Future Generation Computer Systems 56 (2016) 277–283

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Improving network performance on multicore systems: Impact of core affinities on high throughput flows

Nathan Hanford [a,*], Vishal Ahuja [a], Matthew Farrens [a], Dipak Ghosal [a], Mehmet Balman [b], Eric Pouyoul [b], Brian Tierney [b]

[a] Department of Computer Science, University of California, Davis, CA, United States
[b] Energy Sciences Network, Lawrence Berkeley National Laboratory, Berkeley, CA, United States

HIGHLIGHTS

- Affinity, or core binding, maps processes to cores in a multicore system.
- We characterized the effect of different receiving flow and application affinities.
- We used OProfile as an introspection tool to examine software bottlenecks.
- The location of the end-system bottleneck was dependent on the choice of affinity.
- There are multiple sources of end-system bottlenecks on commodity hardware.

ARTICLE INFO

Article history:
Received 20 February 2015
Received in revised form
15 August 2015
Accepted 11 September 2015
Available online 25 September 2015

Keywords:
Networks
End-system bottleneck
Traffic shaping
GridFTP
Flow control
Congestion avoidance

ABSTRACT

Network throughput is scaling-up to higher data rates while end-system processors are scaling-out to multiple cores. In order to optimize high speed data transfer into multicore end-systems, techniques such as network adaptor offloads and performance tuning have received a great deal of attention. Furthermore, several methods of multi-threading the network receive process have been proposed. However, thus far attention has been focused on how to set the tuning parameters and which offloads to select for higher performance, and little has been done to understand why the various parameter settings do (or do not) work. In this paper, we build on previous research to track down the sources of the end-system bottleneck for high-speed TCP flows. We define protocol processing efficiency to be the amount of system resources (such as CPU and cache) used per unit of achieved throughput (in Gbps). The amount of various system resources consumed are measured using low-level system event counters. In a multicore end-system, affinitization, or core binding, is the decision regarding how the various tasks of network receive process including interrupt, network, and application processing are assigned to the different processor cores. We conclude that affinitization has a significant impact on protocol processing efficiency, and that the performance bottleneck of the network receive process changes significantly with different affinitization.

1. Introduction

Due to a number of physical constraints, processor cores have hit a clock speed "wall". CPU clock frequencies are not expected to increase. On the other hand, the data rates in optical fiber networks have continued to increase, with the physical realities of scattering, absorption and dispersion being ameliorated by better optics and precision equipment [1]. Despite these advances at the physical layer, we are still limited with the capability of the system software for protocol processing. As a result, efficient protocol processing and adequate system level tuning are necessary to bring higher network throughput to the application layer.

TCP is a reliable, connection-oriented protocol which guarantees in-order delivery of data from a sender to a receiver, and in doing so, pushes the bulk of the protocol processing to the end-system. There is a certain amount of sophistication required to implement the functionalities of the TCP protocol, which are all instrumented in the end-system since it is an end-to-end protocol. As a result, most of the efficiencies that improve upon current

* Corresponding author.
E-mail addresses: nhanford@ucdavis.edu (N. Hanford), vahuja@ucdavis.edu (V. Ahuja), mkfarrens@ucdavis.edu (M. Farrens), dghosal@ucdavis.edu (D. Ghosal), mbalman@lbl.gov (M. Balman), lomax@es.net (E. Pouyoul), bltierney@es.net (B. Tierney).

http://dx.doi.org/10.1016/j.future.2015.09.012
0167-739X/

TCP implementations fall into two categories: first, there are offloads which attempt to push TCP functions at (or along with) the lower layers of the protocol stack (usually hardware, firmware, or drivers) in order to achieve greater efficiency at the transport layer. Second, there are tuning parameters, which place more sophistication at the upper layers (software, systems, and systems management).

Within the category of tuning parameters, this work focuses on affinity. Affinity (or core binding) is fundamentally the decision regarding which resources to use on which processor in a networked multiprocessor system. The New API for networks (NAPI) in the Linux network receive process allows the NIC to operate in two different contexts. First, there is the interrupt context (usually implemented with coalescing), in which the Network Interface Controller (NIC) interrupts the processor once it has received a certain number of packets. Then, the NIC transmits the packets to the processor via Direct Memory Access (DMA), and the NIC driver and the operating system (OS) kernel continue the protocol processing until the data is ready for the application [2–4]. Second, there is polling, where the kernel polls the NIC to see if there is any network data to receive. If such data exists, the kernel processes the data in accordance with the network and transport layer protocols in order to deliver the data to the Application through the sockets API.

In either case, there are two types of affinity: (1) *Flow affinity*, which determines which core will be interrupted to process the network flow, and (2) *Application affinity*, which determines the core that will execute the application process that receives the network data. Flow affinity is set by modifying the hexadecimal core descriptor in /proc/irq/<irq#>/smp_affinity, while Application affinity can be set using *taskset* or similar tools. Thus, in a 12-core end-system, there are 144 possible combinations of Flow and Application affinity.

In this paper, we extend our previous work [5,6] with detailed experimentation to stress-test each affinitization combination with a single, high-speed TCP flow. We perform end-system introspection, using tools such as Oprofile to understand the impact that the choice of affinity has on the receive-system efficiency. We conclude that there are three distinct affinitization performance scenarios, and that the performance bottleneck varies drastically within these scenarios. First, there is the scenario where the protocol processing and the application process are on the same core, which causes the processing capabilities of the single core to become the bottleneck. Second, there is the scenario where the flow receiving process and the application receiving the data are placed on different cores within the same socket. This configuration does not experience a performance bottleneck, but results in very high memory controller bandwidth utilization. Third, when the flow receiving process and the receiving application process are placed on different sockets (and thus, different cores), the bottleneck is most likely inter-socket communication.

The remaining part of the paper is organized as follows. In the next section, we discuss and summarize our prior research and give the motivation of this study. In Section 3, we describe the related work. In Section 4, we describe the experimental setup used to conduct this study. We discuss the results in Section 5. Finally, in Section 6, we give the conclusions and outline the future work.

2. Motivation

In a previous study, we conducted research into the effect of affinity on the end-system bottleneck [5], and concluded that affinitization has a significant impact on the end-to-end performance of high-speed flows. However, the study did not identify the precise location of the end-system bottleneck for the different affinitization scenarios and hence a clear understanding of why different affinitization leads to significantly different throughput performance. The goal of this research is to identify the location of the end-system bottleneck in these different affinitization scenarios, and evaluate whether or not these issues have been resolved in newer implementations of the Linux kernel (previous work was carried out on a Linux 2.6 kernel).

There are many valid arguments made in favor of the use of various NIC offloads [7]. NIC manufacturers typically offer many suggestions on how to tune the system in order to get the most out of their high-performance hardware. A valuable resource for Linux tuning parameters, obtained from careful experimentation on ESnet's 100 Gbps testbed, is available from [8]. A number of reports provide details of the experiments that have led to their tuning suggestions. However, there is a significant gap in the understanding for these tuning suggestions and offloads.

In this paper, our methods focus on analyzing the parallelism of protocol processing within the end-system. We endeavor to demonstrate the variability of protocol processing efficiency depending on the spatial placement of protocol processing tasks. In this experimental study, we employ *iperf3* [9] to generate a stress test which consists of pushing the network I/O limit of the end-system using a single, very high-speed TCP flow. This is not a practical scenario; an application such as GridFTP [10] delivers faster, more predictable performance by using multiple flows, and such a tool should be carefully leveraged in practice when moving large amounts of data across long distances. However, it is important to understand the limitations of data transmission in the end-system, which can best be accomplished using a single flow.

3. Related work

There have been several studies that have evaluated the performance of network I/O in multicore systems [11–14]. A major improvement which is enabled by default in almost all the current kernels is NAPI [15,3]. NAPI is designed to solve the receive livelock problem [14] where most of the CPU cycles are spent in interrupt processing. When the kernel enters a livelock state it spends most of the available cycles in hard and soft interrupt contexts, and consequently, is unable to perform protocol processing of any more incoming data. As a result of the interrupt queues overflowing, packets would eventually be dropped during protocol processing. This would trigger TCP congestion avoidance, but the problem would soon repeat itself. A NAPI-enabled kernel switches to polling mode at high rates to save CPU cycles instead of operating in a purely interrupt-driven mode. Related studies that characterize packet loss and the resulting performance degradation over 10 Gbps Wide-Area Network (WAN) include [4,16,17]. The study in [18] focuses more on the architectural sources of latency rather than throughput of intra-datacenter links.

Another method of improving the adverse effects of the end-system bottleneck involves re-thinking the hardware architecture of the end-system altogether. NICs optimized for specific transport protocols have been proposed along these lines [19]. End-system architectural reorganization has also been proposed in [20]. Unfortunately, too few of these changes have found their way into the type of commodity end-systems that have been deployed for the purposes of these tests.

3.1. Protocol processing parallelism

Since end-system processor architectures have been scaling out to multiple cores, rather than up in clock speed, systems designers have faced unique challenges in exploiting this parallelism for network-related processing. There have been several related, but essentially discrete methods to this end.