MIDeA: A Multi-Parallel Intrusion Detection Architecture

Giorgos Vasiliadis
FORTH-ICS, Greece
gvasil@ics.forth.gr

Michalis Polychronakis
Columbia University, USA
mikepo@cs.columbia.edu

Sotiris Ioannidis
FORTH-ICS, Greece
sotiris@ics.forth.gr

ABSTRACT

Network intrusion detection systems are faced with the challenge of identifying diverse attacks, in extremely high speed networks. For this reason, they must operate at multi-Gigabit speeds, while performing highly-complex per-packet and per-flow data processing. In this paper, we present a multi-parallel intrusion detection architecture tailored for high speed networks. To cope with the increased processing throughput requirements, our system parallelizes network traffic processing and analysis at three levels, using multi-queue NICs, multiple CPUs, and multiple GPUs. The proposed design avoids locking, optimizes data transfers between the different processing units, and speeds up data processing by mapping different operations to the processing units where they are best suited. Our experimental evaluation shows that our prototype implementation based on commodity off-the-shelf equipment can reach processing speeds of up to 5.2 Gbit/s with zero packet loss when analyzing traffic in a real network, whereas the pattern matching engine alone reaches speeds of up to 70 Gbit/s, which is an almost four times improvement over prior solutions that use specialized hardware.

Categories and Subject Descriptors

C.2.0 [**General**]: Security and Protection

General Terms

Design, Performance, Security

Keywords

Intrusion Detection, Pattern Matching, Acceleration, GPU, NIDS

1. INTRODUCTION

Network intrusion detection systems (NIDS) are commonly classified into *anomaly-based* and *signature-based* systems. Anomaly-based systems are used to detect unknown attacks, but usually generate false positives [8,42]. In contrast, signature-based systems are typically more precise, but cannot detect attacks for which they do not have a signature, and therefore, they require continuous updating [33, 36]. Due to their low false-positive rate and their higher performance, signature-based detection approaches are the basis for the majority of the existing NIDSs. Unfortunately, as the speed of network links increases, keeping up with the inspection of all traffic becomes quite challenging.

A number of approaches have been proposed to address the problem of matching stateful signatures in high-speed networks, both *hardware* and *software* based. Hardware-based implementations offer a scalable method of inspecting packets in high-speed environments [6,9,10,24,26,27,29,45]. These systems usually consist of special-purpose hardware, such as FPGAs, CAMs, and ASICs, that is used to process network packets in parallel. Although the use of specialized hardware achieves high processing rates, most implementations require custom programming, and are usually tied to the underlying device. As a consequence, they are very difficult to extend and reprogram. Additionally, most of these approaches focus on the raw inspection of the network packets alone, without implementing crucial functionalities of modern NIDSs, such as protocol analysis and application-level parsing.

In contrast, software implementations based on commodity processors are low-cost and easily programmable. The advent of multi-core processors has lead researchers to employ them for high-speed traffic processing. Previous approaches have focused extensively on multi-core general-purpose processors [19, 20, 37, 39, 47], in which the NIDSs operations are decomposed to different processing elements. Graphics processors have also been used to boost computationally intensive tasks, like string searching [18, 41, 48] and regular expression matching [49].

The majority of these approaches take advantage of parallelism only at a single level, either through traffic splitting [21, 47], flow-level parallelization [19, 20, 39], or content inspection [18, 41, 48, 49]. In practice, however, the performance of modern NIDSs depends on several operations, including packet capture and decoding, TCP stream reassembly, and application-level protocol analysis. A scalable architecture *must* exploit parallelism for each operation individually, otherwise Amdahl's Law will fundamentally limit the performance that the hardware can provide [34].

In this paper, we present MIDeA, a new model for network intrusion detection systems, which combines commodity, general-purpose hardware components in a single-node design, tailored for high-performance network traffic analysis. Our system takes advantage of the parallelism offered by modern network interface cards, multiple CPUs, and multiple GPUs, to improve scalability and runtime performance. By mapping each operation to the appropriate device, we implemented a NIDS with no serialized components—no component needs to be synchronized and wait for another component to finish its execution, or contend for a shared

CCS'11, October 17–21, 2011, Chicago, Illinois, USA.

resource. This design allows for significant performance gains. On a single box, MIDeA performs stateful packet analysis and inspection at 5.2 Gbit/s, while the raw processing throughput of the computationally-intensive pattern matching operations exceeds 70 Gbit/s when offloaded to the GPUs.

In summary, the main contributions of this work are:

- We introduce a novel multi-parallel architecture for high-performance processing and stateful analysis of network traffic. Our architecture is based on inexpensive, off-the-shelf, general-purpose hardware, and combines multi-queue NICs, multi-core CPUs, and multiple GPUs.

- We present our prototype implementation based on Snort [36], the most widely used open-source NIDS, demonstrating that the proposed model is practical and can be adopted by existing systems.

- We present the design and implementation of a number of system-level optimizations that improve end-to-end performance. We demonstrate that our implementation scales well with the number of processing units.

- We experimentally evaluate our prototype implementation under various configurations, and show that commodity hardware can be used effectively to drastically improve the performance of traffic processing applications. Our evaluation on 10 Gbit/s links demonstrates a significant increase in processing throughput compared to existing approaches.

The rest of the paper is organized as follows. Section 2 presents the design objectives and challenges of our proposed architecture. In Section 3, we describe the architecture of our parallel network intrusion detection system in detail. Section 4 presents optimizations implemented to overcome bottlenecks and reduce specific overheads. In Section 5, we thoroughly evaluate our architecture using different benchmarks and workloads. In Section 6, we discuss the limitations of our system and directions for future work. Finally, we discuss related work in Section 7 and conclude in Section 8.

2. DESIGN OBJECTIVES

We begin by discussing the design principles and practical challenges of mapping the different functional components of a signature-based network intrusion detection system to a multi-parallel system architecture.

Inter-flow Parallelism. Our aim is to design a NIDS architecture that scales with the number of available processing units, enabling it to operate at line-rates without packet loss. The primary role of a NIDS is to passively capture the network packets through the network interface (NIC), process them, and report any suspicious events. Therefore, the main tasks of the NIDS can be summarized as: (i) packet capturing, usually at multi-Gigabit rates, and (ii) packet processing, including TCP stream reassembly, application-level protocol parsing, and pattern matching.

In current hardware NIDS platforms [1, 44], packet processing operates at line-rates, handling a single input port; therefore, the platform must inspect input traffic at several Gigabit per second. Existing software-based NIDS, in contrast, typically follow a multi-core approach and split the traffic at the flow-level to N slices, where N is the number of processing nodes available to the system [39, 47]. Flow-based partitioning achieves an almost even processing load at all processing nodes, without requiring any intra-node communication for processing operations that are limited in scope to a single flow. Traffic is distributed using either an external traffic splitter—which is quite a costly solution—or a software-based load-balancing scheme, where a simple hash function is applied on each captured packet, based on which it is assigned to the appropriate node for processing.

Unfortunately, having many different cores receiving traffic from the same network interface or a shared packet queue, increases contention to the shared resource, which incurs additional delay in packet capturing [19, 20]. This observation leads us to our first design principle: *traffic has to be separated at the network flow level using existing, commodity solutions, without incurring any serialization on the processing path*. In Section 3, we show how our system takes advantage of recent load-balancing technologies such as Receive-Side Scaling (RSS) [3], which allows different cores to receive portions of the monitored traffic directly. This inherently leads us to a multi-core architecture, in which each core runs a separate instance of the inspection engine, processing only a subset of the network flows.

Intra-flow Parallelism. Distributing the monitored traffic to different CPU cores offers significant performance benefits. Recent studies [20, 39] have shown a close-to-linear speedup in the number of cores. However, the CPU is still saturated by the large number of diverse and computationally heavy operations it needs to perform: network flow tracking, TCP stream reassembly, protocol parsing, string searching, regular expression matching, and so on. The problem then is how to further parallelize content inspection on each core, enabling a further increase in the overall traffic processing throughput, *without* incurring any packet loss.

This leads us to our second design principle: *per-flow traffic processing should be parallelized beyond simple per-flow load balancing across different CPU cores*. To enable such "intra-flow" parallelism, network packets from the same flow have to be processed in parallel, while also maintaining flow-state dependencies. In Section 3.2.2, we discuss how our system can take advantage of multiple graphics processors to inspect high-volume traffic concurrently with the CPU cores. Intra-flow parallelism is achieved by buffering incoming packets and transferring them to the graphics card in large batches. Although this buffering scheme adds some latency to the processing path, it pays off in terms of the processing throughput that can be sustained.

By parallelizing both packet pre-processing and content inspection across multiple CPUs and GPUs, the proposed multi-parallel NIDS architecture can operate at line rate in multi-Gigabit networks using solely commodity components. Our parallelization scheme also leads to an architecture that is incrementally extensible in terms of hardware resources. We demonstrate that the overall processing throughput of the system can be increased simply by adding more processing elements.

Resulting Trade-off. A potential issue of our design is the data transfer operations that must take place between the memory address spaces of each device. Specifically, network packets are transferred from the NIC to the main memory of the host, and from there to the device memory of the GPU. However, the extra data transfers between the CPU and the GPU over the PCIe bus can be worth the computational gain offered by the GPU. To further mitigate this data transfer overhead, we have implemented a pipelining scheme that allows CPU and GPU execution to overlap, and consequently hides the added latencies. Although the raw computational power of the GPU offers enough performance benefits even when considering all data transferring overheads, the pipelining scheme that we introduce offers an additional level of parallelism to the overall execution path. In Section 4, we discuss in detail how these optimizations have been implemented in our system.

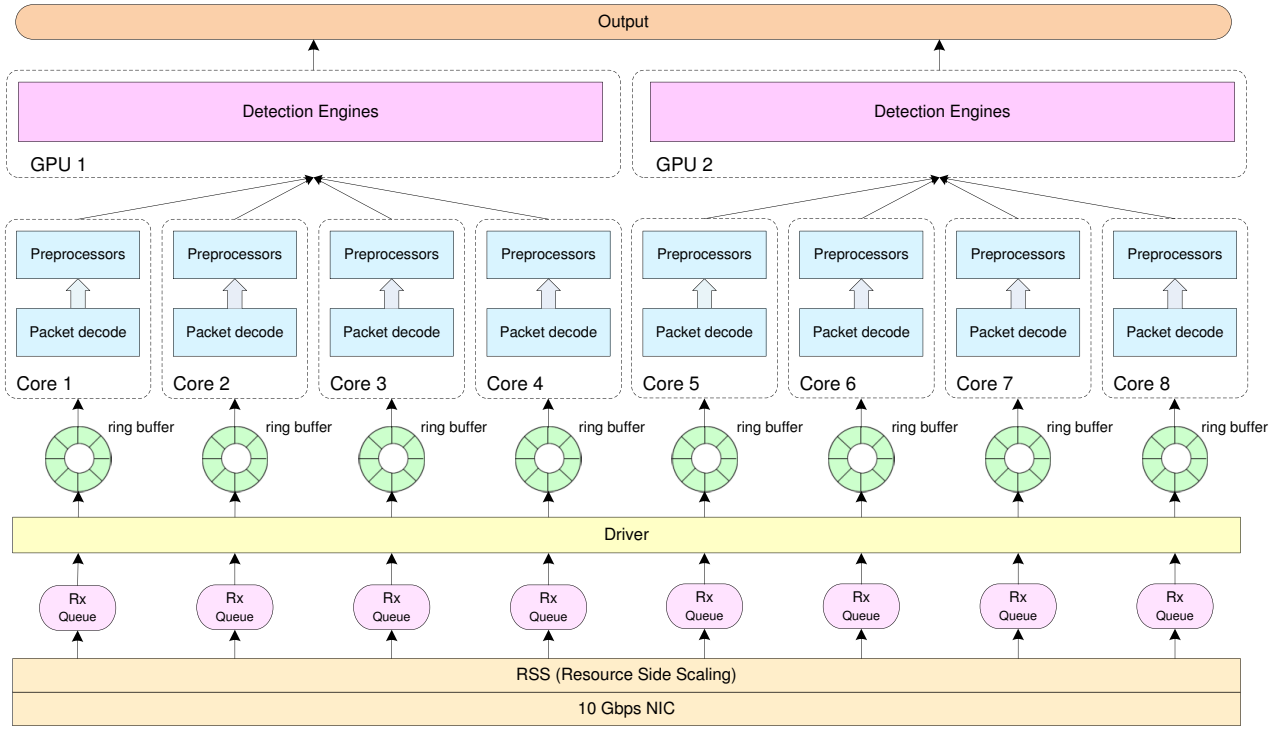


Figure 1: MIDeA architecture.

3. ARCHITECTURE

In this section, we describe the overall design of our multi-parallel network intrusion detection architecture. The key factors for achieving good performance are: (i) load balancing between processing units, and (ii) linear performance scalability with the addition of more processing units. Additionally, for high-performance packet capturing we consider the use of only inexpensive commodity NICs.

As shown in Figure 1, the NIDS application is mapped to the different processing units using both *task* and *data* parallelism across the incoming network flows. In particular, the network interface distributes the captured packets to the CPU-cores, ensuring flow-pinning and equal workload across the cores. Each CPU-core reassembles and normalizes the captured traffic before offloading it to the GPU for pattern matching. Any matching results are logged by the corresponding CPU-core using the specified logging mechanism, such as a file or database.

This design has a number of benefits: First, it does not require any synchronization or lock mechanisms since different cores process different data in isolation. Second, having several smaller data structures (such as the TCP reassembly tables) instead of sharing a few large ones, not only reduces the number of table look-ups required to find a matching element, but also reduces the size of the working set in each cache, increasing overall cache efficiency.

3.1 Packet Capturing

Our system uses 10GbE NICs, which are currently the state-of-the-art general-purpose network interfaces. Capturing packets at these rates is non-trivial and requires the coordinated effort of the network controller and the multi-core CPUs.

3.1.1 Multiqueue NICs

To avoid contention when multiple cores access the same 10GbE port, modern network cards can partition incoming traffic into several Rx-queues [28]. This allows each CPU core to access its own hardware queue independently, while the NIC controller is responsible for classifying incoming network packets and distributing them to the appropriate queue. The Rx-queues are not shared between the CPU cores, eliminating the need of synchronization. Each Rx-queue is dedicated to a specific user-level process that is mapped to a different core, as shown in Figure 1. Each user-level process fetches packets from a single queue and forwards them to the next processing module. The controller can set up a number of Rx-queues equal to the number of available CPU cores (the Intel 82599EB Ethernet controller [2] that we used in our implementation supports up to 128 Rx-queues).

To avoid costly packet copies and context switches between user and kernel space, we use the `PF_RING` network socket [11]. The most efficient way to integrate a `PF_RING` socket with a multi-queue NIC is to dedicate a separate ring buffer for each available Rx-queue [15]. Network packets of each Rx-queue are stored into a separate ring buffer and are pulled by the user-level process through DMA, without going through the kernel's network stack.

We also take into consideration the interrupt handling of each queue. In Linux, interrupts are handled automatically by the kernel through the `irqbalance` daemon. This daemon is responsible for evenly distributing interrupts from each Rx-queue to CPU cores, in a round-robin fashion. Unfortunately, this is not the optimal solution for multi-core systems, because distributing the handling of interrupts from a single Rx-queue to multiple cores results in cache invalidation and performance degradation [25]. This means that `irqbalance` does not guarantee that the interrupt of the next packet of the same flow will be handled by the same core. Therefore, we bind the interrupt handling of each Rx-queue to a specific CPU core by setting the corresponding `/proc/irq/X/smp_affinity` entry (where `X` is the IRQ number of each Rx-queue, which can be obtained from `/proc/interrupts`).

3.1.2 Load Balancing

A major implication when partitioning the incoming traffic to multiple instances is to guarantee that all packets of a specific flow will be processed by the same user-level process. It is also important to distribute the load equally to the different processing cores. Modern NICs [3] support *hash-based* (or *flow-based*), and *address-based* classification schemes. In hash-based schemes, such as Receive-Side Scaling (RSS), a hash function is applied to the protocol headers of the incoming packets in order to assign them to one of the Rx-queues. In address-based schemes, such as Virtual Machine Device Queues (VMDQ), each Rx-queue is assigned a different Ethernet address, to provide an abstraction of a dedicated interface to guest virtual machines.

For our purposes, we choose the hash-based method. The hash function, computed on the typical 5-tuple `<ip_src, ip_dst, port_src, port_dst, protocol>` achieves good distribution among the different queues. The RSS specification [28] allows the explicit parameterization of the tuple fields that will be used to compute the hash. Unfortunately, current RSS-enabled network interfaces (such as the Intel 82599EB that we used) use a fixed hashing type, which only ensures that the packets of the uni-directional streams of a connection will result to the same hash value. This means that the client-to-server stream of the flow will may end up to one Rx-queue, and the server-to-client stream to a different one.

In order to insure that packets of both directions end up into the same ring buffer, a symmetric hashing is further applied on the 5-tuple fields of each packet header. Eventually, all packets of the same flow will always be placed in the same ring buffer, and will be processed by the same user-level process. In addition, we bind the process that reads from each ring buffer to the same core using the CPU affinity of the Linux scheduler (see `sched_setaffinity(2)`), in order to increase cache locality. We assume that the monitored traffic consists of many different concurrent flows (at least as many as the available CPU cores), hence all processes are fed with data. This is not an issue even in small networks, since even a single host usually has tens of concurrent active connections.

3.2 Processing Engine

Incoming traffic is forwarded to the processing engines for analysis. Each processing engine is implemented as a single process and is mapped to a certain CPU core to avoid costs due to process scheduling. The basic functionality of each processing engine is to retrieve the network packets from its assigned hardware queue, decode them and apply higher-level protocol analysis, and finally transfer them to the GPU for content inspection.

3.2.1 Preprocessing

Preprocessing modules are built on top of the decoding subsystem and preprocessor engines of Snort 2.9. The purpose of the decoder is to parse the packet headers according to lower-layer protocols (Ethernet, IP, TCP, and so on). After packets have been decoded, they are sent through a preprocessing stage that includes *flow reassembly* and *protocol analysis*.

TCP packets are reassembled into TCP streams to build the entire application dialog before they are forwarded to the pattern matching engine. Packets that belong to the same direction of a TCP flow, are merged into a single packet by concatenating their payloads according to the TCP protocol. Inspecting the concatenation of several network packets, instead of each network packet separately, enables the handling of overlapping data and other TCP anomalies. This allows the detection engine to match patterns that span multiple packets. Content normalization is also applied for higher-level protocols, such as HTTP and DCE/RPC, to remove potential ambiguities and neutralize evasion tricks.

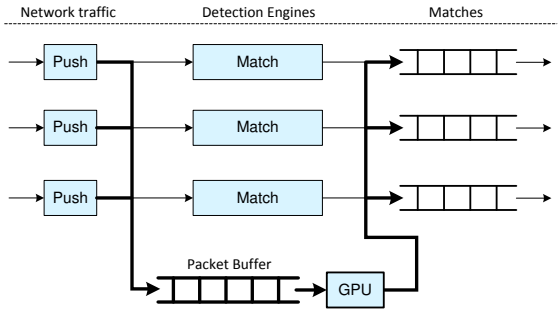


Figure 2: Batching different flows to a single buffer.

Once flow reassembly and normalization is complete, the data is forwarded to the detection engine, which performs signature matching on the incoming traffic. In existing NIDS like Snort [36], the detection signatures are organized in *port groups*, based on the source and destination port numbers of each rule. Additionally, a separate detection engine instance is used to search for the string patterns of a particular rule group. To achieve *intra-flow* parallelization, MIDeA takes advantage of the data-parallel capabilities of modern graphics processors.

Incoming traffic is transferred to the memory space of the GPU in batches. As we discuss in Section 5.2, small transfers results to significant PCIe throughput degradation, hence we batch lots of data together to reduce the PCIe transaction overhead. Also, instead of allocating a different buffer for each port group, we simply mark each packet so that it will be processed by the appropriate detection engine in the searching phase. Consequently, only one buffer is needed per process, instead of one for each port group, as shown in Figure 2. This results to significantly lower memory consumption and reduces response latency for port groups with low traffic. Whenever the buffer gets full, all packets are transferred to the GPU in one operation.

The buffer that is used to collect the network packets is allocated as a special type of memory, called page-locked or "pinned down" memory. Page-locked memory is a physical memory area that does not map to the virtual address space, and thus cannot be swapped out to secondary storage. The use of this memory area results to higher data transfer throughput between the host and the GPU device, because the GPU driver knows the location of the data in RAM and does not have to locate it—neither swap it from disk, nor copy it to a non-pageable buffer—before transferring it to the GPU. Data transfers between page-locked memory and the GPU are performed through DMA, without occupying the CPU.

3.2.2 Parallel Multi-Pattern Engine

A major design criterion for matching large data streams against many different patterns, is the choice of an efficient pattern matching algorithm. The majority of network intrusion detection systems use a flavor of the Aho-Corasick algorithm [5] for string searching, which uses a transition function to match input data. The transition function gives the next state $T[state, ch]$ for a given $state$ and a character ch. A pattern is matched when starting from the *start state* and moving from state to state, the algorithm reaches a *final state*. The memory and performance requirements of Aho-Corasick depend on the way the transition function is represented. In the full representation, each transition is represented with 256 elements, one for each 8-bit character. Each element contains the

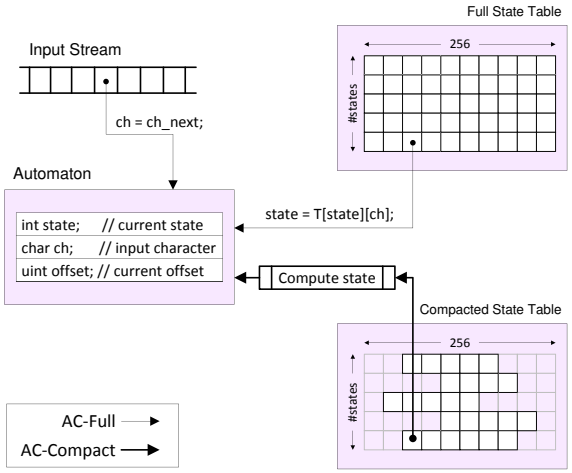


Figure 3: State tables of AC-Full vs. AC-Compact.

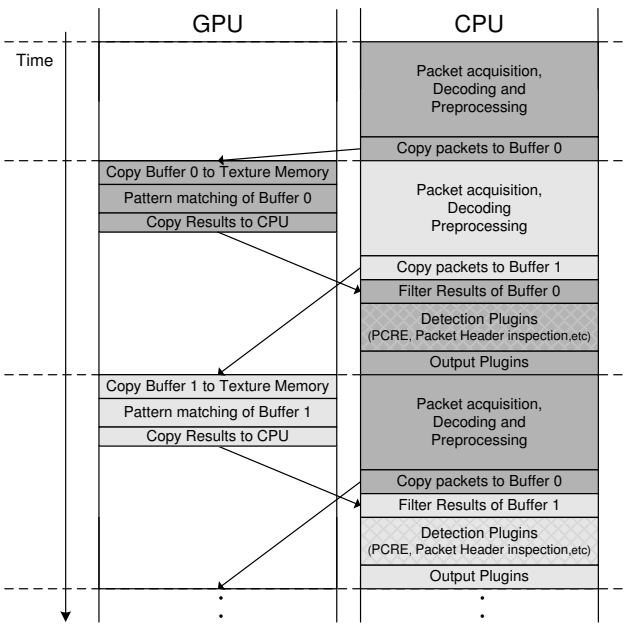


Figure 4: Execution flow of a single CPU process.

next state to move to, hence given an input character, the next state can be found in $O(1)$ steps. This gives a linear complexity over the input data, independently on the number of patterns, which is very efficient in terms of performance.

In the full state representation, hereinafter *AC-Full*, every possible input byte leads to *at most* one new state, which ensures high performance. Unfortunately, a full state representation requires large amounts of memory, even for small signature sets. When compiling the whole rule set of Snort, the size of the compiled state table can reach up to several hundreds Megabytes of memory. On most modern graphics cards, available memory is not a constraint any more, since they are usually equipped with ample amounts of memory—a GeForce GTX480 comes with 1.5GB of memory at a reasonable price. Unfortunately, in the CUDA runtime system [32], on which MIDeA is based, each CPU thread is executed in its own CUDA context. In other words, a different memory space has to be allocated in the GPU for each process, since they cannot share memory on the GPU device. As we discuss in Section 5.2, when using the AC-Full algorithm, only the detection engines of a single Snort instance can fit in the memory space of the GPU. That means that only one Snort instance can fully utilize the GPU at a time.

To overcome the memory sharing limitation of CUDA and maintain scalability, it is important to keep the memory requirements low. Instead of creating a full state table, we use a *compacted state table* structure for representing the compiled patterns [31]. The compacted state table is represented in a banded-row format, where only the elements from the first non-zero value to the last non-zero value of the table are actually stored. The number of the stored elements is known as the *bandwidth* of the sparse table. In our new implementation, *AC-Compact*, the next state is not directly accessible while matching input bytes, but it has to be computed, as shown in Figure 3. This computation adds a small overhead at the searching phase, which is amortized by the significantly lower memory consumption.

Moreover, it is common that many patterns are case-insensitive, or share the same final state in the transition table. Instead of inserting every different combination of lowercase and capital letters for the pattern, we simply insert only one combination (i.e., all characters are converted to lowercase), and mark that pattern in the pattern list as case-insensitive. In case the pattern is matched in a packet, an extra case-insensitive search should be made at the index where the pattern was found. If two patterns share the same final list (i.e., the match list contains more than one pointers to patterns), the patterns contained in the list have to be verified for finding the actual match.

Each packet is processed by a different GPU thread. Packets are stored into an array, which dimensions are equal to the number of the packets that are processed at once and the Maximum Transmission Unit (MTU). Packets that exceed MTU (which is 1500 bytes in Ethernet) are splitted down into several smaller ones, and are copied in consecutive rows in the array. To detect attacks that span multiple rows, each thread continues its search to the following portions of the packet (if any) iteratively, until a *final* or *fail state* is reached.

3.2.3 *Multi-GPU Support*

A key feature of MIDeA is its support for pattern matching using several GPUs at a data-parallel level. Modern motherboards, such as the one we used in our evaluation, support multiple GPUs on the PCI Express bus. MIDeA utilizes the different GPUs by dividing the incoming flows equally and performing the signature matching in parallel across all devices.

By default, MIDeA utilizes as many GPUs as it can find in the system; however, this can be controlled by defining the number of GPUs it should try to use in the configuration file. In the CUDA runtime system, on which MIDeA is based, each CPU process is bound to one device. To make multi-GPU computation possible, several host processes must then be created, with at least one process per device. A static GPU assignment is used for each process. Each process receives a uniform amount of flows, due to the load balancing scheme described in Section 3.1.2, and thus flows are equally distributed to the different GPUs.

4. PERFORMANCE OPTIMIZATIONS

Having described our architecture, we now go into a couple of optimizations that improve: (i) memory accesses on the GPU, and (ii) CPU and GPU execution through pipelining.

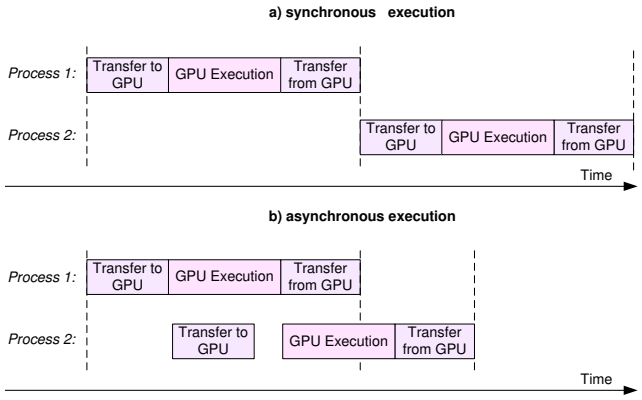


Figure 5: Data transfers and GPU execution of different processes can overlap.

Optimizing GPU Memory Accesses. One important optimization for the GPU pattern matching algorithm is related to the way the input data are loaded from the device memory. Since pattern matching is performed byte-wise, each input symbol is represented with 8 bits. However, the minimum size for every device memory transaction is 32 bytes. Thus, by reading the input stream one byte at a time, the overall memory throughput may be reduced by a factor of up to 32.

We have found that memory is better utilized when multiple bytes are fetched at a time, instead of just one. To that end, we redesigned the input reading process so that each thread accesses data using the `int4` built-in data type (`int4` is a vector type, consisting of 4 integer variables). Data is stored into a 128-bit register, and is accessed a byte at a time. `Int4` is the largest data-type that can be used to read data from the *texture memory* of the device, utilizing up to 50% of the total GPU memory bandwidth.

Pipelined Execution. Our core idea for hiding the pattern matching computation time on the GPU is double buffering. Our architecture improves the achieved parallelism by pipelining the execution of CPU cores and the GPUs. For each process, when the first buffer becomes full, it is copied to a texture bounded array that can be later read by the GPU through the kernel invocation. While the GPU is performing pattern matching on the flows of the first buffer, the CPU processes newly arrived packets, as shown in Figure 4.

Moreover, on recent CUDA-enabled devices, it is possible to overlap kernel execution on the device with data transfers between the host and the device, even for different processes. The dedicated DMA engine of NVIDIA GPUs[1] allows the concurrent execution of a CUDA kernel along with data transfers over the PCIe bus. For example, while one process transfers the data to the GPU, another process can execute the pattern matching operations. This allows better GPU utilization, as depicted in Figure 5. As we discuss in Section 5.2.1, the performance improvement due to overlapping execution in the GPU is up to 330%.

5. EXPERIMENTAL EVALUATION

We evaluated the performance of our system under a variety of workloads. We first describe the experimental testbed (Section 5.1), and then analyze the performance of MIDeA under different scenarios using micro-benchmarks (Section 5.2), as well as high-level end-to-end performance measurements (Section 5.3).

[1]For devices with Compute Capability 1.1 or greater.

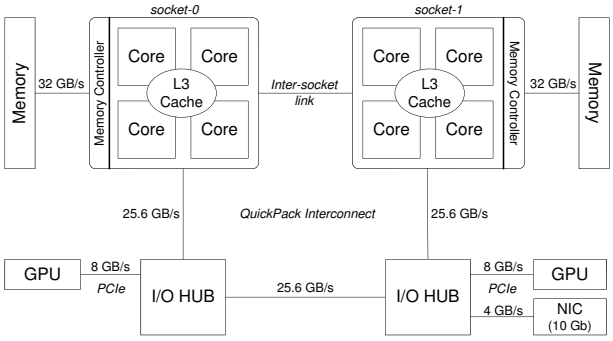


Figure 6: Hardware setup.

5.1 Experimental Setup

Hardware Setup. The overall architecture of our test system is shown in Figure 6. Our base system has two processor sockets, each with one Intel Xeon E5520 Quad-core CPU at 2.27GHz and 8192KB of L3-cache. Each socket has an integrated memory controller, connected to memory via a memory bus; this offers parallelism in memory accesses and, therefore, to higher aggregate and per-CPU bandwidth, as previous studies have shown [13]. The sockets are connected to each other and to the I/O hub via dedicated high-speed point-to-point links. The I/O hub is connected to the GPUs and the NIC via a set of PCIe buses: two PCIe 2.0 ×16 slots, which we populated with two GeForce GTX480 graphics cards, and one PCIe 2.0 ×8 slot holding one Intel 82599EB 10GbE NIC. To cover the needs for PCIe lanes, we acquired a motherboard with a dual I/O hub and a total of 72 lanes. Each NVIDIA GeForce GTX 480 is equipped with 480 cores, organized in 15 multiprocessors, and 1.5GB of GDDR5 memory.

Software. Our prototype runs on Linux 2.6.32 with the `ioatdma` and `dca` modules loaded. The `ioatdma` driver is required for supporting QuickPath Interconnect architecture of recent Intel processors. DCA (Direct Cache Access) is a NIC technology that directly places incoming packets into the cache of the CPU core for immediate access by the application. In all of our experiments we used the default rule set of Snort 2.8.6, which consists of 8,192 rules, comprising about 193,000 substrings for string searching. All default preprocessors, including `frag3`, `stream5`, `rpc_decode`, `ftp_telnet`, `smtp`, `dns`, and `http_inspect`, were enabled.

Traffic Generation. We used two servers for traffic generation to overcome the poor performance of the Linux kernel's network stack when sending small packets. The traffic generation servers and MIDeA are connected through a 10GbE switch. The test traffic, consisting of both generated synthetic traffic as well as real traffic traces, is sent using `tcpreplay` [4].

5.2 Micro-Benchmarks

We begin our evaluation by measuring the computational throughput of MIDeA using a varying number of CPU processes and GPU devices. Each process runs on a different CPU core, therefore we can utilize all cores by creating eight processes.

For the input data stream we used synthetic network traces of varying length with random payload. The data stream was carefully created to exercise most code branches, as well as different parameters of our implementation. To simulate the multi-queue capabilities of the NIC, we loaded the network packets of the trace file into separate queues (one for each core) using a simplified version of the Toeplitz hash function, which is used for RSS in modern NICs [28]. This is the "ideal NIC" case, where no overhead is

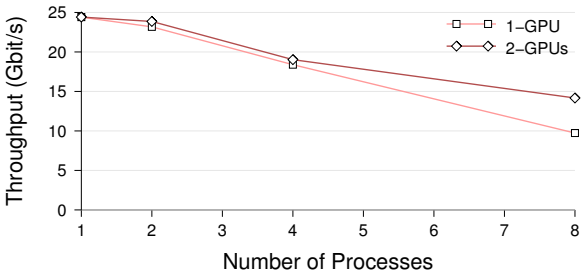


Figure 7: Per-process transfer rate between CPU and GPU.

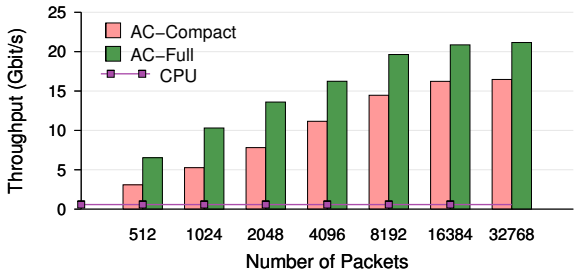


Figure 8: GPU throughput for AC-Full and AC-Compact.

Buffer Size	1KB	4KB	64KB	256KB	1MB	16MB
Host to Device	2.04	7.1	34.4	42.1	44.6	45.7
Device to Host	2.03	6.7	21.1	23.8	24.6	24.9

Table 1: Data transfer rate between host and device (Gbit/s).

#Rules	#Patterns	#States	AC-Full	AC-Compact
8,192	193,167	1,703,023	890.46 MB	24.18 MB

Table 2: Memory requirements of AC-Full and AC-Compact for the default Snort rule set.

added due to the transferring of the packets from the network interface to the host's main memory. All network packets are stored in memory, thus no blocks were transferred from disk when reading packets. We have verified the absence of I/O latencies using the `iostat(1)` tool.

5.2.1 GPU Performance

Data Transfer. Flows are transferred to each GPU device over the shared PCIe ×16 bus. PCIe uses point-to-point serial links, allowing more than one pair of devices to communicate with each other at the same time.

Table 1 shows the transfer rate of one process for moving data to a single GPU device, and vice versa, for different buffer sizes. We observe that with a large buffer, the rate of data transfer to the device is over 45 Gbit/s, while the transfer rate from the device to the host decreases to about 25 Gbit/s. This asymmetry in the data transfer throughput is probably related to the chosen hardware setup (i.e., the interconnection between the motherboard and the graphics cards), and has been also observed by other researchers [17]. We speculate that future motherboards will alleviate this asymmetry.

Figure 7 shows the transfer rate for a varying numbers of processes. The transfer costs include the copy of the network packets to the memory space of the GPU, and the copy of the results from the GPU to the host's memory. We can see that as the number of processes increases, the per-process throughput sustained by the PCIe bus slightly decreases. That is expected, since many processes contend for the same device through the same bus link. However, the aggregate throughput achieved by all processes increases, resulting to better PCIe bus utilization. As shown in Figure 7, for eight processes, the bidirectional PCIe throughput when using a single GPU reaches 9.7 Gbit/s per process, which in aggregate corresponds to 77.8 Gbit/s for all eight processes.[2] Adding one more GPU device results to a much higher throughput of 14.1 Gbit/s per process (113.3 Gbit/s in aggregate for all eight processes), since each GPU device is interconnected through dedicated PCIe lanes.

Computational Throughput. Having examined the data transfer costs, we now measure the GPU performance of the AC-Compact and AC-Full algorithms, described in Section 3.2.2.

Figure 8 shows the sustained throughput for pattern matching on a single GTX480. We fix the packet length to 1500 bytes and vary the number of packets that are processed at once from 512 to 32,768. Our AC-Full and AC-Compact implementations achieve a peak performance of 21.1 Gbit/s and 16.4 Gbit/s, respectively, including the data transferring costs to and from the device. The CPU achieves a performance of 0.6 Gbit/s for the AC-Full implementation, and thus a single GPU instance corresponds to 36.2 and 28.1 CPU cores for the AC-Full and AC-Compact implementations, respectively.

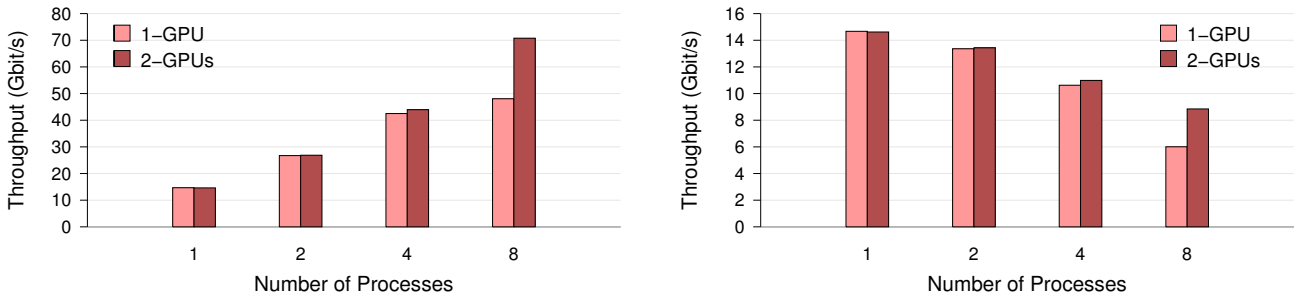
As expected, AC-Full outperforms AC-Compact in all cases. The added overhead of the extra computation that AC-Compact performs in every transition decreases its performance about 30%. The main advantage of AC-Compact is that it has significantly lower memory consumption than AC-Full. Table 2 shows the corresponding memory requirements for storing the detection engines of a single Snort instance. AC-Compact utilizes up to 36 times less memory, which makes it a better fit for a multi-CPU environment, due to CUDA's limitation of allocating a separate memory context for each host thread. Using AC-Compact, a single GTX480 card can store the detection engines of about 50 Snort instances ($50 \times 24.18MB \approx 1.2GB$). The remaining memory is used for storing the contents of network packets. If AC-Full is used, only one instance can fit in device memory. In all subsequent experiments we use the AC-Compact algorithm.

Utilization. We investigate the performance of the AC-Compact algorithm further, by varying the number of CPU processes that feed the GPU devices with data.

Figure 9(a) shows the aggregate data processing throughout of the GPU(s) for an increasing number of CPU processes. Figure 9(b) plots the same data normalized by the number of processes. It is clear that multiple processes offer an improvement even when utilizing only one GPU device. Currently, GPUs support multitasking through the use of "timesliced" context switching: each program receives a time slice of the GPU resources and cannot be suspended. When many processes use the same GPU device, data transfers and GPU execution may overlap, offering better GPU utilization. GPU executions have short run times, ranging from 100–300ns per packet, and hence, the GPU device can be effectively timesliced among the CPU processes.

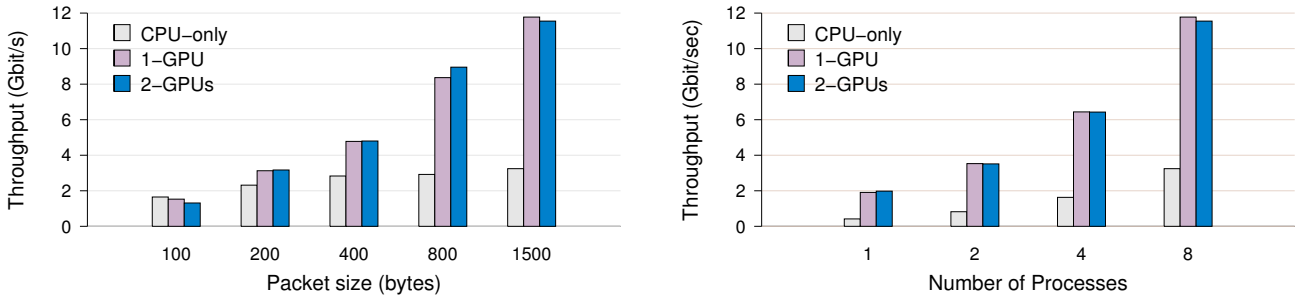
We observe that with two spawned processes, the overhead of the AC-Compact implementation increases, since the 25 Gbit/s throughput achieved is greater than the 21.1 Gbit/s achieved by the AC-Full algorithm, as shown in Figure 8. Increasing to eight processes, a single GPU reaches a maximum of 48 Gbit/s throughput. The PCIe

[2]The capacity of PCIe ×16 v2.0 is 64 Gbit/s *for each* direction. In practice though, the theoretical maximum data rate diverges due to the 8b/10b encoding at the physical layer.



(a) Aggregate (b) Per-process

Figure 9: GPU throughput with an increasing number of CPU-processes up to the number of cores.



(a) Different packet sizes. (b) Different numbers of CPU processes.

Figure 10: Overall sustained throughput for different workloads and configurations.

bus saturation, which was shown in Figure 7, is the main reason for this upper bound. However, since the PCIe bus is a point-to-point link, adding one more GPU device to the system increases the aggregate GPU throughput to over 70 Gbit/s.

5.2.2 *Overall Performance*

Throughput. In our next experiment, we measured the overall processing throughput achieved by our multi-parallel implementation. Figure 10(a) shows the sustained throughput for different packet sizes. We observe that for very small packet sizes, the GPU-assisted design exhibits a slightly worse performance compared to the multi-core approach alone. The main reason for this is that the buffering overheads for very small packets are greater than the corresponding pattern matching costs, as shown in more detail in the following experiment. Therefore, it is better in terms of performance to match very small packets on the CPU, rather than transferring them to the GPU.

As a consequence, we adopted a simple opportunistic offloading scheme, in which pattern matching of very small packets is performed on the CPU instead of the GPU. Thus, only packets that exceed a minimum size threshold are copied to the buffer that is transferred to the GPU for pattern matching. The packets contain already the TCP reassembled stream of a given direction, hence no state needs to be shared between the CPU and the GPU. The minimum threshold can be inferred off line, using a simple profiling measurement, or automatically at runtime. For simplicity we currently use the former method, although we plan to implement an automated solution in the future.

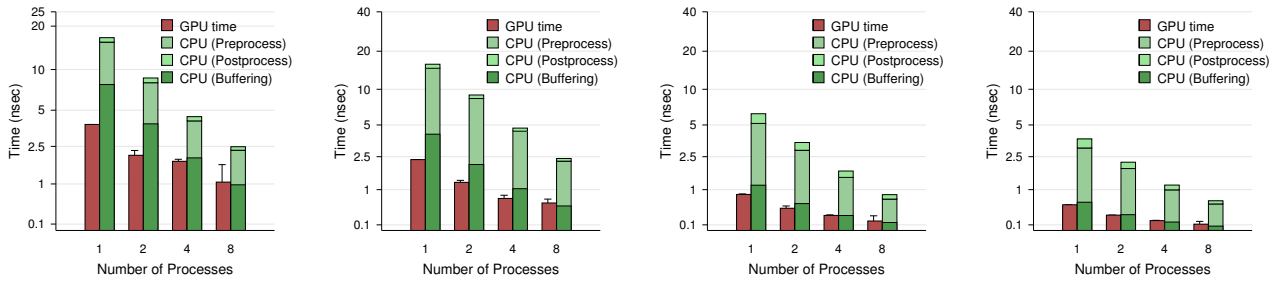
Figure 10(b) shows the sustained throughput for a different number of CPU processes, using 1500-byte packets. We observe that as the number of processes increases, the sustained throughput also increases linearly. When pattern matching is offloaded on the GPU, the throughput of the legacy multi-core implementation is increased 3.5–4.5 times, depending on the number of processes. The maximum throughput achieved by our base system reaches about 11.7 Gbit/s when utilizing all resources—eight CPU cores and two GPUs.

Finally, we observe that switching from one to two GPUs does not offer significant improvements to the overall performance. This can be explained by the fact that GPU communication and computation costs are completely hidden by the overlapped CPU computation, as discussed in the following experiment.

Timing breakdown. We proceed and examine in greater detail the overall performance achieved by profiling each device separately. In Figures 11(a)–11(d) we plot the individual execution times for various packet lengths. We show the times of each device with different bars, since execution is performed in parallel. CPU and GPU execution is pipelined, hence the CPU can continue unaffected while GPU execution is in progress. Each bar represents the execution time of the two GPU devices, while the thin line on each bar represents the corresponding time when utilizing one GPU. We observe that even when a single GPU is used, the cost for the data transfers and the pattern matching on the GPU is completely hidden by the overlapped CPU workload, for all packet sizes.

The extra cost for packet buffering before transferring them to the GPU depends highly on the packet size. Small packets incur higher cost per-byte, due to the start-up overhead of the `memcpy(3)` function. 100-byte packets or smaller induce a prohibitively large overhead, in comparison with the pattern matching cost. We tried to optimize the copies using a byte-by-byte procedure instead of calling the `memcpy(3)`, however the overhead was still higher. Thereupon, we avoid the small-packets penalty by opportunistically offloading pattern matching computation on the GPU depending on the packet length.

Finally, we notice that GPU execution times for small packets



(a) 100-byte packets (b) 200-byte packets (c) 800-byte packets (d) 1500-byte packets

Figure 11: Breakdown of per-byte processing overhead for different packet sizes.

also increase. The main reason for this is that the dimensions of the buffer that is used for transferring the packets to the GPU are fixed, hence it is populated sparsely for small packets.

5.3 Overall Traffic Processing Throughput

In this section, we measure the end-to-end performance of our prototype implementation under realistic conditions.

5.3.1 Synthetic Traffic

Figure 12(a) shows the packet loss ratio for different packet sizes, when replaying traffic at varying rates. We plot values up to the maximum achieved replay rate, hence the smaller the packet size, the lower the replay rate reached. For example, for 200-byte packets, we managed to replay traffic at maximum rate of 1.86 Gbit/s, while for 1500-byte packets we achieved a rate of 7.67 Gbit/s.

Given these traffic replay rates, our prototype system begins to drop packets at 7.22 Gbit/s for 1500-byte packets, which is a 253% improvement over the traditional multi-core implementation. When processing smaller packets, the performance falls to 1.5 Gbit/s, which is slightly higher than the traditional multi-core implementation, although the drop rate is about 6.6 times lower.

Comparing the achieved throughput with the "ideal NIC" case in Figure 10(a), we observe that the NIC adds a variable overhead that depends on the size of the captured packets. It is clear that small packets add more latency to the capturing process than larger ones. For the traditional multi-core approach, we observe an extra overhead of 55% for 200-byte packets, that falls to 18% for 800-byte packets, and 13% for 1500-byte packets. Similarly, the extra overhead for the GPU-accelerated implementation is 110% for 200-byte packets, about 87% for 800-byte packets, and 52% for 1500-byte packets. We observe that the NIC overhead is larger in the GPU-accelerated implementation, and we speculate that this is an issue related to congestion in the PCIe controller.

5.3.2 Real Traffic

In our final experiment, we evaluate MIDeA in a scenario using real traffic. We used a trace of real network traffic (referred to as `UNI`), captured at the gateway of a large university campus with several thousands of users. Specifically, the trace spans 74 minutes, and includes all packets and their payloads, totalling 46 GB. Table 3 summarizes the most important properties of the trace.

To replay the captured trace at high-speed, it has to reside in the main memory of the host to avoid disk accesses. Unfortunately, the main memory of our two traffic generator machines is only 4GB, hence it is impossible to load the whole trace in memory. To overcome this issue, we split the trace to several 2GB parts. While one part is replayed, the other part is loaded into main memory. Since reading from disk is much slower, each part is replayed several times, up until the next part is fully loaded into memory. Using the above pre-fetching scheme, we successfully managed to replay the captured trace with speeds of up to 5.7 Gbit/s.

Packets	73,162,723
Packet size (min/max/avg)	60/1,514/ 679.57
IP Fragments	88,411
TCP sessions	185,642
UDP sessions	174,442
Triggered Snort Alerts	183,050

Table 3: `UNI` trace properties.

Model	Qty	Unit price
NIC: Intel 82599EB	1	$687
CPU: Intel Xeon E5520	2	$336
GPU: NVIDIA GTX480	2	$340

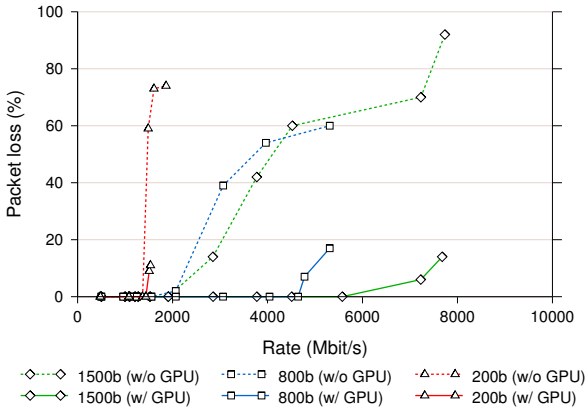
Table 4: Cost of MIDeA components (as of April 2011).

Figure 12(b) shows the dropped packets when increasing the traffic rate. We also annotate the throughput achieved when reading the network packets directly from main memory instead of the NIC. The traditional multi-core implementation starts to drop packets at 1.1 Gbit/s, while the ideal throughput is near 1.4 Gbit/s. When GPU acceleration is enabled, we did not observe any packet loss for speeds of *up to 5.2 Gbit/s*. For comparison, the ideal throughput is 7.8 Gbit/s.

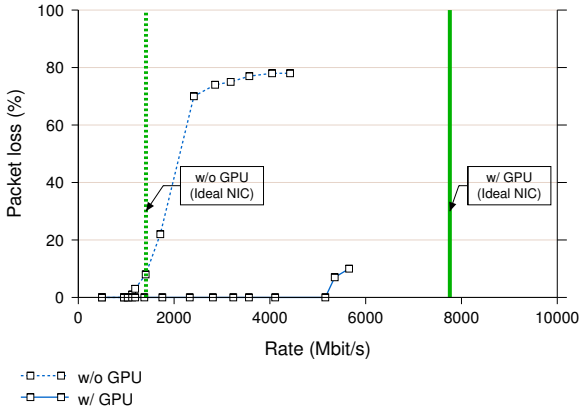
6. DISCUSSION

So far in this paper we went over a detailed description of the design aspects, trade-offs, and performance issues of our proposed architecture. Even though we focused on the parallelization of an intrusion detection system, we strongly believe that the proposed model can benefit a variety of other network monitoring applications, such as traffic classification, content-aware firewalls, spam filtering, and other network traffic analysis systems. With this in mind, we could easily augment a router with multi-parallel network processing capabilities, expanding its functionality without affecting its normal packet routing operations [13].

Price/Performance. For our hardware setup, we have selected relatively low-end devices: two Intel Xeon E5520 processors, two NVIDIA GeForce GTX 480 graphics cards, and an Intel 82599EB 10GbE NIC. Table 4 shows the approximate cost of each component, as of April 2011. The total cost of our base system is about $2739, achieving a throughput per dollar cost of 1.8 Mbps/$.



(a) Synthetic traffic



(b) Real traffic

Figure 12: Observed packet loss for (a) synthetic and (b) real traffic, as a function of the traffic rate. MIDeA can handle real traffic speeds of up to 5.2 Gbit/s without dropping any packets.

Limitations and Future Work. In favor of programming simplicity, we chose to use processes instead of threads for parallelizing the CPU part of MIDeA. We believe that a multi-threaded implementation would further increase the complexity of the design without a significant increase in the overall throughput.

In our flow-based partitioning scheme, we avoid any communication between the cores. Traditional Snort-style signature matching does not require any communication for analysis outside the scope of a single flow. In case a network analysis system needs this functionality, e.g., for detecting DDoS attacks or malware propagation, a lightweight communication scheme needs to be integrated for coordinating the different cores [47].

The buffering of network packets, described in section 3.2.1, introduces an extra copy operation. This is mandatory for our design, considering that most packets have to be processed before matching them against signatures, and that transferring a single packet each time significantly reduces the PCIe throughput.

Finally, each process allocates a different memory space on the GPU, due to the restriction of the CUDA driver for preventing sharing of GPU memory between different processes. Although the same policy applies to threads, we believe that future releases of the CUDA driver will support device memory sharing. In that case, we could easily migrate to the faster AC-Full algorithm. We also believe that a shared GPU memory space would exhibit higher locality and increase the computational throughput.

7. RELATED WORK

Prior work has focused extensively on the use of specialized hardware to augment NIDSs capabilities. The majority of these approaches focus on improving deep packet inspection (DPI), which is the most computationally-intensive NIDS operation, using specialized hardware, such as FPGAs, ASICs, and TCAMs [9, 12, 27, 30, 45, 52]. Recently, Meiners et al. [27] proposed a custom regular expression matching approach based on TCAMs, which achieves a throughput of up to 18.6 Gbit/s; our corresponding pattern matching implementation alone reaches a 70 Gbit/s throughput, which is almost a four-times improvement, using commodity off-the-shelf equipment. Other approaches employ a pre-filtering mechanism based on FPGAs to reduce the amount of traffic sent to a software NIDS/NIPS for inspection [16,43]. Unfortunately, most implementations are tied to a specific architecture, and thus, are particularly difficult to extend and program. Any changes in the rule set require recompilation, regeneration of the automaton, resynthesis, replacement, and routing of the circuits, which is a time-consuming and difficult procedure.

Much work has also focused on improving the performance of detection mechanisms, such as string matching and regular expression matching [7, 22, 23, 40, 46, 51]. These works are orthogonal to ours, and can be integrated to our system to improve memory utilization and performance.

In another line of work, cluster-based approaches have been proposed for keeping-up with the increasing link speeds. Instead of having a single server to process all incoming traffic, a cluster of servers is used instead. The major issue then is how to partition the incoming traffic to the back-end servers, while supporting stateful processing. Kruegel et al. [21] propose a stateful slicing mechanism that divides the overall network traffic into subsets of manageable size, which are then processed by different sensors. Foschini et al. [14] extend that work with a parallel matching algorithm that allows communication between the sensors through a dedicated control plane. SPANIDS [38] uses a specialized FPGA-based switch, that takes into account flow information and the load of each server when redirecting network packets. Xinidis et al. [50] present an active splitter architecture that provides early filtering to reduce the load of the back-end sensors.

Other approaches attempt to improve the performance of NIDS using commodity hardware. Paxson et al. [35] and Valentin et al. [47] implemented a NIDS cluster that scales through the use of parallel nodes. Their system, based on Bro [33], demonstrates the ability to scale beyond the capacity of a single NIDS instance using commodity hardware, with the exception of the special purpose front-end hardware that was used to distribute traffic evenly across the back-end nodes. In Supra-linear Packet Processing [19], a single thread is responsible for packet gathering and dispatching, while many other threads are processing incoming flows in parallel. Thus, each processing thread handles a specific flow in isolation. Unfortunately, a lot of time is spent on context switches between threads, most likely due to high levels of locking contention to the shared packet queue.

To eliminate the excessive contention rates due to packet queue access in the packet processing architectures, flow-pinning is commonly used (i.e., all packets of a flow are "pinned" to be processed by a specific thread) [20]. This approach requires slightly more data storage to keep the incoming packets to separate queues. However, it allows most of the threads to work independently, which is a key characteristic of a good multi-threaded algorithm. Schuff

et al. [39] evaluate different per-flow and intra-flow parallelization approaches. Although their results do not propose a clear winner, it seems that pure flow-concurrent parallelism performs better in almost all cases.

Recently, graphics processors have been used to boost computationally intensive tasks in intrusion detection systems. Gnort [48, 49] was the first attempt that sufficiently utilized the graphics processor for string searching and regular expression matching. Unfortunately, its single-threaded architecture restricts its scalability in the advent of multi-core CPUs. Many other approaches followed the above scheme [18,41], without significant differences in the architecture and the performance benefits.

8. CONCLUSION

In this work, we designed and built a multi-parallel intrusion detection architecture, as a scalable solution for the processing and stateful analysis of network traffic. Our system achieves high performance, with a processing throughput that exceeds 5 Gbit/s for real traffic, and raw pattern matching speeds of 70 Gbit/s, using commodity off-the-shelf hardware, in a single box.

MIDeA has three levels of parallelization, at the NIC, the CPU, and the GPU levels. It consists of a multi-queue NIC, which distributes the incoming traffic across a set of multi-core CPUs for packet processing and analysis. Each CPU core is processing the traffic of only a subset of network flows. Having split the traffic to different cores, MIDeA further offloads the costly content inspection operations to a set of GPUs. By parallelizing both packet processing and content inspection across multiple CPU cores and graphics processors, MIDeA offers a scalable approach for building a multi-parallel NIDS, which can operate at speeds of several Gbit/s, while at the same time it has en extremely low price point.

9. ACKNOWLEDGMENTS

The authors would like to thank Evangelos Markatos for his invaluable comments and suggestions during the preparation of this paper. This work was supported in part by the Marie Curie Actions – Reintegration Grants project PASS, by the FP7-PEOPLE-2009-IOF project MALCODE and the FP7 project SysSec, funded by the European Commission under Grant Agreements No. 254116 and No. 257007, and by the project i-Code, funded by the Prevention, Preparedness and Consequence Management of Terrorism and other Security-related Risks Programme of the European Commission – Directorate-General for Home Affairs under Grant Agreement No. JLS/2009/CIPS/AG/C2-050. This publication reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained herein.

10. REFERENCES

[1] Endace Security Manager. `http://www.endace.com/endace-security-manager2.html`.

[2] Intel 82599EB 10 Gigabit Ethernet Controller. `http://ark.intel.com/Product.aspx?id=32207`.

[3] Receive side scaling on Intel Network Adapters. `http://www.intel.com/support/network/adapter/pro100/sb/cs-027574.htm`.

[4] Tcpreplay. `http://tcpreplay.synfin.net/`.

[5] A. V. Aho and M. J. Corasick. Efficient String Matching: an Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340, June 1975.

[6] Z. K. Baker and V. K. Prasanna. Time and Area Efficient Pattern Matching on FPGAs. In *Proceedings of the 12th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2004.

[7] M. Becchi and P. Crowley. A Hybrid Finite Automaton for Practical Deep Packet Inspection. In *Proceedings of the ACM International Conference on emerging Networking Experiments and Technologies Conference (CoNEXT)*, 2007.

[8] V. Chandola, A. Banerjee, and V. Kumar. Anomaly Detection: A Survey. *ACM Computing Surveys*, 41:15:1–15:58, July 2009.

[9] C. R. Clark, W. Lee, D. E. Schimmel, D. Contis, M. Koné, and A. Thomas. A Hardware Platform for Network Intrusion Detection and Prevention. In P. Crowley, M. A. Franklin, H. Hadimioglu, and P. Z. Onufryk, editors, *Proceedings of the 3rd Workshop on Network Processors and Applications (NP3)*, 2005.

[10] C. R. Clark and D. E. Schimmel. Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2003.

[11] L. Deri. Improving Passive Packet Capture: Beyond Device Polling. In *Proceedings of 4th International System Administration and Network Engineering Conference (SANE)*, 2004.

[12] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood. Deep Packet Inspection Using Parallel Bloom Filters. *IEEE Micro*, 24(1), 2004.

[13] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2009.

[14] L. Foschini, A. V. Thapliyal, L. Cavallaro, C. Kruegel, and G. Vigna. A Parallel Architecture for Stateful, High-Speed Intrusion Detection. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS)*, 2008.

[15] F. Fusco and L. Deri. High Speed Network Traffic Analysis with Commodity Multi-core Systems. In *Proceedings of the 10th Internet Measurement Conference (IMC)*, 2010.

[16] J. M. Gonzalez, V. Paxson, and N. Weaver. Shunting: A Hardware/Software Architecture for Flexible, High-Performance Network Intrusion Prevention. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, 2007.

[17] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. In *Proceedings of the ACM SIGCOMM 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, August 2010.

[18] N.-F. Huang, H.-W. Hung, S.-H. Lai, Y.-M. Chu, and W.-Y. Tsai. A GPU-Based Multiple-Pattern Matching Algorithm for Network Intrusion Detection Systems. In *Proceedings of the 22nd International Conference on Advanced Information Networking and Applications - Workshops (AINAW)*, 2008.

[19] Intel Corporation. Supra-linear Packet Processing Performance with Intel Multi-core Processors, 2006.

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

[21] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In *Proceedings of the 23rd IEEE Symposium on Security and Privacy (S&P)*, May 2002.

[22] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS)*, 2007.

[23] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. In *Proceedings of the ACM SIGCOMM 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, 2006.

[24] 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.

[25] B. H. Leitao. Tuning 10Gb Network Cards on Linux. In *Proceedings of the 2009 Linux Symposium*, July 2009.

[26] R.-T. Liu, N.-F. Huang, C.-H. Chen, and C.-N. Kao. A Fast String-matching Algorithm for Network Processor-based Intrusion Detection System. *ACM Transactions on Embedded Computing Systems*, 3(3):614–633, 2004.

[27] C. R. Meiners, J. Patel, E. Norige, E. Torng, and A. X. Liu. Fast Regular Expression Matching Using Small TCAMs for Network Intrusion Detection and Prevention Systems. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[28] Microsoft Corporation. Scalable Networking: Eliminating the Receive Processing Bottleneck - Introducing RSS, 2005.

[29] A. 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.

[30] J. Moscola, J. Lockwood, R. Loui, and M. Pachos. Implementation of a Content-Scanning Module for an Internet Firewall. In *Proceedings of 11th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2003.

[31] M. Norton. Optimizing Pattern Matching for Intrusion Detection, July 2004.

[32] NVIDIA. NVIDIA CUDA Programming Guide.

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

[34] V. Paxson, K. Asanović, S. Dharmapurikar, J. Lockwood, R. Pang, R. Sommer, and N. Weaver. Rethinking Hardware Support for Network Analysis and Intrusion Prevention. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Security (HotSec)*, 2006.

[35] V. Paxson, R. Sommer, and N. Weaver. An Architecture for Exploiting Multi-core Processors to Parallelize Network Intrusion Prevention. In *Proceedings of the 30th IEEE Sarnoff Symposium*, May 2007.

[36] M. Roesch. Snort: Lightweight Intrusion Detection for Networks. In *Proceedings of the 1999 USENIX Large Installation System Administration Conference (LISA)*, 1999.

[37] D. P. Scarpazza, O. Villa, and F. Petrini. Exact Multi-pattern String Matching on the Cell/B.E. Processor. In *Proceedings of the 5th Conference on Computing Frontiers (CF)*, 2008.

[38] L. Schaelicke, K. Wheeler, and C. Freeland. SPANIDS: A Scalable Network Intrusion Detection Loadbalancer. In *Proceedings of the 2nd Conference on Computing Frontiers (CF)*, 2005.

[39] D. L. Schuff, Y. R. Choe, and V. S. Pai. Conservative vs. Optimistic Parallelization of Stateful Network Intrusion Detection. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2007.

[40] R. Smith, C. Estan, and S. Jha. XFA: Faster Signature Matching with Extended Automata. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P)*, 2008.

[41] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating GPUs for Network Packet Signature Matching. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[42] R. Sommer and V. Paxson. Outside the Closed World: On Using Machine Learning for Network Intrusion Detection. In *Proceeding of the 31st IEEE Symposium on Security and Privacy (S&P)*, 2010.

[43] H. Song, T. Sproull, M. Attig, and J. Lockwood. Snort Offloader: A Reconfigurable Hardware NIDS Filter. In *Proceedings of the 15th International Conference on Field Programmable Logic and Applications (FPL)*, 2005.

[44] Sourcefire. Sourcefire 3D System. `http://www.sourcefire.com/security-technologies/cyber-security-products/3d-system`.

[45] I. Sourdis and D. Pnevmatikatos. Pre-Decoded CAMs for Efficient and High-Speed NIDS Pattern Matching. In *Proceedings of the 12th Annual IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, 2004.

[46] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic Memory-efficient String Matching Algorithms for Intrusion Detection. In *Proceedings of the 23rd IEEE International Conference on Computer Communications Conference (INFOCOM)*, 2004.

[47] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney. The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2007.

[48] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.

[49] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis. Regular Expression Matching on Graphics Hardware for Intrusion Detection. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2009.

[50] K. Xinidis, I. Charitakis, S. Antonatos, K. G. Anagnostakis, and E. P. Markatos. An Active Splitter Architecture for Intrusion Detection and Prevention. *IEEE Transactions on Dependable and Secure Computing*, 3:31–44, January 2006.

[51] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and Memory-efficient Regular Expression Matching for Deep Packet Inspection. In *Proceedings of the 2nd ACM/IEEE symposium on Architecture for Networking and Communications Systems (ANCS)*, 2006.

[52] F. Yu, R. H. Katz, and T. V. Lakshman. Gigabit Rate Packet Pattern-Matching Using TCAM. In *Proceedings of the 12th IEEE International Conference on Network Protocols (ICNP)*, October 2004.