scalability, Haetae adopts the shared-nothing, parallel execution architecture which simplifies overall NIDS task processing. Also, Haetae offloads heavy per-packet computations to programmable network cards and reduces the packet

metadata access overhead by carefully re-designing the structure. Finally, Haetae benefits from dynamic CPU-side flow offloading to exploit all processing power in a given system. We find that our design choices provide a significant performance improvement over existing state-of-the-art NIDSes with great power efficiency. We believe that many-core processors serve as a promising platform for high-performance NIDS and our design principles can be easily adopted to other programmable NICs and many-core processors as well.

Acknowledgments

We thank anonymous reviewers of RAID 2015 for their insightful comments on our paper. This research was supported in part by SK Telecom [G01130271, Research on IDS/IPS with many core NICs], and by the ICT R&D programs of MSIP/IITP, Republic of Korea [14-911-05-001, Development of an NFV-inspired networked switch and an operating system for multi-middlebox services], [R0190-15-2012, High Performance Big Data Analytics Platform Performance Acceleration Technologies Development].

References

1. A hash function for hash table lookup. http://www.burtleburtle.net/bob/hash/doobs.html
2. AMD: OpenCL Zone. http://developer.amd.com/tools-and-sdks/
3. AMD Opteron Processor Solutions. http://products.amd.com/en-gb/opteroncpuresult.aspx
4. Check Point IP Appliances. http://www.checkfirewalls.com/IP-Overview.asp
5. EZchip TILEncore-Gx72 Intelligent Application Adapter. http://tilera.com/products/?ezchip=588&spage=606
6. Intel Data Direct I/O Technology. http://www.intel.com/content/www/us/en/io/direct-data-i-o.html
7. Intel DPDK. http://dpdk.org/
8. Kalray MPPA 256 Many-core processors. http://www.kalrayinc.com/kalray/products/#processors
9. NVIDIA: What is GPU Computing? http://www.nvidia.com/object/what-is-gpu-computing.html
10. PCAP. http://www.tcpdump.org/pcap.html
11. PF_RING. http://www.ntop.org/products/pf_ring
12. SnortSP (Security Platform). http://blog.snort.org/2014/12/introducing-snort-30.html
13. Sourcefire 3D Sensors Series. http://www.ipsworks.com/3D-Sensors-Series.asp
14. Suricata Open Source IDS/IPS/NSM engine. http://suricata-ids.org/
15. The Intel Xeon Processor E7 v2 Family. http://www.intel.com/content/www/us/en/processors/xeon/xeon-processor-e7-family.html
16. TILE-Gx Processor Family. http://tilera.com/products/?ezchip=585&spage=614
17. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Communications of the ACM 18(6), 333–340 (1975)

18. Baker, Z.K., Prasanna, V.K.: Time and area efficient pattern matching on FPGAs. In: Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA). pp. 223–232. ACM (2004)
19. Chen, X., Wu, Y., Xu, L., Xue, Y., Li, J.: Para-snort: A multi-thread snort on multi-core ia platform. In: Proceedings of the Parallel and Distributed Computing and Systems(PDCS) (2009)
20. Han, S., Jang, K., Park, K., Moon, S.: Packetshader: a gpu-accelerated software router. vol. 41, pp. 195–206 (2011)
21. Handley, M., Paxson, V., Kreibich, C.: Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In: USENIX Security Symposium. pp. 115–131 (2001)
22. Huang, N.F., Hung, H.W., Lai, S.H., Chu, Y.M., Tsai, W.Y.: A GPU-based multiple-pattern matching algorithm for network intrusion detection systems. In: Proceedings of the International Conference on Advanced Information Networking and Applications - Workshops (AINAW). pp. 62–67. IEEE (2008)
23. Jamshed, M.A., Lee, J., Moon, S., Yun, I., Kim, D., Lee, S., Yi, Y., Park, K.: Kargus: a highly-scalable software-based intrusion detection system. In: Proceedings of the ACM Conference on Computer and Communications Security(CCS). pp. 317–328 (2012)
24. Jiang, H., Zhang, G., Xie, G., Salamatian, K., Mathy, L.: Scalable high-performance parallel design for network intrusion detection systems on many-core processors. In: Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems(ANCS). IEEE Press (2013)
25. Kuon, I., Tessier, R., Rose, J.: FPGA architecture: Survey and challenges. In: Foundations and Trends in Electronic Design Automation. vol. 2, pp. 135–253. Now Publishers Inc. (2008)
26. Meiners, C.R., Patel, J., Norige, E., Torng, E., Liu, A.X.: Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems. In: Proceedings of the 19th USENIX conference on Security. pp. 8–8. USENIX Association (2010)
27. Mitra, A., Najjar, W., Bhuyan, L.: Compiling PCRE to FPGA for accelerating Snort IDS. In: Proceedings of the ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS). pp. 127–136. ACM (2007)
28. Rizzo, L.: netmap: A novel framework for fast packet i/o. In: USENIX Annual Technical Conference. pp. 101–112 (2012)
29. Roesch, M., et al.: Snort - lightweight intrusion detection for networks. In: Proceedings of the USENIX Systems Administration Conference (LISA) (1999)
30. Smith, R., Estan, C., Jha, S., Kong, S.: Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. In: ACM SIGCOMM Computer Communication Review. vol. 38, pp. 207–218 (2008)
31. Smith, R., Goyal, N., Ormont, J., Sankaralingam, K., Estan, C.: Evaluating gpus for network packet signature matching. In: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software(ISPASS) (2009)
32. Tan, L., Sherwood, T.: A high throughput string matching architecture for intrusion detection and prevention. In: ACM SIGARCH Computer Architecture News. vol. 33, pp. 112–122. IEEE Computer Society (2005)
33. Vasiliadis, G., Antonatos, S., Polychronakis, M., Markatos, E.P., Ioannidis, S.: Gnort: High performance network intrusion detection using graphics processors. In: Recent Advances in Intrusion Detection (RAID). pp. 116–134. Springer (2008)

34. Vasiliadis, G., Polychronakis, M., Ioannidis, S.: Midea: a multi-parallel intrusion detection architecture. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS). pp. 297–308 (2011)
35. Woo, S., Jeong, E., Park, S., Lee, J., Ihm, S., Park, K.: Comparison of caching strategies in modern cellular backhaul networks. In: Proceeding of the annual international conference on Mobile systems, applications, and services (MobiSys). pp. 319–332. ACM (2013)
36. Yu, F., Katz, R.H., Lakshman, T.V.: Gigabit rate packet pattern-matching using tcam. In: Proceedings of the IEEE International Conference on Network Protocols(ICNP). pp. 174–183. IEEE (2004)