High-Performance Many-Core Networking: Design and Implementation

Jordi Ros-Giralt, Alan Commike, Dan Honey, Richard Lethin
Reservoir Labs
632 Broadway, Suite 803
New York, NY 10012

Abstract — Operating systems play a key role in providing general purpose services to upper layer applications at the highest available performance level. The two design requirements—generality and performance—are however in contention: the more general purpose a service layer is, the more overhead it incurs in accessing domain-specific high-performance features provided by the layers beneath it. This trade-off comes to manifest in modern computer systems as the state-of-the-art has evolved from architectures with a few number of cores to systems employing a very large number of cores (many-core systems). Such evolution has rendered the networking layer in current operating systems inefficient as its general purpose design deprives it from a proper use of the large number of cores. In this paper we introduce DNAC (Dynamic Network Acceleration for Many-Core), a high-performance abstraction layer designed to target the maximum network performance available from the network interface in many-core general purpose operating systems.

CCS Concepts

• Computer systems organization~Multicore architectures

• Security and privacy~Network security

Keywords

High-performance networking; closed-loop control systems

1. INTRODUCTION

Computers are systems of growing complexity that combine sophisticated hardware engines (processors, memory modules, network interfaces, etc.) with a vast number of software modules (operating systems, drivers, applications, etc.). As one strategy to deal with such complexity, computers are implemented using a layering approach, whereby services are organized into N layers, with each layer leveraging the services provided by the level beneath to deliver new services to the level above. Although not necessarily always true, users typically interface with computers via the top layer (layer N), while computers interact with other computers most normally via the lowest layer (layer 1). This approach enables the separation of concerns, a design principle that keeps the implementation modular and general, simplifying development and maintenance.

While layering is a powerful strategy upon which many successful general-purpose architectures depend, it comes at a performance cost, as every layer adds computing distance between the application and the hardware. Take for instance a *read* operation on a file. The execution path needs to cross the user-space file system API layer, the system call layer, the kernel-space file system layer and the driver layer in order to retrieve the requested data from storage. The higher the layer is, the richer and more general its services are, albeit at a slower performance. This trade-off of generality versus performance is illustrated in Figure 1.

Within computer architectures, one area that has been the subject of much research and development is the network layer. In a traditional architecture, applications interface with the network via the standard sockets API. This approach provides great flexibility but requires the processing of several layers before data can be transferred between the network and the application, including the socket layer in user space, the system call layer, the socket layer in kernel space, the network layer also in the kernel (e.g., the TCP/IP stack or the PCAP layers), and the lower level driver / direct memory access (DMA) layer. This architecture not only adds excessive amounts of overhead for high-performance computing (HPC) applications, but it also lacks scalability in systems with a large number of cores, as packets need to be demultiplexed and distributed across all cores by the kernel, inducing unnecessary packet copies and creating a kernel bottleneck.

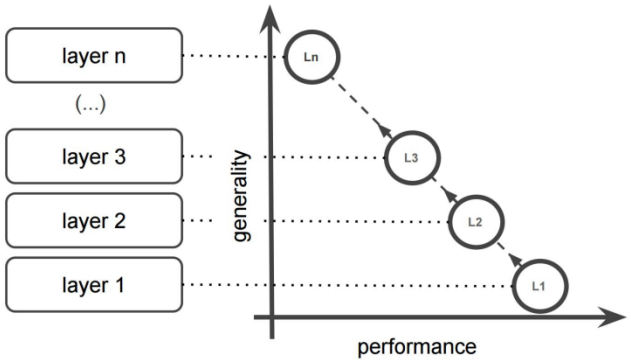

Figure 1. A layered architecture faces the generality versus performance trade-off. The lower level, closer to the physical layer, provides the absolute maximum performance. New layers add more functionality, adding generality to the system, at the expense of losing (at each level) a certain degree of performance.

In our work we present DNAC, Dynamic Network Acceleration for Many-Core, a design and implementation to resolve the HPC problems faced by a general-purpose OS when running on many-

INDIS2015, November 15-20, 2015, Austin, TX, USA
 ISBN 978-1-4503-4002-1/15/11 $15.00
DOI: http://dx.doi.org/10.1145/2830318.2830319

core systems. We argue that an optimized packet handling layer should consider three design requirements:

- *Performance.* The implementation should be able to transfer packets as efficiently as possible, minimizing compute cycles and memory bandwidth spent on each packet.
- *Dynamic packet forwarding.* The optimal forwarding action applied to each packet should in general be dynamic and dependent on the system's current state. For instance, the optimal packet actions taken on a system that is running at 10% of its capacity are in general not the same as those that ought to be taken when the system is running at 100% or above its capacity.
- *Visibility.* In order to make proper packet handling decisions, it is crucial to have a precise and fine grain knowledge of the system's state. For instance, the packet forwarding layer should know whether the system is currently dropping packets and if so, the degree of congestion, in order to make a proper packet forwarding decision.

Much of the work on HPC networking for general computer architectures has focused on performance, leaving a void in the area of optimal packet handling and visibility. Our work provides a comprehensive solution to HPC networking by approaching the problem as a closed-loop control system, integrating visibility and feedback into the packet forwarding engine (control plane), while leveraging all the bare metal acceleration features provided by the underlying hardware (data plane).

The rest of the paper is organized as follows. In Section 2 we introduce the current state-of-the-art by summarizing prior background work. Section 3 introduces the main body of this work with the proposed architecture design and implementation. Section 5 provides some benchmark measurements of the proposed solution, and we conclude in Section 6 with some final remarks and future work.

2. BACKGROUND WORK

High-performance networking in general-purpose computer systems has been the subject of extensive research. For the most part, work has focused on resolving well-known system bottlenecks introduced by the network stack as part of the operating system (OS). Consider for instance the sequence of steps that an incoming packet must follow to reach an application. Starting from the wire, the network interface card (NIC) first captures the packet and transfers it to a buffer in kernel memory using direct memory access (DMA). The NIC then interrupts the CPU which stops its current task in order to serve the associated interrupt service routine (ISR). The ISR performs some light-weight housekeeping work and hands the newly arrived packet to the upper layer before returning control of the CPU. A kernel thread is then responsible to walk the packet through the various network layers prior to reaching the socket. For instance, a TCP/IP packet will first go through the IP layer, then the TCP layer, and finally be delivered to a socket. The socket layer has two levels, one living in the kernel space and the other in user space. The payload of the incoming packet is pushed into the kernel socket, at which point the kernel thread completes its job. Then, the application running in user space issues a read() system call to copy the packet payload from the kernel socket into the user space socket, from where it can finally read its content. Figure 2 provides a diagram of the incoming packet flow, from the wire to the application processes. Notice that a similar set of operations with the same overheads take place when packets are transmitted.

A good number of techniques have been developed to reduce some of the overheads incurred by the OS in receiving and transmitting packets. mmap() can be used to map buffers allocated in user space down into the kernel space in a way that the OS can eliminate the extra packet copies incurred at the socket layer, saving both CPU cycles and memory bandwidth. This technique requires no hardware cooperation and therefore maintains the same degree of generality while improving performance. Examples of technologies employing this approach are libpcap/mmap and vanilla PF_RING [1]. The performance gains are however marginal, and in most HPC applications this approach is not enough. If the NIC hardware can cooperate and provide supporting features like receive side scaling (RSS), the software can be architected to ensure zero-packet copies across the whole network stack. In this approach, the application registers memory using the NIC's native API. The NIC transfers via DMA incoming packets directly to this memory, allowing applications to pull data directly from their buffers. A side benefit of this approach is that it eliminates expensive interrupts, as data is now pulled from the application side. In addition, if the application runs multiple processes and the NIC supports RSS, this approach allows for packets of the same flow to be directly DMA transferred to the right application process, enabling zero-copy load balancing. This approach is implemented in architectures such as PF_RING ZC [2], Intel's DPDK [3], or in other off-the-shelf HPC NIC vendors that provide a native API.

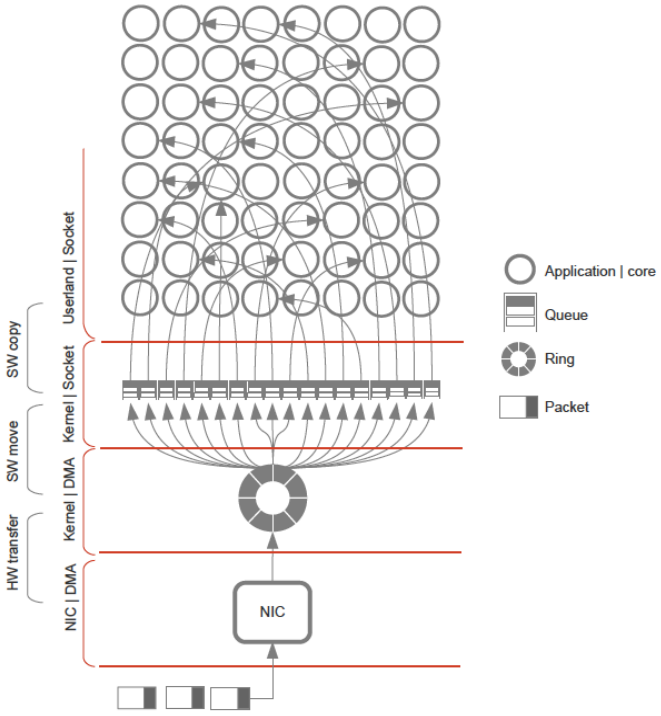

Figure 2. Incoming packets in a general-purpose OS architecture have to traverse many software layers before they can reach their application socket.

Most prior work has focused on the high-performance aspects of the problem. Being able to run applications closer to the hardware layer brings however another relevant advantage: timely fine-grained visibility. This opens up the possibility of incorporating real-time feedback and control to optimize the packet forwarding routines in order to attain higher levels of system performance.

3. ARCHITECTURE

In this section we introduce DNAC, Dynamic Network Acceleration for Many-Core, an architecture and implementation of a high-performance network layer designed to efficiently steer packets from the wire to the application in many-core architectures. While some of the concepts described in this section apply to both the receive and transmit data paths, currently DNAC focuses on applications that only require the processing of incoming packets. Examples of applications that DNAC can support includes network tapping technologies, such as intrusion detection systems, full packet capture engines, or applications dealing with network analysis and monitoring in general.

We structure the architecture discussion in four sections: (1) HPC features and APIs, (2) steering of packets at line rates, (3) visibility and dynamic packet control, and (4) implementation notes.

3.1 HPC Features and APIs

Common HPC NICs and modern operating systems provide different acceleration features as well as APIs to accommodate for application requirements. We start first by providing a brief description of the HPC features available:

- *Receive side scaling (RSS).* Symmetric multiprocessing applications achieve scalability by running identical processes in parallel, each process taking a fair share of the total workload. In a standard OS, the task of load balancing network traffic to each process is carried out by a kernel thread, incurring a critical overhead as the additional CPU cycles and the extra per-packet copy needs to be done in the software. NICs supporting RSS can perform load balancing of traffic in hardware as well as directly DMA transfer packets to the right application process according to a predetermined load balancing algorithm, effectively eliminating the software traffic distribution bottleneck.
- *Kernel bypass.* HPC NICs provide a direct data plane path between the application and the NIC so that packets can flow between these two layers without any kernel intervention. This saves a substantial amount of processing and context switching overhead. As a side effect of this approach and since the kernel is no longer involved, the application is responsible for all packet processing.
- *Packet coalescence.* An implication of establishing a direct communication path between the NIC and the application is that the latter needs to be responsible for managing packet buffers. For instance, in the receive path, as packets are read by the application, new empty packet buffers need to be passed down to the NIC to replenish the ring of buffers so that the NIC can continue to transfer incoming packets. Packet coalescence is a hardware feature that gathers multiple packets together into a buffer and then has the application process the buffer at once. This allows applications to process and replenish packet buffers in larger batches, saving communication overheads between the NIC and the application.

In terms of APIs, applications typically access the HPC NIC services via two interfaces:

- *Bare metal native API.* The native API provides direct access to the NIC's bare metal functions. Using this interface, applications can have access to all the HPC features, albeit at the cost of having to port the application. This also means that applications need to deal on their own with those services otherwise provided by the kernel. For instance, an application that needs to send and receive TCP/IP packets will need to process the TCP and IP layers. To overcome this limitation, some vendors provide as part of their SDK a user-level network stack library that applications can link to [4, 5].
- *PCAP API for packet capture applications.* Applications that only need to capture incoming packets typically use the PCAP API, the standard BSD user-level interface providing access to raw packets (including all headers of the packet) on the receiving end. Because many applications are based on this library, most HPC NIC vendors provide their own PCAP library implementation leveraging the hardware acceleration features. A benefit of this approach is that applications based on the PCAP API don't need to be ported. However, by introducing a new layer, this approach adds some additional overhead and it often does not support all the available bare metal HPC features.

3.2 Steering Packets at Line Rates

Figure 3 presents a view of how packets flow from the wire to the application in the DNAC architecture. As illustrated, the main heavy-lifting work to steer traffic inside the system is carried out by RSS. Because RSS is implemented in hardware, the application effectively receives traffic at wire speeds without incurring any CPU cycles. In comparison with the general purpose OS architecture (Figure 2), in the DNAC architecture we have that:

- Packets are delivered without the need to neither interrupt the CPU nor invoke any interrupt service routine. Instead, the application polls packets from its incoming ring at its own natural pace. This eliminates expensive context switching operations.
- Packets are no longer copied into kernel memory and from the kernel to userland. Instead, packets are directly transferred to the application's userland memory.
- The kernel is no longer involved in the processing of packets, further reducing the number of cycles required to process each packet.
- Expensive system calls to read traffic from the input socket are also eliminated.
- The decision to forward each packet to the right destination process is done in hardware, offloading this task from the CPU. In RSS, packets are forwarded to each process according to the hash value of the IP tuple, in a way that packets of the same flow are forwarded to the same destination process.

DNAC also leverages packet coalescence, which helps reduce per-packet handling costs by delivering packets to the application in batches. The processing of packets using this feature is illustrated in Figure 4 and it works as follows:

1. The application allocates large buffers (typically using Huge Pages from the OS) capable of holding multiple packets and registers them to the NIC. Arriving packets are transferred into a buffer. When the buffer is full or when a timeout since the last packet arrived expires, the NIC posts the event "coalesced packets ready" into a queue of events.
2. The application process reads from the queue to get the new event, and gets a pointer to the buffer holding the batch of packets that triggered the event.
3. The application process dispatches all packets in the buffer until completion.

The application process replenishes a new empty buffer so that the NIC can continue to fill more buffers at line rate.

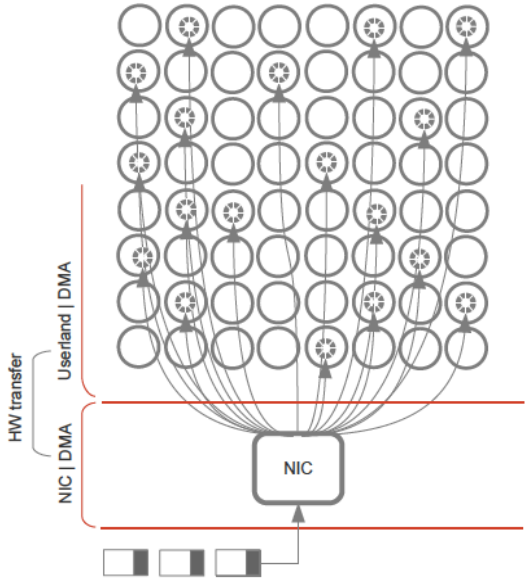

Figure 3. General view of the packet flow in the many-core system, leveraging RSS to bring packets from the wire directly into each process memory at wire speed.

Packet coalescence helps improve performance in two ways: first, the cost of generating and processing each event is amortized across all the packets contained in one batch; second, the application only needs to replenish buffers on a per batch basis, rather than for each packet. These overheads can be important in achieving line rates because both of them involve interactions between the application and the NIC.

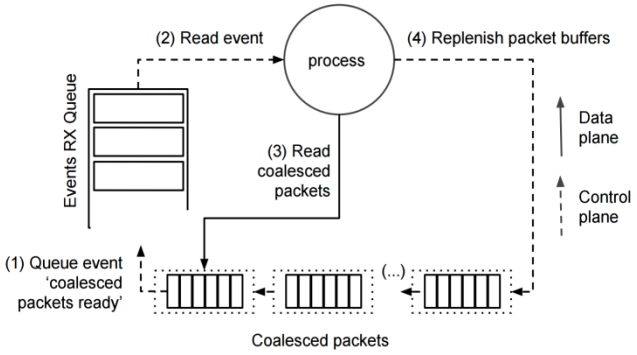

Figure 4. Packet coalescence is used to process packets in batches helping reduce per-packet overheads.

3.3 Visibility and Dynamic Packet Forwarding

Delivering packets at the highest speed available is crucial in order to maximize the total system performance. However, delivering all packets as fast as possible to the application does not always deliver optimal performance. For instance, suppose that the application is overwhelmed to the point that it needs to drop packet P without processing it. Then, rather than delivering packet P to the application, a better strategy is to drop packet P as soon as possible. Another example is a scenario in which by investing a small amount of compute cycles performing some simple operations, the network layer can detect whether a packet is relevant to the application. For instance, it is better to drop all traffic on TCP port 443 as early as possible if it is known that the application is not capable of processing HTTPS traffic. These two examples are in fact part of two general classes of scenarios in which performing dynamic packet forwarding decisions provides a superior strategy. We refer to them as packet congestion and packet relevance, respectively, and summarize their definition in Table 1.

Depending on the type of scenario, the optimal packet forwarding decision may or may not involve real time feedback from the current state of the system. For instance, the packet congestion scenario requires real time information on the degree of congestion the system is experiencing at the time a packet forwarding decision is made. This leads to a closed-loop controller design. On the other hand, certain packet relevance scenarios such as the case of dropping encrypted traffic do not require feedback, leading to an open-loop controller design. Further, not all packet relevance scenarios lead to open-loop systems; for instance, some applications may dynamically decide that a certain type of traffic is no longer relevant, conveying such information in the form of feedback to the network layer, which then can drop the traffic. Table 1 adds a third column for the controller type used in each scenario.

Table 1. Types of scenarios in which dynamic packet forwarding delivers better performance.

Feedback	Description	Controller
Packet congestion:	Packets are dropped by the application as a consequence of the application being congested.	closed-loop control
Packet relevance:	Packets are dropped by the application as a consequence of the packet not being relevant to the application.	open- or closed-loop control

It is well known from control theory that a key to performance is timely and accurate visibility and feedback of the system's state. This observation provides an important justification for basing the design of DNAC on the NIC's native bare metal API. (See Section 3.1 for a description of the NIC API's choices.) By running closer to the hardware, not only can we more efficiently steer packets to the application, but we can also improve visibility and accuracy in measuring the current state of the system. For instance, the alternative PCAP API provides a method to extract statistics, pcap_stats(), such as packet drops. However, this function runs with the full overhead of a system call and the lower-level PCAP layer only refreshes the statistics once a second. On a 10Gbps, up to 15 million small-size packets arrive in one second, rendering pcap_stats() an unfeasible API to implement the closed-loop controller.

The design of the DNAC open- and close-loop controllers is introduced in Figure 5. Packets are first DMA transferred (1) to the process memory as described in Section 3.2. A packet forwarder is located between the incoming ring and the application to make low computational and quick packet steering

decisions. The forwarder utilizes three sources of information to make such decisions: the current level of congestion (5) seen in the ring (feedback type: packet congestion; controller type: closed-loop), any feedback (6) directly received from the application (feedback type: packet relevance; controller type: closed-loop control), and pre-established rules (7) configurable by the network operator (feedback type: packet relevance; controller type: open-loop control). With this feedback, the forwarder then makes a per-packet binary decision: it either forwards the packet to the application or it drops it.

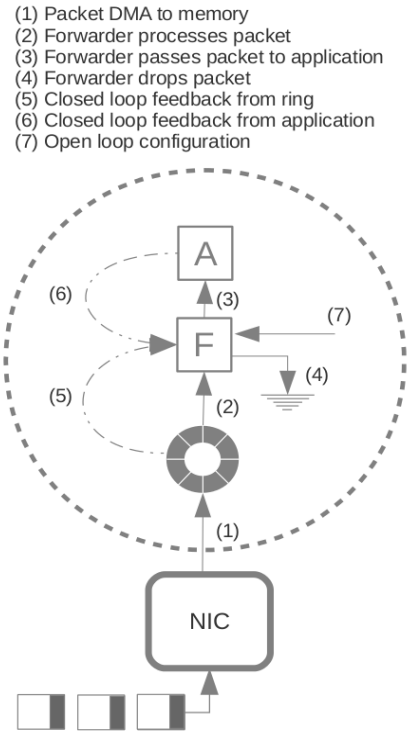

Figure 5. Zoom into the internal architecture of each process in the DNAC architecture: (1) packets are directly DMA transferred to the right core, (2) a forwarder (F) provides intelligent packet dropping decisions based on both open- and closed-loop control; (3) packets are processed by the application (A) running in its own core.

3.4 Implementation Notes

We have implemented DNAC to enable high-performance packet injection into Bro [6], the advanced network monitoring system (NMS) actively developed by computer scientists at the International Computer Science Institute (ICSI) and the National Center for Supercomputing Applications (NCSA). Bro provides a mode of operation called cluster in which multiple workers are run in parallel, each processing a fair share of the total traffic. In this mode, a Bro worker fits the definition of our application building block (represented with the block "A" in Figure 5).

Our implementation of DNAC is architecture independent, providing an abstraction layer between the application and the hardware NIC, and requires no changes to Bro. While it is designed to support multiple NIC vendors, we are currently running DNAC on the SolarFlare SFN7000 series, a 10Gbps HPC network adapter. Our choice for the SFN7000 is based on its support for all the most advanced HPC features described in this paper, including RSS, packet coalescing and support for a bare metal native API. The Solarflare native API is open sourced and available as part of the OpenOnload.org project [4].

The implementation of our closed- and open-loop controllers makes use of several techniques to help substantially improve the performance of the overall system. Table 2 summarizes these techniques and algorithms.

Table 2. Implementation of the control loops

Feedback: packet congestion | Controller: closed-loop
Algorithm: TED Queuing. Tail early dropping is a queuing policy that, upon congestion, drops packets from the tails of each connection, preserving the heads, by dynamically computing a connection cut threshold. This optimization leverages the well-known heavy-tailed nature of traffic [8], which states that on average, connection heads carry higher degrees of information than connection tails. TED queuing is an algorithm developed by the authors and formally presented in [9].
Feedback: packet relevancy | Controller: closed-loop
Algorithm: Packet shunting. The Bro workers have an API that can be used to communicate packet shunting decisions to the forwarder. For instance, if a Bro worker comes to the conclusion that a certain connection is no longer relevant, it can tell the forwarder to drop any future packets from that connection.
Feedback: packet relevancy Controller: open-loop
Algorithm: Packet prioritization. Due to the way protocols are constructed, there exist certain packets that carry higher degrees of information. For instance, dropping a FIN packet not only has semantic implications at the protocol level, but it impacts performance as the upper layer needs to rely on expensive timeouts and hold context information in memory for unnecessary longer periods of time prior to deallocating the connection data structure. DNAC allows network operators to specify fixed rules to prioritize such type of packets.

4. MEASUREMENTS

In this section, we present some initial tests and measurements of the DNAC architecture focusing on two aspects of the solution: (1) single-node performance and (2) the value of closed- and open-loop control. In a forthcoming paper, we will present a more comprehensive set of tests and benchmarks of the proposed solution.

To illustrate the value of dynamic packet forwarding in HPC networking, we define a test that will stress the packet congestion closed-loop controller. This controller implements TED queuing (See Table 2), a technique that dynamically reacts to system congestion by dropping connection tails. Using httperf [7], we synthetically create a packet trace consisting of a population of clients downloading a 1MB file from 25 servers using the HTTP protocol. With this setup, we collect a 65GB trace which we use to stress our implementation by replaying it at various speed-up rates.

Our application runs Bro, a network monitor system that generates logs containing real-time information of events detected from the incoming traffic. We measure performance in terms of the number of events that Bro can detect on the given input trace. While Bro generates events for a large variety of protocols and network analytics, we focus on three types of events:

- http events: generated every time an HTTP transaction is detected.
- files events: generated every time a file download is detected.
- http_track events: generated every time a pair of HTTP REQUEST/REPLY within the same transaction is detected.

The http and file analytics come with stock bro, whereas http_track is a simple analytic that the authors wrote and which can be downloaded from https://github.com/reservoirlabs/bro-scripts/blob/master/bench/benchHttp.bro.

We start by feeding the trace to our system at a rate of 500Mbps. This rate is low enough to ensure that there are no packet drops in the system so that we can take some initial measurements of the trace. Then we run the same test at an input rate of 5Gbps, and measure the effect of enabling DNAC's dynamic packet control versus disabling it. The results are presented in Table 3 and Figure 6.

As shown, at 500Mbps (no congestion scenario), both DNAC and no DNAC configuration perform equally well. Increasing the input rate to 5Gbps triggers the closed-loop control in DNAC to proactively drop connection tails as a function of the congestion level. As a result, system level performance increases by about 3 times (between 2.5X and 3.2X depending on the analytic.)

Table 3. Number of events detected

	500Mbps input rate			5Gbps input rate		
	http	files	http_track	http	files	http_track
w/ DNAC	42449	39594	31300	38425	33137	28400
w/o DNAC	42434	39314	31200	15314	10376	8700
gain	1	1	1	2.5	3.2	3.2

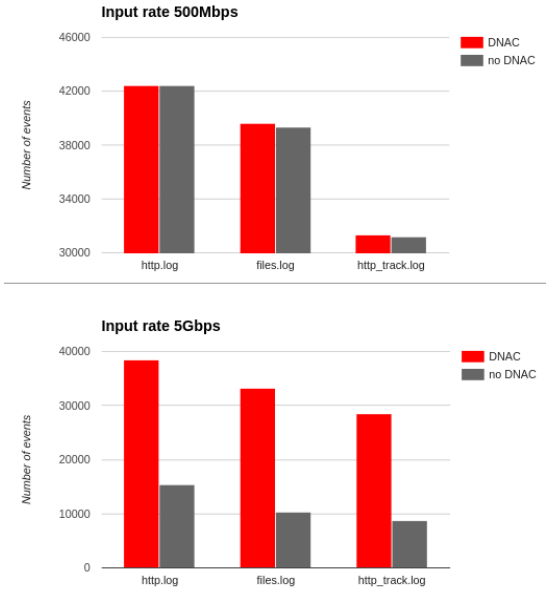

Figure 6. Number of events detected by a single Bro worker when running the input trace at 500Mbps (upper graph) and at 5Gbps (lower graph). At 5Gbps and without dynamic packet forwarding, the system cannot keep up.

For the case of 5Gbps input traffic, Figure 7 presents the number of packets received, dropped and forwarded per second with or without dynamic packet forwarding (marked with the labels 'w/ DNAC' and 'w/o DNAC') as a function of time. Notice that in the 'w/o DNAC' case, all received packets are forwarded to the application, so that the curves for packets received and forwarded collide into a single curve, whereas the number of dropped packets is zero.

We observe the following:

- Both systems ('w/ DNAC' and 'w/o DNAC') start at the same level of performance, accepting about 200,000 packets per second and forwarding all the received traffic to the application.
- Because the system is congested (at 5Gbps a single worker cannot keep up with all packets), DNAC reacts by starting to cut connection tails. The size of the tails cut by the TED algorithm increases until system congestion is eliminated. As indicated by the 'packets dropped w/ DNAC' curve, DNAC intentionally drops packets until reaching a steady state.
- Because DNAC reacts to congestion by proactively dropping packets as soon as congestion is detected, the overall system health increases in that the system can accept about 400,000 packets per second, twice as many as without DNAC at 200,000 packets per second.
- In steady state, we have that DNAC forwards about 70,000 packets per second to the Bro worker, versus 200,000 packets per second when DNAC is disabled. Yet as shown in Figure 6, DNAC delivers about 3 times better system performance as measured by events detected per second. In other words, the per-packet productivity increases by a factor of 8.5X. This illustrates the value of early packet dropping upon congestion: whenever there is congestion, a better strategy is to drop the less relevant packets as early in the network stack as possible.

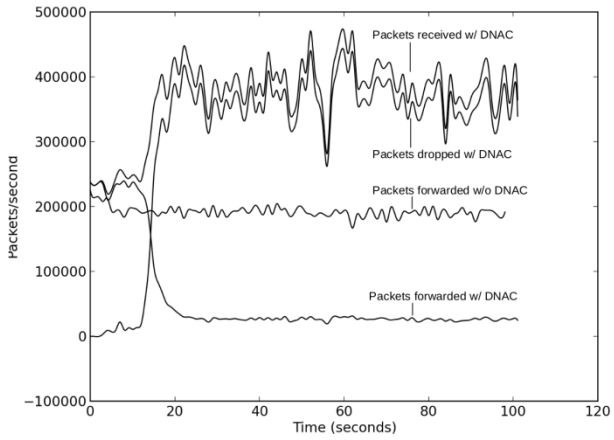

Figure 7. Packets received, forwarded and dropped per second with or without the DNAC layer.

5. CONCLUSIONS

When it comes to designing a packet dispatching layer for high-performance computing (HPC) applications, forwarding packets as fast as possible in a system agnostic manner can be sub-optimal. While the network layer is limited in the type of operations it can perform on a per-packet basis, there exist certain simple actions that it can carry out to proactively react to congestion and substantially improve performance. In this work, we provide a framework to classify such techniques based on the concepts of closed-loop and open-loop feedback control and we apply this framework to the problem of HPC networking for many-core applications. We illustrate via some initial benchmarks

that the overall system performance can be notably increased when the network layer is allowed to take an active role in managing system congestion.

In a forthcoming paper, we will provide a comprehensive set of tests to evaluate and benchmark the performance of the proposed solution in greater detail.

The work presented in this paper is implemented as part of the R-Scope appliance developed by Reservoir Labs [10].

6. ACKNOWLEDGMENTS

The authors want to thank Peter Cullen, Dilip Madathil and Jeff Lucovsky for their valuable comments and feedback.

This work was funded in part by the US Department of Energy under Award Numbers DE-SC0004400 and DE-SC0006343.

7. REFERENCES

[1] Luca Deri, Netikos S. P. A , Via Del Brennero Km, Loc La Figuretta, "Improving Passive Packet Capture: Beyond Device Polling," Proceedings of SANE 2004.

[2] Alfredo Cardigliano, Luca Deri, et al. "vPF_RING: Towards Wire-Speed Network Monitoring Using Virtual Machines," Proceedings of IMC 2011, November 2011.

[3] "High-Performance Multi-Core Networking Software Design Options," Intel, Wind River, White Paper 2011.

[4] OpenOnload high performance network stack, Solarflare Communications, Inc.: http://www.openonload.org/

[5] Rump Kernel TCP/IP stack for DPDK: https://github.com/rumpkernel/drv-netif-dpdk

[6] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," Computer Networks, 31(23-24), pp. 2435-2463, 14 Dec. 1999.

[7] Mosberger, D. and Jin, T. httperf: A Tool for Measuring Web Server Performance". Performance Evaluation Review, 26, 3 (December 1998), 31-37.

[8] Paxson, V. "Empirically Derived Analytic Models of Wide-Area TCP Connections," IEEE/ACM Transactions on Networking, 2, 4 (August 1994), 316-336.

[9] J. Ros-Giralt, A. Commike, B. Rotsted, "Overcoming Performance Collapse for 100Gbps Cyber Security,", In Proceedings of the First Workshop on Changing Landscapes in HPC Security, New York, NY, USA, ACM, June, 2013.

[10] R-Scope: https://www.reservoir.com/product/r-scope/

[11] Wenji Wu, Phil DeMar, "WireCAP: a Novel Packet Capture Engine for Commodity NICs in High-speed Networks," IMC'14, November 05 – 07 2014, Vancouver, BC, Canada.