A High-level Architecture for Efficient Packet Trace Analysis on GPU Co-processors

Alastair Nottingham
Security and Networks Research Group
Department of Computer Science
Rhodes University
Grahamstown, South Africa
Email: anottingham@gmail.com

Barry Irwin
Security and Networks Research Group
Department of Computer Science
Rhodes University
Grahamstown, South Africa
Email: b.irwin@ru.ac.za

***Abstract*—This paper proposes a high-level architecture to support efficient, massively parallel packet classification, filtering and analysis using commodity Graphics Processing Unit (GPU) hardware. The proposed architecture aims to provide a flexible and efficient parallel packet processing and analysis framework, supporting complex programmable filtering, data mining operations, statistical analysis functions and traffic visualisation, with minimal CPU overhead. In particular, this framework aims to provide a robust set of high-speed analysis functionality, in order to dramatically reduce the time required to process and analyse extremely large network traces. This architecture derives from initial research, which has shown GPU co-processors to be effective in accelerating packet classification to up to tera-bit speeds with minimal CPU overhead, far exceeding the bandwidth capacity between standard long term storage and the GPU device. This paper provides a high-level overview of the proposed architecture and its primary components, motivated by the results of prior research in the field.**

***Index Terms*—Data mining, Computer networks, Information security, GPGPU.**

I. Introduction

Network packet traces (also referred to as captures) are repositories for large amounts of useful technical, forensic and statistical information, as they contain a record of all network activity intercepted over an arbitrary period of time at particular network interface. Traces collected over long intervals (spanning months or years) provide a historical record of traffic dynamics, incidents and malicious activity [1]. In addition, long term packet traces are used to store data collected at network telescopes in order to study Internet Background Radiation (IBR) [2], [3], such as the propagation and impact of large scale internet events and worm outbreaks [1]. Unfortunately large traces containing hundreds of millions or billions of packets are computationally expensive and extremely time consuming to process, which makes identifying interesting data tedious, and exploration extremely difficult. To address this, prior research has investigated the use of Graphics Processing Unit (GPU) co-processors to accelerate classification, in order to simplify and accelerate packet classification. This research determined that modern commodity GPUs provide a wealth of a spare massively-parallel processing power which, if used appropriately, can be applied to dramatically improve filtering performance [4].

This paper presents an overview of a GPU based architecture for accelerating and simplifying the analysis of large trace files, based largely on findings and insights gained from prior research. The primary focus of this overview are the four main components used in capture processing: the program compiler, which generates encoded filtering instructions; the capture manager, which marshals data efficiently from long term storage; the GPU classification virtual machine, which executes encoded filter programs and harvests packet data; and the capture visualiser, which provides high-level capture information to the user. As this paper focuses on high-level, abstract architecture, discussion relating to the technical implementation of components is limited. Significant details relating to the actual implementation of many components may however be found in other component-specific publications, which are referenced in text when available.

This paper is structured as follows. Section 2 introduces the domain of packet analysis, describes prior research, and details related work in the field. Section 3 provides an overview of the larger GPU accelerated packet analysis framework, and briefly describes its four primary components. These components include the capture manager, packet filtering virtual machine, program compiler and trace visualiser. Section 4 briefly considers some wider applications of the framework, while Section 5 concludes with a summary.

II. Background

In the context of computational networks, packets are structured units of data which facilitate all remote communication. A packet comprises a set of ordered, protocol-specific transmission directives (called protocol headers), followed by a segment of application-specific transmission data, which together facilitate communication between remote hosts. Each successive protocol header is contained within the payload section of the previous protocol header, with most packets comprising several protocol layers. Protocol headers vary considerably in terms of structure, complexity and variability. While some protocols, such as the User Datagram Protocol (UDP), contain only a few statically defined and predictable field locations, other protocols (such as TCP) include optional and variable length fields which require parsing and evaluating

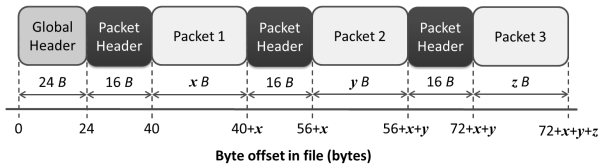


Figure 1. The structure of a PCAP packet trace.

earlier fields in the protocol header to identify. The volume of data contained in a particular packet depends on its purpose, and may vary greatly from simple host name resolution requests to large chunks of multimedia data. As a result, packet lengths may vary significantly, ranging from tens to thousands of bytes.

A. Packet Traces

Packet traces are static records of otherwise transient packet transmissions, typically captured at a specific live network interface over a set period of time. The most common open format is the *pcap* dumpfile format [5], which originated as part of the libpcap UN*X library. The pcap capture format stores packets in a linked list configuration, which supports arbitrary packet sizes without the need padding, and allows for new packets to be appended to the file on the fly, without needing to add or change existing records. The capture file structure [6], illustrated in Figure 1, begins with a 24 byte Global Header detailing information specific to the entire capture, which is followed by a list of packet records, comprising a 16 byte Packet Header followed by the raw packet byte array.

B. Prior Research

The architecture presented in this paper extends and expands upon prior research into efficient, massively-parallel packet header classification on GPU hardware. This research resulted in a prototype general packet classification engine called GPF, designed specifically for efficient GPU-based execution. The prototype algorithm was comprised of two primary execution phases, partitioned into separate CUDA (Compute Unified Device Architecture) kernel functions [7]. Together these provided a memory efficient method of processing multiple statically defined packet header based predicates concurrently. A high-level overview of the architecture of GPF is provided in Figure 2.

GPF filter programs are written in a simple high-level Domain Specific Language (DSL) [8] for specifying filter predicates, which are compiled to two or three highly-optimised low-level integer based instruction arrays. The first integer array encodes all the packet data comparison operations required to evaluate all filter predicates, while the remaining arrays encode predicate expressions. In the first phase of execution, the first integer array is used to sequentially evaluate all comparisons to packet data for all specified filters, caching chunks of data locally to reduce global memory overhead and storing results in coalesced arrays in device global memory. During the second phase, these results are used in combination with

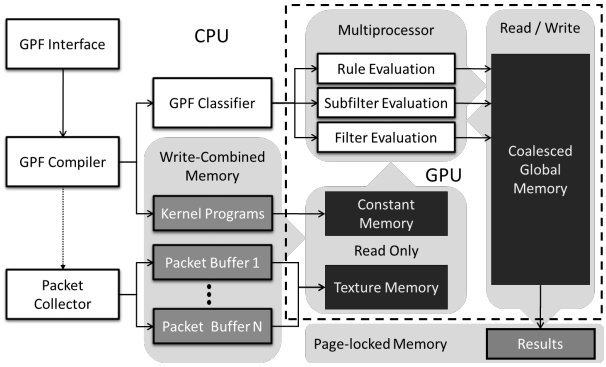


Figure 2. Overview of the architecture of the GPF prototype classifier.

the remaining integer arrays to evaluate all filter predicates and sub-predicates, generating individual arrays of boolean results for every specified filter. The primary benefit of this approach is that it avoids thread divergence by supplying all threads with the same set of instructions, thereby minimising thread divergence while simultaneously providing native multi-match filtering support.

This classification approach, supported by a range of support processes and data structures, ultimately produced promising results which significantly improved upon processing times of comparative CPU classifiers. This research is discussed in detail in [4], and includes a complete description of the construction of the classification algorithm and its associated DSLs. Subsequent research focused on the construction of CaptureFoundry, a capture analysis, manipulation and filtering tool which utilised an enhanced high register pressure [9] version of the classification kernels, an OpenTK based visualisation component, and highly optimised threaded supporting architecture to simplify and accelerate the processing of large packet trace files. This research is described in [10].

C. Related Research

The domains of packet classification and GPGPU (General Processing on GPUs) are both active research areas, and encompass a broad range of diverse research. Literature relating to GPU accelerated packet classification is relatively scarce however, seemingly due, in part, to a perceived high level of incompatibility between traditional packet classification algorithms and GPU hardware design, and and strong trend towards fast IP classification algorithms [11], [12], [13]over more flexible protocol-independent classification [4], [14], [15]. The most closely related significant publication describes Gnort, a GPU implementation of the Snort intrusion detection algorithm [16], was the first GPU accelerated application to accelerate packet analysis operations. Gnort used a fast parallel string matching algorithm to process packet payloads and identify threats using the Snort rule set. Gnort did not provide any GPU accelerated packet classification functions, however, and relied on PCAPs significantly slower CPU classification algorithm to perform pre-filtering, ultimately bottlenecking its performance.

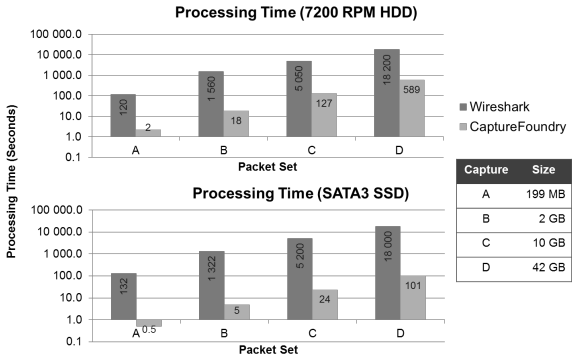


Figure 3. Comparative performance of Wireshark and CaptureFoundry analysing various traces on HDD and SSD mediums.

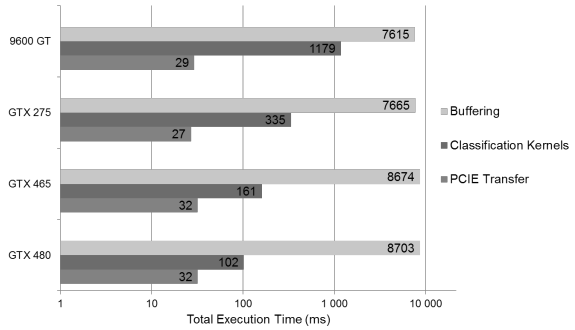


Figure 4. Total number of milliseconds spent buffering, transferring and classifying 10^7 packets against several IP-based filters on various GPUs.

III. Architecture Overview

This section introduces and motivates the proposed GPU based packet processing architecture, and provides an overview of its primary system components and how they relate to each other, in order to supply context for the remaining sections which focus on these components individually. This architecture is intended to facilitate rapid detailed analysis and exploration of extremely large packet trace files spanning hundreds of gigabytes, terabytes, or more, while consuming as few host resources as possible.

A. Architecture Motivation

The architecture described in this section was heavily influenced by the results of the prototype described in Section II-B which illuminated numerous potential opportunities for improvement and optimisation, and highlighted several areas of concern. Many of these observations may be seen in the performance results of the application, particularly when these results are compared with that of Wireshark, which uses pcap for packet classification, across different hard drives. To this end, Figure 3 shows the comparative performance of CaptureFoundry, utilising the prototype filtering architecture, and Wireshark when reading traces from both a Hard Disk Drive (HDD) and a Solid State Drive (SSD) over a SATA3 interface. In addition, Figure 4 shows the length of time spent by the prototype algorithm buffering, transferring and filtering packet capture data.

The performance results in Figure 3 show that while the while Wireshark maintains roughly equivalent performance across both HDDs and SSDs, CaptureFoundry's performance scaled proportionately with respect to the speed of the storage medium utilised. This implies that while Wireshark's classification performance is processor-bound, the performance of the prototype is bandwidth bound. When considered in combination with Figure 4, which shows that prototype filtering engine is capable of processing packet data almost two orders of magnitude faster than the trace reading process is capable of providing it. Additionally, it shows that the transfer process between the host and the GPU is significantly faster than the classification process.

With respect to the GPU classification process this is a highly promising result, as it indicates that the GPU is being under-utilised and left in an idle state for the majority of execution time. As buffering and classification are performed concurrently, the idle time of the GPU classification process can be used to perform more complex analysis of packet data by increasing the complexity of the classification algorithm without sacrificing overall performance. This is of particular significance, as the prototype classification engine is limited in a number of ways. For instance, it is not capable of adjusting classification execution based on data contained within a packet, which makes processing optional or variable length fields accurately and efficiently untenable. Furthermore, packet data cannot be collected, processed, or output to the user, which limits the potential versatility of classification engine. Unfortunately, Figure 4 also indicates a significant trace processing performance bottleneck, as classification performance is bound by how fast packet data can be read from long term storage. Given that the relative processing speed of GPU co-processors is improving at a significantly faster rate than that of long-term storage medium read speeds, and the limited capacity of fast SSD media preventing their use for terabyte sized captures, it is important to find a way mitigate this bottleneck in some way.

The other significant areas of concern with respect to the prototype are the difficulty inherent in programming useful filters using the prototypes simple Domain Specific Language, which requires significant knowledge of header protocols to use effectively, and the lack of a means to impart the results of filtering to the user in a meaningful and understandable way. In order to improve the power, flexibility and speed of the GPU classification system, these observations were used to refine and expand on the prototype architecture. The resultant architectural design is provided in the following section.

B. GPU Platform

The Nvidia CUDA platform is used to support the GPU processing functions used in this architecture. The motivation for this is relatively simple; the development of the prototype classifier, from which this architecture is primarily derived,

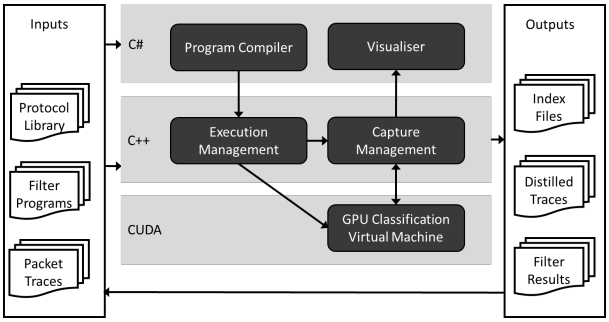


Figure 5. Primary architecture components

began when OpenCL was in its infancy, and suffered from limited functionality and hardware support. The CUDA framework, in contrast, was well into its second major iteration; it was more robust and functionally diverse, and provided greater programmer support. While the project was initially intended to target the OpenCL framework in order to benefit from its portability, the greater maturity, flexibility, availability and accessibility of CUDA at that time made it the most appropriate choice. Due to the similarities between OpenCL and CUDA kernel programs, however, translation between these languages is relatively uncomplicated, and porting the architecture is not a particularly complicated endeavor once development is complete.

C. Architecture Components

The revised classification architecture consists of several general components, each of which has been adapted to address different elements described in the previous section. A simple diagram showing the relationship between components, as well as input and output files, is provided in Figure 5.

Processing is divided between three layers, each operating on distinct platforms. The first layer provides a user interface, and was developed using C#. The major elements of this layer include the capture visualiser, which provides functions for exploring captures at a high level through temporal visualisation, and the program compiler, which converts high-level filter code into optimised filter programs. These components serve to provide useful and illustrative outputs and simplify program creation respectively. The second layer, written in C++, is either invoked through the C# interface or via the command line, and contains data structures and methods optimised for managing Input/Output (I/O) operations, packet buffering and pre-processing, and filter execution management. This layer contains the Capture Management component, which is designed to reduce capture buffering time and generate packet index files. Capture classification and packet processing is facilitated within the third layer by a virtual machine written in CUDA, which extends upon and significantly improves the versatility and flexibility of the prototype classification engine.

D. Indexes, Results and Distilled Traces

The packet capture processing architecture presented depends on an assortment of files to work efficiently. The primary inputs include packet traces (see section II-A), protocol library files and high-level filter code.

During the course of packet processing, the architecture produces additional files which are themselves inputs to the system. These include index files, result files and distilled traces. Packet index files are per-packet arrays which are used to navigate a capture quickly, without requiring repeated re-parsing. There are two distinct types of index file, which are produced concurrently the first time a packet capture is processed. The first is an absolute packet offset index, which is simply an array of the 64-bit byte offsets of each packet in the capture. The second index file type references the absolute packet offset index file, and contains a per-second array of packet indexes which correspond with the number of packets that have arrived at each second in the capture. These files are used to quickly navigate and visualise captures, and allow for the rapid derivation of several useful metrics, including the efficient calculation of packet rate, data rate and average packet size over any arbitrary interval, without the need for re-parsing capture files or storing large volumes of data in host memory.

When packets are processed, they concurrently produce individual filter results files for each filter specified. Each filter result file is an array of bits, where each bit corresponds to the truth value of that filter for a single packet. These result files are extremely small, highly compressible and reusable. In addition, they an be combined with ease through simple bit-wise boolean operations. Packet results files are used as masks for the visualisation component, and may additionally be used to distill simplified or reduced traces. Distilled captures are smaller, focused captures produced by parsing a capture, and using both filter results and/or a temporal window to remove packets which are of no interest. These reduced captures may then be processed in other protocol analysers which do not cope efficiently with large captures.

A detailed consideration of these components may be found in [10].

E. Filtering Language

The GPU classification virtual machine executes filter programs written in a high-level DSL, implemented using ANTLR v3 (ANother Tool for Language Recognition) [8] and C#. The DSL processes the specified filter program, performs a number of optimisations, and emits the resultant compiled filter as an array of commands encoded as integers for execution on the GPU VM (Virtual Machine). The DSL uses a syntax heavily inspired by C style languages, but uses different keywords specific to packet processing, and is intended to be flexible, powerful, and easy to read, modify and write.

To simplify the process of creating filter programs, the DSL supports the creation of protocols and protocol suites, analogous to classes and namespaces in C++ respectively, which encode high-level protocol libraries that describe the structure of collections of protocols and their header fields. They are intended to simplify the process of creating new

filters by providing named fields and filter functions which may be referenced and reused. Protocol libraries operate using a high-level syntax intended to allow flexible and extensible processing of arbitrary protocol headers. These libraries are used to generate optimised filter code, by allowing the filter compiler to determine at compile time which fields need to be evaluated, and which operations need to be performed, in order to generate the results requested by the filter program.

Protocols are evaluated in a chain (similarly to the PathFinder packet classifier [17]), and each protocol may specify which protocol is next in the chain based on the contents of a protocol header field or register value. These links between protocols are used to identify all possible paths between the physical layer protocol indicated in the global header of the packet trace, and any specified high-level protocol. Thus, a program may simply query "TCP.SourcePort == 445", and the compiler will determine all protocols that could potentially precede TCP, and generate filters to detect and navigate them. This greatly simplifies the specification of filters for protocols at higher levels in the protocol stack, and will hopefully help to reduce user error.

Protocols may contain any number of fields or filters, which define the header structure and predefined filter methods respectively. Fields describe the physical layout of protocols in the packet data in terms of bitwise offsets and lengths, and support optional and variable length field types through DSL operations which use the syntax of if statements and while loops respectively. Filters, in contrast, define a general process, and are thus similar to methods in C based languages. Filters can return boolean results or numeric values, either as one or more arrays when invoked from the host, or as single value when invoked from within a protocol executing on the GPU VM. Filters can be used to define reusable filter predicates, or to gather and transform packet header data for use in visualisation or statistical analysis.

In addition, protocols may maintain a limited collection of runtime variables (depending on the GPU being used), which may be used to store and transform packet data for use in calculations and when returning filter results. The allocation and de-allocation of registers is automatically handled by the compiler, which maintains a set of 30 or more numeric registers and 64 boolean registers for each packet. Protocols may also access and manipulate a selection of system registers through special functions. Protocols provide a wealth of other functionality and concepts, including protocol inheritance, field value definitions and a range of mechanisms for controlling the execution of the GPU VM from within protocol code. Unfortunately, due to the complexity and scope of the language and the space limitations of this paper, a detailed discussion is not possible here, and will be deferred to a future publication.

The main process in a filter program is the filter kernel, which performs a similar function to that of a main method in C++. Filter kernels specify which filters should be invoked, which in tern determines which protocols need to be evaluated. Results can be retrieved and stored as a results file from within this method. The compilation of a filter kernel involves parsing and interpreting the structures defined in the protocols it references so that these structures may be used to construct minimalistic and efficient parallel GPU filters. Processing involves parsing all expressions and variable based operations first, and then parsing the field, protocol and suite structure into optimised collections in memory. These ordered collections can then be analysed, manipulated and pruned to emit optimised filter code. Memory reads are optimised so that packet data is loaded as few times as possible by overlapping as many comparison operations and variable loads onto a single data cache read as register space allows.

Once this process completes, the compiled filter kernel is ready to be passed to the classification host process to begin preparation for evaluation.

F. Capture Management

The capture management components of the architecture are host side components which execute concurrently across several threads, and provide the context and inputs necessary to perform all filtering, indexing and distillation operations. The capture management component is the primary I/O component, and is responsible for reading and writing capture, index and filter result files. It was developed using the C++ standard library and the Tiny-thread API (Application Programming Interface).

Due to the bandwidth bottleneck between long term storage and host memory identified in Section III-A, the capture management component's performance heavily impacts the applications overall performance. Care has thus been taken to optimise the performance of this component as much as possible. Each file accessed by the capture management component is read and buffered by a dedicated light-weight thread in 512KB chunks, before being processed by the primary buffer management thread. This raw data is then cycled through in order to concurrently generate packet index files and pass chunks of packet data to the GPU virtual machine, or to the distillation mechanism. Indexing and distillation is covered in more detail in [10].

In order to mitigate the bottleneck presented by the limited performance of individual drives, the capture management component supports a file mirroring mechanism which can be used to reduce execution time at the expense of disk space on independent drives. By specifying multiple identical copies of a specific trace file located on independent disks, the capture manager is able to coordinate and load balance partial capture reads from multiple drives concurrently, increasing the speed at which the capture can be buffered in host memory, in a similar manner to that of a Redundant Array of Independent Disk (RAID) configuration. In order to maintain high capture read speeds across multiple drives, the average read performance of each drive is monitored over a small rolling window, and used to periodically adjust the read load on each drive based on its recent performance. The period between each adjustment is determined using an incremental back-off process. This method ensures that poorly performing

disks do not inadvertently reduce rather than improve the capture parsing speed, without requiring the micromanagement of individual threads.

The file mirroring feature provides the speed benefits of a RAID array without the need to create or maintain one, at the expense of additional disk space.

G. Filtering Virtual Machine

The GPU classification process is handled by a specialised parallel virtual machine which executes the filter kernel operations stored in an integer array in GPU constant memory over segments of packet data collected by the capture manager. The architecture of this VM is derived from the prototype filtering kernels, which used a multistage filtering process, where packet data was processed in two to three course-grained parallel batches in rigid sequence. This course-grained approach was sufficient for static predicates, which could be pre-compiled without access to packet data, but lacked the flexibility needed for more complex and variable protocols where offsets cannot be predicted. The classification VM uses a similar but significantly more flexible approach, relying on fine grained commands (rather than course grained phases) that can occur in any sequence any number of times. This is made possible by the fact that, due to its specialised nature, the virtual machine does not require a wide variety of operations, and many simplifying assumptions can be made about how processing can proceed.

On order to minimise redundant memory loads, the VM interleaves the processing of multiple protocols and evaluates them concurrently. The VM uses a 64 byte packet cache in register memory to hold small chunks of data, which can then be evaluated by multiple protocols simultaneously. Due to the limited register space available to individual threads on many GPUs, it may often not be possible to process all protocols concurrently without spilling registers into device memory. If this happens to frequently used registers, the overall performance of the classification process would be significantly diminished by the high latency inherent in reads from device memory [18]. To minimise this, the per-thread register capacity of the graphics card and the register demands of each protocol are used to split processing into a multiple passes over the packet data (using a simple scheduling algorithm) during compilation. This ensures that the register demands of each batch of concurrent protocol evaluations never exceed the capacity of the execution thread, and thus do not incur significant performance penalties due to high latency.

One of the more considerable complications in processing more flexible protocol specifications is managing thread divergence efficiently. In the prototype algorithm, divergence was eliminated entirely by processing every classification predicated in every packet. Similarly, the approach taken in the VM favours eliminating as much divergence as possible by allowing for redundant processing, rather than trying to forcibly block threads from executing irrelevant classifications, in order to adhere as closely as possible to a SIMD processing model. While each thread attempts to processes all protocols, they are aware of what the current protocol in the packet they are processing is, and use this to determine which operations should be stored, and which should be discarded. In addition, before a protocol is evaluated, each warp may optionally check to ensure that at least one packet in the warp is associated with that protocol using shared memory. If no packets identify with that protocol, that protocol is skipped over by the warp, and all results are set to zero for that protocol.

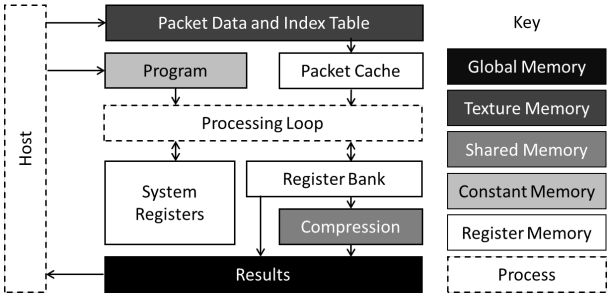


Figure 6. Abstract memory layout of the packet classification virtual machine.

Each thread maintains a bank of registers used to store system and user variables. Each thread maintains 64 1-bit boolean registers, contained in two 32-bit integer registers, and a variable number of 32-bit integer registers, depending on the GPU being utilised. Rather than attempt to allocate and de-allocate registers during execution, the compiler tracks register usage by protocol, and arranges these integers within an array. When a protocol moves out of scope and all its relevant data has been stored, its registers are considered free, and other protocols may use those offsets in the register array. Values which exceed 32-bits are stored across multiple registers.

The virtual machine outputs individual per-packet arrays for each boolean and numeric result returned to the host. Classification results are stored in highly compact and easily compressed single bit format, which may be used as index masks to extract, visualise and distill packets matching particular filter results. By using a bit-based result format, multiple filters can be combined through simple bitwise AND and OR operations on the host, or by a GPU function. Figure 6 shows a high-level overview of the memory architecture of the virtual machine, and the flow of data from memory banks to and from the main processing loop.

H. Visualisation

The user interface provides a unified set of tools for exploring and analysing large long-term trace files, primarily through controls and data structures which facilitate fast visualisation of the underlying packet data. Visualisation depends heavily on index files and filter results generated during packet processing in order to allow for real time exploration of large captures without exhausting host memory. The interface is implemented in C# to facilitate rapid prototyping and development, with visualisations constructed in OpenGL through the OpenTK C# wrapper.

One of the primary problems associated with constructing the visualisation and interaction components of the user inter-

face is the limited access to the underlying packet trace data, which is composed of millions, if not billions, of arbitrarily sized packets whose offsets must be determined by parsing all prior packets sequentially in the trace. As the interface is intended to visualise packet sets that could potentially span terabytes, it is not feasible to store packet data in host memory for analysis due to space constraints. Similarly, repeatedly re-parsing multi-gigabyte capture files for each operation performed in the interface is infeasible as it is far too time consuming. Through the use of the index files generated by the capture manager during capture parsing, and using a tree-like data structure which incrementally caches packet metrics over suitably sized intervals in the capture, the visualisation component is capable of generating detailed overviews of traffic data over any time interval in near-real time, without consuming significant memory resources.

The capture visualisation component displays an overview of the capture by using the per-packet and time-stamp index files to calculate the number of packets and amount of data over various time intervals, which maybe expanded and contracted in real time. Graph rendering is performed by shader programs which execute on the GPU, and the resultant images may be stored in the Portable Network Graphics (PNG) format. Figure 7 shows two images generated by the visualiser, showing both a capture wide (7a) and month-long segment (7b) of visualised traffic volumes. In these images, the black line indicates the number of packets arriving at that time, while the gray area shows the volume of packet data arriving at that time. As the amount of packet data arriving at any given time will always significantly outweighs the number of packets arriving, these graphs have been rescaled to similar sizes. The visualiser supports multiple overlays on the same graph, of various different types, in customisable render orders and colours. In addition, the visualiser is capable of overlaying the results of filters on capture overviews. The visualiser is designed to support additional graph layouts, although these have not been finalised at this time.

IV. Architecture Applications

The high-level architecture presented in this paper is primarily intended to support and accelerate packet trace analysis, and thus discussion has focused on this application exclusively. In many respects, the GPU-based packet processor previously discussed is ideally suited to offline trace-processing tasks, as it is in essence a batch-processing approach. It allows for packets to be buffered and processed collectively to best exploit the SIMD nature of GPU co-processors, and depends heavily on this collective processing to maintain its classification throughput. This does not however preclude its application in online classification processes.

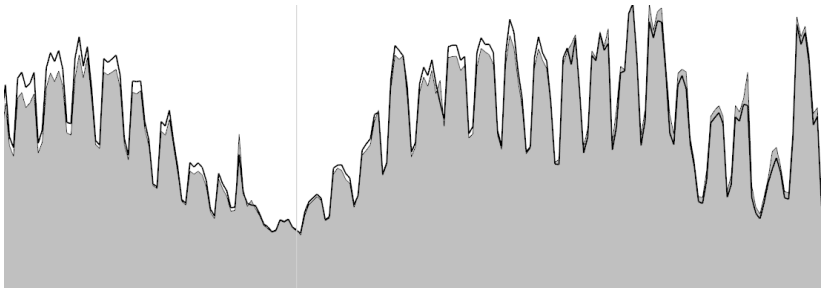
The most significant complication in applying this classification and processing framework to live network traffic is maintaining high throughput while simultaneously minimising the latency between when a packet arrives and when it is processed. This approach may not be viable in applications where minimal latency is paramount, as it is difficult to avoid deferring packets processing entirely. In applications which can afford some latency, however, high-volume near-real-time classification is possible by reducing the size of packet buffers and forcing buffer processing if they are unfilled after a set interval. When network volume is high, buffers will fill quickly and may be processed rapidly, mitigating latency issues. When network volume is low, packets are still guaranteed to arrive and be processed by the device at the end of each set interval, and maximum latency can be reduced by reducing this interval.

With support for live captures and relatively low latency packet processing, the framework is applicable to a wide range of problems, including intrusion detection systems and network monitoring applications. As GPUs are relatively low cost commodity hardware items which run in parallel with a CPU, it is also potentially possible to create and deploy an agent based distributed classification and monitoring sensor network, using available GPGPU capable hosts. The architecture, while packet centric, may in addition be expanded into a more general records processing framework, to be used in processing large log files and similar record sets.

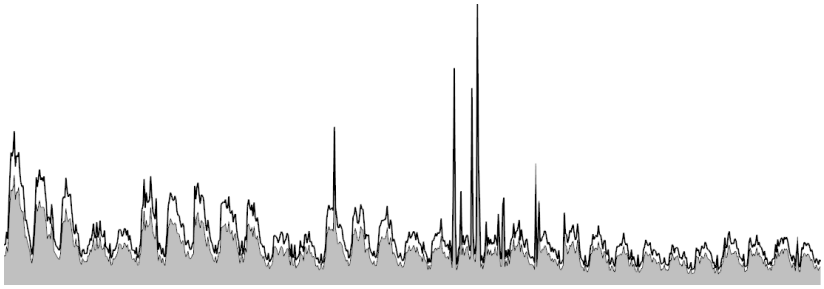
V. Summary

This paper has described an architecture which accelerates the classification, processing and exploration of large packet traces, extending prior research into GPU accelerated packet classification. The described architecture uses four primary components: the program compiler, the capture manager, the GPU classification virtual machine and the visualiser. The program compiler is used to generate optimised virtual machine instructions using a high level domain-specific language to specify protocol structure and filter operations. The purpose of this component is to simplify the creation of classification and analysis programs. The capture manager collects and buffers packet data and generates index files to accelerate visualisation and capture navigation. The capture manager also implements a field mirroring function, which allows multiple copies of capture across distinct drives to be used to significantly accelerate the capture reading process. The filtering virtual machine is used to process the captures buffered by the capture manager using the virtual machine instructions generated by the compiler. The virtual machine can gather data, perform comparisons, execute functions and process optional and variable length fields, returning results as optimised per-packet data arrays. The visualiser uses the index files generated by the capture buffer and the results generated by the classification virtual machine to visualise the capture, and thereby aid in the exploration of complex long term traces.

Together these components provide an efficient low-overhead solution for exploring, analysing and extracting data from extremely large trace files. Future work is expected to focus on refining the process, and expanding analysis to a distributed network of agents which can monitor traffic, issue alerts, and provide remote visualisation and filtering of recent traffic.



(a) 21st October 2010 to 9th May 2011 with 1 day intervals.



(b) 1st December 2010 to 31st December 2010 with 1 hour intervals.

Figure 7. Visualisation showing comparative packet count (black line) and data volume (gray area) over time, at different levels of scale, for a large (43GB) packet trace.

REFERENCES

[1] B. V. W. Irwin, "A framework for the application of network telescope sensors in a global ip network," Ph.D. dissertation, Rhodes University, Grahamstown, South Africa, January 2011.

[2] R. Pang, V. Yegneswaran, P. Barford, V. Paxson, and L. Peterson, "Characteristics of internet background radiation," in *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, ser. IMC '04. New York, NY, USA: ACM, 2004, pp. 27–40. [Online]. Available: http://doi.acm.org/10.1145/1028788.1028794

[3] E. Wustrow, M. Karir, M. Bailey, F. Jahanian, and G. Huston, "Internet background radiation revisited," in *Proceedings of the 10th annual conference on Internet measurement*, ser. IMC '10. New York, NY, USA: ACM, 2010, pp. 62–74. [Online]. Available: http://doi.acm.org/10.1145/1879141.1879149

[4] A. Nottingham, "Gpf: A framework for general packet classification on gpu co-processors," Ph.D. dissertation, Rhodes University, Grahamstown, South Africa, February 2012.

[5] "Winpcap documentation," Online, WinPcap, last accessed: 17/06/2011. [Online]. Available: http://www.winpcap.org/docs/docs_412/html/main.html

[6] G. C. Ulf Lamping, Guy Harris, "Libpcap file format," Online, July 2005. [Online]. Available: http://wiki.wireshark.org/Development/LibpcapFileFormat

[7] Nvidia Corporation, "Nvidia CUDA C Programming Guide, Version 3.1," Online, NVIDIA Corporation, May 2010, last accessed: 09/05/2010. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf

[8] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages*, p3.0 ed. The Pragmatic Programmers, 2008.

[9] V. Volkov, "Better performance at lower occupancy," Online, September 2010, last accessed: 30/05/2012. [Online]. Available: http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf

[10] A. Nottingham, J. Richter, and B. Irwin, "Capturefoundry: a gpu accelerated packet capture analysis tool," in *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, ser. SAICSIT '12. New York, NY, USA: ACM, 2012, pp. 343–352. [Online]. Available: http://doi.acm.org/10.1145/2389836.2389877

[11] W. Jiang and V. K. Prasanna, "Field-split parallel architecture for high performance multi-match packet classification using fpgas," in *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. New York, NY, USA: ACM, 2009, pp. 188–196.

[12] W. Jiang and V. K. Prasanna, "Large-scale wire-speed packet classification on fpgas," in *FPGA '09: Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 2009, pp. 219–228.

[13] D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Comput. Surv.*, vol. 37, no. 3, pp. 238–275, 2005.

[14] A. Begel, S. McCanne, and S. L. Graham, "Bpf+: Exploiting global data-flow optimization in a generalized packet filter architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 4, pp. 123–134, 1999.

[15] S. McCanne and V. Jacobson, "The bsd packet filter: a new architecture for user-level packet capture," in *USENIX'93: Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. Berkeley, CA, USA: USENIX Association, 1993, pp. 2–2.

[16] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort: High performance network intrusion detection using graphics processors," in *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 116–134.

[17] M. Bailey, B. Gopal, L. L. Peterson, and P. Sarkar, "Pathfinder: A pattern-based packet classifier," in *Proceedings of the First Symposium on Operating Systems Design and Implementation*, ser. OSDI '94, Monterey, California, November 1994, pp. 115–123, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.46.1294&rep=rep1&type=pdf.

[18] Nvidia Corporation, "Nvidia CUDA C Best Practices Guide, Version 3.1," Online, NVIDIA Corporation, May 2010, last accessed: 09/05/2010. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_BestPracticesGuide_3.1.pdf