Future Generation Computer Systems ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Algorithms and data structures to accelerate network analysis☆

Jordi Ros-Giralt *, Alan Commike, Peter Cullen, Richard Lethin

Reservoir Labs, 632 Broadway Suite 803, New York, NY 10012, United States

HIGHLIGHTS

- A New queuing algorithm to reduce packet drops in hardware queues.
- New lockless bimodal producer–consumer queue to eliminate multi-thread contention.
- Algorithm to dynamically shunt traffic while maximizing information entropy.
- Lockless hash table with low false negatives to eliminate memory contention overheads.
- Multiresolution priority queues to reduce the complexity of a priority queue down to O(1).

ARTICLE INFO

Article history:
Received 31 January 2018
Accepted 10 April 2018
Available online xxxx

ABSTRACT

As the sheer amount of computer generated data continues to grow exponentially, new bottlenecks are unveiled that require rethinking our traditional software and hardware architectures. In this paper we present five algorithms and data structures (long queue emulation, lockless bimodal queues, tail early dropping, LFN tables, and multiresolution priority queues) designed to optimize the process of analyzing network traffic. We integrated these optimizations on R-Scope, a high performance network appliance that runs the Bro network analyzer, and present benchmarks showcasing performance speed ups of 5X at traffic rates of 10 Gbps.

1. Introduction

System wide optimization of network components like routers, firewalls, or network analyzers is complex as it involves the proper orchestration of at least hundreds of different algorithms and data structures interrelated in subtle ways. In these highly dynamic systems, bottlenecks quickly shift from one component to another forming a network of *micro-bottlenecks*. This makes it challenging to understand which elements should be further optimized to get that extra unit of performance. Moreover, these shifting micro-bottlenecks are interconnected in peculiar ways so that optimizing one of them can often lead to an overall degradation of performance. This is due to internal system nonlinearities such as those found in hierarchical memory architectures. For instance, while optimizing the transfer of packets from the wire to the application is known to be critical, in the limit pushing too many packets to the application is detrimental as packets that eventually need to be dropped will cause a net negative effect by thrashing the processors' local caches, increasing the overall cache miss ratios and hence decreasing system wide performance. The process of performance optimization should therefore be a meticulous one which requires making small but safe steps avoiding the pitfall of pursuing short term gains that can lead to a new and bigger bottleneck down the path.

In this paper we present five of such safe steps that have helped to optimize the performance of R-Scope, a high performance appliance that runs the network analyzer Bro at its core [1]. Each of these steps introduces a new algorithm or data structure designed to accelerate system wide performance, each one addressing a different shifting micro-bottleneck. While we use Bro to demonstrate the efficacy of these optimizations, they are of general purpose and so we believe these techniques can be generally applied to the problem of accelerating network analysis or, to some degree, to optimize other more active network components such as firewalls or routers.

This paper is organized as follows. Section 2 is dedicated to describing the five HPC algorithms in detail, providing algorithmic descriptions of how they work and independent benchmarks illustrating how they help improve performance by decongesting a specific bottleneck. Section 3 provides a system wide benchmark

☆ This work was funded in part by the US Department of Energy, United States under contracts DE-SC0017184, DE-SC0006343 and DE-SC0004400.

* Corresponding author.
E-mail addresses: giralt@reservoir.com (J. Ros-Giralt), commike@reservoir.com (A. Commike), cullen@reservoir.com (P. Cullen), lethin@reservoir.com (R. Lethin).

https://doi.org/10.1016/j.future.2018.04.034
0167-739X/

measuring the performance of all the algorithms working together. We summarize and conclude the paper in Section 4.

2. Algorithms and data structures

2.1. Long queue emulation for packet forwarding

High performance network interface cards (NICs) help accelerate the process of moving packets from the wire to the application by using techniques such as *receive side scaling* (RSS), *zero copy*, *packet coalescence* or *kernel bypass*, among others [2]. These cards achieve higher performance by leveraging hardware at the cost of losing some degree of flexibility and programmability. For instance, one common element of rigidity found in HPC NICs is the amount of memory embedded in their chip, which limits the size of the rings used to temporarily hold packets as they are transferred to the application. As a result, temporary high bursts of traffic that cannot be handled fast enough by the application may overflow these hardware rings leading to packet drops.

A traditional way to address packet drops originated from a limited size ring (LSR) is to dedicate one or more dispatcher threads (DT) to move packets out of the ring and put them into one or more software queues connected to the application threads (AT) residing on the host. Because the host does not have the embedded memory restrictions of the NIC, the software queues effectively have unlimited size. Consequently, packet drops due to bursty traffic are eliminated provided that the dispatcher threads can move packets from the limited size rings (LSR) to the unlimited size queues (USQ) fast enough. This solution is illustrated in Fig. 1.

While this solution seems sound at a high level, in the context of HPC the dispatcher thread introduces the following subtle but important performance penalties:

- *Packet read cache penalty.* If the dispatcher thread (DT) needs to read the packet – for instance, if it needs to compute the hash of the packet's IP tuple to decide which destination queue the packet should be forwarded to – then the packet will need to be loaded into the local cache. Since the application thread (AT) will also need to read the packet for its own processing, the dispatcher model requires loading a packet to the cache twice (one time on the DT's local cache and a second time on the AT's local cache). This negatively impacts performance because cache misses – which require accessing memory to fetch data – are typically ten times slower than cache hits. As a general principle, a good design should aim for a single cache load throughout the lifetime of each packet.
- *Descriptor read cache penalty.* Even if the DT does not need to read the packet – e.g., some implementations can extract the hash of the packet's IP tuple from the packet context information provided by the hardware – the DT will still need to load the packet descriptor onto its local cache. (A packet descriptor is a small software data structure part of all NIC drivers containing a pointer to the packet buffer and additional control metadata such as the packet length or the hash of its IP tuple, among others.) In this case, during the lifetime of a packet, its descriptor needs to be loaded twice, once at the DT's local cache and a second time at the AT's local cache. Just like before, a good design should target one single cache load for each individual packet descriptor.
- *Memory and compute overhead.* Yet another overhead introduced by this approach is the additional memory and compute resources required to run the dispatcher threads themselves.

To avoid the above performance penalties, we propose to use *long queue emulation* (LQE), a simple but efficient technique that eliminates the overhead introduced by the dispatcher thread with the potential to also reduce packet drops.

The main concept behind LQE is to emulate the behavior of the dispatcher thread solution by folding the actions performed by the DT thread into the AT thread. Consider first the pseudocode of the DT and AT threads separately as implemented by the dispatcher model:

```
DtThread()
1   while true:
2       get alls packets from the front of LSR;
3       put the packets to the tail of USQ;
```

```
AtThread()
4   while true:
5       get one packet from the front of USQ;
6       process the packet;
```

While in the dispatcher thread solution the `DtThread()` and the `AtThread()` procedures are run on two independent threads, in the long queue emulation model we fold `DtThread()` into `AtThread()` as a single thread running the procedure `AtLqeThread()`:

```
AtLqeThread()
1   while true:
2       get all packets from the front of LSR;
3       put the packets to the tail of USQ;
4       get one packet from the front of USQ;
5       process the packet;
```

The key characteristic of the `AtLqeThread()` procedure is that it ensures all packets from the LSR ring are moved to the USQ queue before the next packet is processed, effectively giving the highest priority to the ring. This approach emulates the behavior of the dispatcher model with one single thread performing both the DT and the AT procedures. As a result, both packets and packet descriptors are loaded into the cache only once (at the AT's local cache) and there is no additional memory and compute overhead to maintain the dispatcher threads. We illustrate the LQE model in Fig. 2.

More formally, we describe the performance properties of the long queue emulation model in the following lemma:

Lemma 1 (*Long Queue Emulation Performance*)**.** *Let λ and λ_{max} be the average and the maximum packet arrival rate measured at the LSR ring, respectively. Assume for the sake of simplicity and without loss of generality, that the time to process a packet is constant, and let μ_{dt} and μ_{lqe} be the packet processing rate of the DT model and the LQE model, respectively—that is, μ_{dt} and μ_{lqe} correspond to one divided by the time it takes to execute line 6 in `AtThread()` and line 5 in the `AtLqeThread()`. If S_{lsr} is the maximum number of packets that can be held in the LSR ring, then the following is true:*

(1) $\mu_{lqe} > \mu_{dt}$.
(2) *If $s_{lsr}/\lambda_{max} \geq 1/\mu_{lqe}$, the performance of the LQE model is superior to the performance of the DT model.*
(3) *If $s_{lsr}/\lambda_{max} < 1/\mu_{lqe}$ and $\lambda \geq \mu_{lqe}$, the performance of the LQE model is superior to the performance of the DT model.*
(4) *If $s_{lsr}/\lambda_{max} < 1/\mu_{lqe}$ and $\lambda < \mu_{lqe}$, the performance of the DT model is superior to the performance of the LQE model.*

Proof. It is easy to see that $\mu_{lqe} > \mu_{dt}$ because the LQE model does not suffer from DT's performance penalties due to extra cache

Please cite this article in press as: J. Ros-Giralt, et al., Algorithms and data structures to accelerate network analysis, Future Generation Computer Systems (2018), https://doi.org/10.1016/j.future.2018.04.034.