A Programmable Architecture for Scalable and Real-time Network Traffic Measurements

Faisal Khan, Lihua Yuan, Chen-Nee Chuah, Soheil Ghiasi
University of California, Davis
{fnkhan,lyuan,chuah,ghiasi}@ucdavis.edu

ABSTRACT

Accurate and real-time traffic measurement is becoming increasingly critical for large variety of applications including accounting, bandwidth provisioning and security analysis. Existing network measurement techniques, however, have major difficulty dealing with large number of flows in today's high-speed networks and offer limited scalability with increasing link speeds. Consequently, the current state of the art solutions have to resort to conservative sampling of the traffic stream and/or accounting for only a few frequent flows that often fail to provide accurate estimates of traffic features.

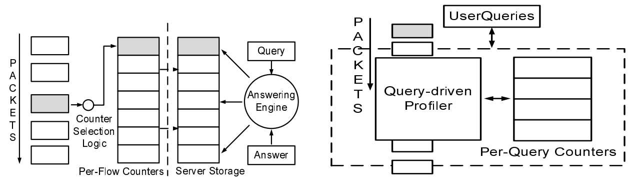
In this paper, we present a novel hardware-software codesigned solution that is *programmable* and *adaptable* to runtime situations offering *high-throughputs* that can easily match current link-speeds. The key to our design is orthogonalization of memory lookups from traffic measurements through our query-driven measurement scheme. We have prototyped our approach on a Xilinx platform using Microblaze soft-core processors integrated with Virtex-II Pro FPGA fabric. We demonstrate the scalability of our architecture and also compare it with a recent offline (non real-time) sampling-based software alternative. The comparison shows that our architecture performs orders better in terms of speed and throughput even while being used as an offline solution.

1. INTRODUCTION

Accurate traffic measurement and monitoring is keystone in a wide range of network applications such as detection of security attacks, traffic engineering, accounting and anomaly detection. A number of network management decisions such as blocking traffic to a victim destination, re-routing traffic, charging customer, or raising alarms to administrators, require extracting real-time statistics of network traffic. A network measurement tool is crucial in judiciously making such decisions [1, 5].

Fundamentally, traffic measurement involves counting number of packets that satisfy some criteria, commonly referred to as *user query* or a *rule*, over a period of time. The traffic is measured in terms of *flows*, where a flow refers to a set of packets that have the same n-tuple value in their header fields. Typical definitions of flow include the 6-tuple: $\{prt, tos, sip, spt, dip, dpt\}$ where, prt is the protocol field, tos is type of service, sip and dip are the source and destination IP addresses and spt and dpt are the source and destination ports, respectively.

Traditional measurement schemes work by maintaining unique "per-flow" based counters on high-density storage media, followed by aggregation of selected counters to answer queries [5]. This mechanism is illustrated in Figure 1(a). Note that there is an inherent and increasingly-widening performance gap between high-density storage access time and network bandwidth. Furthermore, it requires too much processing and I/O overhead to maintain such a paradigm. As a result, traditional schemes do not offer scalable measurement solutions for high data rate networks and have to resort to sampling [13]. The sample is processed *offline* to extract any meaningful information from the collected statistics. Cisco NetFlow [1] is one such widely deployed sample-based traffic measurement solution.



(a) Per-flow statistics collection (b) Query-driven measurement

Figure 1: Measurement Architectures

The key issue with sample based solutions is measurement accuracy [2, 9]. We argue that a scalable[1] solution for real-time and accurate measurement of traffic has to dispose of conventional "per-flow" -based statistics collection. Instead, we propose a "query-driven" measurement methodology that works by profiling passing traffic according to the given query to collect information of interest in real-time. Unlike existing techniques, our solution processes streaming

[1]We use scalability to refer to both capability to handle increasingly many live flows, and implementation feasibility of an architecture with increase in number of components. The context should clarify the exact reference in each case.

ANCS'08 November 6-7, 2008, San Jose, CA, USA.
Copyright 2008 ACM 978-1-60558-346-4/08/0011 ...$5.00.

traffic at link speeds and hence, does not compromise measurement accuracy due to sampling. Figure 1(b) depicts the idea of our proposed solution.

We develop a novel hardware-software co-designed architecture to evaluate our idea. The architecture contains a highly-parallel and scalable array of processing elements that are dynamically programmed according to user queries. The traffic flows through the array at link speed, and information of interest are collected by processing elements. The control unit of the architecture coordinates aggregation of collected information and responding to queries. Furthermore, it oversees reconfiguration of processing elements.

We implemented our architecture on a commodity field-programmable gate array (FPGA) running at modest 100-MHz frequency. We present analytical evaluation to guarantee high accuracy of measurement results in face of processing element reconfiguration latency. Also, we report empirical evaluations to confirm accurate processing of traffic at multi-Gbps link speeds. In addition, we present experimental results to explore the design space of our architecture, and to showcase improvements of our system over an offline software-based counterpart. We further show a case study that utilizes our query-answering engine in identifying heavy (frequent) flows in the network.

2. RELATED WORK

The per-flow based method has been employed in several measurement tools like NeTraMet [5], FlowScan [19], sFlow [18] and Cisco NetFlow [1]. The scalability issues associated with Cisco Netflow [1] have been discussed using static and dynamic adaptive probabilistic sampling in Sampled [2] and Adaptive Netflow (ANF) [9], respectively. Sampling, however, captures arbitrary statistical and systematic variations in the offered traffic and favors large flows. The issue of biasness is alleviated by utilizing sample and hold method, where a flow is sampled based on a bernoulli trial [11]. Another interesting solution proposed is to smartly sample an arbitrary small number of flows that are “best” representative of the traffic stream [8,15]. Some other elegant solutions work by extrapolating the collected sample using statistical inference [7]. Our proposed technique differs from above as we advocate a non-sampled real-time streaming or online solution which does not suffer from inaccuracies of sample based offline solutions.

Due to scalability issues associated with maintaining per-flow schemes, many researchers argue that it is infeasible to accurately measure each and every flow on high speed links. Instead, they advocate focusing on few most frequent flows [11,12]. Detecting such *heavy hitters*, or elephant flows, is an important monitoring task for both security-purposes (example, denial-of-service attacks) and traffic engineering (example., route selection). Work has been done in producing traffic summary or identifying hierarchical heavy hitters. Aguri [6] and Autofocus [10] are traffic profilers that aggregate flows and report if they are above certain threshold. These are offline tools that work by aggregating per-flow statistics bottom-up, i.e. from longest prefix to the shortest. In a DDOS type attack where there could be substantial number of low frequency flows (mice), the bottom-up strategy involves storage and aggregation over huge number of flows before the victim can be isolated. This problem is studied by Zhang et al. in [23] by treating flow accounting problem as IP packet classification problem [20] using a top-down multibit-trie based approach. However, the tries require multiple memory accesses that makes the method infeasible for real-time processing while using DRAMs at current link-speeds [20]. Some other elegant solutions are based on coincidence [14] that exploit the fact that one is more likely to observe n consecutive packets from the same flow if the flow is large or long-lived. These techniques also favor large flows without knowledge of user requirements.

A recent solution, ProgME [22], addresses programmability and scalability by aggregating flows based on user's queries. Our technique differs from ProgME that we do away with its computational paradigm that is tuned for software based implementation. Another solution [21] performs similar aggregation for collecting distributed statistics on network processors. Instead, we employ a novel query-driven architecture as shown in Figure-1(b) that can perform network measurements at a node on passing by traffic in real-time.

3. MOTIVATION AND BACKGROUND

3.1 Motivation and Challenges

The core problem faced with per-flow schemes to operate in real-time is their inability to fit the flow statistics in faster SRAMs. The SRAMs sizes are far too small and their density does not scale with access times [4].

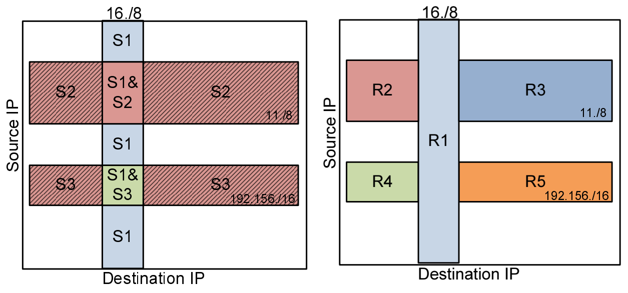
The query-driven measurement solution addresses the scalability problem by only maintaining statistics for what is needed by the network managers. However, by bringing up-front the query-processing stage, the incoming packet stream may now need to be evaluated with potentially multiple queries until a match is found. Once a match is found, its corresponding record(s) needs to be updated in the memory. Thus the query-driven approach trades off memory scalability with increase in computing requirements per packet. The computations and the subsequent memory updates also need to complete at latencies driven by link-speeds for the traffic measurement system to operate in real-time.

If packet arrival rate is R (in gigabits per second), the minimum packet size being P bits, and C counters are updated by Q queries with processing overhead of T ns/query, the memory needs to respond (for both read and write) in $(P/[R(Q+2C)]-T)$ns. Let's consider an example of minimum sized TCP headers of 40-bytes coming on OC-192 link with link-speed of 10-Gbps. This gives 32ns to process the packet. On a 2-GHz CPU, this translates to 64 clock cycles to do the processing using which only a small set of queries can be processed in real-time. In contrast, the number of queries are increasing at a constant pace. A uni-processor model thus faces query-scalability problem using query-driven measurement. This coupled with difficulties in programming multi-core solutions [16], some researchers argue employing hardware solutions for network measurements [17]

4. PROBLEM DEFINITION

Definition 1 (Superflow)**.** A superflow is a set of contiguous flows that share portion of their CIDR prefixes. More precisely, they define a range on the values of their individual flow dimensions that they share as a superflow.

As an illustration, the regions S_1, S_2 and S_3 in Figure-2(a) describe three sets of contiguous flows representing corresponding CIDR prefixes. User queries are usually composed of a combination of such prefixes. For example, a query-q may require statistics for all the flows passing through both the superflows S_2 and S_3 except those that also match S_1. However, the semantics of superflows are restricted by well defined structures that can only describe contiguous set of flows that are described by regular CIDR prefixes. Query-q cannot be described using any single superflow and would require complex computations if superflow based aggregation is used. We hereby define a new concept, *superegion*, that addresses the shortcomings in superflow based representation.



(a) Superflows (b) Superegions

Figure 2: Visualization of Suprflows and Superegions in 2-tuple header-space

Definition 2 (Superegion)**.** A superegion is a set of contiguous flows that can entail any irregular shaped n-dimensional structure in an n-tuple header space.

As an illustration, the query-q above can now be described as a union of superegions, R_2, R_3, R_4, R_5, shown in Figure-2(b). This division is however arbitrary which in the given case is tuned for quick answering of query-q as described in Table-2(b). Note that all superflows can by definition be described using superegions but the reverse is not true.

4.1 Problem Statement

Given an input stream $I = \{(k_i, u_i)\}$, and user query $\{(R_i, \tau_i)\}$ where R_i is a set of flows and $\tau_i \in \mathbb{N}$ is time allocated for answering the query, the query(rule) answering problem is to find SUM$= \sum_i u_i$ for all $k_i \in R_i$ such that time, $t \leq \tau_i$

The problem at hand is to design a network measurement system that is scalable, efficient and has the ability to process user queries in real-time and online on streaming network traffic. We can then use this platform to explore other networking problems.

5. PROGRAMMABLE SOLUTION FOR NETWORK TRAFFIC MEASUREMENTS

In this section, we describe the details of our solution for the problem of real-time traffic measurements. To the best of our knowledge, we have developed the first system that not only supports real-time network measurements but can also admit new queries during normal operation.

We addresses the bottleneck posed due to query lookups by hardwiring them in a custom processing engine. The problem due to slower DRAM updates is tackled using a two-pronged strategy. First, we reduce the number of updates using our query-driven approach. Secondly, we parallelize the query processing such that the incoming data can be processed in real-time, thereby finenesses the need for its sampling. Consequently, our focus has been on simplifying the common case, i.e. the *flow identification* or query answering. We use superegions to define the rules that are not only more powerful than superflows but also simpler to process. Their only downside being extra checks required to process. However, we utilized the mutual exclusiveness of the checks to parallelize them in the processing engine. We will shortly discuss the details of evaluating a query when we discuss our datapath.

The measurement solutions detailed in literature mostly place their processing kernels in either a complete software or a hardware domain. However, we note that the problem has an interesting blend of contrasting requirements such as user level configurability as well as need for fast real-time response. Whereas configurability and control can be efficiently handled in software, a hardware domain is more appropriate for fast real-time processing. We therefore exploit the advantages of either domain by mapping the operations that are best suited for respective goals: a general purpose processor for processing control and configurations on a hardware datapath that matches incoming flows in real-time with the configured superegions. A high level diagram of our co-designed architecture is presented in Figure-3. We next discuss its details.

5.1 Parallel and Pipelined Programmable Architecture

The design presented in Figure-3 employs a high degree of parallelism that can be scaled according to the needs of the deployment. It consists of a control processor and a hardware datapath that is composed of parallel and pipelined *Processing Elements, PEs.* The synchronization between the datapath and the control unit is done using a *timer* while interaction between the cycle-accurate hardware and non-cycle accurate software is streamlined using the *Glue-Logic.*

The PEs are arranged in parallel and pipelined datapath and can be independently configured for concurrent matching of different superegions. The task level parallelism of a PE is combined with architectural pipelining to maintain scalability of the datapath. Note that a PE can independently keep track of a superegion. The pipelining ensures that data and control information streams through the PEs unidirectionally on separate channels. Thus datapath size alteration requires simple plugging-in or removal of PEs within a chain during design synthesis which has no effect on operating effectiveness of others. We will further discuss scalability of the datapath when we evaluate our design.

5.1.1 Control Processor

The control processor orchestrates both spatial and temporal allocation and programming of PEs to service incoming user queries. By orthogonalizing query-control from its corresponding statistics-collection, our architecture can be dynamically configured during normal operation to admit new user queries or switch its focus from one application to another. This dynamic change in behavior does not involve an FPGA (or device) level (re)configuration which is prohibitively expensive for networking applications. In con-

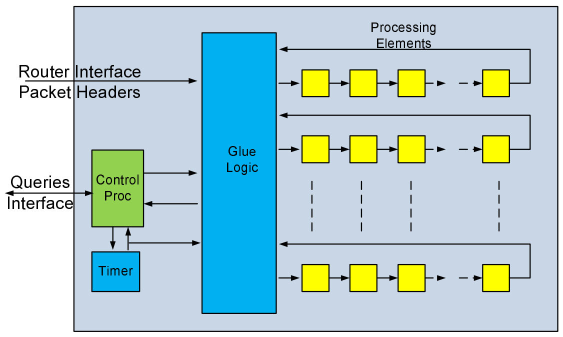


Figure 3: Architecture

trast, this is done through a relatively simple changing of values in PE registers. These changes are controlled by the control processor. We will discuss latency of these changes under reconfiguration-latency when we discuss the results.

5.1.2 Programmable Processing Element

We call the basic unit of hardware that can independently keep a log of a superegion as Processing Element. It keeps runtime statistics of incoming flow by using one or more 32-bit counters. PE architecture is shown in Figure-4.

PEs are parameterizable and programmable. They can be statically parameterized (static configuration) during hardware synthesis to employ 2^n number of counters and dynamically programmed by the control processor(dynamic configuration) during runtime to track different superegions. PEs can also be dynamically programmed to operate in either a single or multiple counter mode. A single counter can accommodate statistics-collection for a defined superegion. In certain networking applications, like *Heavy Hitter Identification* (discussed later), there is a need to track subregions within a superegion for finer resolution of statistics. The multiple counter mode is useful in such situations. When used in multiple counter mode, a PE auto sub-divides the search space into as many superegions, thereby avoiding allocation of dedicated PEs. We refer to the level of sub-divisioning as *zoom ratio (ZR)*.

The PEs have separate data and control channels. The data channel is 32-bits wide and passes-on the incoming data at the output irrespective of the PE's operating mode or state (discussed shortly). A PE also passes on the control codes and signals to the next PE in the chain unless it is waiting for configuration codes. By keeping data and control on separate channels, a PE in a row can remain active while other PEs in the same row are being configured.

The PEs also employ a *timer counter* that represents the number of interrupts (or timer expirations) after which a PE's collected statistics are due to be returned to the control processor. All the dynamic configurable parameters of a PE are controlled through the control processor.

5.1.3 Operation

A PE logs superegions by comparing incoming flows with the four configured end points of the superegion for a configured user defined duration. On a successful match, one of the 2^n counters in the PE is incremented. As mentioned earlier, multiple counters serve to sub-division a defined superegion. This sub-divisioning is further made dynamically configurable to be active or inactive. If enabled, it can be further configured to occur on either source or destination address space. A sample 4-way partitioning occurring at destination address space for the superegion $R2$ given in Figure-2(b) is shown in Figure-6. The selection of the partition is being done by utilizing some of the bits in the incoming flow. Exactly which bits are responsible for this selection is again programmed by the control processor. The partitioning and subsequent count update is controlled within the PE by *Counter Index Logic*.

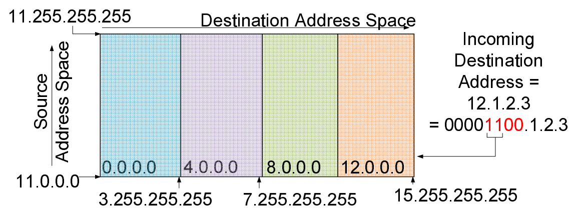


Figure 6: Auto Sub-Divisioning

5.1.4 Configuration

Configuration of a PE involves returning collected statistics and rearming it with new configuration information. As mentioned earlier, configuration of a PE is done dynamically through the control processor. A PE configuration consists of superegion boundaries (*S1, S2, D1, D2* in the figure), duration for which statistics are to be collected (Timer), weather the PE is to be configured for auto sub-divisioning or single count mode (*Is_HH*) and if source or destination space sub-division is used (*Is_SD, Shift*). The duration for statistics collection is encapsulated in a timer count value that represents number of interrupts to occur before the collected statistics of a PE are due to be returned at the control processor. Thus the user configured value set in the system timer serves as the minimum resolution at which the circuit can be observed and operated.

Configuration was one of the challenging tasks in our design. The difficulty arises as multiple PEs might be concurrently expecting configuration information or wishing to send their statistics on the same datapath channel. We solve the problem using our communication synchronization protocol which is a three state state-machine shown in Figure-5.

During the normal operation, a PE keeps collecting statistics corresponding to a configured superegion in the *Collect* state. Once the timer count expires, the PE statistics are due to be returned in *Send* state. In this state, a PE first checks to see if there are any incoming control values (*Incoming Ctrl* signal). These values may represent statistics being returned by PEs that are earlier in the chain to the current PE. The current PE first forwards these values and then sends its own statistics. Thus the values belonging to a PE ahead in the chain are also received earlier than the remaining at the control processor. When all the count values of a PE are passed-on to the next PE in the chain, the PE goes in *Rearm* state where it waits for control codes representing reconfiguration. Any control code that the PE now receives is assumed to be for itself and is not forwarded to the remaining PEs in the chain. In this way, the synchronization protocol ensures that reconfiguration also takes place in the same sequence as statistics were returned, i.e. the PE earlier in a chain is configured before the remaining.

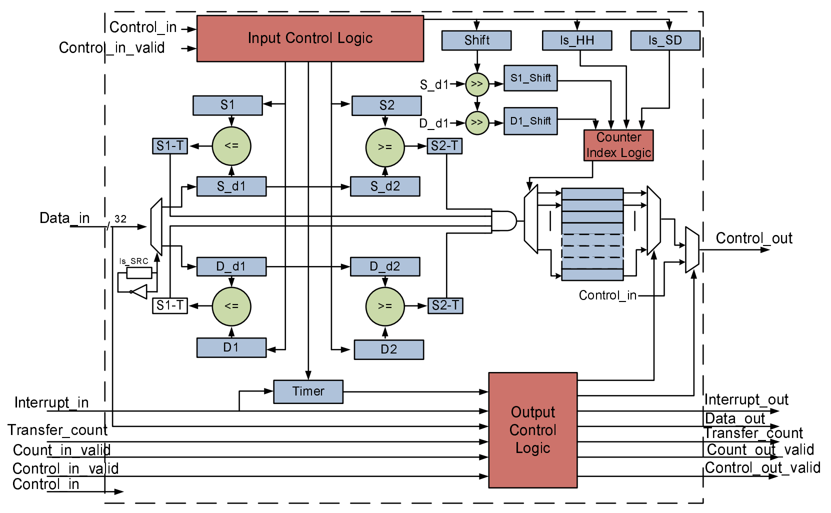
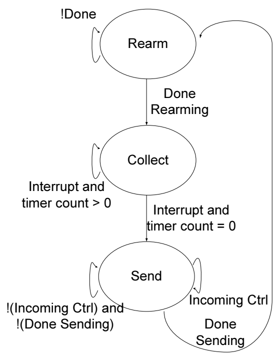


Figure 4: Processing Element



!Done
Rearm
Done Rearming
Collect
Interrupt and timer count > 0
Interrupt and timer count = 0
Send
!(Incoming Ctrl) and !(Done Sending)
Incoming Ctrl
Done Sending

Figure 5: Synchronization Protocol

5.1.5 Glue Logic

Besides configuration, another core problem we faced is the classic co-designed problem in interfacing fast, cycle accurate and parallel hardware with non-cycle accurate and slow software. The *Glue-Logic* addresses the problem by acting as a handshaking interface between the two domains and synchronization protocol (SP). It receives and acknowledges data and control codes from software-processor and passes on these values to the datapath-engine. As software-processor are slower than the hardware, they may take multiple clock cycles to send a value resulting in empty bubbles where no valid value is present in a cycle. The presence of these bubbles can fail the synchronization protocol and are therefore taken care of in the glue logic that sends the information to the datapath along with certain extra signals for proper interpretation of the information.

Besides acting as an interface, another function of the glue logic is to facilitate the control processor in properly configuring PEs. On occurrence of an interrupt, there could be multiple PEs in different rows that are ready to return their collected statistics. One naive way to interface the controller with the datapath would be to use as many return links as datapath rows. However this design not only uses redundant links, as the controller can only service one link at any given time, but is also unscalable[2]. We therefore use a single return queue interfaced to the control unit and employ an ordering sequence for row configuration. This sequence is dynamically calculated in the control unit and is sent to the glue-logic that uses this information to activate a required row in the datapath.

6. EMPIRICAL EVALUATION

6.1 Experimental Setup

A prototype of the presented design has been developed using Xilinx Virtex II 2VP30 Pro FPGA and Xilinx Embedded Design Kit (EDK). We used the FPGA for performing a rapid exploration of statically parameterizable components of the architecture. The EDK provides soft-core Microblaze processors that can be readily instantiated in the designs and mapped on FPGA devices. It further provides point to point Fast Simplex Links (FSLs) for communication that act like FIFOs whose sizes can be configured. We used EDK Version 9.1 to map our controller on a Microblaze soft-core processor and used FSLs to connect it with datapath logic. The prototype was operated at a clock frequency of 100-MHz

We employ data from CAIDA [3] to test our design and employed another Microblaze processor for mimicking network data source. The data processor reads in the dataset and sends it on to the datapath using an FSL. However since reading and writing comes with its own overhead, the network mimicker was seen to be utilizing only 20% of our datapath's processing capacity/bandwidth. We were also restricted on the 64k size of the memory that can be allocated to a microblaze and therefore used a part of the dataset of 5000 (source, destination) tuples. As part of our later experiements, we integrated CompactFlash with Microblaze processor and were able to test using datasize as big as $410,000$ packets. The query set consisted of randomly generated primitives that are maintained in the control processor. However, it should be noted that the data-source setup is only meant for comparison with a software alternative and is not a necessity for our design.

We selected ProgME [22] as our comparison baseline solution as it incorporates state of art in traffic measurement technology. For making the comparison fair, we employ an identical dataset on both ProgME and our architecture that is iteratively reprocessed until the queries are answered. However, there exist some core differences between the two solutions, like the usage of BDD based data structures, flowsets, by ProgME and superegions in our case. These along with some other minor differences make an accurate comparison between the solutions quite difficult. We will highlight the impact on the figures when we discuss the results. The results reported by ProgME were taken on an AMD Opteron-285 running at 2.6 GHz.

[2]For example, Xilinx's Microblaze processor has an upper limit of 8 point to point links

6.2 Design Space Exploration

6.2.1 Area

There are a number of statically configurable parameters in our design like the amount of parallelization and pipelining (rows and columns) of the datapath and the ZR (number of counters) of a PE. To limit the huge number of possible test cases, we synthesize an equal number of rows and columns and only vary the counters in the PE. We were not able to test cases employing 4 and 16 counters on the 5×5 architecture due to FPGA area limitations.

This area statistics are summarized in Figure-8 which shows logic utilization in terms of FPGA slices for different configurations. It can be noticed that a 3×3 array with 16 counters/PE approximately takes as many slices as a 5×5 implementation with 2 counters/PE. The question of either to use small number of counters with higher number of PEs or vice versa in a design is application dependent. We will shortly demonstrate potential uses of employing more PEs or counters.

6.2.2 Configuration Time

As discussed in section-5.1.4, (re)configuration involves collecting statistics from PEs and re-arming them with new boundaries by the control processor. The time it took for reconfiguring our various implementations is presented in Figure-9. Each point in the chart involves complete reconfiguration of the datapath, for example all the 9 PEs in the case for 3×3 implementation. It can be seen seen that configuration overhead increases significantly as ZR is increased to 16. This is due to significant increase in number of counters being returned in the latter case.

An important thing that can be deduced from the values in the figure is that our highest configuration time translates to missing out 160 packets[3], which is significantly smaller if complete FPGA were to be reprogrammed. Moreover, this is only a one time cost that can be further scaled down by overlapping configuration of a PE with statistics gathering using another identically configured PE.

6.2.3 Scalability

The scalability of our architecture depends on how area and processing overheads scale with an increase in the number of PEs and their counters. These are also presented in Figure-8 and Figure-9 for area and configuration time respectively. It can be seen that for different ZRs, the increase in either area or configuration time is sub-linear with the increase in number and size of PEs, or in other words, the initial cost of area and configuration time is higher than subsequent increments. The trend in area is due to the fact that a significant portion of the circuit, involving control logic and interconnections, remains ineffected with the increase in datapath size. Moreover, a PE's area also has a sub-linear growth rate with increasing ZRs as shown in Figure-7. The rate of increase in configuration time is sub-linear because a good chunk of processing by the control processor involving configuration is overlapped while the datapath is busy collecting samples. These sub-linear growth rates of area and configuration overheads demonstrate the scalability of our design.

[3]Assuming 500-bytes sized packets on OC-192 link

6.3 Performance

6.3.1 Speed

We next analyze the case where number of queries exceed the available resources in our prototype solution. Although we believe that this situation should not arise in the practice, as the system is to be pre-synthesized with as many PEs as there could be potential requirement, the case is however interesting as it would basically involve storing the dataset and its reiteration after re-arming the PEs with new superegions. In such a situation, the system would behave as any other offline solution.

We perform such a comparison between latency of our technique with ProgME using a dataset of 100-k headers as number of queries are varied. This is presented for our implementation in Figure-10. ProgME's computation paradigm makes it difficult to have a one-to-one comparison. The matching process takes place sequentially with a number of flowsets mapped to a hash table until a match is found. The depth of the match dictates the overall latency. This can be seen in Figure-11 where the x-axis represents the number of hash table entries compared for matching a flowset. The first case is very fast as it does not require building the BDD based representation. Although quite data dependent, it is expected that in steady state there would be around 4 candidates in the hash table which basically translates to a latency of 4 sec taken by ProgME.

6.3.2 Throughput

Our architecture takes 2 clock cycles to process a two-tuple header in real-time. Thus while processing as online measurement tool, the throughput of our prototype is 50 million packets/second which is more than enough to process in real-time at current link speeds as reported in Table-1.

We also compared steady state performance of our experimental prototype where data was sent using data mimicker. The throughput in this case was determined by the speed of the mimicker, which was quite slow. It was seen to be supporting 7.68 million headers/second. ProgME in contrast supported 50-k headers/second that essentially translates to a sampling factor of 50 on a 10-Gbps link using an average packet length of 500 bytes.

Finally, we analyze and compare throughput of our solution with ProgME when both are used as offline tools. The offline processing essentially means that we process a sampled snapshot of network data. This situation can arise if the prototype design have less PEs than number of queries, which would necessitate reiteration of the dataset and rearming its PEs with new queries in successive iterations. We realize such a situation using query-set of size 100 on our various prototype sizes. Such an analysis and its comparison with ProgME is presented in Table-2.

As expected, the throughput of our design is increasing with number of PEs. Though not shown in the table for brevity, increasing the number of counters/PE, or having a finer resolution on flow distributions in the assigned query space, has no effect on the throughput. Another important observation that can be gleaned from the table is that our design is orders (1-2) faster than ProgME despite using the slow data mimicker. Based on these observations, we contend that a designer would like to have a higher number of single-counter PEs in the design if her need is only to evaluate flow statistics streaming through the network. The

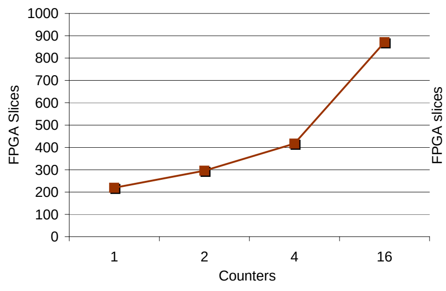


Figure 7: PE Area Comparison

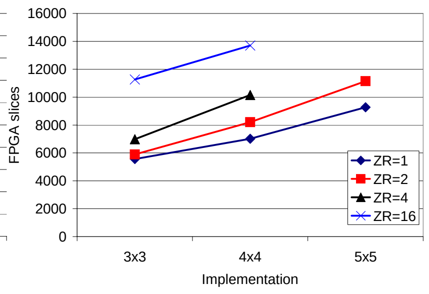


Figure 8: Area Scaling with ZR

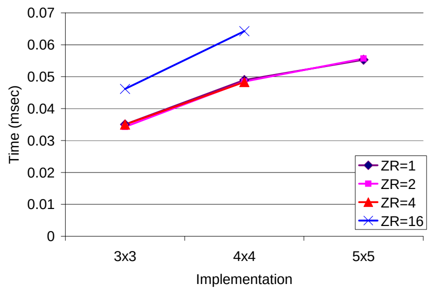


Figure 9: Configuration Time (per iteration) Scaling

Media	Link Speed (Mbps)	Packets / second (Millions)			
		PL = 40	PL = 500	PL = 1000	PL = 1500
OC3	150	0.4687	0.0375	0.0187	0.0125
OC12	625	1.9531	0.1562	0.0781	0.0521
OC48	2500	7.8125	0.625	0.3125	0.2083
OC192	10000	31.25	2.5	1.25	0.8333

Table 1: Data Throughput for different media and Packet lengths (PL)

ProgME	3x3	4x4	5x5
0.0616	0.6347	0.9902	1.7211

Table 2: Throughput Comparison for Offline Processing(Millions packets/sec)

cut-off value of the PEs is dependent on device limitations and user requirements.

7. CASE STUDY: IDENTIFYING HEAVY HITTERS IN PASSING TRAFFIC

Query processing, outlined in section-4.1, is the key function that an online measurement system has to perform at wire-speed. A system with real-time query processing capability can also be utilized to identify heavy hitters in passing traffic. Given a system that can answer queries, we utilize a recursive top-down query generation hueristic to detect heavy hitters. The scheme, called Multi-Resolution Tiling (MRT) algorithm [22], relies on the simple but powerful observation that if a superflow does not contain θ fraction of the entire traffic, then no flow of that superflow can be a heavy hitter. That is in search for the elephants, if a superflow is not a heavy hitter, all of its constituting flows can be discarded from further consideration.

An iteration of the algorithm is illustrated in Figure-12. Initially, the two-dimensional sample space is equally partitioned into four sub-regions using a *zoom ratio (ZR)* of four. Corresponding queries are generated to collect statistics for every sub-region. Only those sub-regions that exceed the threshold, marked with a cross in the figure, are selected for further zooming-in in the next-iteration. MRT thereafter continues iterating between partitioning, statistics-collection and zooming-in phases until a desired level of resolution is achieved.

We modify the MRT algorithm for our hardware-software codesigned architecture. We define regions of interest in our architecture using superegions. The superegions that meet the θ-threshold requirements are then queued in software that are later mapped to query answering hardware engine as the resources get available. Another interesting modification in our architecture is efficient use of sub-divisioning heuristic that helps expedite the HHI process. The heuristic makes intelligent use of ease in checking individual bits in hardware, thereby trading area with identification latency of a HH.

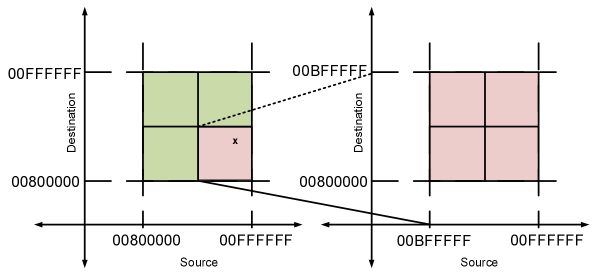


Figure 12: MRT with zoom ratio of four

7.1 Analytical Evaluation

In this section, we analytically discuss our design and provide measures to estimate its performance. The next section discusses our empirical performance. We first define some notation that is used in our subsequent discussion. Let

θ denote the threshold in terms of link usage.

T_s denote the inspection time during which incoming packets are logged.

T_c denote the configuration time during which the PEs are configured.

n_i^s represent the number of packets observed for superegion i during T_s

n_i^c represent the number of packets for superegion i that could not be observed during T_c

λ denote packet arrival rate (We assume Poison arrival).

λ_i represent the arrival rate of packets within a superegion i under consideration. Thus $\lambda_i = q * \lambda \leq \lambda$, where q represent fraction of total flows passing through the region under consideration. We refer to this parameter as the *flow ratio*.

Then the minimum number of packets needed for a flow/superegion to be declared as containing a heavy hitter is given by

$$N = \theta * \lambda * (T_s + T_c)$$

However, packets can only be observed during T_s, during which

$$N_i^s = \theta * \lambda * T_s$$

packets at least need to be logged in order to declare the presence of a heavy hitter.

As there could be mismatches between N_i^s and N, it raises the possibilities of false negatives and positives that are discussed next. Note that n_i^s and n_i^c are independent of each other according to the memoryless property of Poisson distribution.

7.2 False Positive:

A false positive is a false indication that a heavy hitter is found. It can occur if $n_i^s + n_i^c < N$ when ${N_i}^s \leq n_i^s$. Thus the probability of false positive can be given as

$$\begin{aligned}
P_p &= P[n_i^s + n_i^c < N | {N_i}^s \leq n_i^s] \\
&= \frac{1}{P[N_i^s \leq n_i^s]} * P[n_i^s + n_i^c < N \ \& \ N_i^s \leq n_i^s < N] \\
&= \frac{1}{\sum_{l={N_i}^s}^{\infty} P[n_i^s = l]} * \sum_{j=N_i^s}^{N-1} P[n_i^c < N - j \ and \ n_i^s = j] \\
&= \frac{1}{1 - \sum_{l=0}^{{N_i}^s - 1} P[n_i^s = l]} \sum_{j={N_i}^s}^{N-1} [\sum_{k=0}^{N-j-1} P[n_i^c = k]] * P[n_i^s = j]
\end{aligned}$$

using independence. After a few more steps we get

$$P_p = \frac{e^{-\lambda_i \cdot T_c} e^{-\lambda_i \cdot T_s}}{1 - e^{-\lambda_i \cdot T_s} \sum_{l=0}^{N_i^s - 1} \frac{(\lambda_i . T_s)^l}{l!}} \sum_{j=N_i^s}^{N-1} [\sum_{k=0}^{N-j-1} \frac{(\lambda_i . T_c)^k}{k!}] \frac{(\lambda_i . T_s)^j}{j!} \quad (1)$$

The numerator in the above equation defines probability that the threshold requirements were seen to meet during inspection time but it was an incorrect observation which is divided with the probability that the first argument holds, i.e. threshold requirements were seen to meet. This numerator itself defines probability of incorrectly detecting a heavy hitter and can be given as:

$$P'_p = e^{-\lambda_i \cdot T_c} e^{-\lambda_i \cdot T_s} * \sum_{j=N_i^s}^{N-1} [\sum_{k=0}^{N-j-1} \frac{(\lambda_i . T_c)^k}{k!}] \frac{(\lambda_i . T_s)^j}{j!} \quad (2)$$

7.3 False Negative:

A false negative is defined as incorrect absence of a heavy hitter indication during inspection period. This could occur when $n_i^s + n_i^c \geq N$ when $n_i^s < N_i^s$. Thus the probability of false positive can be given as

$$P_n = P[n_i^s + n_i^c \geq N | n_i^s < N_i^s]$$

and following the same line of derivation as used above gives

$$P_n = \frac{1}{\sum_{l=0}^{{N_i}^s - 1} \frac{(\lambda_i . T_s)^l}{l!}} \sum_{j=0}^{{N_i}^s - 1} [1 - e^{-\lambda_i \cdot T_c} \sum_{k=0}^{N-j-1} \frac{(\lambda_i . T_c)^k}{k!}] \frac{(\lambda_i . T_s)^j}{j!} \quad (3)$$

The numerator in the above equation defines probability of missing a heavy hitter which can be given as follows:

$$P'_n = e^{-\lambda_i \cdot T_s} \sum_{j=0}^{{N_i}^s - 1} [1 - e^{-\lambda_i \cdot T_c} \sum_{k=0}^{N-j-1} \frac{(\lambda_i . T_c)^k}{k!}] \frac{(\lambda_i . T_s)^j}{j!} \quad (4)$$

7.3.1 *Analysis*

The variations in false positive and negative are next plotted along with probabilities of missing-out detection of heavy hitter, i.e. numerators of the corresponding expressions for false positive and negative. These are presented in Figure-13 and Figure-14 respectively. The significance of these curves lie in identifying confidence of reported results. They moreover help a user to tune-in some key parameters in obtaining a desired level of accuracy from the architecture.

The curves are plotted using parametric values as tabulated in Table-3 unless given in the plots themselves. It can be noticed that polarities of false negative and positive switch at the threshold point. This is expected as it represents a true absence of alarm for false negative below the threshold point. Similarly, the shooting up of false positive alarm above the threshold depicts the probability of an incorrect alarm. Ideally, we would like to have sharp slopes of these curves at the thresholds to maximize the confidence. The sharpening of the slopes can be observed with increasing inspection times while having an inverse relation with packet lengths and configuration times. This is because with a greater inspection time, there is a larger collected sample using which a higher degree of confidence can be deduced. Similarly, increasing the average packet length or configuration time decreases the proportion of inspected traffic with respect to uninspected traffic that translates to lesser confidence level on the collected statistics. We did not plot the variations with increasing link speeds as their pattern can be easily deduced along the lines of above argument, i.e. with increasing link speeds, one should be seeing more steeper switchings quite similar to decreasing packet lengths as there would be now more packets to base a decision. This is significant as it shows that the performance of our design scales with link speeds.

It should be noted that a higher probability of false negative above the threshold point does not represent an inferior performance of our design. It only represents possibility that the absence of the alarm could be incorrect and has to be seen in conjunction with probability of actually missing out a HH (given in dotted lines in the graphs). A similar argument holds for the case of false positive and probability of incorrect HH detection (also plotted in the graphs).

The plots in Figure-13 show extremely low chances of evaluating an incorrect HH in our architecture (the dotted lines). Furthermore, it can be seen that an increase in inspection times to 1ms considerably reduces these chances which almost get reduced to zero for sampling time of 32ms. A similar observation can be seen for probability of missing out a heavy hitter in Figure-14(a).

7.4 Empirical Evaluation

We discussed how our solution performs when used as an offline or online measurement solution in section-6.3.2. Here we will detail processing time of our solution as an offline tool for HHI. This is plotted in Figure-15 where a sampled dataset of $5,000$ headers is reiterated for a given MRT iteration. However, we stress again that this setup is not a

θ	1%
T_c	0.05 msec
T_s	32 msec
Packet Length	500 bytes
Link Speed	10 Gbps

Table 3: Parameters

necessity of our design but merely done as a means for comparison with ProgME which utilizes a different algorithm for its MRT progression. A ZR of unity in the figure describes a design where partitioning or zooming-in is done using software by mapping different sub-regions at different PEs. We employ software zooming-in of ratio 2.

As expected, the delay is seen to be decreasing with increasing ZRs. However interestingly, the increase is not linear with corresponding increase in ZRs. ProgME in contrast was seen to be taking 963 msec with a ZR of 16 on an AMD Opteron-285 running at 2.6 GHz, which is still a magnitude more than our worst performing case.

7.5 Higher ZR or More PEs

The HHI problem demonstrated benefits of employing higher ZRs. However, a higher ZR comes with a price in area overhead as noticed in Figure-7. We note that a 3×3 datapath prototype with 16 counters takes approximately as much area as 5×5 datapath prototype using 2 counters. However, the performance of former is significantly better for HHI than the latter. This is in contrast to using a higher PEs and less ZRs type datapath for a query-answering type of application. A correct balance between ZR and PEs is therefore application dependent. We leave the selection of operating balance between the two at the discretion of the user.

8. CONCLUSION

We have presented a programmable, adaptable and scalable hardware-software co-designed solution for real-time network traffic measurement. We prototyped our design on a Virtex-II Pro platform operating at a mere 100-MHz and showed the superiority of our query-driven measurement paradigm against per-flow based solutions. We demonstrated heavy hitter identification problem as one of the potential networking problems to utilize the query-answering platform. To the best of our knowledge, this is first real-time network measurement solution, offering a high degree of accuracy and performance. Comparison with a recently proposed network measurement solution validates superiority of our prototype.

9. REFERENCES

[1] Cisco NetFlow. `http://www.cisco.com/warp/public/732/Tech/netflow`.

[2] Sampled NetFlow. `http://www.cisco.com/en/US/docs/ios/12\_0s/feature/guide/12s\_sanf.html`.

[3] CAIDA: Cooperative Association for Internet Data Analysis. `http://www.caida.org/home/`.

[4] B. Amrutur and M. Horowtiz. Speed and power scaling of srams. In *IEEE J. Solid-State Circuits*, volume 35(2), 2000.

[5] N. Brownlee, C. Mills, and G. Ruth. Traffic Flow Measurement: Architecture. RFC 2722, 1999. `http://www.ietf.org/rfc/rfc2722.txt`.

[6] K. Cho, R. Kaizaki, and A. Kato. Aguri: An Aggregation-based Traffic Profiler. In *Proc. Quality of Future Internet Services*, 2001.

[7] N. Duffield, C. Lund, and M. Thorup. Estimating Flow Distributions from Sampled Flow Statistics. In *SIGCOMM*, 2003.

[8] N. G. Duffield, C. Lund, and M. Thorup. Flow Sampling Under Hard Resource Constraints. In *SIGMETRICS*, 2004.

[9] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a Better Netflow. In *SIGCOMM*, 2004.

[10] C. Estan, S. Savage, and G. Varghese. Automatically inferring patterns of resource consumption in network traffic. In *SIGCOMM*, 2003.

[11] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems*, 21(3):270–313, 2003.

[12] C. Estan, G. Varghese, and M. Fisk. Bitmap Algorithms for Counting Active Flows on High Speed Links. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, 2003.

[13] W. Fang and L. Peterson. Inter-as: Traffic Patterns and their Implications. In *Proceedings of IEEE GLOBECOM*, 1999.

[14] M. Kodialam, T. Lakshman, and S. Mohanty. Runs bAsed Traffic Estimator (RATE): A simple, memory efficient scheme for per-flow rate estimation. In *INFOCOM*, 2004.

[15] A. Kumar and J. Xu. Sketch Guided Sampling – Using On-Line Estimates of Flow Size for Adaptive Data Collection. In *INFOCOM*, 2006.

[16] J. Mudigonda, H. M. Vin, and S. W. Keckler. Reconciling performance and programmability in networking systems. In *SIGCOMM*, pages 73–84, 2007.

[17] V. Paxson, K. Asanovic, S. Dharmapurikar, J. Lockwood, R. Pang, R. Sommer, and N. Weaver. Rethinking hardware support for network analysis and intrusion prevention. In *USENIX Hot Security*, August-2006.

[18] P. Phaal, S. Panchen, and N. McKee. InMon corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks, 2001. RFC 3176.

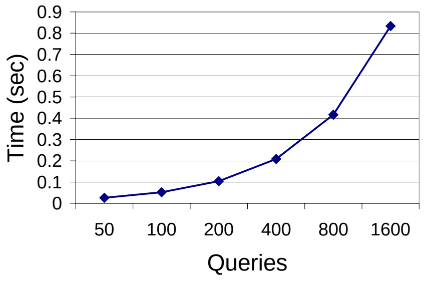
[19] D. Plonka. FlowScan: A Network Traffic Flow Reporting and Visualization Tool. In *USENIX LISA*, pages 305–317, 2000.

[20] G. Varghese. *Network Algorithmics*. Morgan Kaufmann, 2005.

[21] T. Wolf, R. Ramaswamy, S. Bunga, and N. Yang. An architecture for distributed real-time passive network measurement. In *MASCOTS*, pages 335–344, 2006.

[22] L. Yuan, C.-N. Chuah, and P. Mohapatra. ProgME: towards programmable network measurement. In *SIGCOMM*, pages 97–108, 2007.

[23] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund. Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications. In *IMC*, 2004.



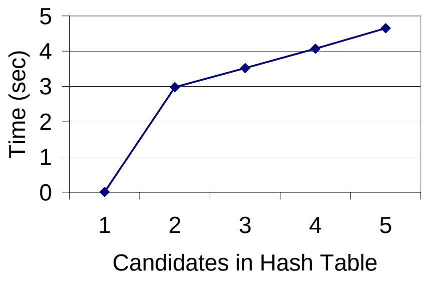
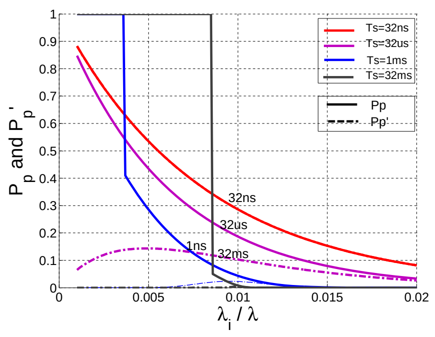
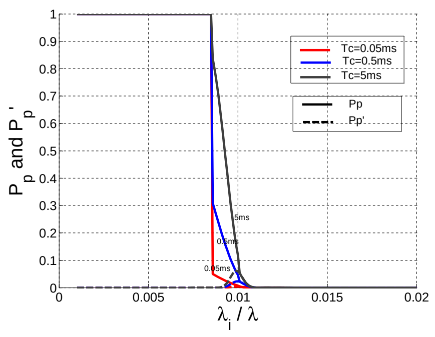


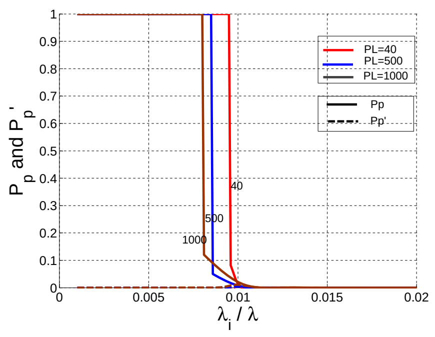
Figure 10: Queries and latency **Figure 11: Flowsets and latency**



(a) Varying Sampling Time

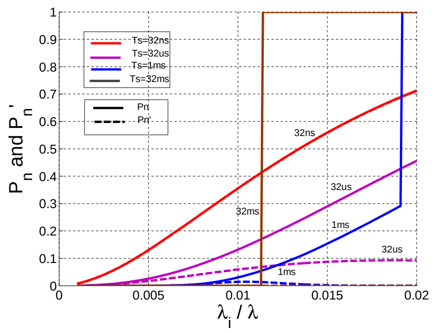


(b) Varying Configuration Time

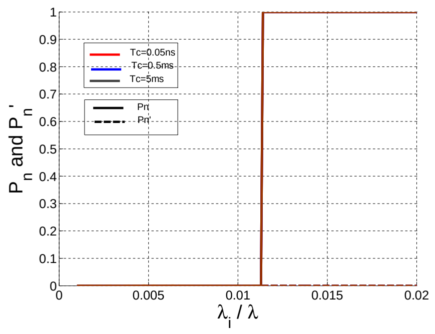


(c) Varying Packet Lengths

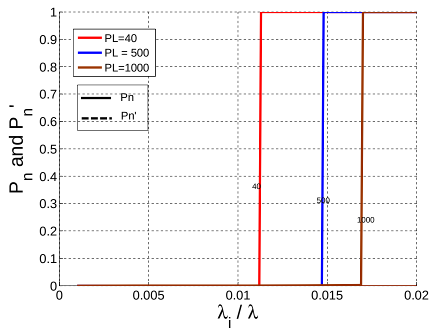
Figure 13: False Positives



(a) Varying Sampling Time



(b) Varying Configuration Time



(c) Varying Packet Length

Figure 14: False Negatives

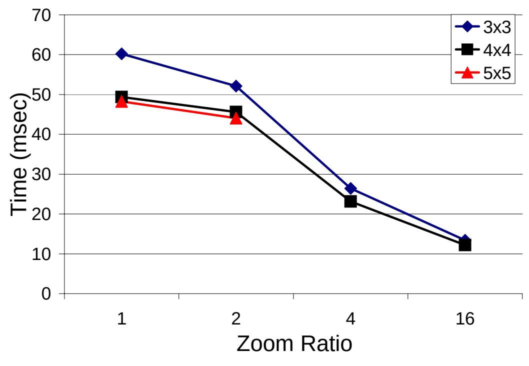


Figure 15: HHI Time Comparison