INVESTIGATING THE EFFECT OF GENETIC ALGORITHMS ON FILTER OPTIMISATION WITHIN FAST PACKET CLASSIFIERS.

Alastair Nottingham[1] **and Barry Irwin**[2]

Security and Networks Research Group
Department of Computer Science
Rhodes University, Grahamstown

[1]**anottingham@gmail.com,** [2]**b.irwin@ru.ac.za**

ABSTRACT

Packet demultiplexing and analysis is a core concern for network security, and has hence inspired numerous optimisation attempts since their conception in early packet demultiplexing filters such as CSPF and BPF. These optimisations have generally, but not exclusively, focused on improving the speed of packet classification. Despite these improvements however, packet filters require further optimisation in order to be effectively applied within next generation networks. One identified optimisation is that of reducing the average path length of the global filter by selecting an optimum filter permutation. Since redundant code generation does not change the order of computation, the initial filter order before filter optimisation affects the average path length of the resultant control-flow graph, thus selection of an optimum permutation of filters could provide significant performance improvements. Unfortunately, this problem is NP-Complete. In this paper, we consider using Genetic Algorithms to 'breed' an optimum filter permutation prior to redundant code elimination. Specifically, we aim to evaluate the effectiveness of such an optimisation in reducing filter control flow graphs.

KEY WORDS

Genetic Algorithms; Packet Classification; Permutation Optimisation

INVESTIGATING THE EFFECT OF GENETIC ALGORITHMS ON FILTER OPTIMISATION WITHIN FAST PACKET CLASSIFIERS.

1 INTRODUCTION

This paper details a preliminary investigation into the use of genetic algorithms in improving the efficiency and performance of complex packet classification tasks. This paper serves to motivate the inclusion of such techniques in the design of a GPGPU-based offline packet classifier, intended for fast classification of network telescope data. We are thus less concerned about filter update latency than we are about classification performance, as it is assumed that filter sets will rarely change. Furthermore, we are tolerant of the significant initialisation overhead required by genetic algorithm based solutions, if this may significantly improve performance, for obvious reasons.

In this section, we provide a brief overview of packet filters, and the specific optimisation which we intend to investigate.

1.1 A Note on Terminology

Packet filtering and classification may refer to a number of different, domain specific operations perfromed on packet data in order to derive or retreive useful information. These include, but are not limited to, IP routing, demultiplexing, analysis and intrusion detection [14, 12, 4]. In this paper, packet filtering and classification refer to the analysis of arbitrary packet header information within an architecture compatable with application level packet demultiplexing.

1.2 Brief History

The field of packet filtering and classification has a long history of research and development, pioneered by the CMU/Stanford Packet Filter (CSPF), a memory-stack-based packet filter [10], and later by 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 [10].

BPF was succeeded by several similar packet filters, engineered to improve both classification efficiency and flexibility. This began with the Mach Packet Filter (MPF), targeted at the Mach micro-kernel, which introduced packet fragment handling and packet matching optimisations [16], and was followed closely 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 structure [11].

The successes of both MPF and PathFinder paved the way for the Dynamic Packet Filter (DPF), which leveraged 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, BPF+, in the form of JIT compilation [4]. BPF+ also introduced a significant number of concurrent and interdependent optimizations, including constant folding, predicate 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. The Extended Packet Filter (xPF) incorporated simple extensions for statistics collection into the BPF model [9], while the Fairly Fast Packet Filter (FFPF) used extensive buffering to reduce memory overhead, among other optimisations [5]. Finally, the Swift packet filter used CISC based pseudo-machine to minimise filter update latency, further reducing instruction overhead and command interdependence [15]. 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.

1.3 Problem Statement

The number of redundant operations that may be eliminated from a packet classification control flow graph is often dependent on the permutation of filters prior to optimisation. Thus, by finding an optimum permutation, a minimum control flow graph may be created, improving the efficiency of the filter program. As finding an optimum permutation of filters is analogous to the traveling salesman problem, there exists no polynomial time solution. In this paper we conduct a pilot study to assess the applicability and possible performance improvements that may result from implementing a genetic algorithm to approximate an optimum filter permutation.

1.4 Optimising Filter Permutation

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 [4]. 1 provides an illustration of a simplified filter program comprising three separate filters, represented as a control flow graph.

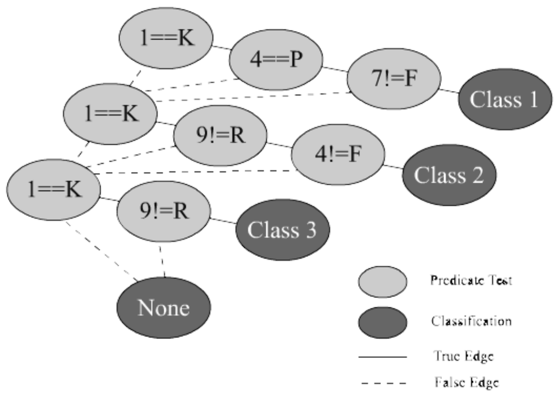

Figure 1: Filter Control Flow Graph

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 often dependent on the order in which header values are tested, which corresponds to the order of filters prior to optimisation [13]. 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 [10]. As constructing an optimal binary decision tree is NP-Complete [8], finding an optimum filter permutation is a non-deterministic operation. As an example, 3 shows how the optimisation results produced from two different permutations of the filters illustrated in 2.

While a set based heuristics solution which utilises an adaptive pattern matching algorithm to find a near-optimal decision tree has been detailed

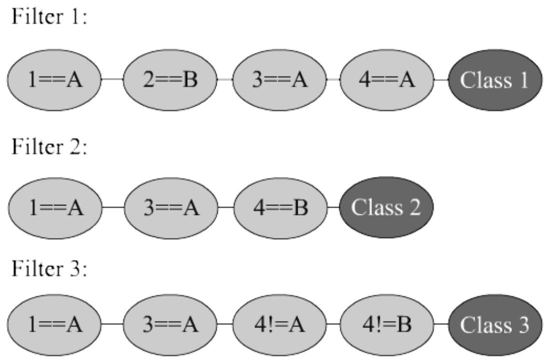

Figure 2: Filter Specification, adapted from [13].

in the literature [13], 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 genetic algorithm. As genetic algorithms have proven effective in NP-Complete problem spaces, including those analogous to the traveling salesman permutation problem [2], we intend to assess their effectiveness in finding an optimum filter permutation.

In this paper, we consider the feasibility of this approach by constructing a prototype simulation system to measure the effectiveness of the genetic algorithm optimisation in an abstract filter environment. As previously indicated, we are primarily interested in calssification performance, as it is assumed that filter permutations will remain relatively static, with billions of packets being classified using the same control flow graph.

2 SIMULATION SYSTEM

This section discusses the simulation system, a rapidly developed prototype used to guage the benefits and weaknesses of permutation optimisation using genetic algorithms.

2.1 Motivation

The simulation system is intended to aid in evaluating the potential for optimisation by altering filter permutation. To this end, it is necessary to specify a measure of performance that is not subject to run-time constraints, such

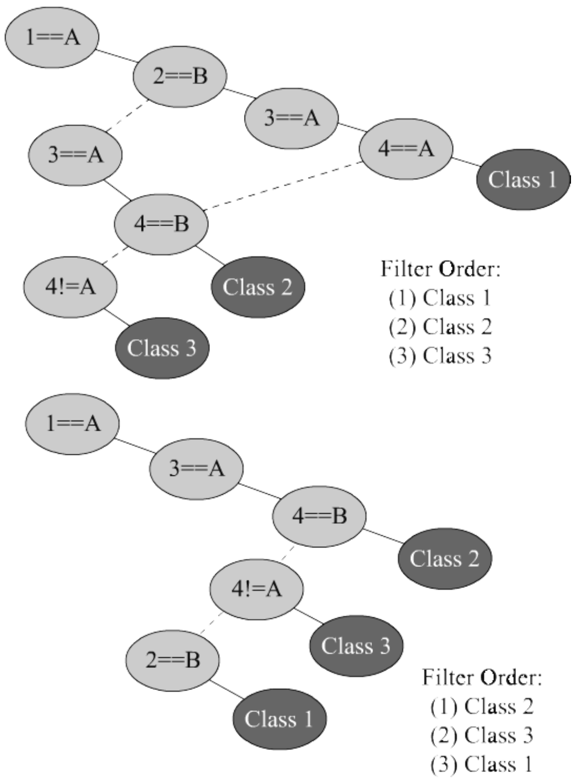

Figure 3: Control Flow Graph Reductions Using Different Permutations, adapteed from [13].

as CPU clock speed or memory latency. Noting that each tree will be generated from a permutation of filters, and that the optimisation techniques used cannot create new nodes or connections, but simply remove redundant ones, we can assume that the node count of each optimised filter tree provides a satisfactory indication of the performance of that filter tree, as it indicates the absolute difference in redundant nodes eliminated by optimisation.

Using the node count as our basis for comparison, we construct a simple, abstract control flow graph generator which accepts an arbitrary permutation of a filter set as input, converts the filter permutation to a control flow graph, and reduces the resultant graph using predicate assertion propagation and static predicate prediction [4]. By comparing the node count of two trees generated from different permutations of the same filter set, where one permutation is provided by a genetic algorithm, we can infer potential performance gains without having to implement a run-time filtering process.

The following sections consider the filter specification language and optimisation techniques used in generating the filter control flow graphs.

2.2 Filter Design

Prior to discussing the generation of filter sets in order to test optimisation, it is first necessary to elaborate on the design of the filter language used. Filters in the simulation system operate on packets containing a fixed length character array, or string. The character array is populated with random uppercase alphabetical characters, such that at any given index within the character array, there exists a random character that can be tested for equivalence against some constant character value. This forms the basis for the filter abstraction used.

Filters in the simulation system are equivalent to a chain of predicates, where each predicate compares a given packet index to a particular value, and returns true or false. In the interest of simplicity, only two comparative tests are available, namely equality and inequality. Should a predicate return true, the next predicate in the chain is evaluated until all predicates in the chain have returned true, at which time the packet is classified by the chain. If a predicate should fail, then the packet is not a member of the classification set, and is thus rejected by the filter, and processing begins on the next filter. If no filters match the packet, then the packet is not classified as a member of any set. See 1 for an illustration.

This filter design was adopted as it is both similar to typical packet filtering [13], and easy to map into a control flow graph. Furthermore, as the packets to be tested do not contain any inter-packet dependencies, in that every character in the packet is random, we are free to generate random filters without concern for relational constraints. This greatly simplifies the implementation of the simulation system, while providing for the trivial generation of random filters.

2.3 Automatic Filter Generation

In order to test the effectiveness of permutation optimisation in reducing the filter control flow graph, a set of filters is first required. In the interest of both generality and efficiency, we have implemented a filter generator capable of creating a filter set of arbitrary cardinality, composed of a bounded but variable number of filter predicates. In the interest of optimisation, filters may be generated such that the indices to be tested are sorted in ascending order, improving the effectiveness of optimisation [6, 4].

In typical filtering scenarios, large volumes of packets may share common tests [6, 4], such as for TCP or IP protocols, and thus a mechanism for ensuring a specified proportion of packets contain a particular test is desirable. The filter generator facilitates these requirements, allowing for an arbitrary number of predicates to be specified and associated with an occurrence percentage value. We term these *specified predicates.* Once all specified predicates have been processed, the remainder of the chain are populated by randomly generated predicates.

The automatic filter generation component thus allows for an arbitrary set of filters to be generated, with each filter containing a number of predicates within a specified predicate-chain-length range, and a user specified degree of overlap for specific predicates.

2.4 Filter Optimisation

The simulation system employs a subset of typical filter optimisations, focusing in particular on the reduction of nodes within the control flow graph through a combination of predicate assertion propagation and static predicate prediction. These optimisations require the calculation of the *node dominator relationship* [1, 4] between nodes, in order to ensure accuracy. A node n is said to dominate another node m if and only if for every path to node m, node n is in that path. If node n dominates node m, then the predicate in node n is known at node m regardless of the path taken. Note that by this definition, a node implicitly dominates itself, and all nodes are dominated by the root node of the control flow graph[1]. For our purposes, a nodes dominator set may be found recursively, by finding the intersection of the dominator sets of all parent nodes.

Predicate assertion propagation, in its most basic form, involves the use of predecessor dominator node predicates to eliminate redundancy within in a particular path. Specifically, if an edge from a node n points to a predicate node m whose result may be determined from the dominator set of n, then the node m may be bypassed by redirecting the edge from n to the appropriate child node of m [4]. The result of this process is the minimisation of redundancy within a particular path.

Static predicate prediction, for the purposes of our system, is similar to predicate assertion propagation, in that it uses the results of dominator nodes to ascertain the result of a predicate computation without evaluating the predicate. It differs in that, while predicate assertion propagation considers the explicit computational results of a dominator predicate in optimising a path, static predicate prediction infers an implicit computational result instead [4]. Specifically, if a predicate dominator node in a path returns false, the converse of the predicate is assumed to be true, and used as an optimisation parameter in the rest of the path. Together, predicate assertion propagation and static predicate prediction provide for significant optimisation opportunities.

For instance, if a dominator node n contains the predicate “5==J”, then predicate assertion propagation ensures that any redundant computation of this explicit predicate is removed from the path from the true edge of n. Similarly, static predicate prediction infers the implicit predicate “5!=J”, and attempts to remove redundant computation of this predicate in the path from the false edge of the node.

3 GENETIC ALGORITHM STRATEGY

In this section, we discuss the particulars of the genetic algorithm employed to test our hypothesis.

3.1 Introduction

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 [3]. 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 [3]. 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 randomly, 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 [3]. 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 [3].

As finding an optimum filter permutation is similar in both complexity and structure to the traveling salesman problem, we have applied a genetic algorithm to attempt to improve filter permutations such that the resultant control flow graph is minimised. In the following subsections, we discuss the specifics of the algorithm used.

3.2 Chromosome Representation

Before detailing the specifics of the employed genetic algorithm, it is first necessary to briefly describe our permutation representation. Chromosomes are represented as integer arrays, with a length equivalent to the total number of filters within the filter set. We then assign a unique numeric value to each filter, where each value corresponds to the filters position in the original filter set. The goal of our genetic algorithm is to permute these values into an optimum order, such that when the filters within the filter set are placed in the order of their identifiers within the chromosome, a near-minimal control flow graph is generated. Thus, the fitness of a chromosome is equal to the number of nodes eliminated from the resultant control flow graph after optimisation of the chromosomes filter permutation.

3.3 Selection

Chromosome selection is the process of selecting suitable parents from the current chromosome generation, in order to breed a new generation. The primary goal of the selection process is two-fold, namely improving upon current solutions, and exploring yet undiscovered solutions to a particular optimisation problem [3]. Numerous selection methods exist, from simple elitist methods which select the best performers from the chromosome population, to more sophisticated techniques such as adaptive selection and tournament selection [3]. For the purposes of our prototype, a proportional mechanism known as Roulette Wheel selection is used.

Roulette wheel selection is one of the most common selection mechanisms used today, and is considered as one of the simplest to meet the requirements of both improvement and exploration of the solution space. Simply put, roulette wheel selection is analogous to a roulette wheel, where the chance of selecting a chromosome as a parent is directly proportional to its fitness. To achieve this, the total fitness of the chromosome population is calculated by finding the sum of the fitness of each individual. We then designate a slice of this total to each chromosome, such that each slice is equivalent in size to the fitness of that chromosome. Finally, we randomly select a value between zero and the total fitness value, and determine the chromosome associated with the slice that value falls into.

While roulette wheel selection is sufficient for our purposes, more sophisticated methods may improve convergence to an optimum permutation, and the quality of the final result.

3.4 Crossover

The crossover operation is responsible for the re-composition of two parent chromosomes into their constituent child chromosomes. Much like selection, a myriad of crossover algorithms exist, each boasting particular strengths and weaknesses [3]. At its most simplistic, crossover involves splitting the parent chromosomes in a designated way, and then using the resultant pieces to create two child chromosomes, where each child contains pieces of both parents. For binary encoded problems, it is often sufficient to simply split each parent at a random point, creating two pieces, and then using these pieces to create the child chromosomes such that each child contains the first piece of one parent, and the second piece of the other parent. As this particular method is not well suited to permutation problems, we have opted for a more sophisticated crossover method, tailored for permutation optimisation. We detail this method below.

Recall that our chromosome representation is a simple array of unique numeric identifiers. Our first step is to compare both parent chromosomes, and locate all those identifiers which are in the same index position within the identifier array, and copy these values to the same position within both child chromosomes. This allows for the conservation of the most beneficial permutations. This leaves a set of n identifiers, where n is less than or equal to the number of filters in the filter set. We then take the first $\frac{n}{2}$ remaining identifiers from the first parent, and place them in order into the first $\frac{n}{2}$ available indices of the first child, and fill the remaining indices with unused identifiers in the order they appear in the second parent. This process is repeated for the second child, with the order of parents reversed.

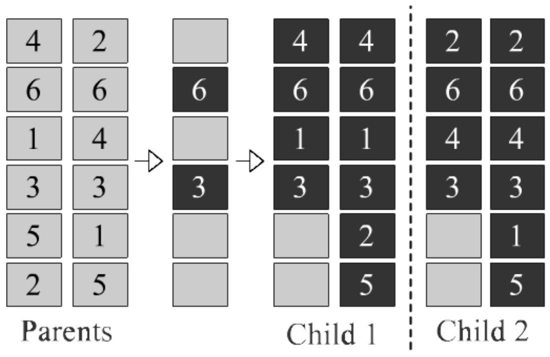

Figure 4: Crossover Operation

While crossover is of vital importance to the success of a genetic algorithm, it is often beneficial to allow a few parent chromosomes to survive intact into the next generation. To facilitate this, we use a crossover rate percentage of 70%, where the remaining 30% of crossovers result in child chromosomes identical to their respective parents.

3.5 Mutation

Mutation is responsible for small changes in children that are not inherited from their parents, and operates on individual components of a chromosome. Due to the purely random nature of mutation, the chance that a particular index of a chromosome is mutated, termed the mutation rate, is typically very small, to ensure that information inherited from parents is not regularly and excessively contaminated.

As mutation occurs at the component level within a chromosome, we iteratively cycle through the indices of the chromosome, allowing a 0.3% chance of mutation at each index position. This involves selecting another chromosome index at random, and swapping the filter identifiers contained within them. This ensures that no identifier occurs more than once within a chromosome, preventing corruption.

4 PRELIMINARY RESULTS

In this section, we discuss the preliminary findings regarding the use of a genetic algorithm to improve filter efficiency.

4.1 Filter Improvement

Of primary interest, with respect to this pilot study, is the potential for increased filter efficiency through permutation optimisation alone. In this regard, preliminary findings are optimistic, with notable improvements measured in a number of test cases. To investigate the hypothesis that permutation impacts significantly on filter permutation, we generate a random filter set of a specified size and composition, and compare the node count of the resultant control flow graphs constructed using both the original and optimised filter permutation configurations. To this end, we have created three distinct test sets, each containing several results. These are enumerated below.

The first test set is conducted on ten filters, with each filter containing five to ten predicates. There is an 80% chance that a filter contains the predicate "2==K". Five tests were run with the predicates in index order, with the other five in random order. Results are shown in figure 5. In almost all test cases, the genetic algorithm reduced the standard reduction node count by over 20%. Note that in the second trial, the genetic algorithm was unable to find a better permutation solution than that of the standard permutation, making it the best known solution.

The second test set is intended to illustrate performance over a large number of longer filters. Results, provided in figure 6, show considerable improvement in several trials, at the expense of increased computation time.

Finally, we consider a complex filter environment, lacking any explicit redundancy. Furthermore, both population size and generation count are increased, to obtain high quality solutions. While the processing cost was significant, resulting from the need to construct 125 000 relative complex

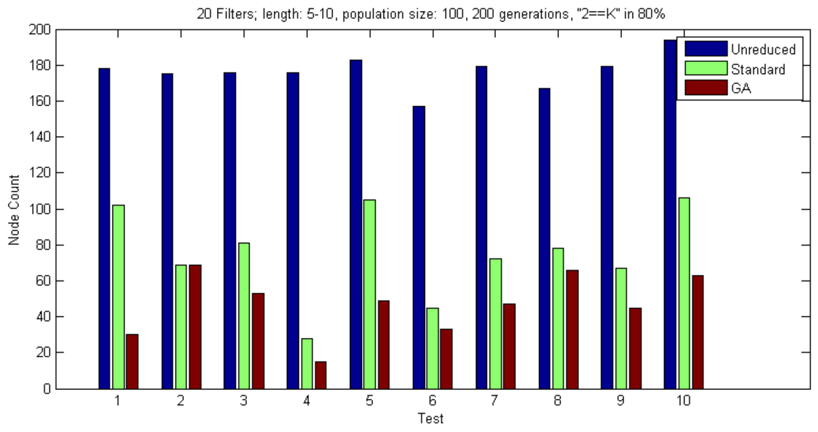

Figure 5: Test Set 1: 20 filters; length: 5-10; population size: 100; 200 generations; "2==K" in 80%

trees, as opposed to the 20 000 trees required by the algorithm in the second test set, considerable improvement is noted in several trial cases,as illustrated in figure 7. This implies that the algorithm performs well in complex environments, given sufficient computational resources.

4.2 Genetic Algorithm Performance Considerations

The genetic algorithm has shown promise in filter optimisation. However, the computation time necessary is roughly proportional to both the complexity of the filter environment, and number of independent chromosomes in the genetic algorithm, as these correspond to the time necessary to reduce a single tree, and the number of such trees that need to be constructed respectively. If the chromosome population size is m and the algorithm runs for n generations, then a total of nm trees need to be reduced in order to calculate node count. Thus, we may reduce the computational time necessary to complete permutation optimisation in two distinct areas, namely the time spent optimizing an individual graph and counting its nodes, and the number of chromosome evaluations performed before a suitable solution is presumed to be found. We consider these briefly.

By improving the efficiency of the tree reduction and counting mechanisms of the control flow graph, we may reduce the number of calculations by roughly kmn, where k represents the average reduction in graph optimisation and node counting cost.

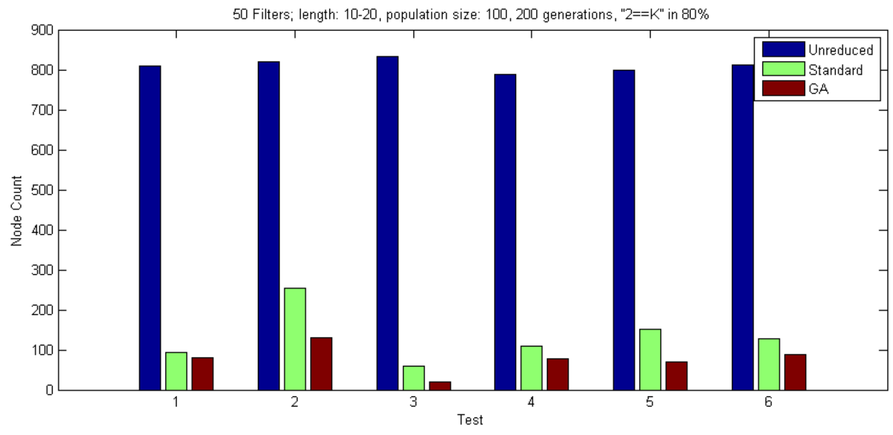

Figure 6: Test Set 2: 50 filters; length: 10-20; population size: 100; 200 generations; "2==K" in 80%

In order to reduce the number chromosome evaluations performed, we need to improve upon the values n and m. Firstly, the use of chromosome operators better suited to the task at hand is warranted. By utilising operators tailored to this problem, the rate of convergence to an optimum solution may be increased, reducing the requirements placed on both population size and generation count [3]. Specifically, Fuzzy Adaptive Genetic Algorithms (FAGAs) have shown significant potential in similar problem spaces, as they adapt breeding parameters dynamically in order to improve results [7]. This is, however, beyond the scope of this paper. Secondly, as genetic algorithms are parallel in nature [3], an implementation targeted at a parallel architecture such as multicore GPUs may reduce breeding time by a factor of m, as the entire population may be computed concurrently, spread over the number of cores available. An added benefit of this is that the population size may be increased to the number of cores available, without incurring significant delay.

While minimising filter generation time is important to some degree, rapid generation is not imperative, given its intended purpose as an offline packet classifier. Thus, we consider a generation time of several hours to be undesireable, but acceptable.

4.3 Limitations of Findings

As this paper represents a pilot study into the applicability of genetic algorithms to filter permutation optimisation, several limitations are evident.

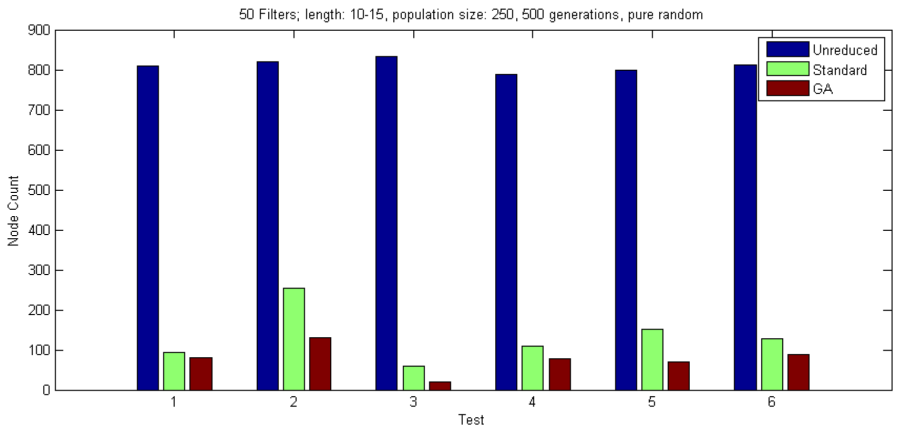

Figure 7: Test Set 3: 50 filters; length: 10-15; population size: 250; 500 generations; pure random

Firstly, we have considered the final node count within a control flow graph as the only measure of chromosome fitness. While a reduced final node count is an acceptable indicator of improvement, such improvement is not guaranteed. Numerous factors affect control flow graph efficiency, including the average path length, and the composition of incoming packets. While we have not used these measures to guage fitness within our pilot genetic algorithm, such metrics may be incorporated into a more sophisticated implementation.

Secondly, due to the simplicity of the implemented filter system, results only indicate an approximate potential for improvement. Given that an actual packet filtering system relies upon numerous and diverse predicate and computational operators which may ultimately influence the level of success attainable by optimising filter permutation, our results do not guarantee similar performance in an actual filtering system, but simply indicate that such performance improvements are conceivably possible.

Finally, we note the limitations of the simple filtering system implemented. As the generation of filters is performed at random, the possibility of generating overlapping definitions, or unreachable classifications, is not only possible, but highly probable in large filter sets, given the limitations on indices, comparison operators, and character values. This further implies that a change in permutation may ultimately change the classification of a packet, if two similar filters have their order reversed. While such instances are of concern, in an actual filtering system they may be mitigated through the use of both detection functions, and the ability to enforced ordering of filters.

5 CONCLUSION AND FUTURE WORK

The simulation system discussed in this paper, while limited in many respects, demonstrates the possibility for significant filter efficiency improvement through the application of a well tailored evolutionary permutation optimisation approach. The simple genetic algorithm employed to test this hypothesis produced numerous control flow graphs containing less nodes than their unoptimised counterparts, illustrating the potential of such an approach in a real packet classification system. Given the breeding overhead required by genetic algorithms, such techniques may not be beneficial in dynamic filter environments, but is well suited to those in static environments. We thus intend to apply this knowledge to the development of a packet filter architecture which efficiently leverages genetic algorithms in unison with GPU processing, with the express goal of improving offline packet classification performance in complex filter environments.

References

[1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison Wesley, August 2006.

[2] Back, T., and Hoffmeister, F. Adaptive search by evolutionary algorithms, 1992.

[3] Bck, T. Evolutionary algorithms in theory and practice, Feburary 1994.

[4] Begel, A., McCanne, S., and Graham, S. L. Bpf+: exploiting global data-flow optimization in a generalized packet filter architecture. *SIGCOMM Comput. Commun. Rev. 29*, 4 (1999), 123–134.

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

[6] Engler, D. R., and Kaashoek, M. F. 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, 1996), ACM, pp. 53–59.

[7] Herrera, F., and Lozano, M. Fuzzy adaptive genetic algorithms: design, taxonomy, and future directions. *Soft Computing - A Fusion of Foundations, Methodologies and Applications 7*, 8 (August 2003), 545–562.

[8] Hyafil, L., and Rivest, R. Constructing optimal binary decision trees is np-complete. *Information Processing Letters 5* (1976), 15–17.

[9] Ioannidis, S., and Anagnostakis, K. G. 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.

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

[11] McMurchie, L., and Ebeling, C. Pathfinder: A negotiation-based performance-driven router for FPGAs. In *FPGA* (1995), pp. 111–117.

[12] Taylor, D. E. Survey and taxonomy of packet classification techniques. *ACM Comput. Surv. 37*, 3 (2005), 238–275.

[13] Tongaonkar, A. S. Fast pattern-matching techniques for packet filtering. Tech. rep., 2004.

[14] Vasiliadis, G., Antonatos, S., Polychronakis, M., Markatos, E. P., and Ioannidis, S. 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, 2008), Springer-Verlag, pp. 116–134.

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

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