gPF: A GPU Accelerated Packet Classification Tool

Alastair T. Nottingham[1] and Barry Irwin[2]
Security and Networks Research Group
Department of Computer Science
Rhodes University
E-Mail: [1]anottingham@gmail.com, [2]b.irwin@ru.ac.za

***Abstract*—This paper outlines the design of gPF, a fast packet classifier optimised for parallel execution on current generation commodity graphics hardware. Specifically, gPF leverages the potential for both the parallel classification of packets at run-time, and the use of evolutionary mechanisms, in the form of a GP-GPU genetic algorithm to produce contextually optimised filter permutations in order to reduce redundancy and improve the per-packet throughput rate of the resultant filter program. This paper demonstrates that these optimisations have significant potential for improving packet classification speeds, particularly with regard to bulk packet processing and saturated network environments.**

***Index Terms*—Packet Classification, GP-GPU, Genetic Algorithms**

I. Introduction

Fast and accurate packet classification is a vital component in modern network environments. Classification is widely used with regard to both network security, differentiated service offerings, traffic shaping and packet stream demultiplexing for delivery from the network tap to listening applications. As available bandwidth continues to grow [1], and traffic volumes over network infrastructures increase, the need for greater processing efficiency regarding packet classification becomes ever more necessary.

This paper provides a brief, high level overview of the conceptual design of gPF - a fast packet classifier intended to leverage the parallel processing capabilities of current generation Graphics Processing Units (GPUs). This system is used to reduce both the redundancy in filter programs through the use of genetic algorithms, and improve overall packet processing speed through parallel execution of packet classification. gPF is intended primarily as a bulk packet analyzer for the fast processing of large recorded packet dump files, but may further provide significant improvements when applied to live packet demultiplexing. The initial implementation is focused on the processing of IP based traffic, but could be applied to other packet based transmission systems.

This paper is organized into six sections. Section II provides relevant background information regarding packet filter history. Section III briefly discusses the GPU general processing and the selection of OpenCL as an implementation framework. Section IV deals with the optimisation of filter permutation using an evolutionary approach, providing an overview of genetic algorithms, their advantages with regard to the problem space, and a selection of preliminary results regarding their application to filter optimisation. Section V provides a motivation for GPU based packet processing, and considers a few associated challenges. Finally, conclusions are discussed in section VI.

The authors would like to acknowledge the financial support of Telkom SA, Comverse, Stortech, Tellabs, Amatole Telecom Services, Mars Technologies, Bright Ideas 39 and THRIP through the Telkom Centre of Excellence in the Department of Computer Science at Rhodes University

II. Background

A. Brief Overview of Packet Filtering

In this paper, we consider a GPU (Graphics Processing Unit) accelerated packet analyzer, derived from filters used in the context of packet demultiplexing. Packet filtering for demultiplexing purposes essentially refers to the process of identifying the most appropriate end-point for a packet arriving at a network interface on a particular host, such that the packet may be delivered to the correct waiting application, in a timely manner [2], [3], [4]. This should not be confused with the similar field of packet filtering for IP routing, which refers to the classification of incoming packets, typically on routing hardware, for the purposes of selecting the appropriate path along which to forward the packet, such that it arrives at its destination, again in a timely manner[5]. In both instances, the incoming packet is compared to a priority ordered set of rules, and either dispatched according to the first matching rule, or discarded if no matches are found.

The most notable difference between demultiplexing algorithms and IP routing algorithms is the degree of flexibility provisioned. As their name suggests, IP routing algorithms typically focus explicitly on the IP header of incoming packets, as this header is sufficient for identifying the correct rout along which to forward a packet. As such, these algorithms are often designed rigidly, and thus do not natively support classification against arbitrary protocols. In contrast, packet demultiplexing algorithms are constructed to specifically support classification against arbitrary protocols. This generality is useful when performing packet analysis, as it allows consideration outside of the typical IP 5-tuple of source and destination address, source and destination port numbers, and protocol [5]. This however comes at the expense of reduced throughput. Furthermore, as demultiplexing algorithms do not target a specific protocol, they typically avoid exploiting protocol specific fields in their architectural design[2]. Despite these limitations, the flexibility provided for by demultiplexing algorithms makes them appropriate for use in a packet analysis context.

B. History

The field of packet demultiplexing has a long history of research and development, pioneered by the *CMU/Stanford Packet Filter* (CSPF), a memory-stack-based packet filter [2]. CSPF was followed by the *BSD Packet Filter* (BPF), which provided the foundation for modern register-based filter machines [6], [4]. BPF implemented a RISC based pseudo-machine, in which filters were created using a low level assembler language, and translated into a directed acyclic control flow graph (CFG) for packet processing[2]. This deviated from the method used by CSPF, which used a boolean expression tree abstraction, a model suited to stack based environments. BPF has to a large extent become the de facto system for packet matching on most modern Unix-like operating systems.

BPF was succeeded by several similar packet filters, engineered to improve both classification efficiency and flexibility. This began with the *Mach Packet Filter* (MPF) in 1994, targeted at the Mach micro-kernel, which introduced packet fragment handling and packet matching optimisations[3], and was followed in the same year by the *PathFinder* packet classifier, which leveraged a declarative packet-masking mechanism to match a packet against a line of cell patterns within a directed acyclic graph (DAG) [7].

The successes of both MPF and PathFinder paved the way for the *Dynamic Packet Filter* (DPF) in 1996, a sophisticated packet filter which employed dynamic code generation to exploit run-time information at compile time, thus improving the efficiency of filter operation [6], [4]. Dynamic code generation proved successful in reducing redundancy, often significantly, and thus was incorporated into a subsequent BPF descendant in 1999, *BPF+*, in the form of JIT compilation [4]. BPF+ also introduced a significant number of concurrent and interdependent optimizations, including constant folding, register propagation and partial redundancy elimination, in order to reduce the number of nodes in its filter tree[4]. As a result, significant improvements to overall performance were noted.

Since the introduction of BPF+, there has been relatively little development within the field of packet demultiplexing filters. The *Extended Packet Filter* (xPF) incorporated simple extensions for statistics collection into the BPF model [8] in 2002, while the *Fairly Fast Packet Filter* (FFPF) used extensive buffering to reduce memory overhead, among other optimisations [9] in 2004. Finally, the *Swift* packet filter, introduced in 2008, used CISC based pseudo-machine to minimise filter update latency, further reducing instruction overhead and command interdependence [10]. Despite this, a number of alternative methods for improving filter performance still remain relatively unexplored. One such method is that of filter permutation optimisation prior to control flow graph construction. The gPF system described in this paper develops upon the features described above, introducing an innovative evolutionary optimisation approach, and leveraging the parallel processing capabilities of current generation GPUs.

III. Implementation Language

Prior to the discussion of implementation specifics, a brief introduction to GPU General Processing (GP-GPU), as well as the implementation framework OpenCL, is warranted. With the introduction of the programmable shaders found in current generation commodity graphics hardware, GPUs have become a viable and comparatively inexpensive platform for fast, general purpose parallel processing [11]. In order to utilise the processing power of programmable GPU hardware, a processing framework is necessary. While vendor specific implementations, such as CUDA for nVidia based graphics cards, and ATI's STREAM framework, provide the necessary flexibility and power, they lack portability between architectures, reducing their desirability. In contrast, the OpenCL specification ensures portability between all OpenCL compliant devices [11], allowing for the same GPU kernel program to execute on a wide range of vendor architectures and processor types. As such, the OpenCL C language, a C language derivative with parallel processing extensions, will be leveraged as primary implementation language, allowing for superior portability.

IV. Minimizing Decision Trees

Packet classification techniques typically comprise comparing a subset of a packets header information to a set of static values in order to identify a packet as a particular type, thus determining its destination, as well as other relevant information. When a packet filter contains more than one filter program, comparison overlap in multiple filters is nearly unavoidable. For instance, a significant proportion of protocols utilise IP within their network layer to facilitate and maintain connections, and thus filters for these protocols will all test for an IP header, introducing significant redundancy [6], [4].

While the compiler optimisation techniques utilised in packet filters such as DPF and BPF+ typically eliminate a significant proportion of these redundancies, the effectiveness of optimisation is largely dependent on the order in which header values are tested, which corresponds to the order of filters prior to optimisation [12]. Finding an optimum ordering of filters is thus equivalent to finding an optimum directed acyclic control flow graph, which is in turn comparable to binary decision tree [2]. As constructing an optimal binary decision tree is NP-Complete [13], finding an optimum filter permutation is a non-deterministic operation.

While a set based heuristics solution which utilises an adaptive pattern matching algorithm to find a near-optimal decision tree has been detailed in the literature [12], the effectiveness of the permutation optimisation is limited when the number of filters grows large. In this regard, an attractive alternative is to breed a near-optimal filter permutation using a generation restricted, GPU-based genetic algorithm, as discussed in the following section.

A. Genetic Algorithms Overview

Genetic algorithms are a form of adaptive algorithm modeled on natural evolution. The concept of an evolutionary algorithm was first introduced over fifty years ago as a mechanism

for finding good solutions to problems within vast search spaces [14]. One such early attempt was that of an automatic programming algorithm, which attempted to evolve a binary encoded computer program capable of performing simple computational tasks, such as finding the sum of two bits [14]. Due to the lack of computational power at the time, success was somewhat limited, but subsequent technological developments and various algorithmic improvements have only supplemented the capabilities of genetic algorithms, increasing their applicability to a variety of optimisation problems.

A genetic algorithm is essentially composed of a population of individual chromosomes, where each chromosome represents, or encodes, a particular solution to a problem. Each chromosome is assigned a fitness value, representing the efficiency of its particular solution. The initial population is typically generated at random, and subsequent generations created by selecting two parent chromosomes from the population pool, and using them to create two new child chromosomes, each containing parts of both parents. These child chromosomes then enter the population pool, often replacing chromosomes with the lowest fitness in the process. By repeating this process for a number of generations, chromosomes with greater fitness values are slowly evolved. At some point, the process is stopped, and the best performing chromosome is selected as the solution [14]. While genetic algorithms do not guarantee an optimum solution, they are considered effective at finding near optimum solutions in relatively short time periods, making them attractive alternatives for NP-Complete problems, such as the traveling salesman problem [14].

As finding an optimum filter permutation is an NP-Complete problem [13], we intend to apply a genetic algorithm to attempt to improve filter permutations, implemented such that the resultant control flow graph is optimally minimised for the network context, thus shortening the execution time of the actual classification process. In the following subsections, we discuss the specifics of the algorithm used.

B. Contextual Advantages

With respect to the minimisation of packet classifier control flow graphs, the application of genetic algorithms provide for a number of significant advantages. Set based solutions, while capable of approximating a minimum control flow graph, are less useful when large numbers of filters are present, due to the NP-Complete nature of the problem [13]. Furthermore, as they are blind to the environment, they are relatively inflexible and unsuited to self-optimisation within the context of a particular network environment.

In contrast, genetic algorithms scale well with regard to filter set size, with the bulk of computation concerned with permutation evaluation. The evaluation procedure is executed roughly nm times each time the genetic algorithm is run, where n is the number of chromosomes in the chromosome population, and m is the number of generations over which the genetic algorithm is active. Due to the parallel nature of genetic algorithms [14], further improvement to performance is possible through the utilization of General Processing on GPU devices (GP-GPU). By performing evaluation over multiple cores, utilising a *Single Program Multiple Data* [11](SPMD) configuration, the number of evaluations is reduced to m per multiprocessor, as long as $n \leq n_{max}$, where n_{max} is the number of available multiprocessing cores. This results from the concurrent and independent execution of the fitness evaluation algorithm over the available multiprocessors, eliminating the sequential execution bottleneck. This should greatly improve genetic algorithm performance, with the possibility of achieving near-linear speedup in the best case.

A further advantage of using an evolutionary approach is autonomous adaptability to diverse network run-time environments. In order to optimize a particular filter set, it is first necessary to define what constitutes an optimum control flow graph, such that individual permutations may be compared. One possible, arguably simplistic, solution would be to consider the number of individual nodes in the resultant control flow graph as an effective performance measure, as this implies a minimal longest path in most cases. Alternatively, average, maximum and minimum path lengths may be considered, along with many other alternatives. However, network traffic is rarely, if ever, evenly distributed over the a filter set, and thus certain classifications will be made more frequently than others. If the most frequently occurring classifications appear near the end of a minimized control flow graph, performance may be adversely affected. Thus, performance is heavily influenced by network traffic composition, which is difficult to predict, and changes over time.

By recording a sliding window of recent network traffic, we may introduce the use of heuristic measures to determine which performance evaluators are likely to yield the most efficient control flow graph structures for a particular network context. We may then breed one or more permutations in parallel, using the most promising performance measures, and finally select the best performing permutation for use within the classifier. This process may be performed in the background, at set intervals, thus allowing the filter to adapt to changing network environments without user intervention or complicated user interaction.

C. Challenges

There are several challenges associated with filter permutation optimisation, primarily concerned with semantic changes and permutation breeding time. Regarding semantic changes, general filters may hide more specialized filters if they occur first, and are composed of a subset of the specialized filters predicates. By changing the permutation of filters, it is possible that the semantics of the filter program may be changed. Thus, it is necessary to be able to declare non-negotiable filter hierarchies, and to detect and eliminate hidden solutions, such that consistency is maintained.

To achieve these goals, a number of facilities need to be incorporated. Firstly, a mechanism to enforce a hierarchy of evaluation is necessary. This may be achieved by simply specifying expected results from preceding filters. While it is imperative that such specifications do not create loops [4], the nature of a permutation abstraction inherently prohibits cycles, and so such risks are mitigated. The remaining hidden solutions

should be identified and eliminated if possible, or alternatively identified and reported as possible errors. If no solutions can be found to reveal a hidden solution, the permutation of all relevant node denominators should be enforced. Once dependencies have been established, one may simply swap all elements which are in the wrong order until such time as an acceptable solution is found. Due to the simplistic nature of filter specifications, this should be sufficient to ensure consistency between permutations.

A similar problem arises when two filters do not contradict one another. In such cases, it is possible for a filter to match both cases, and thus the result of classification is dependent on permutation. Again, such problems may be mitigated by allowing for explicit ordering, and otherwise ensuring that such permutations are left in their original order. While these complexities may alternatively be eliminated by allowing for multiple classifications per packet, a useful feature in some scenarios, such a solution is both expensive, and generally provides little if any improvement to classification accuracy.

Finally, we consider the impact of breeding time within the context of permutation improvement. As previously indicated, the required breeding time for a genetic algorithm depends heavily on the both the size of the chromosome population, and the number of generations over which the population evolves[14]. Thus, while larger populations and generation counts may increase the likelihood of finding a near-optimal filter permutation, this comes at the expense of execution time. However, as an optimal permutation need only be evolved once, before being deployed indefinitely, such expense is justified.

In the following subsection, we discuss preliminary results obtained through a simplified simulation system.

D. Preliminary Results

In this subsection, we discuss preliminary results gathered through a simplified predicate classification system, which indicate significant potential for CFG optimisation using evolutionary techniques. The implemented filter system defines a filter as a set of simple predicates to be tested against a string of 20 random uppercase characters. A predicate is of the form $I[==|!=]K$, where I is the relevant packet index, and K is the character value to be tested at that index. A filter may contain several predicates, and concludes with a classification. A filter classifies a packet only if all filter predicates evaluate to true. If a predicate in a filter evaluates to false, the next filter is processed until either the packet is classified, or the last filter fails to classify the packet. The simulation system also allows the specification of predicates, and the percentage of filters which should contain them, to ensure redundancy. The genetic algorithm used is relatively simple in nature, employing the node count of the resultant control flow graph as a measure of fitness. It is important to note that the simulation system does not implement any GPU based optimisations.

Figure 1 shows the node counts of ten different randomly generated filter programs, and their reduced node count with and without genetic algorithm optimisation. Each filter program contained 20 distinct filters, each testing between five and ten predicates, with the predicate $2 == K$ having an 80% occurrence probability in any given filter. The genetic algorithm was run for 200 generations, over a population of 100 chromosomes, with processing time varying between 36 seconds and 78 seconds.

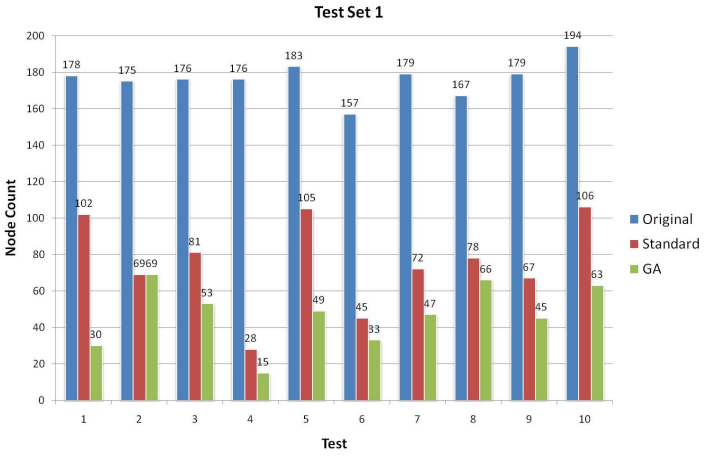

Figure 1. 20 Filters; length: 5-10, population size: 100, 200 generations, "2==K" in 80%

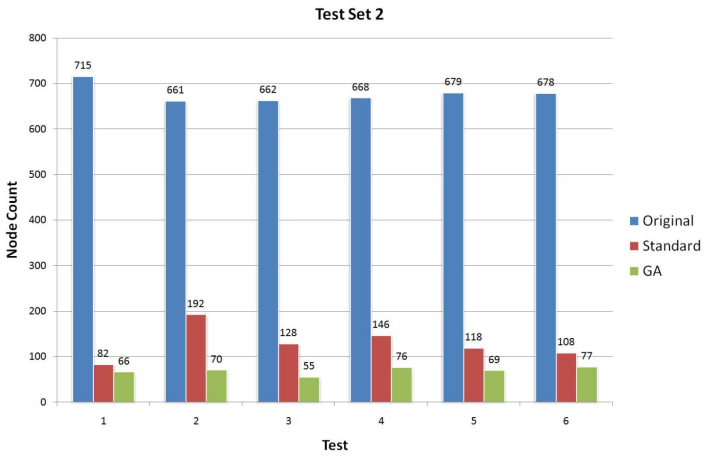

Figure 2. 50 Filters; length: 10-15, population size: 250, 500 generations, no explicit redundancy

Figure 2 shows a more complex case, where 50 filters, each comprising 10 to 15 predicates, is processed using 250 chromosomes over 500 generations. Unlike figure 1, no explicit redundancy was incorporated. Processing time for these more complex instances was significant, ranging between 1744 seconds and 2115 seconds, but through proper use of GPU parallel processing, these values should be significantly reduced.

Both figures 1 and 2 demonstrate the possibility for significant optimisation through reordering permutation alone. While such impressive improvements may not be achievable in a real world system, this method appears viable as an optimisation technique, particularly given that the genetic algorithm implemented was relatively generic. It is important to note, as is evident in these graphs, that the use of genetic algorithms does not ensure significant improvement in all instances, as, for instance, the initial permutation may already be near optimal.

V. Parallel Packet Classification

This section provides a brief overview of the motivation behind the creation of a GPU based parallel packet classification system, as well as a consideration of the challenges associated with its implementation. It is important to note that at this stage, no algorithm for parallel packet process has been properly considered, and thus explicitly stating performance improvement estimates is premature. We may however hypothesize, given the implicit parallelism of packet processing, that performance gains resulting from a GP-GPU implementation will be significant, and thus worth pursuing.

A. Motivation

Packet stream multiplexing and classification is essentially a parallel task, in that a static filter set is repeatedly compared against a continuous flow of diverse packets. For the purposes of classification, each packet can be regarded as being essentially independent of all other packets, although it is recognized that certain applications require the use of stateful inspection. This inherent parallelism may be exploited by many current GPU coprocessors, given their inherent SIMD (Single Instruction, Multiple Data) [15], [16], [11] and SPMD (Single Program, Multiple Data) [11] capabilities, allowing for a dramatic improvement to packet classification rate. Such improvement is warranted, as many packet sets, such as those collected at network telescopes, are too large to be processed interactively unless significant classification speed improvement is achieved. Furthermore, as current implementations, such as the BPF based PCAP, typically take longer to process a packet set than the time over which the packet set was originally captured, it becomes infeasible to analyze packet sets collected over long time-spans.

General Processing on GPU (GP-GPU) has been successfully leveraged by the gNORT application, a parallel implementation of the SNORT [15] network intrusion detection system. This implementation offloads string matching operations to the GPU to improve packet processing rates, showing performance improvements up to an order of magnitude faster than sequential implementations [15]. The SNORT network intrusion detecting system is architecturally and functionally similar to that of a typical packet classifier, in that it depends on the matching of packet data to string patterns in order to detect threats. However, while SNORT and gNORT operate primarily on packet payloads [15], packet classification tasks operate primarily on packet headers [2], [4], requiring substantially less memory copying overhead for each individual packet.

Thus, the utilization of GP-GPU technologies as a means of viable packet classification acceleration is supported by the promising results detailed in the literature [17], [15].

B. Packet Processing Challenges

As previously indicated, both OpenCL and the architecture of GPUs facilitate fast SPMD[11] processing, allowing for a single packet classification program to be run concurrently on multiple cores against diverse independent packets. While SIMD processing, as used in gNORT[15], proves sufficient for static string matching, a CFG implementation relies on divergent filter execution, and thus necessitates an SPMD implementation. Specifically, as different packet classifications correspond to different paths through a particular CFG, it follows that each individual packet requires an independent instruction register in order to traverse the CFG independently. As SIMD processing requires each processor element to execute commands from a single global instruction register, such independent traversal is not suited to SIMD processing. In order to ensure optimum performance within a GPU architecture, consideration needs to be given to packet buffering, and the efficient management of packet fragments.

In order to process packet headers on a GPU, these headers first need to be transferred and mapped to GPU memory. Doing this on a per-packet basis however introduces significant overhead, and thus a more elegant approach is necessary. By buffering packet headers for batch transfer to the GPU, this issue is mitigated, but introduces issues regarding packet fragmentation handling.

Packet fragmentation occurs when a particular datagram is larger than the Maximum Transmission Unit (MTU) of a facilitating network. For instance, the MTU in Ethernet networks is 1500 bytes, while in Token Ring networks, it is 4096 bytes. Such datagrams are split into multiple IP fragments smaller then the MTU, which are transmitted over the network independently, collected at the destination network interface, and finally reconstituted to form the original datagram. The datagram is then passed up the protocol stack for further processing. Each IP fragment header contains a unique fragment stream id, a fragment offset value indicating the fragments position in the datagram, and flags indicating whether more fragments are expected.

As the Internet Protocol only provides for best effort delivery, both packet loss and non-sequential fragment arrival are possible. As we are not necessarily concerned with packet reassembly, and higher-level protocol headers are almost exclusively contained within the initial packet fragment, successive fragments cannot be classified until such time as the first fragment has been successfully processed [3]. Thus, complications arise from an increased probability that packet fragments will be evaluated prior to, or at the same time as, the first packet fragment in the stream.

A simple solution is to maintain a fragment buffer, controlled by the CPU, which stores all fragments until such time as the first fragment of the stream is processed. Once an initial fragment is identified and recorded by the GPU packet classification algorithm, the fragment buffer is polled for all associated fragments, such that these may be copied to the GPU in the next bulk transfer if necessary. Of course, if subsequent fragments are not needed to complete classification, then these fragments may be discarded. If no first fragment arrives before a set time interval, or the filter set fails to classify the first fragment, associated fragments may be purged from the fragment buffer. This solution ensures that all packet fragments processed on the GPU are necessary, and can be associated with their appropriate context, eliminating the overhead of copying unnecessary and unclassifiable packets

to GPU memory. As data transfer is a bottleneck in GP-GPU processing [11], [15], this is warranted.

VI. Conclusion

In this paper, we have shown that the task of packet classification may be optimised and improved through the intelligent application of GP-GPU technologies to the problem space, both in actual classification and in the optimisation of filter permutation. We have shown that the use of genetic algorithms as a mechanism for control flow graph reduction is both promising, given the results of our simple predicate based filter optimisation simulation, and amenable to GPU based implementation, due to their inherently parallel nature. Furthermore, given that most packets on a network are distinct from one another and may be classified separately, a GPU based classification algorithm may greatly improve classification performance, due to the SPMD capabilities of the OpenCL framework and current generation programmable GPUs. Given the success of gNORT, such a hypothesis is supported by the literature [15], [17]. Furthermore, as OpenCL is an open standard, portability is guaranteed between diverse GPU architectures, as long as the device architecture in question supports OpenCL. Thus, the creation of a bulk packet classifier based upon the principles and ideas detailed in this paper has significant potential in accelerating packet classification, while retaining acceptable portability.

References

[1] J. Nielsen. (1998) Nielsen's law of internet bandwidth. Website. http://www.useit.com/alertbox/980405.html. [Online]. Available: http://www.useit.com/alertbox/980405.html

[2] S. Mccanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture," 1993, pp. 259–269.

[3] M. Yuhara, B. N. Bershad, C. Maeda, J. Eliot, and B. Moss, "Efficient packet demultiplexing for multiple endpoints and large messages," in *In Proceedings of the 1994 Winter USENIX Conference*, 1994, pp. 153–165.

[4] 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.

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

[6] D. R. Engler and M. F. Kaashoek, "Dpf: fast, flexible message demultiplexing using dynamic code generation," in *SIGCOMM '96: Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 1996, pp. 53–59.

[7] L. McMurchie and C. Ebeling, "Pathfinder: A negotiation-based performance-driven router for FPGAs," in *FPGA*, 1995, pp. 111–117. [Online]. Available: citeseer.ist.psu.edu/mcmurchie95pathfinder.html

[8] S. Ioannidis and K. G. Anagnostakis, "xpf: packet filtering for low-cost network monitoring," in *In Proceedings of the IEEE Workshop on High-Performance Switching and Routing (HPSR*, 2002, pp. 121–126.

[9] H. Bos, W. D. Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis, "Ffpf: Fairly fast packet filters," in *In Proceedings of OSDI04*, 2004, pp. 347–363.

[10] Z. Wu, M. Xie, and H. Wang, "Swift: a fast dynamic packet filter," in *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008, pp. 279–292.

[11] *The OpenCL Specification, Version 1.0*, Online, Khronos Group Std., Rev. 33, April 2009. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.0.33.pdf

[12] A. S. Tongaonkar, "Fast pattern-matching techniques for packet filtering," Tech. Rep., 2004.

[13] L. Hyafil and R. Rivest, "Constructing optimal binary decision trees is np-complete." *Information Processing Letters*, vol. 5, pp. 15–17, 1976.

[14] T. Back, "Evolutionary algorithms in theory and practice," Feburary 1994.

[15] 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.

[16] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2008, pp. 73–82.

[17] D. R. Horn, M. Houston, and P. Hanrahan, "Clawhmmer: A streaming hmmer-search implementation," *SC Conference*, vol. 0, p. 11, 2005.

Mr Alastair Timothy Nottingham completed his BSc (Hons) in Computer Science in 2008 . His research interests include high-performance parallel processing, computer security, and artificial intelligence, as well as computer graphics and data visualization. Alastair is currently in his first year of MSc Study focusing on High performance packet classifiers at Rhodes University, within the Security and Networks Research Group (SNRG).