An Evaluation of Parallel Optimization for OpenSolaris® Network Stack

Hongbo Zou[*§], Wenji Wu[§], Xian-He Sun[*], Phil DeMar[§], Matt Crawford[§]
[*] Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616
[§] Computing Division, Fermi National Accelerator Laboratory, Batavia, IL 60510
{hzou1, sun}@iit.edu, {wenji, demar, crawdad}@fnal.gov

***Abstract*— Computing is now shifting towards multiprocessing. The fundamental goal of multiprocessing is improved performance through the introduction of additional hardware threads or cores (referred to as "cores" for simplicity). Modern network stacks can exploit parallel cores to allow either message-based parallelism or connection-based parallelism as a means to enhance performance. OpenSolaris has redesigned and parallelized to better utilize additional cores. Three special technologies, named Softring Set, Soft ring and Squeue are introduced in OpenSolaris for stack parallelization. In this paper, we study the OpenSolaris packet receiving process and its core parallelism optimization techniques. Experiment results show that these techniques allow OpenSolaris to achieve better network I/O performance in multiprocessing environments; however, network stack parallelization has also brought extra overheads for system. An effective and efficient network I/O optimization in multiprocessing environments is required to cross all levers of the network stack from network interface to application.**

Keywords-Network Stack; OpenSolaris;

I. INTRODUCTION

As network bandwidths continue to increase at an exponential pace, network stacks for uni-processor architecture cannot keep pace with such growth in order to efficiently utilize that bandwidth. At the same time, the growing challenges of power consumption and heat dissipation on single-core processor make the computing industry shifting to multi-core architecture. Therefore, it is a trend to use parallel processing core to optimize network stack on multi-core architecture to make up for the loss in performance growth of uni-processor architecture.

The network stacks of mainstream operating systems have already been parallelized. However, parallelization has also brought extra overheads for OS (Operating System): contention for shared resources, software synchronization, and cache inefficiency [2]. Therefore, how to reduce these overheads have been the leading challenges on network stack parallel optimization. Investigations [1] indicate that the coordinated affinity scheduling of network stack processing and network applications on the same processor can significantly reduce the extra overheads. The coordinated affinity scheduling of network processing and network applications on the same core has three goals: interrupt affinity, flow affinity, and network data affinity. Interrupt affinity implies that network interrupts of the same type should be directed to a single core. Flow affinity means that packets of each data flow should be processed by a single core. Network data affinity means that TCP/IP network processing and network applications should be scheduled on the same core to maximize cache efficiency.

The OpenSolaris network stack architecture (internally named FireEngine) went through multiple transitions by which the core pieces (e.g., socket layer, TCP, UDP, IP and device driver) using Softring Set (SRS), Soft ring and Squeue (serialization queue) [4]. In the hope of maximizing network stack parallelism, FireEngine employs more threads on network stack processing [6]. In some degree, this strategy addresses the increasing performance demands of network-centric applications and workloads. However, this strategy has brought potential side effects that more threads are bound to incur more context switch and decrease the possibility of cache affinity on certain circumstance. This paper will explore the strength and weakness of OpenSolaris network stack in multiprocessing environments. Because high-performance TCP data receiving is a challenge to network stack performance, our study mainly focus on OpenSolaris TCP packet receiving process.

The rest of the paper is organized as follows: Section II gives an introduction of Solaris network stack history. Section III studies the OpenSolaris packet receiving process and its optimization techniques. The evaluation and experimental results are presented in Section IV. Finally, Section V concludes the paper.

II. BACKGROUND

The network stack of Solaris 1.x was a BSD variant and was very similar to the BSD Reno implementation. Solaris 2.x migrated to AT&T SVR4 architecture. With SVR4, the network stack went through a transition from a BSD style stack to a STREAMs-based one [5]. During the late 90s, with the prevalence of the multi-processor and high speed NIC, how to parallel process high speed network flow on multiple processors became an urgent problem to OSes [1]. OpenSolaris network stack went through one more transition to improve its parallelism.

The FireEngine is the new network stack developed by Sun to meet the current and future networking needs. Three special technologies: SRS, Soft ring and Squeue are introduced for stack optimization. SRS is responsible for collecting incoming packets from Rx ring and classifying them into each soft-ring. Soft-ring is a software abstraction of hardware Rx ring to enhance system parallelism granularity. In addition, Squeue tackles multiple threads synchronization and mutual

This research was supported in part by the Universities Research Association (URA) Visiting Scholars Program and IIT Fieldhouse Research Fellowship.

exclusion from IP to sockfs. These techniques allow OpenSolaris network stack possible to achieve much better performance in multiprocessing environments.

OpenSolaris is a thread-based, fully preemptible system. To achieve the objectives of fairness, interactivity and efficiency, the Solaris kernel implements a global priority model to schedule threads running in a prioritized round robin manner. There are totally 170 values for thread priority assignment. Interrupt threads (160-169), Real-time (RT) threads (100-159), system threads (60-99), and user threads (0-59) are assigned with different priorities, respectively. A higher-priority thread preempts a lower one running on a core in runtime. OpenSolaris network stack are executed with threads. Using multi-threads to handle packet processing can maximize network stack parallel processing and reduce overall respond time [3]. However, preemption in packets processing and extra context switch incur some potential negative effects on network performance, especially in the high-speed networking environments.

III. SOLARIS PACKET RECEIVING PROCESS

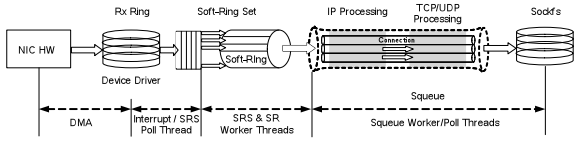


Figure 1. Solaris Networking Packet Receiving Process

An OpenSolaris network stack TCP packet receive path is illustrated in Figure 1. On packet reception, there are totally six types of threads working cooperatively in the network stack to deliver an incoming packet from its ingress to its final application. SRS, Soft-ring, and Squeue are the main components of OpenSolaris network stack. They are actually data structures to queue incoming packets at different protocol levels. Every data structure has an attached worker or poll thread to perform the corresponding protocol processing. According to the scope of these threads, the packet reception can be roughly classified into three stages [5]: (1) Packets are received and classified from network interface card (NIC) to MAC layer's SRS, which are driven by Device Driver Processing; (2) packets are spread to multiple Soft-Rings in SRS by SRS worker thread. And then, every SR worker thread continues the MAC Layer Processing to deliver packets to IP interface; (3) depending on the speed of packets arriving to Squeue, either Squeue worker or poll thread will execute merged TCP/IP modules in Squeue exclusively. Finally, Packet data is copied from the socket receive buffer to the application. The following section details these three stages.

A. Device Driver Processing

The Device Driver performs the layer 1 and 2 functions of the OSI 7-layer network model. When packets are received by the NIC (assuming, NIC does not support RSS technology), the NIC will generate interrupts and inform the cores to respond to the interrupt requests. A responding core then suspends its current thread and invokes the corresponding NIC interrupt thread to execute interrupt handler. After the interrupt handler is completed, then interrupt thread surrenders the core to the interrupted thread and goes to sleep till a new packet arrives. An interrupt thread with higher priorities is scheduled and preempted as a normal one to classify and forward incoming packet to SRS queues. After the packet delivery is completed, the interrupt thread wakes up a SRS worker thread to take over the next processing (shown in Figure 2i a). The process is called the interrupt mode. If network interrupts come too fast and the SRS worker thread is not able to handle incoming packets timely, a poll thread will be waked up by the worker thread to stop further NIC interrupts and pull a chain of packets from the Rx ring time to time, which is named the poll mode. The poll thread will switch the NIC back to the interrupt mode when there are no packets queued in Rx-ring. On the receive side, a one to one mapping exists between a SRS and a NIC hardware Rx ring. Thus, each individual SRS can switch the hardware Rx ring processing between the interrupt thread and the poll thread to control incoming path bandwidth without impacting each other. After packets have been delivered into SRS, a software classification is executed by the SRS worker thread to spread incoming packets to different soft rings.

An RSS-enabled NIC supports multiple Rx rings. Each ring is assigned a separate network-interrupt, and hence an interrupt thread. To further improve the network-processing efficiency. OpenSolaris network stack applies a fast path mechanism that the SRS and soft ring data structures will be bypassed by networking processing if the NIC supports the RSS technology (Figure 2i b). Under such conditions, a Squeue has a one to one mapping with an Rx ring. When network packets arrive, the RSS-enabled NIC classifies and steers incoming packets into different Rx rings. The associated interrupt thread for each Rx ring delivers incoming packets to the mapping Squeue directly. In addition, Squeue poll thread replaces SRS thread to pull the incoming packets from the corresponding Rx rings owned by each Squeue and dynamic switches the Rx ring between interrupt and polling mode to control the rate of interrupt and packet receiving.

B. Mac Layer Processing

Once the SRS worker thread has been waked up (non-fast-path case), the software classification will be executed to sort incoming packets as previously mentioned. A full-featured software classification is used when the NIC is not capable of classifying based on L3/L4 headers, or is out of hardware Rx rings. Software classification is performed by interrupt thread (interrupt thread could continue to handle packets delivered if there are no other packets backlogged on

SRS queue) or SRS worker thread very early to assign a packet to one of soft rings associated with the SRS according to load spreading policies (e.g., hash of src IP address or TCP/UDP protocol).

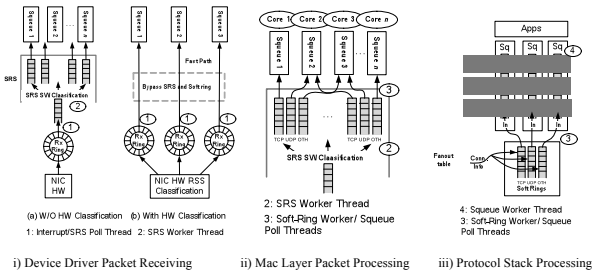


Figure 2. Three Stages of the Packet Receiving

The software classification functionality of the SRS and soft rings provides the classification capability in the MAC layer. These software modules aim to improve network stack parallelism if RSS is not available. Usually, a single Squeue is assigned to a SRS in the absence of soft rings, or to each soft ring within a SRS. With this assignment approach, Squeue could control its bandwidth with its queue backlog state through dynamic switching between soft ring worker thread and Squeue poll thread. However, Soft rings within a single SRS can be assigned to different Squeues. This function is named fan-out. Since each Squeue is tied to a separate core, the fan-out function spreads incoming traffic to different cores and improves OpenSolaris network stack parallelization in the MAC layer if RSS is not available. After classification, SRS worker thread will wake the corresponding soft ring worker thread up to process its incoming packets on different cores in parallel and independently. With this mechanism, TCP processing load could be spread to multiple cores. Figure 2ii illustrates packets classifying process with the software classification engine.

If an RSS-enabled NIC can classify incoming packets to different Rx rings, the packets could be handed over to IP layer directly by means of function calls. The entire MAC layer processing will be bypassed. This is the fast-path mechanism that we have discussed earlier. Such conditions happen in the contexts that interrupt thread sends the incoming packets to Squeue directly or the Squeue polls the packet chain from the Rx ring with Squeue poll thread for flow control.

C. Protocol Stack Processing

Before incoming packets are further delivered by soft ring worker thread to Squeue (or pulled by Squeue poll thread), IP connection classifier creates or looks up a connection structure for each inbound packet. Based on the classification, each packet is attached to a connection state and queued in the Squeue to which its connection is bound (Figure 2iii). After being delivered to Squeue, the packet could be processed directly by soft ring worker thread, or Squeue poll thread, or queued for later processing by other threads. The choice is determined by the Squeue entry point and the state of the Squeue. A thread can enter Squeue for immediate processing only when there is no other thread accessing the same Squeue [4]. The Squeue only allows an external thread (e.g., Soft ring worker thread, interrupt thread on fast-path case) to do processing with a finite duration, after which it switches processing to Squeue worker thread. In the Squeue, Squeue poll thread can be waked up at any time to take over current processing when packets have been backlogged. Because OpenSolaris only allow a single thread to enter Squeue at any given time, threads have been serialized to access the TCP connection structure in the merged TCP/IP modules. This mechanism protects the whole connection state from IP to sockfs. Finally, application thread will be woken up to get its data packets on its sockfs buffer.

IV. Analysis and Experiment

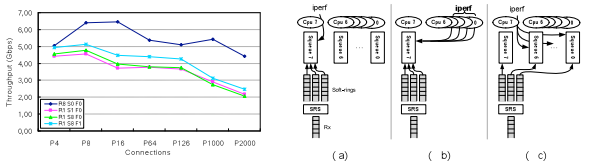
We ran data transmission experiments on two machines connected back-to-back with optical fibers. In the experiments, iperf transfers TCP data in the direction from the sender to the receiver. The sender has two Intel Xeon Dual-core 3.80GHz processors, 8GB memory, and a Myri-10G NIC, running Linux 2.6.23. The receiver has two Intel Xeon Quad-core 2.66GHz processor, 16GB memory, and a Myri-10G NIC, loaded with OpenSolairs SNV129.

A. Maximize Parallelism

The experiments aim to evaluate the effectiveness of various parallelism techniques of OpenSolaris network stack. In the experiments, iperf transmits TCP data for 100 seconds. The following parameters are varied: 1) the number of Rx rings in the NIC; 2) the number of soft rings; 3) enabling/disabling the fan-out function. For simplicity, we labelled an experiment as R*a*S*b*F*c*. Here, R*a* represents *a* Rx rings; S*b* refers to *b* soft rings; and F*c* refers to whether enable (*c*=1) or disable (*c*=0) the fan-out function. There are totally four experiments, R8S0F0, R1S8F1, R1S8F0, and R1S1F0. In addition, we varied the number of TCP connections. Consistent results are obtained across repeated runs. The experiment results are as shown in Figure 3i. It can be seen that R8S0F0 achieves much higher throughputs than other experiments. This is because the RSS-enabled NIC will effictively spread connections to different cores, and hence increases the overall parallelism in the receiver. The throughput of 16 connections reaches a peak of 6.47 Gbps. When the number of connections is further increased, the achieved throughput starts to decrease. This is because that the increasing parallelism brings extra overheads for OS, which finally offsets the parallelism gains.

It also can be seen that the software classification functionality of the SRS and soft rings improves network stack parallelism if RSS is not available. The experiments show the effectiveness of this mechanism. R1S8F1 achieves better throughputs than R1S1F0 and

R1S8F0. In addition, we evaluate the fan-out function by comparing R1S8F0 with R1S8F1. As for R1S8F0, since the fan-out function is disabled, the 8 soft rings are actually tied to a single core, the interrupt-handling core. As a result, only one thread can access these soft rings at any given time, with limited network stack parallelism. As for R1S8F1, the fan-out function actually assigns the 8 soft rings to multiple cores; multiple threads can assess these soft rings at any given time, which maximizes network stack parallelism. The experiment results verify our argument and clearly show that the throughputs of R1S8F1 are much higher than those of R1S8F0.



i) Network Throughput on 4 Configuration ii) Comparison Experiment Logical Flowchat

Figure 3. Comparison Experiment

B. *Enhance Core Affinity*

OpenSolaris network stack supports interrupt affinity and employs the per-core Squeue mechanism to ensure flow affinity. We further evaluated network stack performance from the perspectives of data affinity. We designed three experiments as shown in Figure 3ii: (a) network interrupts and iperf are bound to core 7, and the soft ring fan-out function is disabled. (b) The network interrupts are bound to core 7 and the soft ring fan-out function is disabled. Iperf is bound to a core set (from core 0 to 6). (c) iperf and NIC interrupts are bound to the core 7 and the fan-out function is enabled. In the experiments, iperf transmit 64 parallel TCP connections from the sender to the receiver for 100 seconds. The performance metrics are collected in the receiver. The metrics of interest are: 1) smtx - the number of times cores failed to obtain a mutex immediately; 2) migr - the number of thread migrations to between cores; 3) l2_miss – the number of L2 cache misses; 4) bus hitm – the number of memory transactions with caused HITM to be asserted; 5) sq_lock - the number of Squeue adaptive mutex hold events. Consistent results are obtained across repeated runs. The experiment results are listed in Table I (the data is normalized by network bandwidth).

In (a), since the network interrupts and iperf are bond to a single core and the soft ring fan-out function is disabled, network processing and application are executed on a single core. In the experiments, we observed that Core 7's CPU utilization ratio reaches 100%, and other cores' CPU utilization ratios are almost 0%. The observation confirms that network processing and application are executed on a single core. Therefore, (a) guarantees data affinity in network processing. In (b), core 7's CPU utilization ratio now reduces to 67%; the average workload on the core set (core 0 – 6) is 23%. The throughput of (b) is almost twice that of (a). In (c), we bind iperf to core 7 and enable the soft ring fan-out function. As a result, the incoming network traffics are spread across to different cores. For both (b) and (c), the network processing and application are scheduled on different cores. However, the hardware performance metrics clearly show that scheduling iperf and network processing on the different cores will cause significant extra costs (see Table I). When network processing and application are scheduled on different cores, it will cause three negative side effects: (1) Concurrent threads contention for shared resources (see the smtx and sq_lock comparison); (2) Software synchronization overheads (see the bus_hitm comparison); (3) Inefficient cache usage (see the l2_miss comparison). As a general purpose OS, the OpenSolaris scheduler prioritizes such properties as load-balancing and fairness over core affinity in network processing. As a result, it is more likely that network processing and applications are scheduled on different cores with more network parallelisms introduced in multiprocessing environments and leads to increased network processing overheads.

TABLE I. DATA AFFINITY ANALYSIS UNDER FIGURE 3II.

	smtx	*migr*	*l2_miss*	*bus_hitm*	*sq_lock*
(a)	**0.04**	**0.02**	**5.88**	**4.34**	**52.20**
(b)	**2.96**	**0.12**	**34.53**	**26.41**	**158.32**
(c)	**1.34**	**0.04**	**51.94**	**36.18**	**223.83**

V. CONCLUSIONS

In this paper, we study the OpenSolaris packet receiving process and its parallelism optimization techniques. Experiment results show that these techniques allow OpenSolaris to achieve better network performance in multiprocessing environments; however, network stack parallelization has also brought extra overheads for OS. A more effective and efficient parallel optimization still needs for improving the current network stack**.**

REFERENCES

[1] A. Foong, et al. "Architectural Characterization of Processor Affinity in Network Processing," In *Proceedings IEEE International Symposium on Performance Analysis of Systems and Software*, 2005.

[2] P. Willmann, et al., "An Evaluation of Network Stack Parallelization Strategies in Modern Operating System," In *Proceedings of USENIX Annual Technical Conference*, 2006.

[3] R. McDougall, et al., Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture, 2nd Edition, Prentice Hall, 2006, ISBN-10: 0-13-148209-2.

[4] S. Tripathi, FireEngine – A New Networking Architecture for Solaris™ Operating System. *A Technical White Paper*, 2004.

[5] S. Tripathi, et al., Crossbow: A Vertically Integrated QoS Stack. In *Proceedings of WREN'09,* 2009.

[6] W. Wu, et al., The Performance Analysis of Linux Networking – Packet Receiving. *Computer Communications* 30 (5) 2007.