Introducing Scalability in Network Measurement: Toward 10 Gbps with Commodity Hardware

Loris Degioanni
Computer Science Department
University of California, Davis
One Shields Avenue – 95616 Davis, CA
ldegioanni@ucdavis.edu

Gianluca Varenni
Dipartimento di Automatica e Informatica
Politecnico di Torino
Corso Duca degli Abruzzi, 24 – 10129 Torino, Italy
gianluca.varenni@polito.it

ABSTRACT
The capacity of today's network links, along with the heterogeneity of their traffic, is rapidly growing, more than the workstation's processing power. This makes the task of measuring traffic more problematic every day, especially when off-the-shelf hardware is used. A general solution adopted by the computer industry to achieve better performance is to partition the processing among different computing units, exploiting the implicit or explicit parallelism available on today workstations. Parallelism is in fact growing in two dimensions: physical and logical CPUs (e.g. HyperThreading). Unfortunately, most network measurement systems are engineered to process data in a set of sequential tasks; thus, completely ignoring any form of parallelism provided by the hardware. This paper introduces a new approach to build high performance and scalable network measurement tools. It discusses the problem of dispatching packets to different processing entities and describes a technology able to distribute the flow of incoming packets among different processors in an effective and configurable manner, that avoids any copy and optimizes resource usage.

Categories and Subject Descriptors
C.2.3 [Computer Communication Networks]: Network Operations –*network monitoring*.

General Terms
Measurement, Performance.

Keywords
High performance, scalability, software tools.

1. INTRODUCTION
The need for powerful, and at the same time versatile, appliances for network measurement has grown significantly in the last few years. The demand for such systems and tools is driven by network operation, research purposes, and network security. Traditionally, networks have been measured in two different ways:

- Using software tools, running on off-the-shelf workstations and network cards. This approach, although cheap and versatile, is usually not considered suitable for high speed networks. The strength of purely software tools lies in the fact that people know how to use them effectively.
- Using custom hardware. This solution, although able to deal with the highest bandwidth networks, is usually very expensive and poorly versatile.

Recently, a novel hybrid approach has become popular: it is based on standard workstations supported by special purpose traffic acquisition cards. This solution grants the same versatility of purely software architectures while at the same time achieving excellent performance by off-loading the task of acquiring traffic to a customized piece of hardware. Endace Dags[1] are the best known capture cards available on the market.

Existing literature proposes several techniques to enhance the performance of network measurement. From the software point of view, past approaches focused on the lower level layers, i.e. the ones responsible for capturing the packets from the link. The best known are:

- Optimized filtering [2].
- Shared buffers between the capture engine and the applications [3].
- Interrupt delaying and device polling, in order to reduce (or even eliminate) the number of interrupts per packet [4][5][6].

From the hardware point of view, the main trend in the workstations and processors industry is to achieve better performance by exploiting parallelism, in the form of multiple physical CPUs or logical processing units per dye (HyperThreading).

It is clear that software and hardware improvements do not follow the same path. This is a major drawback, because all the approaches, whether purely software or hardware aided, reveal that the real bottleneck of real-time traffic measurements is represented by user level applications. The work presented in this paper aims at solving this problem by proposing a novel technique that is able to speed-up and grant a certain degree of scalability to PC-based network measurement. The proposed solution has been implemented in a device driver for Dag cards, and tested over a fully loaded OC-48 link.

IMC'04, October 25–27, 2004, Taormina, Silicy, Italy.

The paper is organized as follows: Section 2 briefly details the architecture of a Dag card, Section 3 describes the new SMP (symmetric multi-processor) Dag driver developed by the authors, which is able to distribute the load among many concurrent CPUs, Section 4 illustrates the problems which arise when multiple consumers need to share a common packet buffer, and describes a solution for such situations, Section 5 shows the results of a series of performance tests, and Section 6 discusses the conclusions and describes future work.

2. DAG CARD ARCHITECTURE

Dags are passive network measurement systems originally created by the Waikato Applied Network Dynamics (WAND)[1] research group at the University of Waikato. Basically, a Dag is a PCI network card that can be plugged in a traditional PC.

The architecture of a Dag from the software point of view is shown in Figure 1. The device driver is responsible of allocating a large contiguous section of physical memory, called a *memory hole*, which will be filled by the card with the network data. The size of this buffer is variable, but usually ranges between 32 and 128MB. When an application initiates a capture, the Dag driver performs two memory mappings: one to create a view of the memory hole in the address space of the application and another to export the card I/O space to the application. From that moment, the user-level program has complete control of the card and carries out the capture without any intervention from the kernel. The dashed arrow in Figure 1 indicates that the application has direct access to the card registers; thus, it is directly responsible for giving feedback to the hardware. Of course, this means that only one application can use the board at any precise moment.

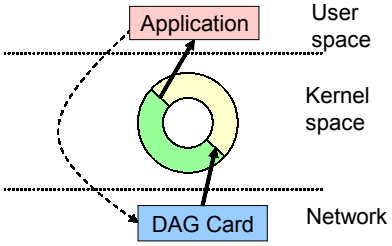

Figure 1. Dag software architecture.

During the capture, the card and the application share the memory hole: the card takes the packets from the network and puts them in the hole while the application removes them from the hole and processes them. No interrupts take place within this process. Therefore the application must poll the card registers to understand if new packets are available.

3. NOVEL SOFTWARE ARCHITECTURE FOR DAGS

This section describes a new experimental driver for Endace Dag cards, which distributes the processing load among the CPUs of a SMP workstation in order to improve the performance of the traffic measurement systems built upon this card while also adding a certain amount of scalability to the measurement process. The idea is to move some complexity back into the kernel, to increase utilization of the available hardware. The driver was developed to run inside Windows operating systems and supports Windows 2000, Windows XP, and Windows Server 2003.

Before describing this solution and presenting the results, it is important to point out that the concepts presented in this paragraph are general and fit most architectures for network analysis. We are currently working to apply the architecture presented in this paper to systems based on off-the-shelf network cards and the results seem to be encouraging.

3.1 SMP Dag Driver

The general architecture of the SMP Dag driver is shown in Figure 2. If we compare it with Figure 1, it is easy to see that two modules have been added at the kernel level: the *buffer monitor* and the *memory hole scheduler*. These two modules are integrated in the Dag driver.

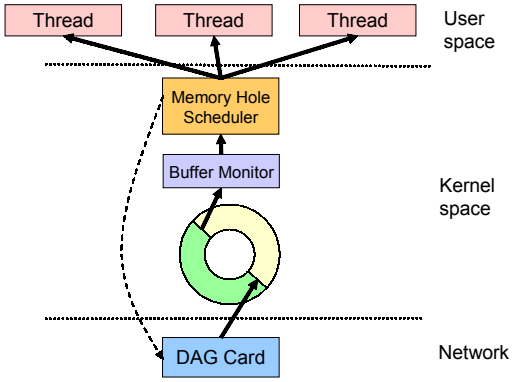

Figure 2. SMP Dag driver architecture.

The *buffer monitor* is a kernel thread, running in the context of the operating system. Its purpose is to periodically check the card registers in order to understand the availability of new captured packets from the network and the amount of these data. The *hole scheduler* manages the content of the buffer and splits it among different user level threads according to a specified policy. There are no constraints on the location of these threads: they can be in the same process or in different processes. The scheduler is essentially a state machine: it receives input, in the form of messages, from the buffer monitor and from user-level applications and produces output for the applications (typically buffers with a certain amount of packets) and for the card (the new position inside the hole). Different schedulers, each one implementing a different policy, can be present in the driver and it is possible to switch from one to another while the driver runs, using a registry key.

As depicted in Figure 2, the dashed line no longer starts from the applications. This means that the control of the card is now delegated to the scheduler, which in fact acts as a centralized card arbiter running at kernel level. The fact that the management of the memory hole is now centralized and arbitrated implies that there is no problem with having more than one application accessing it (the approach shown in Figure 1 clearly forces a single entity at the same time). Moreover, the modular architecture makes it easy to change the scheduler in order to satisfy different needs.

An important note is that the memory hole is still mapped in the address space of the user level processes. When a thread requires some data, the scheduler returns an offset inside the hole

and any process has a direct view over it. This means that the network packets reach the applications without being copied: such an approach raises some synchronization issues and requires careful engineering to avoid bottlenecks due to slow consumers (Section 4 will address this problem in detail). However, it is essential and unavoidable if high performance has to be guaranteed.

As shown in Figure 3, the SMP Dag Driver subdivides the memory hole into three sections: section A is the data currently processed by the consumers; the interval between `SchedEnd` and `CardPointer`, marked with the letter B, contains fresh data, i.e. packets that are waiting to be processed by the applications; section C, between `SchedStart` and `CardPointer`, is the free buffer space. Note that `SchedStart` is the stop point of the card, i.e. the last writeable position in the memory hole.

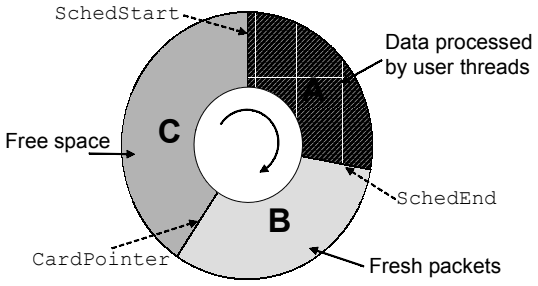

Figure 3. Memory hole organization.

Two different schedulers have been developed: the *Multiple Full-Delivery Scheduler* and the *Load-Balanced Delivery Scheduler*.

The Multiple Full-Delivery Scheduler delivers all the packets to all the consumers, so that anyone has a chance to process every incoming packet. The scenarios in which this scheduler can be adopted are the following:

- A multithreaded application that is able to split the processing into a certain number of lighter threads, each of them needing the complete network traffic. The typical case is an IDS (Intrusion Detection System) or a traffic measurement system that is able to split its rules into subsets, each of them being processed by a different thread.
- Several instances of the same network tool. In a configuration similar to the previous one, several measurement programs could be running on the machine, each one analyzing a different aspect of the traffic.
- Several different network tools. A common case is a user needing to run both a security and a traffic monitoring tool on the same machine.

The overall behavior of the Multiple Full-Delivery Scheduler is rather simple:

1. When the scheduler receives a `sc_newpackets` message from the hole monitor, it awakes every waiting consumer immediately.
2. As a consequence, every consumer starts to request new packets, sending a `sc_readdata` message.
3. The scheduler replies to this message by passing the calling thread a slice of the hole that starts at the last byte processed by the consumer and ends at `CardPointer`, i.e. all the data in the hole that the consumer has not processed yet.
4. The scheduler updates, if necessary, the `SchedStart` pointer.

The Load-Balanced Delivery Scheduler adopts a different distribution approach. When it is used, every consumer receives an independent slice of the memory hole. This means that a packet coming from the network is processed by one and only one consumer and the portions of the buffer processed by different user-level threads are never overlapping. The typical scenario in which this scheduler is adopted is a parallelizable tool that performs stateless analysis on the network traffic. The application spawns several threads, each one processing a subset of the total traffic: network monitoring tools, stateless firewalls, and traffic loggers are typical examples. The behavior of the Load-Balanced Delivery Scheduler is as follows:

1. When the scheduler receives a `sc_newpackets` message from the hole monitor, it cycles among the waiting consumers. It picks the consumer with lowest starting point (i.e. the one whose buffer starts at the offset `SchedStart` inside the memory hole) and awakens it.
2. On the counterpart, the consumer requests new packets as soon as it is ready, sending a `sc_readdata` message.
3. As a response to this message the scheduler returns the slice of the buffer that contains all the fresh data.
4. The scheduler updates, if necessary, the `SchedStart` pointer.

In practice, the unprocessed packets available in the hole will only be delivered to the first thread able to process them.

3.2 Location of Modules

A question that arose during the development was whether it was better to put the buffer monitor and the hole scheduler at user level or inside the kernel. We adopted the latter solution for the following reasons:

- In order to have packet distribution at user level, the equivalent of a daemon needs to be implemented. This requires an ad-hoc protocol to exchange data, the use of slow IPC primitives, and non-trivial inter-process memory mappings. Our kernel level implementation makes use of the standard and efficient OS I/O primitives. This makes the code simple and easy to integrate with existing systems like `WinPcap`[7].
- Handling hardware at user level is seldom a good choice, because of the stability and security implications that arise.

3.3 Interactions the Driver and Supported Programs

The application does not need to cope with messages, working modes, and all the other details needed to interact with the hole scheduler since a library is provided for this purpose. The name of this library is *Dag Capture Library*, also known as `dagc` library or `libdagc`.

Under Windows, `WinPcap` [7] has been extended to use `libdagc`. As a consequence all WinPcap-based tools (i.e. the vast majority of the Windows network tools, among which

`Ethereal` [8], `Snort` [9], and `WinDump` [10]) run over the SMP Dag driver without any change or recompilation since `libdagc` is a dynamic link library.

4. HOLE SHARING AND PERFORMANCE ISSUES

The first obvious drawback of the above system architecture is that it is limited by the speed of the slowest consumer. Since the memory hole is shared by all consumers, if the last running thread is very slow, it can be a bottleneck for all the other ones. This section proposes a solution aimed at mitigating the problem and describes how it has been applied to the Load-Balanced Delivery Scheduler. The evaluation of the working implementation will be presented in Section 5.

4.1 Solution Description

The basic idea is to give only a small, controlled, amount of all the fresh data available in the hole to each thread, so that slow threads do not block the buffer for a large amount of time.

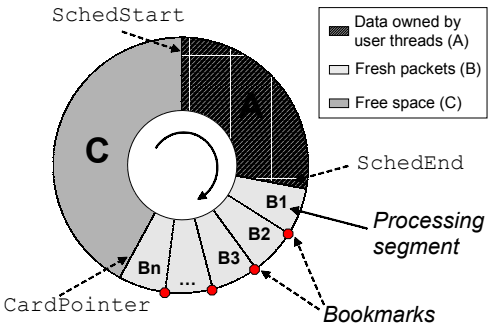

Figure 4. Overall memory hole organization with bookmarks.

If we consider the Load-Balanced Delivery Scheduler, given the basic structure described in previous paragraphs (see Figure 3), when a consumer requests a new slice of the memory hole, it receives all the fresh packets, i.e. the whole portion of buffer labeled with the letter B in Figure 3. If that portion contains a large amount of data, two drawbacks come up:

- The consumer spends a lot of time processing such data, blocking the forward movement of `SchedStart`, and possibly `CardPointer`. As a consequence the board cannot put any packet in the memory hole, even if the other consumers have already released their portion of memory, and the slow consumer itself has already processed a part of its slice but without updating the `SchedStart` pointer.
- When the other consumers finish their work, if the traffic is low they can remain without any packet to process, while one of the threads is busy with a large portion of the memory hole. In this case, parallelism is not exploited and most processing capabilities are wasted.

In order to solve these problems, the concept of *bookmark* has been introduced. As shown in Figure 4, a bookmark delimits a portion of fresh data in the memory hole, called a *processing slice*, whose size is defined as a constant when the scheduler is loaded. Bookmarks are created by the hole monitor thread that inserts each of them in a FIFO queue, called `bookmark table`. When a user-level thread requests some data, the Load-Balanced Delivery Scheduler checks to see if at least one bookmark is present in the `bookmark table`, extracts it, and returns the corresponding slice (marked as B1 in Figure 4) to the calling thread.

4.2 Impact of the Solution and Parameters Dimensioning

An obvious problem with this solution is related to how to calculate a proper value for the slice size. From a purely theoretical point of view, the smaller a slice is, the more granular the allocation of the packets in the memory hole to the threads processing them, thus exploiting the available parallelism in the best way. As a matter of fact, some factors impose a lower bound to the slice size: the overhead to obtain each slice from the scheduler, the memory required for the `bookmark table`, and the overhead in the hole scheduler to calculate the bookmarks.

The approach we followed to give an estimation of the acceptable values for the slice size was pragmatic. We compared the overall performance of two sample tests for different slice sizes and without the buffer slicing. The results of such tests are shown in paragraph 5.2.

A more formal approach to the problem is currently under development and will be presented in a future paper. It uses an analytical model to predict the slice size while taking into account consumers speeds and network bandwidth.

5. PERFORMANCE EVALUATION

This section presents the results of a series of tests carried out at the Endace labs in Hamilton, New Zealand, on different flavors of the Dag driver. The goals were:

- To understand the performance that a PC can reach as a network analysis station with the aid of a Dag board.
- To compare the efficiency and the scalability of the different Dag driver incarnations, including the SMP one described in this paper.
- To determine the impact of the buffer slicing technique on a real world implementation.

The test-bed is formed by a traffic generator and a receiver directly connected with a multi-modal optical fiber link. The traffic generator is a SmartBits 6000B, with an OC-48c output port. The receiver is a DAGMON network monitoring system. It mounts two 2.8 GHz P4 Xeon processors with Hyper-Threading technology: this means that there are four virtual processors. A Dag 4.3 card with an ATM/POS interface is plugged inside the PC through the 133 MHz PCI-X bus. The link between the two systems is an OC-48, i.e. 2488 Mbps. The SmartBits traffic generator is configured to send constant frame rate bursts of minimum-sized POS packets (42 bytes) in a Cisco HDLC encapsulation. The network load during bursts is approximately 100%: this configuration is able to produce slightly more than 6 millions packets per second to be fed to the DAGMON monitoring system. We did not use more complicated traffic patterns because the most interesting situation to measure the system is the worst one from the computational standpoint: high bandwidth with small packets.

5.1 libpcap Real-time Processing

This first test involves measuring the performance of a custom `libpcap`-based[11] application that runs on Windows

by means of the `WinPcap` library. The application under test has the traditional `libpcap` structure, with a callback function invoked for every captured packet. This callback performs several accesses to the packet data (15 scans of the whole packet) with some actual processing, recreating the situation of a demanding network tool. The whole content of the packets is captured and no BPF filters are set.

Note why the test did not make use of a real network tool (like `tcpdump`, `Ethereal` or `Snort`) is that these tools tend to execute a lot of display and I/O operations that hide the weight of actual packet acquisition and processing. Therefore, they are not suitable for the goals of this paper and have been excluded for lack of space.

The platforms under test used the following operating systems and software configurations: Debian Linux (this system is used as a reference since it is the one that Endace sells to its customers), Windows with the vanilla 2.5 Dag driver, Windows with a modified 2.5 Dag driver that includes the SMP support. The test has been repeated with a variable number of consumers, in order to understand the effective functionality of the kernel scheduler architecture. "Different consumers" in this case means that more than one `libpcap`-based application is running during the test. The architecture supports different threads in the same process as well but this situation is not presented here. A 32MB memory hole is configured under all the platforms. The results are shown in Figure 5 and deserve some comments.

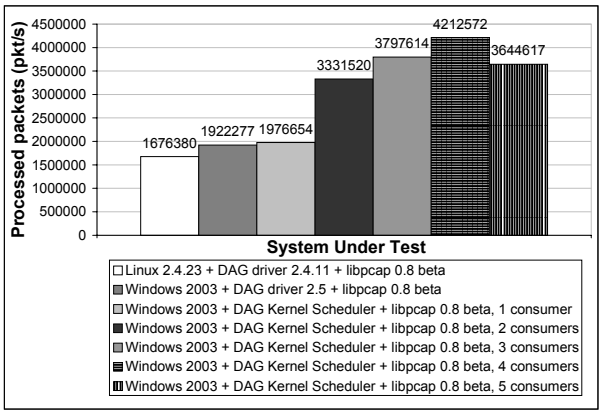

Figure 5. Performance of a libpcap-based application.

The first interesting result is that the SMP driver with a single consumer is a bit faster than the vanilla one. The first reason is that the subdivision of the memory hole into smaller slices gives better performance than an unregulated management. The second one is the increased locality yields better instruction cache usage. These factors compensate the overhead due to the increased synchronization.

With two consumers, the system scales very well and the number of processed packets grows approximately to 170%. With three and four consumers the speedup is constant but less remarkable. This behavior is consistent with other sources [12], which state that the actual speedup of the second hyper-threading processor ranges normally between 15% and 50%. With five consumers the performance starts decreasing, because their number is greater than the number of processors. In the best case, more than 4.2 million packets per second are processed and the overall improvement with respect to the original Endace driver is 2.5X.

5.2 Impact of buffer slicing

This paragraph shows the results of a group of tests that aim at measuring the influence of buffer slicing (see Section 3) on the overall system performance and behavior.

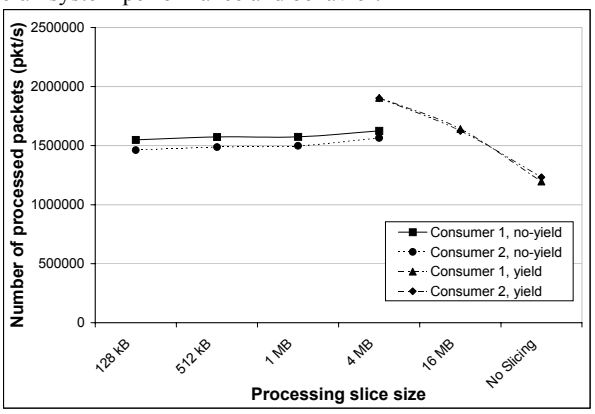

Figure 6. Performance of a libpcap-based application for different processing slices - two consumers.

Figure 6 shows the behavior of a system with two identical consumers. Each consumer is an instance of the same `libpcap`-based application used in previous tests. First let us explain what we call the "yield/no-yield" concept. As explained above, the hole monitor is implemented as a kernel-level thread, therefore it shares the processors with the other user-level applications without any specific privilege. Its purpose is to detect the amount of data present in the memory hole and this operation is carried out by periodically looking at a register inside the I/O space of the Dag card. Between reads of this register, the hole monitor is idle and sleeps. In the "yield" case, the hole monitor uses an operating system wait function, that causes a yield of that thread; in the "no-yield" case, the monitor uses a busy wait loop, instead[1].

The load is well distributed between the consumers: the performance increases to a processing slice size of 4MB, with a remarkable difference between the yield and no-yield version. At larger values, like 16MB, evidence shows expected performance reduction, which increases if no slicing is present.

The diagram in Figure 7 shows a system with two consumers with different speeds. The first of them is the same `libpcap`-based application used in the previous tests while the second is a very slow application that performs 50 scans of the whole packet, performing different kind of operations on the payload. The situation here is the opposite than in previous tests: the overall performance is higher with small slice sizes. In fact, smaller slices allow a better distribution of the hole content, while as the size increases, the speeds of the consumers tend to adapt to the slowest one.

[1] A more thorough treatment of the subject can be found in [13]

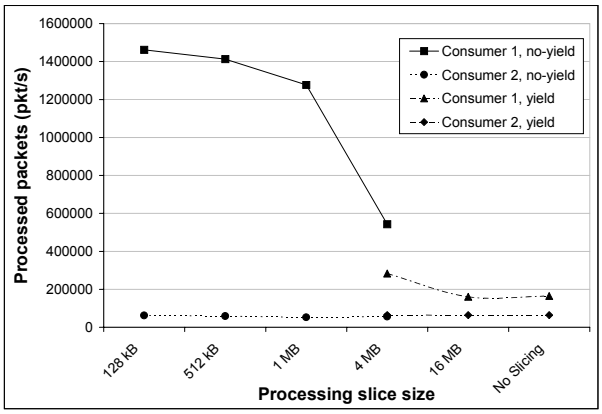

Figure 7. Processing performance of a `libpcap`-based application for different processing slices - two different consumers.

The difference between the yield and the non-yield version is remarkable again, but in this occasion the better one is the "no-yield" approach because it guarantees a more precise buffer distribution.

6. CONCLUSIONS

The work presented in this paper addresses the top of an iceberg that will necessarily emerge in the next few years. PC-based network monitoring and measurement tools (including free ones) are growing in quality and importance. They are widely adopted, their users are able to effectively benefit from them, and they want to be able to also use them on fast networks. Optimized hardware usage and scalability, which imply load balancing across multiple processors, are basic requirements for operation on high speed networks. Achieving this is not trivial and requires reconsidering the process of analyzing networks completely. Our paper proceeds in this direction from the foundation: the acquisition of network traffic.

The work just described is only the first step toward this application redesign: it highlights the problem in a situation that exacerbates it, proposes a solution, and substantiates its advantages by implementing and evaluating a prototype.

Despite the remarkable fact that an improvement of 250% has been obtained with a purely software optimization, the main result is not absolute performance. The main result is the degree of scalability that has been introduced. This paves the way, for the first time, to achieve full rate analysis of 10 GBit/OC-192 links, or even faster ones, by means of a PC.

7. ACKNOWLEDGEMENTS

The authors would like to acknowledge Endace Technology, Ltd. for supporting this research and providing the resources (development systems and test equipment) to carry it out.

The authors would also like to thank Fulvio Risso <fulvio.risso@polito.it> and Mario Baldi <mario.baldi@polito.it> for the valuable discussions, challenging questions, suggestions and support throughout this work.

Finally, special thanks go to Bruce Grisham <bruce@accumatics.com> for building and configuring the test environment and encouraging this research.

8. REFERENCES

[1] University of Waikato. The *Dag Project*. Available at http://dag.cs.waikato.ac.nz/.

[2] McCanne, S., and Jacobson, V. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the 1993 Winter USENIX Technical Conference*, San Diego, CA, January 1993, USENIX.

[3] Wood, P. *libpcap-mmap*. Available at http://public.lanl.gov/cpw. Los Alamos National Labs.

[4] Rizzo, L. Device Polling Support for FreeBSD. Available at http://info.iet.unipi.it/~luigi/polling/. In *BSDCon Europe Conference 2001*, Brighton, United Kingdom, November 2001.

[5] Mogul, J. C., and Ramakrishnan, K. K. Eliminating Receive Livelock in an Interrupt-Driven Kernel. In *ACM Transactions on Computer Systems*, 15, 3, (August 1997), 217—252.

[6] Kim, I., Moon, J., and Yeom, H. Y. Timer-Based Interrupt Mitigation for High Performance Packet Processing. In *5th International Conference on High-Performance Computing in the Asia-Pacific Region*, 2001.

[7] Risso, F., and Degioanni, L. An Architecture for High Performance Network Analysis. In *Proceedings of the 6th IEEE Symposium on Computers and Communications (ISCC 2001)*, Hammamet, Tunisia, July 2001.

[8] Combs, G. *Ethereal*. Available at http://www.ethereal.com.

[9] Roesch, M. Snort – Lightweight Intrusion Detection for Networks. In *Proceedings of Usenix Lisa '99 Conference*. Available at http://www.snort.org.

[10] The Netgroup at Politecnico di Torino. *Windump*. Available at http://windump.polito.it.

[11] Jacobson, V., Leres, C., and McCanne, S. *libpcap*, Lawrence Berkeley Laboratory, Berkeley, CA. Initial public release June 1994. Currently available at http://www.tcpdump.org.

[12] Vianney, D. *Hyper-Threading speeds Linux*. Available at http://www-106.ibm.com/developerworks/linux/library/l-htl/?ca=dgr-lnxw01HyperThread, January 2003.

[13] Degioanni, L., and Varenni, G. *Toward 10Gbps with Commodity Hardware*. Technical Report DAUIN200403. Available at http://netgroup.polito.it/gianluca/dags_2004.pdf. Dipartimento di Automatica e Informatica, Politecnico di Torino, Italy, Mar. 2004.