University of Crete
School of Sciences and Engineering
Computer Science Department

Implementing an architecture for efficient network traffic processing on modern graphics hardware

Lazaros Koromilas

Master's thesis

Heraklion, November 2012

University of Crete
School of Sciences and Engineering
Computer Science Department

Implementing an architecture for efficient network traffic processing on modern graphics hardware

Thesis submitted by
Lazaros Koromilas
in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

Author:

Lazaros Koromilas

Committee approvals:

Evangelos Markatos
Professor, Thesis Advisor

Sotiris Ioannidis
Principal Researcher, Thesis Coadvisor

Maria Papadopouli
Associate Professor, Committee Member

Departmental approval:

Angelos Bilas
Professor, Director of Graduate Studies

Heraklion, November 2012

Abstract

Network traffic processing is necessary in order to develop active components in the infrastructure of the network, such as routers, or passive applications, such as network intrusion detection systems. However, in today's high-speed network links this has become a very challenging task in terms of computational resources. Custom hardware appliances that can handle high packet rates are rather expensive and offer limited programmability.

This work presents the design and implementation of a high-performance software packet processing system using high-speed network interfaces, multi-core processors and many-core graphics hardware. The massive parallelism of modern graphics chips is exploited for efficient packet processing, which effectively frees cycles on the main processor. The development of the system focuses on high-throughput data movement techniques, memory access optimizations, domain specific data structures and the configuration of a large set of parameters that enables high processing rates. The evaluation of the system has shown that it can passively process real-world traffic at 18 Gbps, with latency in the order of few milliseconds. In active mode, where packets are forwarded, the system can achieve a 13.5 Gbps rate. There is an up to 15 times increase in throughput when compared to traditional approaches.

Περίληψη

Η επεξεργασία δεδομένων από το δίκτυο είναι αναγκαία για την ανάπτυξη είτε ενεργών κομματιών μέσα στην υποδομή του δικτύου, όπως δρομολογητές, είτε παθητικών εφαρμογών, όπως συστήματα ανίχνευσης δικτυακών επιθέσεων. Παρολαυτά, στους συνδέσμους μεγάλης ταχύτητας του σήμερα αυτό έχει γίνει μια πολύ απαιτητική διεργασία όσον αφορά τους υπολογιστικούς πόρους. Ειδικά σχεδιασμένες συσκευές οι οποίες αντέχουν σε μεγάλους ρυθμούς πακέτων είναι μάλλον ακριβές και προσφέρουν περιορισμένες δυνατότητες προγραμματισμού.

Η εργασία αυτή παρουσιάζει το σχεδιασμό και την υλοποίηση ενός υψηλής απόδοσης συστήματος επεξεργασίας πακέτων μέσω λογισμικού. Το σύστημα χρησιμοποιεί κάρτες δικτύου μεγάλης ταχύτητας, πολυπύρηνους επεξεργαστές και κάρτες γραφικών με πολλούς επεξεργαστές. Η τεράστια παραλληλία που προσφέρουν οι σύγχρονες αρχιτεκτονικές γραφικών αξιοποιείται για αποτελεσματική επεξεργασία πακέτων, και έτσι ελευθερώνονται κύκλοι στον κύριο επεξεργαστή. Η ανάπτυξη του συστήματος επικεντρώνεται σε τεχνικές μετακίνησης δεδομένων με μεγάλη απόδοση, βελτιστοποιήσεις πρόσβασης στην κύρια μνήμη, δομές δεδομένων ειδικού πεδίου και την ρύθμιση ενός εκτενούς συνόλου παραμέτρων που επιτρέπει υψηλούς ρυθμούς επεξεργασίας πακέτων. Η αξιολόγηση του συστήματος έχει δείξει ότι μπορεί να επεξεργαστεί παθητικά πραγματική κίνηση δικτύου στα 18 Gbps, με καθυστέρηση λίγων χιλιοστών του δευτερολέπτου. Σε ενεργή λειτουργία, όπου τα πακέτα προωθούνται, το σύστημα καταφέρνει ρυθμούς της τάξεως των 13.5 Gbps. Σε σύγκριση με τις καθιερωμένες μεθόδους το σύστημα έχει μέχρι και 15 φορές καλύτερη απόδοση.

Acknowledgements

I am grateful to my supervisor Prof. Evangelos Markatos for his guidance in my studies at the University of Crete and the chance to be a part of the Distributed Computing Systems Lab at FORTH-ICS. I would also like to thank my co-advisor Dr. Sotiris Ioannidis for seeing my potentials and believing in my work. This thesis would not be possible without their invaluable comments and contribution.

Credits go to Giorgos Vasiliadis for a great collaboration on the project that this thesis is part of, Spiros Thanasoulas for introducing me to the UNIX way, and the endless conversations on IRC and problem solving over the phone, Giannis Manousakis for those little chats on all sorts of unexplained phenomena, Antonis Papadogianakis for helping in system setups and then sabotaging our experiments (just kidding, that was me), Giorgos Chinis for preserving the balance between computers and real life, Christos Papachristos for always keeping our morale up and running.

My family gets my deepest gratitude for always beeing by my side and supporting me through my studies and life. Special thanks go to Hara, Christos, Andreas and Kalliopi for just being there.

This work has been performed at and is supported by the **Distributed Computing Systems** laboratory, **Institute of Computer Science (ICS)**, **Foundation for Research and Technology – Hellas (FORTH)**.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer networks have been growing in size and complexity; connection speeds, especially in access and backbone links, typically reach multi-Gigabit per second rates. At the same time, networking applications become diversified and traffic processing gets more sophisticated. Coping with the increasing network capacity and complexity necessitates pushing the performance of network data processing applications as high as possible.

Programmable special-purpose components, such as FPGAs (Field Programmable Gate Arrays) and NPUs (Network Processing Units), increase the performance of network traffic processing systems and have been successfully used in routers and NIDSes (Network Intrusion Detection Systems). However, the cost of these components is usually very high and, more importantly, most system setups require specialized programming and are tied to the underlying architecture. Developing new applications on top of these devices is both challenging and non-portable. In contrast, software implementations based on commodity processors have low cost, and are more accessible to application developers. The emergence of many-core architectures, such as multicore CPUs (Central Processing Units) and modern GPUs (Graphics Processing Units) has proven to be a good solution for accelerating network applications, and has led to their successful deployment in high-speed environments [16, 19, 42]. Unfortunately, the lack of programming abstractions and the absence of GPU-based libraries tailored for network traffic processing —even for simple tasks such as packet decoding and filtering— increases significantly the programming effort needed to build, extend, and maintain high-performance GPU-based network applications. Moreover, the absence of adequate OS support increases the cost of communication between the host and I/O devices. Addressing such problems requires introducing OS-level support and an appropriate programming interface.

1.1 Background

From the early days of networking, computers were connected to computer networks using special hardware called *network adapters* or NICs (Network Interface Card / Controller). Computers manufactured today usually have a NIC built-in on the motherboard because of Ethernet's wide adoption (Fast Ethernet or Gibabit Ethernet). Also, the faster adapters that support the 10 Gigabit Ethernet (10 GbE) standard become increasingly popular because of their advanced capabilities for a reasonable price. However, it is not straightforward how to fully exploit modern NICs in order to develop applications that operate at such speeds.

Network support in modern operating systems has been designed with general-purpose networking in mind. Although this is suitable for most networking abstractions and applications, it is largely inefficient for high-speed packet processing. While the per-packet overhead is negligible when large packets are received or transmitted it becomes important for small packets. Unfortunately, the worst case (minimum Ethernet packet) is what many critical applications have to cope with. Such applications include NIDSes (Network Intrusion Detection System), network monitoring systems and software routers. For each packet the network device driver typically allocates a buffer to hold the packet data. The buffer is queued into kernel data structures in order to be processed by all interested subsystems (firewall, prioritization system). It's also copied between the kernel and user space for each application that should receive the packet. The buffer's memory is freed as soon as the packet has reached all clients. For the transmit path, the same operations happen until the packet gets written on the wire. The per-packet overhead introduced by all the relevant layers in the operating system has become a bottleneck for processing packets using high-speed 10 GbE devices.

Some efforts have been made to improve the networking performance in general purpose operating systems by using techniques to amortize or eliminate some of the costs mentioned above. By applying *zero-copy* techniques in various places of the I/O path, moving data between kernel and user space can be eliminated; this is the concept behind `sendfile(2)` and `splice(2)` implementations. Furthermore, new socket types like `AF_PACKET` [3] and `PF_RING` [11] have come to address a different problem: the limited buffers used in the packet capturing process and the excessive use of system calls (at least one per packet reception/transmission). With the latter frameworks, the packets are placed in a circular buffer that is exposed in user space using the `mmap(2)` system call. This way the copies are eliminated and many I/O operations can be batched together to hide the system call overhead. Although these are valid approaches, further optimization is necessary in order to push high-speed network hardware to its full potential and, more importantly, enable cooperation with accelerators like GPUs in the packet analysis process.

The use of graphics hardware for general purpose computation is not new. Commodity computer graphics chips have long been appealing to programmers for they provide huge computational power and memory bandwidth, while at the same time being inexpensive. The GPGPU (General Purpose computing on Graphics Processing Units) programming community has been using GPUs for over a decade, to take advantage of their unique features mostly to accelerate various data parallel applications. For a survey of applications using the GPGPU model see [32]. However, in the early days of GPGPU computing the programs had to exploit the *graphics pipeline* by using graphics APIs. That means that the problems had to be expressed in terms of vertex coordinates, textures and shader routines. Also, scattered memory operations and some arithmetic operations were not supported. All these facts greatly increased development complexity, restricted the programming model and limited wider adoption.

More recently, graphics chips manufacturers began to develop architectures that exhibit different levels of programmability. They have moved from the programmable shader architecture to general purpose processors that support an SIMD (Single Instruction, Multiple Data) instruction set. Furthermore, the programmer can now write in high-level programming languages like C and use more generic and intuitive APIs such as those provided by CUDA [29] (Compute Unified Device Architecture) or OpenCL [15] (Open Computing Language). Through these frameworks, it is possible to manage the memory and control flow of code that runs on the graphics processor from the host, and perform high-throughput transfers to and from the device's memory. Note that the terms "device" and "graphics processor" are used interchangeably throughout the text. More details about the specific GPU architecture used in this work can be found in Section 3.7.

1.2 Contributions

Software packet processing solutions are viable whenever there is enough processing power and bandwidth in the system to meet the capabilities of network hardware. It's often difficult to achieve maximum packet I/O performance when the processing on the network traffic is particularly demanding. This work proposes the use of graphics processors to add computing resources to the system, even though this possibly spares some of the bus and memory bandwidth. Thanks to a set of design decisions presented in this work it's possible to effectively use commodity graphics chips for packet processing at high speed rates, while at the same time showing acceptable latencies. Using this approach not only frees many CPU cycles, but also enables computation that would otherwise be impossible without specialized hardware.

The most important contributions of this work are as follows.

- Practical optimizations and performance evaluation of software packet processing using graphics hardware and high-speed networking equipment.
- Implementation and evaluation of a novel zero-copy technique for data movement between the network adapter and the graphics processor.
- Insights about various configuration parameters and design choices, such as the placement of buffers and processes, interrupt request mappings, data structures and programming practices.

The rest of the text is organized as follows. Chapter 2 presents related work from the fields of networking, GPU computing and beyond. Chapter 3 introduces and analyses the various techniques used and technologies adopted for the making of the system. The challenges faced in the process of implementing the system are addressed in Chapter 4. Chapter 5 shows results from micro-benchmarks for the memory, the network adapter and the graphics card, as well as the overall performance of the system. Chapter 6 concludes the thesis and discusses possible future work.

Chapter 2

Related Work

Several frameworks, research projects, tools and infrastructures share ideas with this work. Many of them are strictly on the field of networking, while others are related to graphics hardware programming for general purpose computations. Furthermore, there are works that study the applicability of GPU processing on the fields of cryptography, storage technologies and even biology, the most important of whom are presented in this section. Finally, a comparison of works that deal with network processing on graphics cards is presented.

2.1 Networking

This section lists important advances of the last decade in networking infrastructures and the techniques used to achieve high performance. Click [27] is a software architecture for building configurable routers. It consists of modules that perform simple functions. Those functions include packet classification, scheduling, queueing, as well as interfacing with the hardware. This way, complex network setups are possible and relatively easy to compose. Click successfully demonstrated the need and the importance of modularity in software routers. Kernel-based packet forwarding can operate at high speeds but cannot keep up for small packet sizes, even on 100 Mb Ethernet. This work targets packet processing on 10 Gb Ethernet links.

Recent Linux releases come with `AF_PACKET` [3], a kernel infrastructure that implements a new communications domain for sockets. [1] It provides access to raw packets at the device driver level. The infrastructure also includes socket-level options to manipulate packet storage by creating and managing ring buffers that can be later mapped to userspace. This technique of amortizing the system call overhead over many packets is common to this work. The *libpcap* library [2] uses these facilities in Linux.

[1] See the `packet(7)` manual page of the Linux Programmer's Manual.

[2] You can visit the homepage at `http://www.tcpdump.org/`.

The `PF_RING` [11] is a new socket type that brings the idea of `AF_PACKET` one step further. It provides fast packet capturing facilities by not giving the received packets to the upper layers of the operating system. Operating in this mode, increases performance because no processing is done by the kernel. Similar tactics are used in this work for efficient movement of packet data. The software is available for use with Linux. [3]

The `PF_RING` DNA (Direct NIC Access) solution first described in [12] targets wire speed packet forwarding at user space by taking over control of the network adapter. With the help of a library, the software directly controls the hardware queues of the network adapter and completely bypasses the operating system. Although an extremely efficient approach, there are not many compatible drivers yet. The library is only available under a restrictive license, but it can be used for evaluation purposes. The system of this work operates under the same basic principles, but direct access to the hardware is avoided due to complexity and consistency constraints.

RouteBricks [13] investigates the problem of scaling software routers with the recent advances in networking technologies and processors. The authors propose an architecture that parallelizes router functionality both across multiple machines and across multiple processor cores inside the same machine. They report considerable improvement against previous proposals, but computation intensive tasks are still non-trivial and interfere with packet I/O. This work aims to fill this gap with the computing power of the GPU.

A new addition to the set of capturing frameworks is netmap [33]. It provides a generic API for fast packet I/O, by utilizing known techniques such as packet batching and memory mapped ring buffers. It is similar to DNA in terms of performance, but without exposing hardware registers to the user application. The software is available in FreeBSD and has been also ported to Linux under an open-source license. Actually, this work uses the Linux version of netmap with minor modifications.

2.2 GPU Computing

Some general purpose frameworks and systems designed to harvest the power of the GPU are outlined here. The main idea behind them is to provide operating system abstractions and programming interfaces, in order to integrate the GPU to the heart of the system. An important step in the evolution of interfaces for general purpose computations on graphics cards is BrookGPU [7]. The system builds upon the graphics APIs like OpenGL and DirectX, and provides a compiler and runtime system that abstracts and virtualizes many aspects of the graphics hardware. This is, more or less, tha same model adopted by OpenCL implementations and CUDA, of course without the need to communicate with the underlying graphics interface.

[3] It can be downloaded from `http://www.ntop.org/`.

PTask [34] implements a set of abstractions, that make use of GPU devices as generic computing resources at the operating system level, and support a dataflow programming model. Through the new interface, the system is able to provide traditional scheduling guarantees like fairness and isolation. Having a global view of all GPU applications, data movement operations can be optimized. Those optimizations are critical for this work also, as unneeded data transfers greatly impact performance.

Another recent system that handles grahics processors as "first-class" computing devices is Gdev [20]. Gdev uses advanced memory management techniques to implement IPC (inter-process communication); it also provides scheduling facilities with virtualization in mind. Another significant contribution is the step forward to a usable, open-source device driver for the NVIDIA graphics chips. Although Gdev is a generic systems framework, it shares many basic concepts with this work, mostly related to considerations about data transfers.

Similar to the frameworks mentioned above is the kgpu [4] project. However, kgpu instead of dealing with reverse engineering in order to directly communicate with the GPU, it uses a user-space daemon as a middle layer. This daemon, in turn, talks to the GPU through the normal framework for developing CUDA applications. This work does not attempt any direct communication also, and the data is passed through user space.

2.3 GPU Applications

Probably the most obvious field of computer science that can greatly benefit by the exploitation of graphics processors as computing devices is *cryptography* and *pattern matching*. CryptoGraphics [8] is among the earlier works that implement cryptographic primitives for symmetric key encryption on the GPU. Although the actual implementations did not outperform CPU implementations at the time, they clearly showed that the main problem was the lack of expressive enough APIs. With new generations of graphics processors supporting integers natively, Harrison et al. reported considerable speedups with their AES implementation. [17] Another work on crypography is SSLShader [19], which provides accelerated versions of the RSA, AES and SHA-1 cryptographic primitives. The project aims to offload the expensive operations required by the SSL (Secure Sockets Layer) protocol. The authors report that the system greatly outperforms state-of-the-art CPU configurations. Among others, there is a suite of freeware tools called hashcat tools [5] that utilizes graphics hardware for password cracking. It mainly targets cryptographic hash functions for password recovery, by implementing a number of attacks (brute force, dictionary based, rule based) on the GPU.

[4] The project's home can be found at `https://github.com/wbsun/kgpu`.

[5] It is available for download at `http://hashcat.net/`.

Gnort [40] is a GPU-accelerated prototype based on the open-source Snort [6] NIDS (Network Intrusion Detection System). It shows that the system can cope with twice the processing rate than the original Snort software. The performance improvement comes from its parallel regular expression matching engine that offloads expensive computations to the GPU. Furthermore, Smith et al. confirm that signature matching with regular expressions can run several times faster on the GPU, depending on the underlying automaton implementation. [37] GrAVity [41] is a massively parallel antivirus engine. It uses a modified version of the open-source ClamAV [7] antivirus software that performs two orders of magnitude better than the original.

There are some notable works in the field of storage systems as well. GPUstore [38] is a framework for encryption and RAID recovery using the GPU. The system is part of and built around the infrastructure of the kgpu project within Linux and shows substantial improvement over the default implementations. Shredder [4] is also a system that provides computational facilities for storage systems. More specifically, it is designed for content-based chunking, a technique used in data deduplication and incremental computation schemes. Again, most of the optimizations discussed by the authors are implemented or evaluated for use in this work.

The power of GPU computing has began to enter new areas, such as *molecular biology* and genetics. The Smith-Waterman is a dynamic programming algorithm used to find the optimal local alignment of two DNA sequences. However, its computational complexity makes for a challenging task. Some of the most noteworthy projects of this area are SW-CUDA [24] and CUDASW++ [22] that accelerate the algorithm's execution using the parallel engines of modern graphics hardware. CUSHAW [23] GPU-accelerates a sequence alignment algorithm based on the Burrows-Wheeler transform and the Ferragina-Manzini index.

Few projects have surfaced that try to enhance networking performance using GPU computing. Nuclei [36] utilizes GPUs for network coding, an alternative approach to routing (store and forward), that is known to have better performance on a variety of networks. Despite its appealing properties, network coding has high computational demands and that's why it's a good fit for the parallel computing model of the GPU. In this work, however, compatibility with other network components is important and no optimizations are considered at this level.

[6] The project's webpage is located at `http://www.snort.org/`.

[7] The source code is available at `http://www.clamav.net/`.

2.4 Networking using the GPU

This section refers to the most relevant systems to this work. All of them have similar features or goals and share the use of the GPU to achieve better packet processing throughput. The closest relative of this work appears to be PacketShader [16], a software router framework with support for packet processing on the GPU. PacketShader has it's own I/O engine (PSIO) that uses techniques similar to netmap in order to optimize packet data movement, but it is built around and supports one specific network adapter. Compared to it, the system developed in this work introduces some novel optimizations concerning packet data movement to minimize DMA transfers and improve maximum performance. Additionally, a new zero-copy mode is implemented and evaluated.

MIDeA [42] is an NIDS that exploits modern architectures to parallelize the task of detection. It maps the multi-queue NIC to multiple multi-core processors with RSS (Receive Side Scaling) and many GPUs to distribute the system's load. MIDeA uses `PF_RING` for its packet capturing and queue handling facilities and provides a modified Snort system. Kargus [18] further improves the aggregate throughput of a GPU-accelerated NIDS. The authors use the PSIO engine (from PacketShader) to boost packet reception and also introduce multiple network adapters to push the system further. Both systems operate in passive mode and their needs do not involve high-speed packet forwarding but only efficient reception.

Chapter 3

Methodology

The development of a system that bridges the networking device with the graphics processing unit is bound to face many challenges and must employ sophisticated techniques in order to overcome the various problems. This section describes the technologies, methodologies and issues that surface during this process. In short, these include choices of data structures, hardware requirements, common optimizations in networking and graphics processing, data movement models, operating system internals, architectures and security considerations.

3.1 Ring Buffers

A *ring buffer* is a data structure that uses a single fixed-sized buffer for storage, as if it were connected end-to-end. Writing to the buffer, advances a pointer or counter, which wraps-around when the end is reached. Reading from the buffer also moves forward a respective pointer or counter, freeing up space. This structure and model lends itself easily to buffering data streams. It is efficient because of its inherent FIFO (First In First Out) principle. Furthermore, there is no need for allocation and deallocation of memory when new data is inserted or removed; old data is instead overwritten by the new and no data shifting is required.

It is because of these properties, that both hardware and software related to packet transmission use ring buffers in the I/O pipeline. Most high-performance network chips implement ring buffers for packet descriptors that manage packet data. Shadow copies of the descriptor rings are kept in main memory as a means of synchronization for software (such as the device driver) with the hardware. Moreover, the actual packets are queued in ring buffers to avoid the cost of the per-packet allocation, that can be significant for the operating system at high packet rates. Ring buffers can be further optimized to benefit from DMA (Direct Memory Access) by using contiguous memory for the underlying buffer. In addition, contiguous memory allocated by the

kernel can be mapped (using `mmap(3)`) to a process's virtual memory space to be used directly. For details on how the assumption of a linear memory space can affect programming practices see Section 4.2.

3.2 Use of Packet Slots

Modern high-speed network adapters have special capabilities but they also have special requirements. They are designed to have their receive data storage in main memory in order to cope with the high packet rates and avoid packet loss due to heavy system load. This design forces packet buffers to be large enough to hold even the largest of packets, because there is no way to know the packet length beforehand. When packet buffers are pre-allocated on a larger ring buffer they are called *slots*. Fixed-sized slots have many advantages and lead to simple designs because of their manageability. Slots make indexing fast, for buffer offsets in the ring descriptors can be pre-computed. Additionally, they are a good match with alignment requirements often found in DMA engines of peripheral devices like NICs.

Although the slotted design seems like a reasonable solution, it is very bad in respect to space efficiency. The size of the memory occupied by packet data starts to become important when batches of packets need to be transferred over to the GPU's memory. As previous studies have shown [16, 40, 42], contrary to NICs, current GPU implementations suffer from poor performance for small data transfers (this is verified in Section 5.3). To improve PCIe throughput, several packets are transferred at once. However, the fixed-size slots scheme leads to redundant data transfers for traffic that consists of small packets. For example, a 64-byte packet utilizes only the 1/32 of the available space in a slot of 2048 bytes (the default buffer size in many device driver implementations). This introduces an interesting trade-off: whether it is better to use a single buffer that is shared between the NIC and the GPU (zero-copy approach) or stick with the traditional model and copy the packets into an intermediate buffer along with packet indexes, as described in Section 4.1. As will be seen in more detail in Section 5.4, there are cases where it is better to utilize a second buffer, where the packets are stored in a compact manner.

The process of re-arranging the packets for consuming less space in memory is illustrated in Figure 3.1. The buffer in the top represents the original storage used by the network adapter, partitioned in equally sized slots. The arrows are memory copies targeting the buffer at the bottom. This introduces a very specific access pattern that involves read operations with a stride equal to the slot size and sequential writes. In symmetry, after packet processing, packets inside a compacted buffer are put in slots when sent for transmission. The effects of compaction/de-compaction in the data path are discussed in Section 5.2.

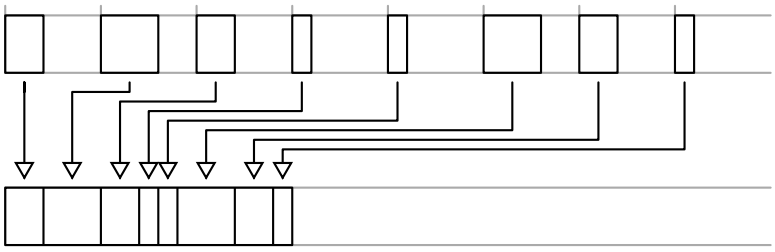

Figure 3.1: Performing compaction of packet buffers. Each packet is copied to a new location right after the previous packet, resulting to a more space-efficient storage scheme.

The packet buffer slot size was chosen to be 1536 bytes because of the following reasons. First, packet slots should be large enough to hold maximal packets along with the Ethernet headers, which adds up to 1518 bytes (see Section 5.2). Second, slot buffers should conform to alignment restrictions imposed by the hardware [9] and the size is important because it is desired for the slots to be adjacent. Last, the total memory allocated for packet slots should be minimal, not only for space efficiency, but also for performance, because, as discussed above, unused space inside the slots leads to worse "goodput" in zero-copy transfers to the GPU. Additionally, the NIC issues fewer requests for buffers when many packets reside in the same memory block, and slightly improves performance for small packets where the effect is at its peak.

3.3 Interrupts and Polling

The most straightforward way to implement the device driver for a NIC (Network Interface Card) is using interrupts. The device interrupts the kernel by issuing an *interrupt request* (IRQ) and the kernel services this request by executing the appropriate *interrupt service routine* (ISR). However, handling IRQs is expensive in terms of both computational resources and time. Considering that on interfaces with link speeds of many Gigabits thousands of packets arrive per second, such an implementation would be very inefficient. Furthermore, this phenomenon known as an "interrupt storm" can lead to even more serious problems. The system can become completely unresponsive because the kernel consumes all CPU time to service interrupts, and no user process can be scheduled. Even worse, because interrupt handlers cannot be preempted, the system may enter a state called *livelock* in which it does not manage to do any useful work. [26] Hopefully, the many cores of today's processors are not so easy to saturate altogether, but that does not solve the problem.

To cope with the problem, some operating systems and network adapter device drivers include a *device polling* mode. In polling mode, normally enabled when the network load increases, the interrupts are disabled on the interface and the driver receives packets in batches. This works better because there always are packets available for reception on busy links. For instance, Linux has introduced NAPI [35] (New API) as an infrastructure for polling NIC drivers; FreeBSD has a similar feature and KPI (Kernel Programming Interface). However, many of today's modern NICs support *interrupt moderation* in hardware. With this feature enabled the device waits for a number of packets to come or a timeout before generating an interrupt. The exact algorithm is vendor-specific but allows for simpler and more robust driver implementations. The bottom line is that on high-speed interfaces, like the 10Gb NICs used in this work, the per packet interrupt overhead cannot be tolerated and one of the above interrupt mitigation mechanisms has to be adopted. For practical observations and results regarding how interrupts are handled please go to the bottom of Section 5.2.

3.4 Asynchronous Transfers

The nature of performing computation on a co-processor such as the graphics chip requires several extra considerations when compared to a program that does the same computation using the CPU. These include transferring the data from the host memory to the GPU memory, executing the kernel program using the parallel processing model, and transferring the results back to the host memory. All these operations take time to complete, therefore the tasks of scheduling and synchronization need to be taken care of. As a result of the GPU processing model, development requires adhering to specific coding patterns. Details about synchronization techniques and related data structures are given in Section 4.1.

A key attribute in configuring the runtime of the system is data movement. Memory transfers to and from the device can be either *synchronous* or *asynchronous* while kernel launches are always asynchronous with respect to the user program. Using synchronous operations blocks the control flow waiting for them to complete; this greatly impacts the performance of the system. Packet buffers fill-up, causing packet drops and the overall performance degrades. To avoid this behaviour and better utilize resources an asynchronous model is adopted.

Figure 3.2 illustrates the pipeline of packet reception, processing and transmission. At any given time slot of the program's execution incoming packets are written to the receive ring buffer and outgoing packets are read by the network adapter's DMA engine and sent on the wire. At the same time, packets received in the previous time slot are copied over to the GPU, processed and written back, ready to be sent on the next time slot. Buffer

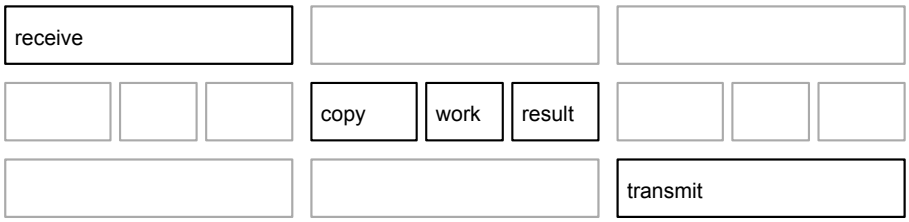

Figure 3.2: The I/O and processing pipeline. The packets received at the current time slot will be processed at the following time slot and finally sent during the time slot after that.

swapping techniques, similar to *double buffering* are used to achieve acceptable performance. More information about the caveats involved are given in Section 4.1.

For the sake of completeness, note that there is another mode for data movement between the host and the device that is implemented by mapping host address ranges to the device memory space. This way only memory accessed by the graphics chip is actually transferred. The transfers are transparent to the user process, but introduce some extra overhead. Although this specific functionality can be useful in many applications, the fact that network processing may require all packet data, and not only a fraction of it, to be accessed by the processing routine renders this method useless. Details about this style of mapped memory operation can be found in [29].

Systems targeting performance take extensive measures to minimize memory allocations and memory copying. The former is easily solved by pre-allocating buffers and managing those buffers in the context of the application. The point of this technique is to avoid the overhead introduced by the memory allocator, because it can be of significant cost if it is on the critical path of a program. The buffers are not freed, but are re-used instead. Even though buffers can be re-used, memory copies are even more resource-consuming, as they occupy both the CPU and the memory controller. However, on many occasions data movement can be optimized using *zero-copy* techniques. Figure 3.3 shows how the DMA engines of the NIC and the GPU communicate data to and from the main memory. In general, the memory used for DMA operations is dedicated to one device and the CPU handles the data movement on the main memory. In other words, a packet is received with DMA in a buffer, it is copied to another buffer, and then it is sent with DMA to the GPU. The intermediate copy can be avoided by using a shared buffer for both devices. This approach involves allocating memory in the kernel by the NIC's device driver, exposing it to the user program, and "registering" this memory area as DMA-able by the GPU. More in depth information about how this is achieved is given in Section 4.3.

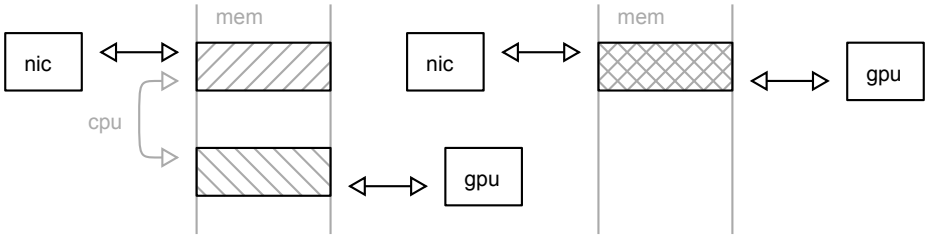

Figure 3.3: Packets are transferred using DMA from the NIC to the main memory, from the main memory to the GPU device, and all the way back. The buffers on main memory are distinct and the CPU copies all packets between them, or the buffers are shared.

3.5 The Host's Network Stack

Normally, the operating system configures and manages the various aspects of a network interface's operation. This includes statistics about packets received and transmitted, active TCP flows, in-kernel firewall state, etc. However, when designing applications to work inside the network, as a network component, the cost imposed by the kernel infrastructures, often becomes the bottleneck and the system cannot keep up with the high rates supported by the hardware. When performance is the ultimate goal, the operating system's stack can be circumvented; the user application can do all management work with the minimal required effort.

The problem becomes even more prevalent when trying to make efficient use of other resources, such as the GPU co-processor, in the I/O path. To utilize the GPU processing power, data needs to be transferred also through user space. Note that the latter is not inherent to the graphics hardware, but related to the GPU programming abstractions available at the moment. Although, kernel space operation would probably be more efficient, graphics device drivers neither export a computing API for use in the kernel, nor their source code is available. Gdev [20] is a very recent work towards an open interface for GPU programming at the operating system level.

Another important parameter that should be taken care of is I/O batching. As confirmed by previous work, packet batching is essential in order to hide the various per-packet costs associated with packet reception. [33, 21] By choosing a large-enough batch size, the system call overhead used for synchronization becomes negligible. Furthermore, packet batching maps very well with the GPU processing model, because the effectiveness of the latter is more apparent with large amounts of data.

3.6 Multiprocessor Systems

Computer architectures have grown into complex systems that require much effort to understand in depth and configure appropriately. Aiming at maximum aggregate performance, current systems consist of many processors, and each processor may contain many individual processor cores, that form groups or *nodes*. Typically, all cores that reside in the same node share some resources among their siblings, such as the level 3 (L3) cache, while having separate level 2 (L2) caches for example. All CPUs also share the main memory and run under the same operating system.

When the processors of the system are homogeneous with respect to the memory and the I/O devices, the system is an SMP (Symmetric Multiprocessor) system, enabling parallel execution flows. However, modern architectures have introduced some heterogeneity on the memory interface and hierarchy in order to address the speed limits imposed by the memory when many processors need to operate on it. NUMA (Non-Uniform Memory Access) systems have come to solve the problem of the shared memory bus by providing separate memory to each CPU node, and thus forming *memory domains*. All nodes, in turn, are linked together using high-speed interconnects. Although access to every memory address is possible by any processor, there exist *local* and *remote* domains with respect to the processor a program runs on.

All these additions have given significant performance boost to computers, but also created systems more challenging to program, configure, and use at the maximum of their capabilities. Blagodurov et al. give some insight and solutions to the problem of scheduling programs to run on the same memory domain where their data reside [6]. This property is very important to this work where multiple buffers are read and written simultaneously by the NIC, the GPU and the user process, so the correct scheduling and even the physical placement of the peripheral interface cards should be considered and optimized.

Figure 3.4 displays an abstraction of the internal architecture of the system used in this work. To discover the best combination of CPU core, memory allocations, and physical placement, many valid combinations were tested starting with the more intuitively probable. One NIC and one GPU were placed on the same I/O hub in order to belong to same memory domain and have lower access times to the same memory. The most common principle that applies to such configurations is to minimize the cross-node memory accesses. This proved to work good enough, but did not give the peek performance at all times. In fact, the configuration that achieves the maximum possible throughput is not the same for both large and small packets! Quantitative information on the matter is given in Section 5.2.

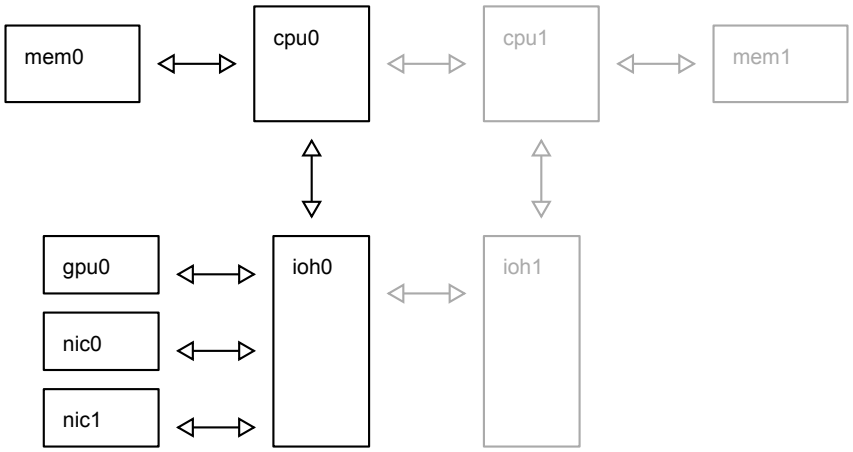

Figure 3.4: The architecture block diagram. All four links among processors and the I/O hubs are high-speed interconnects. The I/O hubs communicate with peripherals via PCI links. Cross-node memory accesses are more expensive due to the extra hop.

3.7 The Modern GPU Architecture

In this section, the key features and design details of NVIDIA's CUDA (Compute Unified Device Architecture) hardware and software architecture are outlined. Although this information is based on a very specific family of products, many of these concepts are shared by a broader group of vendors that design graphics chips and general purpose computing interfaces. All information is extracted from official documentation and the CUDA user manuals [29, 28, 30].

As mentioned in the introduction, the GPUs can execute programs written in high-level programming languages with some extensions. For example, for the C and C++ languages a modified compiler is provided by the CUDA SDK that takes care of all GPU-related code and optimizations. A CUDA program calls parallel *kernels*. A kernel executes in parallel across a set of parallel *threads* on the device. These threads are organized in *blocks* and many blocks can form multi-dimensional *grids* of blocks depending on the needs and semantics of the problem the application is designed for. In this configuration, a thread runs an instance of a kernel and can be addressed using multiple identifiers: *(i)* the thread id which is unique inside a thread block, *(ii)* the block id within its grid and *(iii)* the grid id.

Furthermore, each level of abstraction is matched with different levels of caches, registers and memory. Figure 3.5 displays how the memory organization of the GPU relates to the execution model. Each thread within a block has its own program counter, registers, private memory, inputs and outputs.

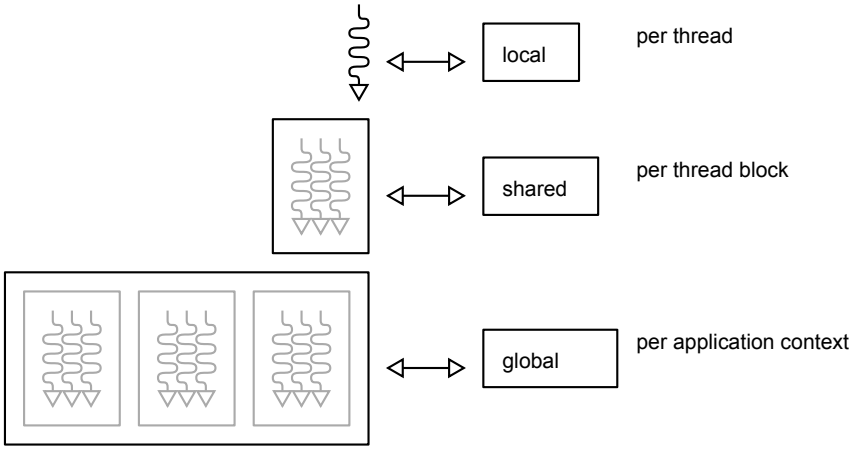

Figure 3.5: The memory model of the CUDA architecture. Threads have access to private local memory, share a cache with the other threads inside the same block and perform operations on the global memory concurrently from all grids.

A thread block is a set of concurrent threads that can cooperate using barrier synchronization and shared memory. A grid is an array of thread blocks that execute the same kernel and do I/O on the global DRAM of the device; they synchronize between dependent kernel launches.

A CUDA enabled device is built around a scalable array of *streaming multiprocessors* (SMs). The organization of threads maps to a hierarchy of SMs on the GPU. A multiprocessor is designed to execute hundreds of threads concurrently and to do so it employs an architecture that follows the SIMT (Single Instruction, Multiple Threads) paradigm. This is closely related to SIMD, with the former being a bit more expressive, as it enables random memory accesses and control flow divergence. These features, of course, come at a cost as discussed below. Each SM executes one or more thread blocks, while the cores in the SM execute the actual threads. Threads are executed in groups of 32 parallel threads called a *warp*. Performance is greatly improved if all threads in a warp follow the same code path and access nearby addresses in memory. If one thread diverges the rest of the warp should wait for it to converge back in order for execution to continue; this introduces considerable latency.

The data parallel architecture and the flexible programming model introduced by the modern graphics processors maps well with the huge amount of network traffic that flows on today's network infrastructures. In it's simplest and most efficient form the partitioning scheme used for packet processing on the GPU is to map each packet to a different kernel thread. Among the pros of this approach is that every packet is processed independently and

the code that does the work is fairly straightforward. The main suspected drawback is that execution times among threads may differ depending on the variation of packet sizes in a given warp, but that is mostly hidden by the warp schedulers. Further details about the kernels' format can be found in Section 4.4.

3.8 Security Concerns

The use of the GPU to store, even temporarily, and manipulate data that come from the network, naturally raises some security questions. "Are the packets on the GPU safe?" Safe in this case would mean that no other process can access the same memory used by the system for packet storage. NVIDIA GPUs not only use virtual memory addressing, but also manage some sort of process context (that includes an identifier). Device pointers are validated through an address translation scheme together with the context, thus there is not a straightforward way to manipulate device memory. [5] However, by reverse engineering the creation of virtual address space objects and context objects, there is a good chance that these objects can be forged in order to trick the pointer validation mechanism.

Another issue that is present when using the CUDA library is the fact that it does not respect the system's resource limits. All function calls of the interface that allocate memory at the host, provide page-locked memory regions, effectively reducing the memory available to the operating system for paging. Although the developer's manual advises against the excessive use of these functions due to possible performance degradation, the fact that any program can lock large amounts of system's memory through this mechanism —without requiring special privileges— should be noted.

Given a hardware configuration and the graphics/network adapter physical placement, the processes executed in the system deliver significantly different results according to which processor they got scheduled to run on. To obtain optimal performance, the execution flows should be bound to a specific node, as close as possible to the buffers and the devices. However, hard binding programs to specific processors can result in starvation of other critical programs. This is really a job that the scheduler should do, but there are many parameters involved in this specific use case and the scheduler was not designed to consider all of them. The answer to all the security related concerns expressed here is that the system was not designed to accommodate multiple users; it is rather a prototype of a general purpose networking appliance. So the internals of the system are considered part of a trusted environment.

Chapter 4

Implementation

This chapter describes the system implementation and gives details about data structures, special procedures and programming interfaces. Furthermore, in depth information is provided on the data movement operations to and from the devices, and on the computation model used for the packet data processing.

4.1 Data Structures

In order to address the coordination and synchronization issues between packet reception/transmission by the network adapter and packet processing by the graphics chip, some basic data structures are needed. Furthermore, the lightweight layer provided by them helps to transparently implement other techniques such as buffer swapping, that are essential for performance.

The definition of the core data structure used in this work is displayed in Figure 4.1. The inner structure called `pktbuf` defines a simple buffer (`buf`) intended to store packet data, which is accompanied by metadata, counters and limits. The `off` array holds the offsets to the packets stored inside the buffer, the `len` array holds their lengths, and `usr` is designed for generic metadata storage, i.e. the associated interface identifier. The choice of offsets over pointers is for the packets to be accessed portably in the different address spaces. The buffer has a capacity for `max` packets or `bufsiz` bytes; the buffer is considered full when either of them runs out. The `cnt` counter is about the current number of packets while `used` designates the total bytes occupied. Finally, `pos` is used internally as an iterator for consuming packets.

On an upper level, there are many packet buffers managed in the context of an `xpktbuf`. There is the input, output, device and temporary packet buffers, namely `in`, `out`, `dev` and `tmp`. This is illustrated in Figure 4.2 with the faded color selected to designate that the device buffer is optional; it's not used when running in zero-copy mode, as explained below. At any given time

```
struct pktbuf {
        unsigned char   *buf;   /* Packet storage */
        int             *off;   /* Packet offsets */
        unsigned short  *len;   /* Packet lengths */
        unsigned char   *usr;   /* Packet user data */
        unsigned int    max;    /* Max number of packets */
        unsigned int    cnt;    /* Packets counter */
        unsigned int    pos;    /* Packets iterator */
        unsigned int    bufsiz; /* Buffer size */
        unsigned int    used;   /* Amount used */
};

struct xpktbuf {
        struct pktbuf   *in;    /* Input buffer */
        struct pktbuf   *dev;   /* The device copy */
        struct pktbuf   *out;   /* Output buffer */
        struct pktbuf   *tmp;   /* Swap buffer */
        stream_t        stream; /* Async transfers */
        int             flags;  /* Operation mode */
#define XPKTBUF_SYNCBUF (1 << 0)
#define XPKTBUF_SYNCOFF (1 << 1)
#define XPKTBUF_SYNCLEN (1 << 2)
#define XPKTBUF_SYNCUSR (1 << 3)
#define XPKTBUF_SYNCALL \
    (XPKTBUF_SYNCBUF | XPKTBUF_SYNCOFF | \
     XPKTBUF_SYNCLEN | XPKTBUF_SYNCUSR)
};
```

Figure 4.1: The definition of the host-device hybrid packet buffer data structure. It holds packets and records offsets, lengths and user data. Normally, the input, output and temporary buffers are allocated from the host's memory and there exists one copy of the latter on the device.

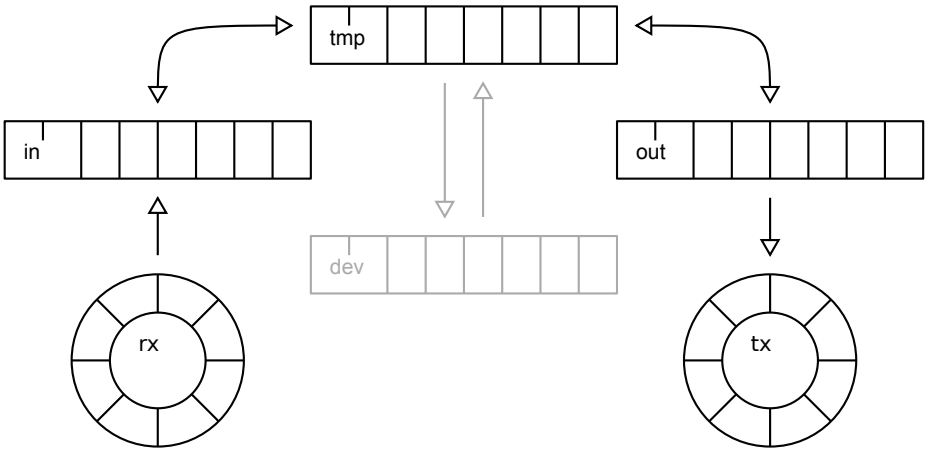

Figure 4.2: The receive and transmit path showing the buffer swapping techniques used to achieve asynchronous I/O. Once the input buffer is full and the output buffer is empty, the buffers are swapped for processing to continue safely.

during the system's execution three major things do happen: *(i)* packets are moved from the receive ring (rx) to the input buffer, *(ii)* packets are moved from the output buffer to the transmit ring (tx), *(iii)* packets are processed on the device and the results are moved back to the temporary buffer. When in normal mode, only the device buffer lives in the address space of the GPU and is synchronized with the temporary buffer once in every time slot of the I/O pipeline, by means of a DMA transfer. The synchronization point is reached when both the input buffer is full and the output buffer is empty. At this point two *buffer swaps* occur (indicated with unidirectional arrows). The first one happens between the temporary buffer and the output buffer in order to move the new results to the output queue. The second one happens between the input buffer and the temporary buffer in order to send the newly received packets for processing. Because those two swaps are performed at the same time the buffers are effectively "rotated".

The `flags` field defines the level of synchronization enabled between the device and the host when the result of the processing is obtained. All data is transferred back to the host by default (`XPKTBUF_SYNCALL` flag), however that might not be necessary in all applications. For example, a processing procedure may only need to filter packets. In that case, the buffers, packet lengths and user data are not touched. By providing "invalid" offsets for the packets that should be dropped, only getting the offsets back is enough for the application's demands, and this is exactly what `XPKTBUF_SYNCOFF` does. In the same manner, the flags `XPKTBUF_SYNCBUF`, `XPKTBUF_SYNCLEN` and `XPKTBUF_SYNCUSR` enable the synchronization of the buffer, lengths and

user data respectively. This helps configuring the system for maximum performance by minimizing unneeded transfers. However, what is needed or not is open to interpretation by the program developer. The `stream` field is an implementation dependent stream identifier that aids in the serialization and synchronization of transfers. Its use and operational model are described in Section 4.3.

The system also implements a zero-copy variation for the data transfers between the host and the device. Instead of copying every packet from the receive ring to the input buffer, a batch of packets is transferred to the input buffer that now lives on the device. The same applies to the output and temporary buffer, while the fourth buffer is not needed as the data is already on the device. This way, the only buffers that remain on the host are the receive and transmit rings, and all `memcpy` calls are effectively replaced by, already present, DMA transfers. The performance of this approach, though, is not straightforward because of the holes inside the original packet buffers. The reason for their presence is discussed in Section 3.2. For experimental results and discussion on this technique see Section 5.4.

The interface provided to handle the data structures presented previously is shown in Figure 4.3. The `xpkbuf_new` and `xpktbuf_free` functions create and destroy packet buffers for both host and device use. All parameters correspond to buffer configuration and runtime options. Additionally, the `xpktbuf_setstream` function initializes the stream identifier for the cross-device transfers to get enqueued in a stream other than the default. The `xpktbuf_addtoin` function appends one packet of given length to the input packet buffer. The `xpktbuf_addmanytoin` function also manipulates the input buffer but instead transfers a batch of packets directly from a memory location like the receive ring; this requires all lengths and offsets to be provided. The `xpktbuf_syncstreams` function synchronizes the device operations and does buffer swapping to prepare for the next round, as described above. The `xpktbuf_copyintodev` and `xpktbuf_copydevtoout` functions initiate (or enqueue) the DMA transfer operations from the host to the device or from the device to the host respectively; these functions are aware of buffer swapping and should be called once in every time slot and in that order. The `xpktbuf_nextout` function gets the next packet from the output buffer. The `xpkbuf_nextmanyout` function gets many packets by transferring them from the device; the batch of packets gets written sequentially on a memory location like the transmit ring. The `xpktbuf_isfullin` and `xpktbuf_isemptyout` functions report on the full status of the input buffer and the empty status of the output buffer respectively. Note that the functions having "many" in their name are suitable and optimized for use with the zero-copy approach.

```
struct xpktbuf *
xpktbuf_new(unsigned int max, unsigned int bufsiz, int flags);

void
xpktbuf_free(struct xpktbuf *xpb);

void
xpktbuf_setstream(struct xpktbuf *xpb, stream_t stream);

int
xpktbuf_addtoin(struct xpktbuf *xpb, unsigned char *pkt,
    unsigned short len);

int
xpktbuf_addmanytoin(struct xpktbuf *xpb, unsigned char *buf,
    unsigned int bufsiz, int *offary, unsigned short *lenary,
    unsigned int n);

void
xpktbuf_syncstreams(struct xpktbuf *xpb);

void
xpktbuf_copyintodev(struct xpktbuf *xpb);

void
xpktbuf_copydevtoout(struct xpktbuf *xpb);

int
xpktbuf_nextout(struct xpktbuf *xpb, unsigned char **pkt,
    unsigned short *len, unsigned char *usr);

int
xpktbuf_nextmanyout(struct xpktbuf *xpb, unsigned char *buf,
    unsigned int bufsiz, int *offary, unsigned short *lenary,
    unsigned int n);

int
xpktbuf_isfullin(struct xpktbuf *xpb);

int
xpktbuf_isemptyout(struct xpktbuf *xpb);
```

Figure 4.3: The packet buffer API. The complete interface used to handle packets and transfer them between the host and the device. Buffer swapping is performed by the "syncstreams" function which matches the start of a time slot in the I/O pipeline.

```
base = NULL;
while (cur = next(buf, &len)) {
        if (base == NULL)
                base = cur;
        if (cur < base || cur - base > limit) {
                transfer(dst, base, prev - base);
                base = cur;
        }
        prev = cur + len;
}
```

Figure 4.4: Generic sample code to monitor a ring buffer. The call to a "next" function returns a pointer inside the buffer, where the next item lives. By saving a base pointer and the end of the previous item, a transfer is triggered when a certain limit is reached.

4.2 Special Procedures

Assuming that received packets are buffered in a ring buffer using a virtually contiguous memory region, the logic of monitoring the status of the packet buffer and transferring data to the device is shown in Figure 4.4. The function `next` returns a pointer to the start of the slot holding the next available packet (`cur`). Using pointer arithmetic the offset of the current slot (`cur - base`) is calculated, using a base address obtained in the first iteration. Also, a `limit` smaller than the size of the whole ring buffer is chosen. Synchronization between the host's and the graphics adapter's memory happens when that limit is reached or the ring buffer wraps-around. The memory area being transferred over starts at the `base` and is of length `prev - base`. A pointer to the end of the preceding packet (`prev`) is necessary in order to compute the area length in the event of a wrap-around (`cur < base`).

4.3 Data Movement

All data movement operations between the network adapter's queue and the host's main memory happen with DMA. The netmap [1] framework is used for packet I/O. It provides abstractions for synchronizing with the hardware, but also gives enough freedom to control the state of the receive/transmit queues. Packet reception and transmission are achieved using `poll(3)` on a set of file descriptors, one for each interface, and manipulating data structures that handle packet slot data. For an overview of netmap's kernel infrastructure and its user level library see [33].

[1] `http://info.iet.unipi.it/~luigi/netmap/`

Data transfers between the host and the graphics adapter are done using CUDA's asynchronous memory copy functions (`cudaMemcpyAsync` and friends [2]) which are always initiated by the program running on the host. In fact, all operations including asynchronous transfers and computation (kernel launches) are queued in the stream designated by the `stream` identifier of the packet buffer instance. A *stream* is a sequence of commands that execute in order. The stream of a packet buffer is explicitly synchronized when packet I/O is finished. That means that program execution waits until all preceding commands have completed, for the system's pipeline to advance.

However, for the GPU to accelerate a transfer, the host memory in question has to be non-pageable and also it must be added to `nvidia` driver's internal tracking mechanism. For CUDA v3.2 or lower this meant that the memory had to be allocated using the API provided by the framework (`cudaMallocHost`). With the release of CUDA v4.0 the addition of the `cudaHostRegister` function call gave a solution. Now memory regions can be registered as memory eligible for accelerated DMA transfers. Buffers that that do not fulfill these requirements are first copied to some staging memory, which slows things down significantly. Before the new functionality came out, a workaround was used in this work. A dummy buffer of the same size with the target buffer was allocated using the proprietary CUDA API, and the page table of the process was altered to make the dummy buffer point to the physical pages of the original target buffer. With this hack, the GPU was tricked to perform accelerated transfers from/to generic non-pageable memory. The memory translation and re-mapping facilities were provided by a Linux kernel module that was developed for this exact purpose.

Either way, the zero-copy approach needs the packet storage of the capturing framework to be registered as GPU buffers. In the case of netmap's exposed NIC rings this was not possible "out-of-the-box". The problem seems to originate from the `nvidia` driver and the flags and/or structure associations it expects to be present on the virtual memory area passed to `cudaHostRegister`. However, the driver comes as a binary blob with no source code available. In order to overcome this problem, the netmap kernel module, and more specifically the code that implements the `mmap(3)` functionality, was modified. The patch simply changes the routines used to export the allocated pages to userspace. Apparently, the problem here is the use of `remap_pfn_range` which produces raw PFN (Page Frame Number) mappings and the pages do not have a `struct page` associated with them. Using `vm_insert_page` for every page instead works.

[2] `http://developer.download.nvidia.com/compute/cuda/4_2/rel/toolkit/docs/online/sync_async.html`

4.4 Computation Units

Probably the most appropriate unit to perform computation on in the current system is the packet. Generally, an application needs to know *which* packet matches certain criteria or rules in order to act. Other applications only need to know the *flow* a packet belongs to or simply *whether* there is a match or not. Therefore, the finer granularity of per-packet operation is adopted in order to satisfy all scenarios. Each packet is assigned to a different thread that runs on a separate stream processor.

Figure 4.5 demonstrates how the packet buffer data structure interfaces with GPU kernel launches. Note that the original code is slightly altered for presentation purposes. The function `pktproc` is a wrapper around the `kpktproc` GPU procedure (the `k` in the name as well as a special qualifier before the return type designate that this is a kernel function). The kernel is configured for parallel execution using the `<<< >>>` syntax; the packet buffer is partitioned in blocks of 512 packets (with the exception of the last one that may contain less). Choosing that kind of parameter is a matter of balancing latency hiding and multiprocessor utilization and requires some experimentation. The value of 512 threads per block is considered a sane default but in no case an optimal value for all applications. The pointers passed to the kernel function should point to valid device memory locations and thus the packet buffer `buf` and helper arrays `offary`, `lenary` and `retary` are device copies of the originals that reside in the host. The `bufsiz` and `n` arguments are used for handling the possibly incomplete last block and sanity checks. Each thread accesses a specific packet using its `pktid` identifier, which is constructed using the built-in variables (with the `block` and `thread` prefix) provided by the runtime environment. This skeleton kernel function can be thought as the equivalent of a libpcap callback function, as it eventually provides the same functionality: a pointer to the packet data and metadata such as the packet length.

```
__kernel__ void
kpktproc(unsigned char *buf, unsigned int bufsiz,
    int *offary, unsigned short *lenary, int *retary,
    unsigned int n)
{
    unsigned int pktid;
    unsigned char *pktp;
    int off;
    unsigned short len;
    int *retp;

    pktid = blockidx * blockdim + threadidx;
    if (pktid >= n)
        return;

    off = offary[pktid];
    if (off >= bufsiz)
        return;

    pktp = buf + off;
    len = lenary[pktid];
    retp = retary + pktid;

    /* do process packet */

    return;
}

void
pktproc(struct pktbuf *pb)
{
#define CEILDIV(x, y) ((x + y - 1) / y)
#define BLKSIZ 512
    kpktproc<<<CEILDIV(pb->cnt, BLKSIZ), BLKSIZ, pb->stream>>>
        (pb->buf, pb->used,
         pb->off, pb->len, pb->usr, pb->cnt);
}
```

Figure 4.5: The skeleton of the kernel function definition for processing packet buffers. The wrapper function is common practice for doing synchronization and hiding the execution configuration from the user.

Chapter 5

Evaluation

There exist numerous parameters that affect the performance of the developed system. Most of them are isolated and evaluated separately to come up with the optimal configuration. For example, benchmarks have been written to experiment with the memory bandwidth (Section 5.1) and the performance of transfers between the host and the device (Section 5.3). Also, the rate achievable by the network adapters in many configurations is measured in Section 5.2. In more detail, options that are configured include the size of the packet slots, as seen in Section 3.2, the number of packet descriptors used by the network driver, the batch size for packet reception and transmission, the batch size of packet data transfers to and from the graphics hardware. Moreover, not only the local or remote placement of the hardware components between each other, but also the placement of the software resources relative to the hardware is configured. The rest of the parameters are the packet size used for the synthetically generated packets, the mode of system operation (passive or active), the method of data movement (compaction-first or zero-copy), the data transfer optimizations (on or off) and the number of concurrently active network interfaces.

Three machines were used for the evaluation of the system. The base machine is equipped with two Intel Xeon E5520 Quad-core CPUs at 2.27 GHz and 12 GB of DDR3 1066 MHz RAM (6 GB per memory domain). Each CPU is connected to the peripherals via a separate I/O hub, linked to several PCIe slots. A GeForce GTX 480 graphics card and several Intel 82599EB 10 GbE single- and dual-port network adapters have been used for the evaluation of the system. Two similar machines with Intel Xeon 5120 CPUs and 6 GB of memory are used as traffic generators for synthetic data, as well as the replaying of real network traces. All machines run 64-bit Linux 3.0.0, with the `netmap` kernel module and a modified `ixgbe` driver. For the purposes of packet generation, the sample `pkt-gen` application that comes with the netmap release is used. It's functionality has been extended to also support pcap trace file replay.

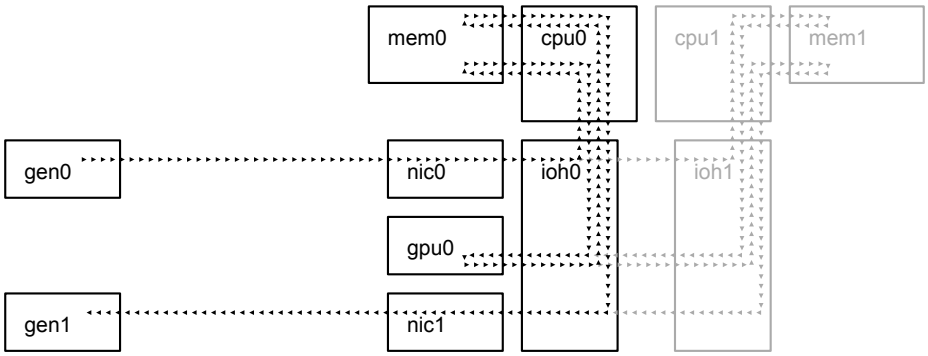

Figure 5.1: The experimental system setup. The arrows show the flow of packets from the point they enter the system up to the point they leave. The memory accesses can be local or remote.

Figure 5.1 shows the layout of the experimental setup. The network interfaces of the generator machines are displayed on the left, while a more detailed picture of the base machine is on the right. The arrows indicate the path followed by the data inside the system. Consider, for example, that the first machine (`gen0`) acts as a packet source, while the second (`gen1`) as a sink. The packets are received by the first port (`nic0`) and into memory. From there they are copied to the GPU for processing; the results are transferred back to memory and from there the packets get written to the output network port (`nic1`). When following the black arrows all memory accesses are done to the local memory (`mem0`) with respect to the devices' placement. In contrast, the gray arrows hit the remote memory (`mem1`).

5.1 The Memory

The following experiment aims to simulate the packet reception process by issuing writes at memory locations that are slot size apart. The `memset` function is called in succession for different packet sizes (the amount of data written). The results are displayed in Figure 5.2. Note that the black solid line corresponds to writing to the local memory domain, while the gray line is for writing to locations on the remote memory domain. There is an at least 30% performance gain when writing to local memory, for the specific block sizes (Ethernet packets). The destination buffer was chosen to be large enough in order to eliminate any major caching effects.

Continuing with the memory bandwidth benchmarks, the performance of `memcpy` was evaluated using the same arrangement of slots. A packet is read from a slot and is written to another slot of another buffer; it can be thought as mirroring two buffers by only copying the useful parts of the

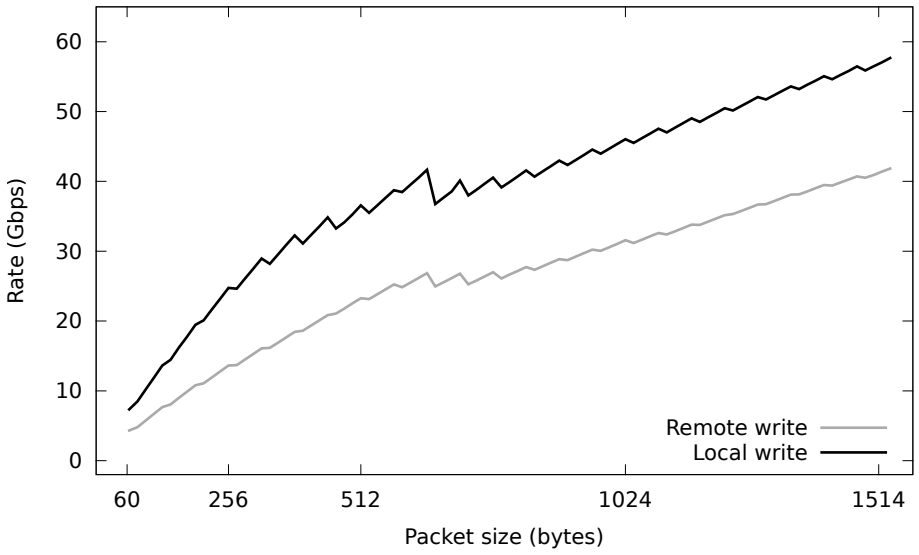

Figure 5.2: Memory write throughput for different packet sizes when the data is put in fixed sized slots. Local memory should be preferred because it always provides better performance than remote memory.

original to the copy. When both buffers belong to the same memory domain the performance is the same independently of the processor the program runs on. The solid lines in Figure 5.3 represent those cases and match exactly. The dotted lines show the performance of cross-domain copies. For the most part they deliver the same rate, except for packets smaller than 256 bytes or larger than 1024 bytes where the remote to local copy performs better than the local to remote copy.

By changing the destination addresses to be length bytes apart, the obtained access pattern closely resembles the *compaction* process (see Section 3.2). The results are displayed in Figure 5.4. For small packet sizes, processor caches benefit performance because of the sequential writes (obvious for packets ranging from 64 to 256 bytes). This happens because for a large number of writes the target pages are already in the cache. The divergence of the dotted black line from the solid black line shows the fading of the caching effect, as the packets become bigger and fewer of them fit the caches. Again, the remote to remote copy performance is the same as the local to local copy. However, the remote to local copy cannot benefit as much, and is worse from the optimal by about 3 Gbps at all times. It's still better than the local to remote copies for packets larger than 1024 bytes. The jigsaw-like effect here is caused by the specific implementation of the `memset`/`memcpy` function. Properly aligned packet sizes give better results because the whole memory hierarchy is optimally utilized.

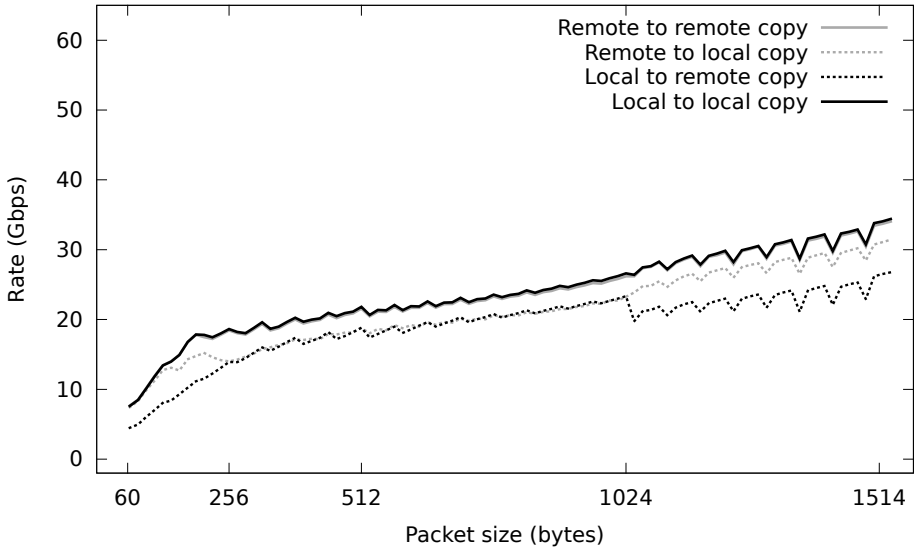

Figure 5.3: Memory copy throughput for different packet sizes when the data is put in fixed sized slots. Best performance is achieved when both source and destination are on the same domain. Otherwise it's preferable for the destination to be local.

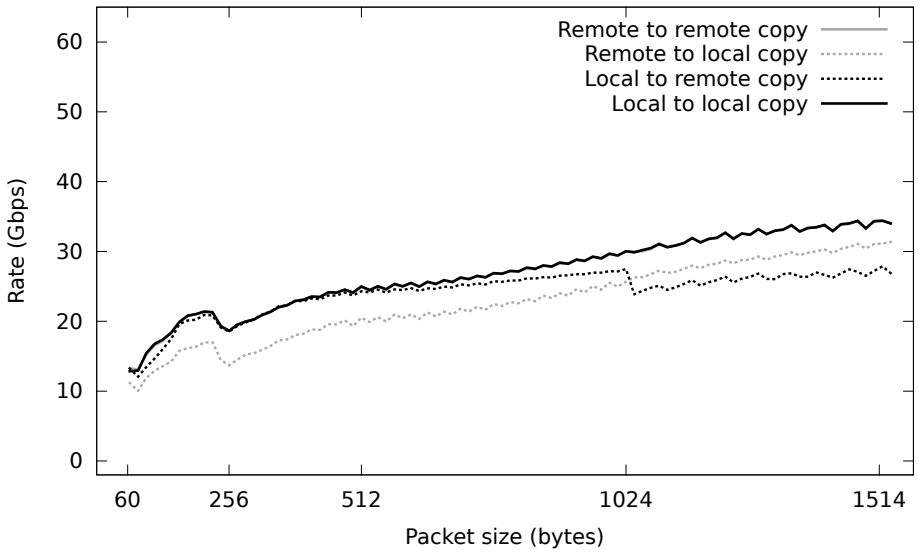

Figure 5.4: Memory copy throughput for different packet sizes when the data is read from slots but written sequentially. The pronounced boost in performance for small packets is a caching effect.

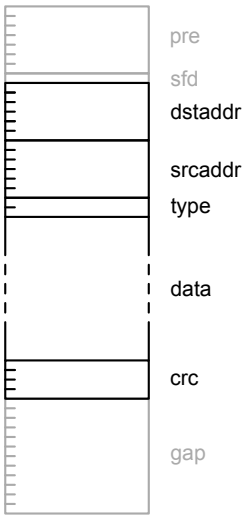

Figure 5.5: The Ethernet frame and the extra costs that are coupled with packet transmission. The preamble and interpacket gap are used for synchronization and collision detection.

5.2 The Network Adapter

An Ethernet packet, as defined by IEEE 802.3 [31], starts with a 7-octet preamble, an 1-octet start frame delimiter (SFD) field, source and destination MAC addresses of 6 octets each and a 2-octet length or type field. Then follows the client data and the packet ends with a 4-octet frame check sequence (FCS) that is actually a cyclic redundancy check (CRC) used for data verification. The minimum length of the data field of a packet sent over an Ethernet link is 46 octets for basic frames, while the maximum length is 1500 octets. An optional pad is appended after the data to meet the minimum length requirement, an artifact of the CSMA/CD protocol operation. Furthermore, for the correct transmission of packets an interpacket gap of minimum length of 96 bits is needed. Note that an Ethernet frame does not include the preamble, SFD and of course not the interpacket gap, but they are valid transmission costs. The CRC field typically belongs in the frame but is most likely stripped by the hardware or early in the operating system's stack. This way it never reaches the application so it can be excluded as well for simplicity. Figure 5.5 displays the format of the Ethernet frame.

The theoretical minimum and maximum frame rate and throughput of a saturated link can be computed as follows. The total length of transmitted bits associated with the sending of one frame would be:

$$\text{packetlen} = \text{pre} + \text{sfd} + 2 \cdot \text{addr} + \text{type} + \text{data} + \text{crc} + \text{gap}$$
$$\text{framelen} = 2 \cdot \text{addr} + \text{type} + \text{data}$$

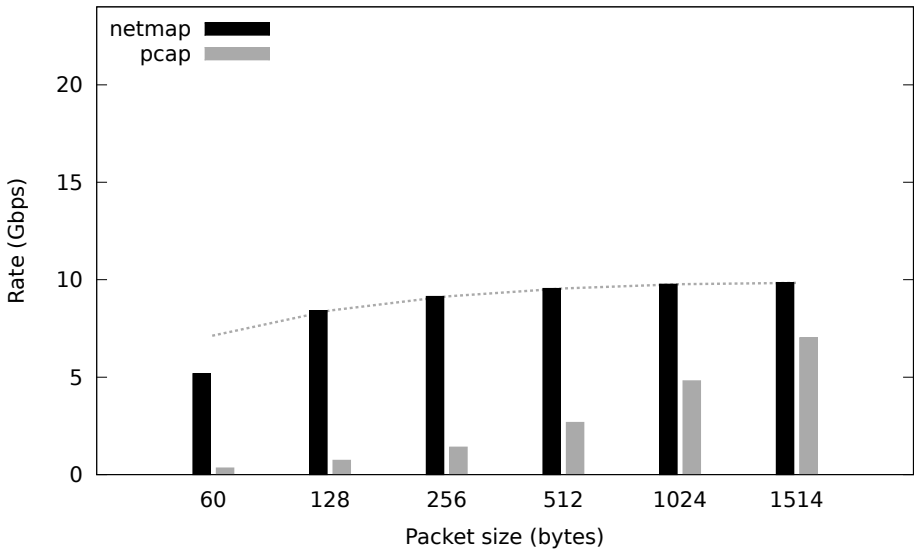

Figure 5.6: Comparison of the new packet I/O engine over the traditional approach of packet capturing. Up to 15 times speedup is obtained for small packets.

The minimum and maximum length for frames and packets in bytes are:

$$\begin{aligned}
\text{packetlen}_{\text{min}} &= 7 + 1 + 2 \cdot 6 + 2 + 46 + 4 + 12 = 84 \text{ bytes} \\
\text{packetlen}_{\text{max}} &= 7 + 1 + 2 \cdot 6 + 2 + 1500 + 4 + 12 = 1538 \text{ bytes} \\
\text{framelen}_{\text{min}} &= 2 \cdot 6 + 2 + 46 = 60 \text{ bytes} \\
\text{framelen}_{\text{max}} &= 2 \cdot 6 + 2 + 1500 = 1514 \text{ bytes}
\end{aligned}$$

For an adapter with a link speed of 10 Gbps, the maximum and minimum frame rate as well as data throughput for each direction are:

$$\begin{aligned}
\text{framerate}_{\text{max}} &= \frac{\text{speed}}{\text{packetlen}_{\text{min}}} = \frac{10 \text{ Gbps}}{84 \cdot 8} = 14.88 \text{ Mfps} \\
\text{framerate}_{\text{min}} &= \frac{\text{speed}}{\text{packetlen}_{\text{max}}} = \frac{10 \text{ Gbps}}{1538 \cdot 8} = 0.81 \text{ Mfps} \\
\text{throughput}_{\text{min}} &= \frac{\text{framelen}_{\text{min}}}{\text{packetlen}_{\text{min}}} \cdot \text{speed} = \frac{60}{84} \cdot 10 \text{ Gbps} = 7.14 \text{ Gbps} \\
\text{throughput}_{\text{max}} &= \frac{\text{framelen}_{\text{max}}}{\text{packetlen}_{\text{max}}} \cdot \text{speed} = \frac{1514}{1538} \cdot 10 \text{ Gbps} = 9.84 \text{ Gbps}
\end{aligned}$$

To demonstrate the need for the new packet I/O engine, a comparison with the straightforward approach is given in Figure 5.6. The gray bars indicate the rate achievable when using libpcap for packet capturing, while the black bars show the rates possible when using the netmap packet I/O

GPU memory	NIC memory	Packet size	Rate
local	local	max	9.32
		min	2.22
local	remote	max	9.84
		min	2.00
remote	local	max	9.84
		min	1.98
remote	remote	max	6.10
		min	2.16

Table 5.1: Selected rates (in Gbps) from different configurations of memory allocations for the GPU and the NIC. The memory is local or remote with respect to the devices. Varying the packet size shows that there is no single "best" configuration.

API and kernel/driver infrastructure. For minimal packets the reception is over 15 times faster. The ratio becomes smaller with larger packet sizes; note however that netmap saturates the interface for packet sizes above 128 bytes. The dotted gray line shows the theoretical Ethernet upper bound.

Returning to the discussion about what is the optimal configuration that gives the best performance on a NUMA system, more benchmarks were conducted. The QPI (QuickPath Interconnect) links that are shown on Figure 3.4 have a theoretical maximum throughput of 12.8 GB/s per direction while PCIe 2.0 links can go up to 8 GB/s per x16 link. Table 5.1 gives some indicative results that lead to the following observations. In the case where large packets are handled, the memory bandwidth on the single node is not enough for the specific access pattern and it's beneficial to put some extra load to the I/O interconnects to also take advantage of the remote memory bandwidth. This is done by placing the NIC's ring buffer there. However, small packets saturate the I/O interconnects due to the small and frequent transactions and cannot afford any remote accesses. Small packet workloads need all buffers on the same memory domain, and preferably the local with respect to the slots where the GPU and the NIC are placed. This is in contrast with previous work on SMP systems by Egi et al. that identified the bottleneck for small packets to be related to memory latency [14]. By physically placing the NIC on a different node than the GPU the results are more or less the same; on some cases the performance slightly degrades for large packets and marginally improves on small packets.

These deductions were also verified using the performance event monitoring infrastructure available to the Nehalem micro-architecture. [10] Using a set of model-specific performance counters a selection of parameters can be monitored and measured. For example, the number of L3 cache misses is significantly larger for minimal packets than maximal packets. This way

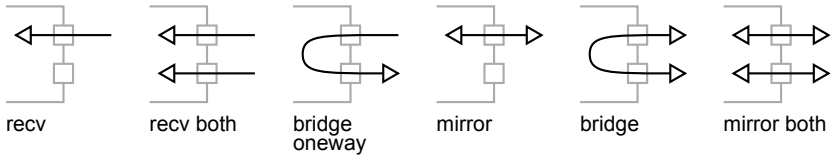

Figure 5.7: Schematic representation of the different configurations possible with two network adapter ports, ordered by performance. The ports can be part of the same physical dual port adapter or of completely separate adapters. Performance may differ depending on what the case is.

Packet size	recv	recv both	bridge oneway	mirror	bridge	mirror both
Dual port adapter						
max	100%	100%	100%	100%	95%	93%
min	71%	45%	43%	33%	32%	31%
Separate adapters						
max	100%	100%	100%	100%	100%	96%
min	71%	70%	44%	33%	32%	31%

Table 5.2: Performance results for different setups of network adapter ports in percentages of the calculated optimum rate. There is a benefit in using two physically separate NICs over a dual port NIC in some cases.

much more load is put to the memory controller and some resource in the subsystem is saturated leading to sub-optimal performance.

Dual port cards can receive at wire rate when maximum packets are transmitted at both interfaces simultaneously. However, with minimal packets, wire rate reception is possible only when one of the two interfaces is active. With both interfaces receiving small packets concurrently, 63% of the sum of their measured individual performance is utilized; that is about 45% of the network adapter's expected performance based on it's advertised capabilities. Similar behaviour is observed by Manesh et al. in [25]. The various configurations evaluated on dual port NICs are illustrated in Figure 5.7; the same tests were done for ports of separate adapters, with both interface cards carefully placed on the same node and memory domain. Detailed performance results for all cases are presented in Table 5.2. The percentages are obtained considering the calculated rates for minimum and maximum packets presented above. Notable results include the case where two ports are bridged (incoming packets in one port are sent on the other and vice versa) and maximal packets are passed through; the system can do the bridging in wire rate only with separate adapters, otherwise it can only reach 95% of the top rate.

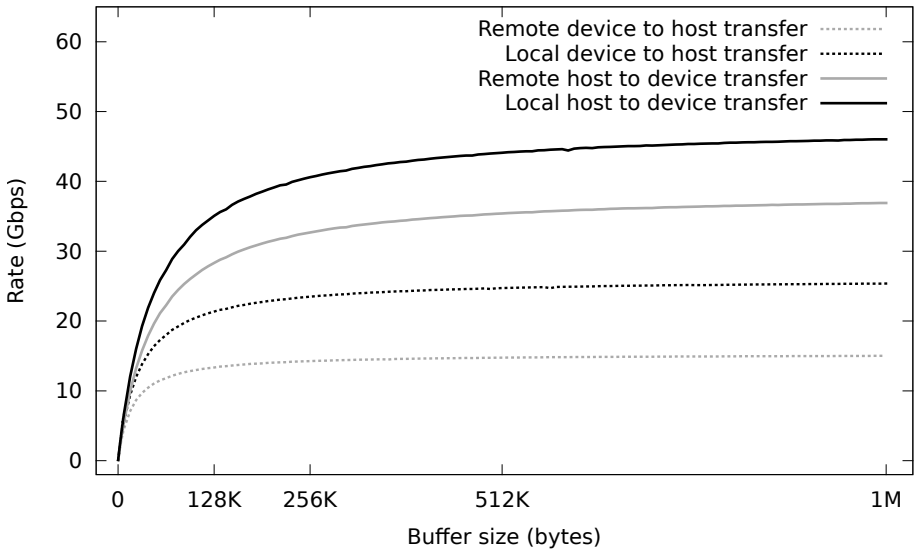

Figure 5.8: The effect of buffer size on the host to device transfer throughput. Choosing a small buffer size can significantly impact the data transfer performance.

The transmission of small packets produces a large amount of interrupts, even with interrupt mitigation techniques enabled. After experimentation with interrupt balancing and static interrupt masks, it is found that it is better to have interrupts serviced at cores of the same processor node as the running process. Specifically, setting the interrupt mask to a sibling core, marginally, gives the best performance. All setups involving real network interfaces follow this rule to avoid complicated problems like those discussed in Section 3.3.

5.3 The Graphics Processor

The following experiment helps understand and quantify the overheads involved in data transfers between the graphics processing device and the host memory. The transfer rate was measured by repeating each transfer many times and using different buffer sizes. The transfers are preformed by the DMA engine of the device and thus no caching effects should be observed. Figure 5.8 displays the results for all possible configurations of transfer direction and memory domain. About 10 Gbps of the throughput is lost in remote transfers for both host to device and device to host transfers. An important point made by this benchmark is that transferring small buffers, such as a single packet, over to the GPU is an overkill and packets must be batched in the order of megabytes for adequate performance. Furthermore,

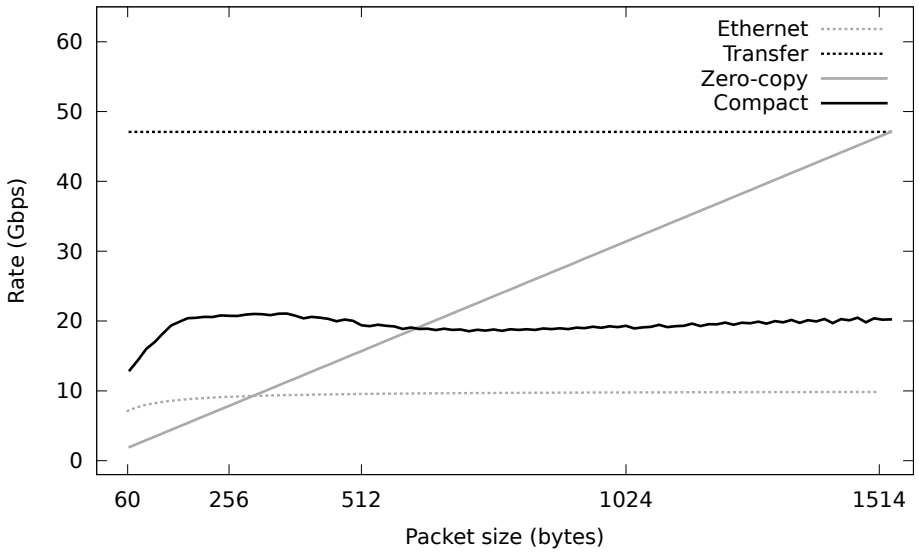

Figure 5.9: The effect of packet size on host to device transfers when the data is put in fixed sized slots. It's often faster to spend some time to compact the data before transferring than using the whole buffer together with unwanted bits.

the current system displays asymmetric behaviour when it comes to the direction of the transfer. Host to device transfers produce a rate of 46 Gbps whereas device to host transfers only reach 25 Gbps, for large buffers on the local domain. This is thought by other researchers to be an issue with multi-socket motherboards [16] and it has been verified that other systems with different setups do not display the problem. Specifically, a GeForce GT 640 on an AMD FX-6100 Six-Core processor system has a slightly larger device to host throughput.

The next benchmark is focused on the compaction of packet buffers. There is the scenario where the whole buffer (6 Mb in size) is transferred over to the device, unfortunately, along with the holes at the tail of packet slots (zero-copy mode). There is also the approach to compact the packets, which produces fully dense buffers. Figure 5.9 displays the results of transferring compacted buffers with the solid black line, and the "goodput" of the zero-copy method with the solid gray line. Also, for comparison, the dotted black line shows the maximum rate achievable with host to device transfers, whereas the dotted gray line shows the theoretical maximum 10 Gb Ethernet rate for the given packet size. Using a slot size of 1536 bytes, it's impossible for the zero-copy mode to handle packets smaller than 300 bytes at wire rate. However, using compaction there are enough resources to keep-up with such rates. Actually, there is a certain point at which the zero-copy mode

outperforms the compaction-first mode; at this graph this appears to be around 600-byte packets, but in reality this point may be moved further away because of other parameters, such as device to host transfers, memory bandwidth, interconnect available bandwidth, and so on.

5.4 Overall Performance

Results of the complete system testing are presented here. Stressing the system with synthetic data produced by the modified netmap packet generator are shown first. Figure 5.10 displays performance of handling traffic using one network adapter. The black bars correspond to passively receiving and processing the packets, whereas the gray bars show how the system responds to the extra effort of forwarding all packets. Some degradation is observed for small packets when forwarding, but for packet sizes larger than about 500 bytes, full speed is achieved too. Note that these measurements apply to the cases where packet data is not rewritten, so that no buffers need to get back to the host from the device. The results are cut down to simple flags, and the original packet buffers are re-used for transmission.

Similar results for two network adapters used concurrently are shown in Figure 5.11. The aggregate rate does not double when compared to using one adapter for all cases; this is because of the memory bandwidth limitations posed by the current system. However, some "ideal" cases, such as with 1024-byte packets, almost achieve wire rate. Packet forwarding uses even more of the available memory bandwidth and the system never reaches wire rate in this mode, but performs at about the 3/4 of the passive mode. The final data point for the receive mode shows a small fall because the irregularity of 1514-byte packets becomes important at those high rates. This is related to the efficiency of the memory copy implementation using specific buffer lengths. Again, buffers are not synchronized; this is a key optimization as it avoids redundant data transfers on the slower device to host path.

Figure 5.12 compares the compaction-first approach with the zero-copy method. Black bars are for the former while gray bars indicate the zero-copy mode performance. The zero-copy approach is bound to perform worse for small packets, because of the holes introduced by the packet placement in slots. Also, because the buffer swapping tricks are performed on the device, no copies of the packet buffers are kept at the host, and device to host transfers are needed. The current system suffers from the asymmetries in device transfers discussed in Section 5.3, so the full potential of the zero-copy mode cannot be realized. For this reasons, in order to do a fair comparison, all measurements were taken by using buffer synchronization after processing. This degrades performance in general, but shows that for very large packets the zero-copy mode can outperform the normal mode (only visible for 1514-byte bars).

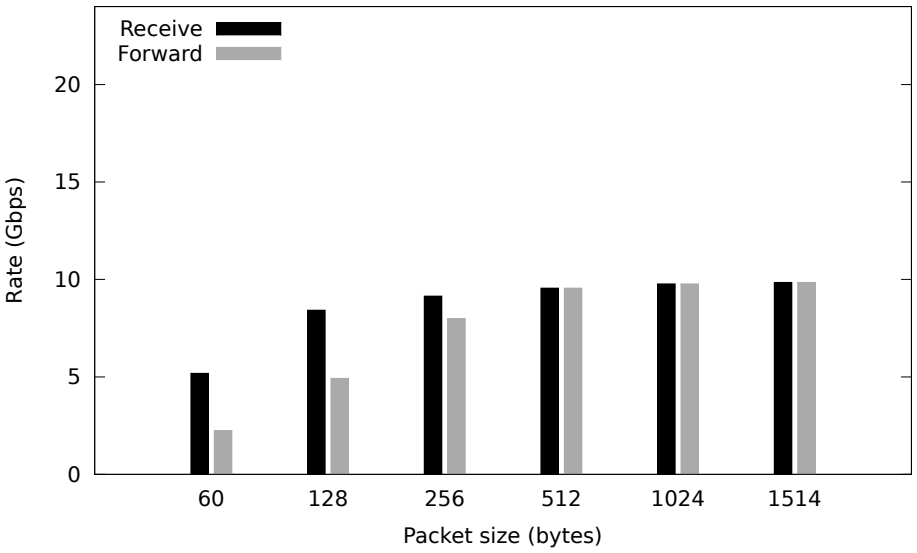

Figure 5.10: The throughput of packet reception and forwarding for various packet sizes using one network adapter. Buffers are not synchronized after processing.

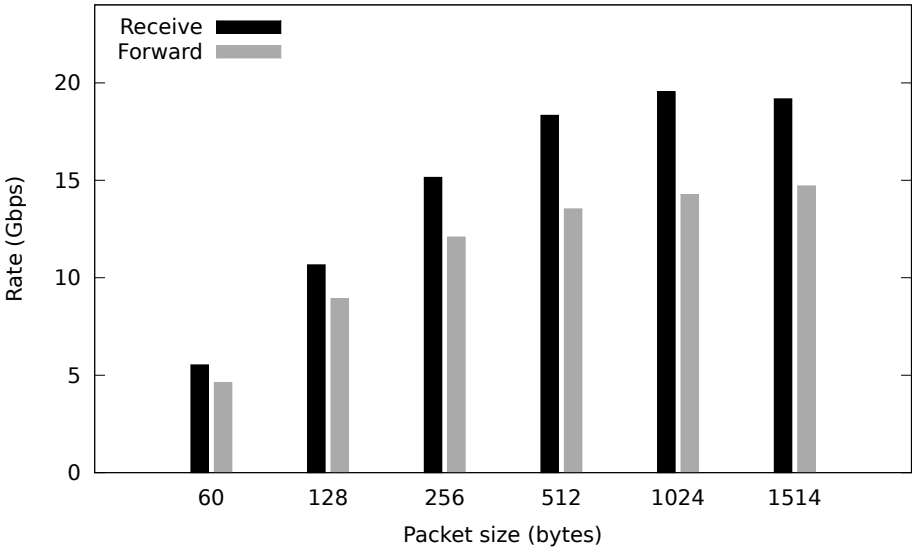

Figure 5.11: Aggregate throughput of packet reception and forwarding for various packet sizes using two network adapters. Buffers are not synchronized after processing.

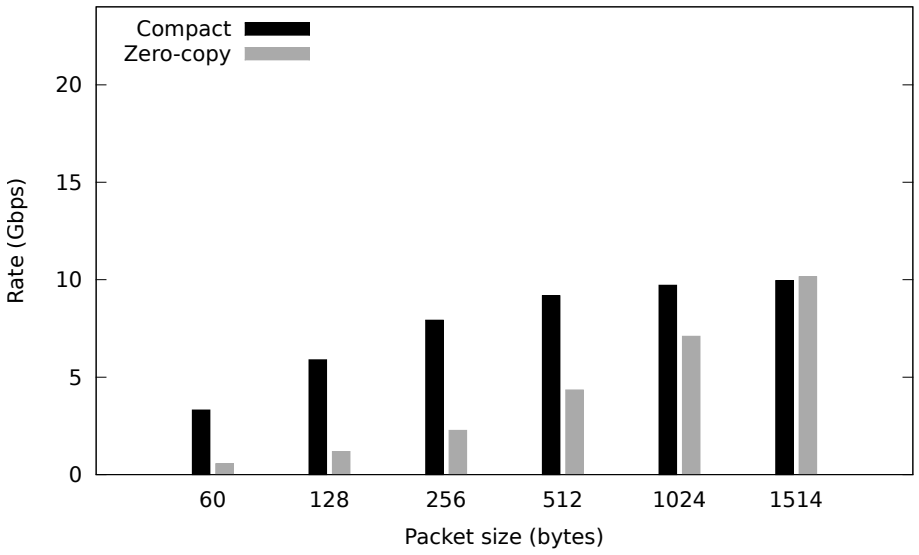

Figure 5.12: Comparison of the zero-copy technique to the normal compaction-first approach. Aggregate throughput of packet reception is shown. Buffers are synchronized after processing.

Although synthetic data is good for system testing and development, evaluation with real packet traces provides important information about the system's behaviour in real world scenarios. For this purpose, recent sample traces were selected from the repositories of CAIDA [2] (The Cooperative Association for Internet Data Analysis) and the MAWI [1] (Measurement and Analysis on the WIDE Internet) Working Group. Each trace is replayed using the packet generator. However, because the traces only contain the headers of each packet, whatever data is present in the slot buffers is used for the payload. This way, most of the properties of the trace remain, including the length distribution that is important for the evaluation.

Results for most system configurations are presented in Table 5.3. Operation is either passive ("receive") or active ("forward") running on one network interface or two interfaces concurrently. The mode corresponds to how data movement is performed and is either using compaction or with the zero-copy solution. The leftmost results column (full) is for the case where buffers are synchronized after processing, whereas the rightmost column (optimized) is for the case where they are not. The latter means that buffers are transferred to the device but not synchronized back with the host copy; the original buffers are re-used (if needed) according to the packet metadata. The metadata is always part of the result in all cases. Full synchronization always results in lower rates due to the bottleneck introduced by the slower device to host transfers. The zero-copy mode gives worse performance than

Operation	Interfaces	Mode	Full	Optimized
CAIDA trace				
Receive	One	Compact	9.65	9.65
		Zero-copy	5.52	9.65
	Two	Compact	9.07	18.01
		Zero-copy	5.13	12.37
Forward	One	Compact	8.84	9.65
		Zero-copy	5.54	n/a
	Two	Compact	8.29	13.67
		Zero-copy	4.65	n/a
MAWI trace				
Receive	One	Compact	9.68	9.68
		Zero-copy	5.95	9.68
	Two	Compact	9.09	17.72
		Zero-copy	5.57	13.04
Forward	One	Compact	8.94	9.68
		Zero-copy	5.88	n/a
	Two	Compact	8.38	13.58
		Zero-copy	5.22	n/a

Table 5.3: Results using real traffic from recent packet traces from CAIDA and MAWI. Rates in Gbps are displayed for all combinations of system operation, number of interfaces, buffering mode and optimizations used.

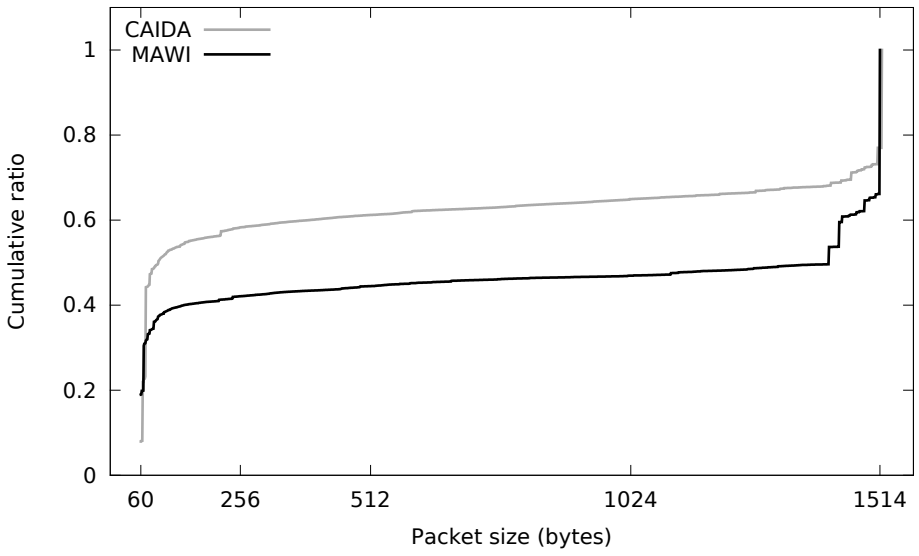

Figure 5.13: Cumulative packet size distribution in traffic traces. Although large packets account for the bulk of the total bytes transmitted, small packets are still significant if not dominant.

		Batch size				
Operation	Packet size	512	1024	2048	4096	8192
Receive	max	0.629	1.259	2.519	5.039	10.079
	min	0.047	0.094	0.189	0.379	0.758
Forward	max	1.259	2.519	5.039	10.079	20.158
	min	0.094	0.189	0.379	0.758	1.517

Table 5.4: Sample average latency (in milliseconds) for various batch sizes (number of packets) assuming minimum or maximum packet sizes. This gives an estimation of the latency range for a given batch size.

the compact mode because of the excessive data transfer sizes. The traces still contain a significant amount of small packets, as is shown in Figure 5.13, thus the system cannot operate in wire rate in zero-copy mode, except for the case of optimized synchronization running on only one interface. The CAIDA trace contains roughly 20% more small packets than the MAWI trace and that is why the system performs slightly better in zero-copy with the latter. Please note that the zero-copy mode is incompatible with the optimized version when forwarding packets, simply because there exist no intermediate buffers to act as reusable queues.

Latency is affected both by the packet size distribution and the selected packet batch size used for processing. In passive mode the average latency corresponds to the time from when a packet is received till a result for the same packet is obtained. This corresponds to one time slot in the pipeline of Figure 3.2 because the processing stage operates on a batch of packets. On the other hand, in forwarding mode the average latency is computed as the time needed for a packet to be received, processed and written to an output interface. These two are different definitions of latency, but they are considered the most appropriate. Selected results for various batch sizes are displayed in Table 5.4. Small batch sizes result in lower latencies, however too small batch sizes will make inefficient use of the parallel capabilities of the graphics processor, and negatively affect the overall performance. Using larger batch sizes achieves better utilization at the cost of the increased latency. The batch size is an important configuration parameter that bounds latency on saturated links. Additionally, for real-time applications —for which latency is critical— a timeout can be configured to trigger the processing of the incoming packets instead of waiting for buffers to fill up over a specified threshold.

Chapter 6

Conclusions

This thesis presented the design, implementation and evaluation of a programmable and high-performance system for developing network traffic processing applications. The design space of combining the massively parallel architecture of graphics processing units with 10 Gigabit Ethernet network interfaces is explored. This includes the experimentation with optimization techniques such as buffer swapping, redundant transfer elimination, zero-copy data movement, packet buffer compaction. Furthermore, the configuration of various parameters such as buffer placement in non uniform memory architectures, process scheduling and synchronization has been investigated. The design of simple yet efficient data structures and the programming practices followed is also an important part of the system's implementation.

The overall system performance was evaluated not only with synthetic data, but also with real traffic from public packet trace sources. The system achieves wire-rate or near wire-rate processing speeds in the best configuration, for the typical packet size distributions obtained from real-world workloads. Latency is bound to increase, but is kept at acceptable levels, in the order of milliseconds. Specifically, the system reaches 9.5 Gbps with one network interface and 18 Gbps with two interfaces in passive mode, while forwarding mode achieves rates of 9.5 Gbps and 13.5 Gbps with one and two interfaces respectively.

6.1 Future Work

Many of the challenges faced in this work come from the need for packet buffers to be partitioned in slots. The rise of this requirement, or a workaround for better placement of packets on the ring buffers is part of future work. A plausible scenario is to reuse the remaining space at the end of the slots to store more packets using a prediction scheme. However, this will increase the complexity of packet descriptor management and possibly latency in the network driver.

Another part of the system that can take improvement is the packet buffer placement on NUMA systems. Currently, the configuration is manual, and there are significant extra costs involved in "wrong" placements. The detection of the node where each component physically resides could be automated, so that the rest of the resources be allocated in an optimal way.

Caching effects should be analysed in more detail and possibly optimized. Caching does improve the compaction of small packets, however the zero-copy mode does not actually touch the data on the host. The following question needs investigation: "Could disabling or bypassing the processor caches benefit the performance of the zero-copy mode?"

Although the system uses DMA for data movement between the NIC and the GPU, all memory copies used in compaction are done using the normal synchronous API, which may hurt performance in specific cases. Alternative approaches for memory copies in main memory have been proposed, such as the asynchronous `memcpy` of Vaidyanathan et al. that uses Intel's I/OAT (I/O Acceleration Technology). [39] Evaluation of such methods has not been done yet.

Future work will also deal with the integration of the current system with virtualization technologies. The virtualization of the GPUs and NICs is already there, however the abstraction of a virtual packet processing engine/accelerator may be desired.

Moreover, there is a plan to upgrade the current capacity to 40 Gigabit, using the same hardware architecture. By utilizing the second node and memory domain in a symmetric manner the system will easily scale to something close to 40 Gbps.

Finally, it would be very useful to include long term costs in order to have a complete economic analysis of this solution. The hardware is inexpensive, but its energy consumption and maintenance costs are also of interest. The logical next step of this work will provide such measurements.

Bibliography

[1] MAWI Working Group Traffic Archive – 2012-09-26, September 2012. `http://mawi.wide.ad.jp/mawi/`.

[2] The CAIDA UCSD Anonymized Internet Traces 2012 – 2012-08-16, August 2012. `http://www.caida.org/data/passive/passive_2012_dataset.xml`.

[3] Ulisses Alonso Camaro and Johann Baudy. Linux packet mmap. In Linux source code at `Documentation/networking/packet_mmap.txt`, 2009.

[4] Pramod Bhatotia and Rodrigo Rodrigues. Shredder: GPU-Accelerated Incremental Storage and Computation. In *Proccedings of the 10th USENIX Conference on File and Storage Technologies*, pages 171–185. USENIX FAST, San Jose, CA, USA, February 2012.

[5] Nick Black and Jason Rodzik. My Other Computer is your GPU: System-Centric CUDA Threat Modeling with CUBAR. Retrieved from `http://nick-black.com/dankwiki/index.php/CUBAR`, 2010.

[6] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A Case for NUMA-aware Contention Management on Multicore Systems. In *Proceedings of the 2011 USENIX Annual Technical Conference*, pages 1–15. USENIX ATC, Portland, OR, USA, June 2011.

[7] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, 2004.

[8] Debra Cook, John Ioannidis, Angelos Keromytis, and Jake Luck. CryptoGraphics: Secret Key Cryptography Using Graphics Cards. In *Topics in Cryptology - The Cryptographers' Track at the RSA Conference 2005*, pages 334–350. CT-RSA, San Francisco, CA, USA, February 2005.

[9] Intel Corporation. *Intel 82599 10 GbE Controller Datasheet, Revision 2.0*, July 2009.

[10] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*, May 2012.

[11] Luca Deri. Improving Passive Packet Capture: Beyond Device Polling. In *Proceedings of the 4th International System Administration and Network Engineering Conference*. SANE, Amsterdam, The Netherlands, October 2004.

[12] Luca Deri. nCap: Wire-speed Packet Capture and Transmission. In *Proceedings of the 3rd IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services*, pages 47–55. IEEE E2EMON, Nice, France, May 2005.

[13] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 15–28. ACM SOSP, Big Sky, Montana, USA, October 2009.

[14] Norbert Egi, Adam Greenhalgh, Mark Handley, Mickael Hoerdt, Felipe Huici, and Laurent Mathy. Towards High Performance Virtual Routers on Commodity Hardware. In *Proceedings of the 2008 ACM Conference on Emerging Network Experiment and Technology*. ACM CoNEXT, Madrid, Spain, December 2008.

[15] Khronos OpenCL Working Group. *The OpenCL Specification, Version 1.2*, November 2011.

[16] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-accelerated software router. In *Proceedings of SIGCOMM 2010*, pages 195–206. ACM SIGCOMM, New Delhi, India, September 2010.

[17] Owen Harrison and John Waldron. Practical Symmetric Key Cryptography on Modern Graphics Hardware. In *Proceedings of the 17th USENIX Security Symposium*, pages 195–209. USENIX Security, San Jose, CA, USA, August 2008.

[18] Muhammad Jamshed, Jihyung Lee, Sangwoo Moon, Insu Yun, Deokjin Kim, Sungryoul Lee, Yung Yi, and KyoungSoo Park. Kargus: a Highly-scalable Software-based Intrusion Detection System. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*. ACM CCS, Raleigh, NC, USA, October 2012.

[19] Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon, and KyoungSoo Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, pages 1–14. USENIX NSDI, Boston, MA, USA, March 2011.

[20] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: First-Class GPU Resource Management in the Operating System. In *Proceedings of the 2012 USENIX Annual Technical Conference*, pages 401–412. USENIX ATC, Boston, MA, USA, June 2012.

[21] Joongi Kim, Seonggu Huh, Keon Jang, KyoungSoo Park, and Sue Moon. The Power of Batching in the Click Modular Router. In *Proceedings of the 3rd ACM SIGOPS Asia-Pacific Workshop on Systems*. ACM SIGOPS APSys, Seoul, South Korea, July 2012.

[22] Yongchao Liu, Douglas Maskell, and Bertil Schmidt. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes*, 2:73, 2009.

[23] Yongchao Liu, Bertil Schmidt, and Douglas Maskell. CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows-Wheeler transform. *Bioinformatics*, 28(14):1830–1837, 2012.

[24] Svetlin Manavski and Giorgio Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC bioinformatics*, 9(Suppl 2):S10, 2008.

[25] Maziar Manesh, Katerina Argyraki, Mihai Dobrescu, Norbert Egi, Kevin Fall, Gianluca Iannaccone, Eddie Kohler, and Sylvia Ratnasamy. Evaluating the Suitability of Server Network Cards for Software Routers. In *Proceedings of the ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of Tomorrow*. ACM SIGCOMM PRESTO, Philadelphia, USA, November 2010.

[26] Jeffrey Mogul and Kadangode Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[27] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click modular router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 217–231. ACM SOSP, Kiawah Island Resort, SC, USA, December 1999.

[28] NVIDIA. *CUDA C Best Practices Guide, Version 4.1*, January 2012.

[29] NVIDIA. *CUDA C Programming Guide, Version 4.2*, April 2012.

[30] NVIDIA. *OpenCL Programming Guide for the CUDA Architecture, Version 4.2*, March 2012.

[31] LAN/MAN Standards Committee of the IEEE Computer Society. IEEE Std 802.3-2008: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications, September 2008.

[32] John Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron Lefohn, and Timothy Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113.

[33] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Annual Technical Conference*, pages 101–112. USENIX ATC, Boston, MA, USA, June 2012.

[34] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: operating system abstractions to manage GPUs as compute devices. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 233–248. ACM SOSP, Cascais, Portugal, October 2011.

[35] Jamal Hadi Salim and Robert Olsson. Beyond Softnet. In *Proceedings of the 5th Annual Linux Showcase & Conference*, pages 165–172. USENIX ALS, Oakland, CA, USA, November 2001.

[36] Hassan Shojania, Baochun Li, and Xin Wang. Nuclei: GPU-Accelerated Many-Core Network Coding. In *Proceedings of the 28th IEEE International Conference on Computer Communications*, pages 459–467. IEEE INFOCOM, Rio de Janeiro, Brazil, April 2009.

[37] Randy Smith, Neelam Goyal, Justin Ormont, Karthikeyan Sankaralingam, and Cristian Estan. Evaluating GPUs for Network Packet Signature Matching. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 175–184. ISPASS, Boston, MA, USA, April 2009.

[38] Weibin Sun, Robert Ricci, and Matthew Curry. GPUstore: Harnessing GPU Computing for Storage Systems in the OS Kernel. In *Proceedings of the 5th Annual International Systems and Storage Conference*. ACM SYSTOR, Haifa, Israel, June 2012.

[39] Karthikeyan Vaidyanathan, Lei Chai, Wei Huang, and Dhabaleswar K. Panda. Efficient asynchronous memory copy operations on multi-core systems and I/OAT. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 159–168. IEEE CLUSTER, Austin, Texas, USA, September 2007.

[40] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, pages 116–134. RAID, Cambridge, MA, USA, September 2008.

[41] Giorgos Vasiliadis and Sotiris Ioannidis. GrAVity: A massively parallel antivirus engine. In *Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection*, pages 79–96. RAID, Ottawa, Canada, September 2010.

[42] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. MIDeA: a multi-parallel intrusion detection architecture. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 297–308. ACM CCS, Chicago, Illinois, USA, October 2011.