

Packet Event Framework for the OpenSolaris Operating System

Yu Xiangning

Sun China Engineering and Research Institute
Beijing, P.R.China
eric.yu@sun.com

Bruce Curtis

Sun Microsystems Inc.
Menlo Park, CA 94025, USA
bruce.curtis@sun.com

ABSTRACT

This paper presents Packet Event Framework (PEF), an event driven framework designed for OpenSolaris networking stack. PEF introduces a layer-based approach for message delivery between different network layers or sub-layers, and it is suitable for achieving parallel processing of networking stack on modern computer systems. In order to demonstrate the efficiency and flexibility of PEF, we integrate an implementation of TCP into the framework, briefly, to split the protocol processing into three major layers representing IP, TCP, and SOCKFS. Experiments show that we gain significant performance improvement on Chip Multi-Threading (CMT) systems. We begin with a brief introduction to different approaches to networking stack parallelism. Then we reveal the related works on the architecture of parallel networking stacks. Finally, we provide the design and implementation of PEF, as well as the performance evaluations.

1. INTRODUCTION

The exponential growth of network bandwidth demands that software, networking stack in particular, to catch up with the speed of network interfaces (NIC). Nowadays, in a uni-processor computer system, a conventional 10G NIC would result in a single execution unit, a CPU, or a core, or even a hardware thread, being the bottleneck. And the industry begins to seek new ways by replacing uni-processor with multi-core processors. Sun's Niagara series are typical examples, the first generation of Niagara, UltraSPARC-T1[3], contains eight cores on the single chip and each capable of running four hardware threads simultaneously, and its follow on Niagara II, supports eight hardware threads. Therefore, it is quite necessary for networking stack to do scalable parallel protocol processing.

Previous work on different approaches of networking parallelism can be found in [7, 9, 6, 11, 10]. Nahum, Yates, Kurose, and Towsley[6] evaluated the performance issue in parallel network protocols. According to their findings, preserving order in TCP pays for performance degradation. What is more, they revealed that single connection TCP parallelism is limited in packet-based parallelism both on the send side and the receive

side, while multiple connection TCP parallelism scales by eliminating connection state lock. Finally, they confirmed that lock contention and cache behavior remain as crucial factors that greatly influence performance.

Willmann, Rixner, and Cox[9] evaluated different strategies of networking stack parallelism in modern operating systems. They mainly analyzed three types of approaches: packet-based, connection-based using thread, and connection-based using lock. They found that any of the above approaches outperforms a uni-processor. Also, they found that even though with different approaches to parallelism, locking overhead, cache efficiency, and scheduling overhead remain as the key factors to prevent networking performance on multi-core processors from perfectly speeding up. However, they didn't address the layer based approach to networking stack parallelism.

Tripathi and other engineers from Sun designed and implemented the FireEngine[7] architecture for Solaris 10, which follows a connection-based approach in networking parallelism. This approach has greatly improved multiple connections performance on the Solaris operating system. However, it does not address single connection performance. Also, FireEngine introduces a per-execution unit synchronization mechanism that guarantees there is only one thread can do the protocol processing at any given time even on a multi-execution unit computer system. Thus, when handling single TCP connection on computer systems with multiple execution units, one or two of the execution unit might be very busy processing while the rest remain idle.

Having witnessed the issue of single connection performance on multi-execution unit computer systems, therefore, in this paper, we propose the Packet Event Framework[2], which is currently under development in Sun Microsystems Inc. and is already open sourced at the OpenSolaris community[1]. The main contributions of PEF are the following:

- PEF builds a new infrastructure for parallel processing of network protocols on top of current OpenSolaris[1] networking stack.

- PEF explores networking performance on multi-core computer systems, especially single connection TCP performance.
- PEF enlarges the Solaris networking feature sets by providing new kernel APIs that allow researchers and developers to integrate new protocols easily.

The remainder of this paper is structured as follows: Section 2 provides the background on different approaches to networking parallelisms and a brief introduction to FireEngine. Section 3 presents the design and implementation of the Packet Event Framework. The performance evaluation is presented in Section 4, followed by future work and conclusions.

2. NETWORKING PARALLELISM

Researchers have proposed and designed different approaches to exploit networking stack parallelism in modern operating systems, and these approaches can be classified to three main categories: message or packet based parallelism, connection based parallelism and layer based parallelism, each of which is discussed as follows:

Packet based parallelism has been widely deployed as most of modern operating systems, such as Linux and FreeBSD, allows multiple threads to process different packets from same or different connections. This approach results in poor data locality on multi-execution unit servers for the reason that any given packet can be processed unpredictably on different execution units. Meanwhile, packet based approach brings much synchronization overhead to maintain packet order between different stack layers.

Connection based approach explores the stack parallelism by classifying packets early in the networking stack in a per-connection basis and assigning each connection to an execution unit. This per-connection parallelism provides better data locality for protocol processing, as packets belonging to multiple connections can be processed separately on different execution units, and consequently greatly improved multiple connections parallelism. However, it does not improve the throughput of single connection for the reason that single connection utilizes limited execution units in this case.

The FireEngine networking stack in Solaris 10 is a typical example of connection based parallelism. It merges all protocol layers into one fully multi-threading STREAMS[8] module, which was separated modules in earlier releases of Solaris. And FireEngine combines all the connection specific data structures into one data structure called "conn_t". FireEngine also introduces a per execution unit synchronization mechanism with the name "Vertical Perimeter", which is implemented as a serialization queue called "squeue_t" and provides all the necessary synchronization and mutual exclusive

access to connection specific data structures, thus guarantees that only a single thread can process a given connection at any time. What is more, FireEngine introduces an early classification mechanism that does IP classification right after the packet arrives at IP.

Layer based parallelism splits the networking stack horizontally into different layers or sub-layers, each of which is processed on different execution units, resulting in that the packets are processed between layers through inter-process communication via a pipeline fashion. Thus no packet order problem is involved compared to packet based approach, and the performance of single connection can be greatly improved on multi-execution unit servers compared to that of connection based approach.

3. DESIGN AND IMPLEMENTATION

In this section, we begin with the architecture design of Packet Event Framework. Then we present the implementations in detail. TCP is used as a demonstration throughout the discussion.

3.1 Architecture

As we discussed in Section 2, the FireEngine networking stack introduces a per-execution unit synchronization mechanism called `squeue_t`. On top of that, we expand the `squeue_t` semantics and design a framework with fine-grained modularity of the networking stack based on the execution of a series of protocol event functions specified in the IP classifier. Each event links together to be a single list (the event list), each of which represents a layer or sub-layer in the network stack.

We achieve the art of networking parallelism by allowing different layers of protocol processing to be done under the synchronization of different `squeue_ts`. Thus the processing of different layers can scale on multiple execution units. The followings are two fundamental approaches of networking parallelism in PEF:

- **Horizontal Parallelism**

In this scenario, we split the protocol processing horizontally into different layers. Take TCP for instance, it can be split into three major layers representing IP, TCP, and SOCKFS respectively, each of which is processed on a designated execution unit. Therefore, when a packet traverses through the stack, the processing of different layers are done on different execution units. And consequently, the processing of packets belonging to the same connection scale over multiple execution unit. Fig 1 (a) depicts the scenario of per-network layer per-event in the receive side. In this case, the `squeue_t` perimeter actually becomes the "Horizontal Perimeter".

The Solaris networking stack that is prior to Solaris 10 FireEngine also introduced a mechanism

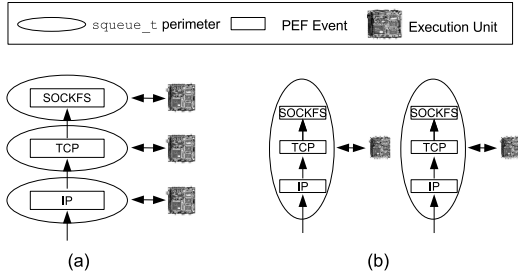


Figure 1: Different Approaches of Parallelism in PEF.

of horizontal parallelism. It is a STREAMS-based stack and each protocol is implemented as a STREAMS module, and the processing of each protocol layer can be spread on multiple execution units as well. However, the STREAMS by nature lacks of CPU affinity, and the switching processing from one execution unit to another leads to poor data locality and code locality. While with PEF, the processing of each protocol layer can be designated to any execution unit, hence there is less switching of protocol processing.

- **Vertical Parallelism**

This scenario is identical to that of FireEngine. By assigning the whole stack to the same `squeue_t`, we achieve parallelism in a connection-based style for multiple connections. As illustrated in Fig 1 (b), the IP, TCP, and SOCKFS processing are all within the same *Vertical Perimeter*, such that the same connection gets processed on the same perimeter, and the processing of multiple connections scales on different execution units.

By default, PEF randomly chooses a `squeue_t` for all the protocol layers as FireEngine already does. While PEF will allow application developers specify different execution units for different protocol layers.

3.2 Implementations

In the following sections, we will provide a detailed discussion of each PEF component one by one:

3.2.1 Event and Event List

The PEF event is a simple data structure that contains a callback function, representing the processing of a protocol layer or sub-layer in the network stack, and all the events link together to represent the whole stack. Also, each connection possesses its own event list both on transmit side and receive side. Further more, we allow the framework to dynamically alter this event list on the fly.

As mentioned above, the event structure contains an event callback function that can be called by PEF, the event function accepts a `pkt_t` pointer, which we will discuss later, as the only arguments and returns an enumerate type. Also, each event can optionally specify a `squeue_t` perimeter, which allows it to utilize the synchronization and mutual exclusive access provided by the `squeue_t` framework. Otherwise, the event will be executed outside of any perimeter.

PEF introduces a group of APIs to allow researchers and developers to alter the event list, and these APIs can be summarized in the following two categories:

- **Static Registration** This API is used by other modules within the OpenSolaris kernel to register their own default event at startup time. For example, when TCP, or SOCKFS is loaded at boot time. The API is defined as follows:

```
pef_evt_register_event()
```

- **Dynamic Altering**

After a PEF enabled connection is established, this API can be used to dynamically alter the connection's default event list. For instance, we can provide better observability for a given TCP connection by inserting a trace event in the connection's event list. The API is defined as follows:

```
pef_evt_insert_front() - Insert an event to the beginning of a major layer for a given connection, for instance, if the event is an TCP event, insert as the first TCP event.
```

```
pef_evt_insert_back() - Insert an event to the end of a layer in the event list of a given connection.
```

```
pef_evt_disable() - Disable an event in the connection's event list that has already been inserted.
```

Please note that so far this paper is written, these APIs are still during evolution, and they can be changed in the future.

3.2.2 PEF Socket Options

PEF will introduce several experimental socket options to allow application developers to issue certain operations on a socket. These socket options are defined as follows:

level	name	description
IPPROTO_IP	SO_PEFON	activate PEF on a TCP socket
IPPROTO_IP	SO_SETSQP	designate an execution unit for a certain protocol layer

3.2.3 PEF Classification

PEF classification inherits the FireEngine IP classifier, and it sits at the bottom of the IP stack on the receive side. When an IP packet arrives at IP, it is classified by looking up in the connection hash tables. If the classification succeeds, the packet, which means a STREAM message in Solaris, will be encapsulated into a new data structure called `pkt_t`. A `pkt_t` contains the following information: (1) All the packet meta data including per-layer data pointers from Ethernet to application payload. (2) The connection (`conn_t` pointer) it belongs to. (3) The first event to execute, which is initialized as the first element of event list on the receive side.

Since the main purpose of PEF classifier is to establish the relationship between a packet and the connection's in-kernel transport layer data structure `conn_t`, there is no need to do classification on the transmit side because the SOCKFS already knows about the transport layer.

3.2.4 PEF Execution

PEF execution serves as the core component of the Packet Event Framework. It can be invoked in the well designed code positions throughout the stack: On the receive side, IP will pass the control to PEF execution after the packet is successfully classified. On the transmit side, given a user buffer, SOCKFS encapsulates the data into a `pkt_t` in the user context, and passes the control to PEF execution. Also, PEF execution interacts with the `squeue_t` framework such that the `squeue_t` worker thread, which is bound to an execution unit, can pick up the protocol processing.

Then the `pkt_t` traverses through the event list as the only argument to each PEF event on transmit side or receive side respectively. PEF execution ensures that each event function is executed sequentially. Also, according to the return values from the event function, PEF execution will take different actions described as follows:

- **NEXT:** PEF execution will continue to fetch the next event in the event list, and pass in the current `pkt_t` as the argument. If the last event in the event list is NULL, the processing for the current `pkt_t` is done.
- **DONE:** The processing of the current `pkt_t` is done, and PEF will stop executing on this `pkt_t` even if there are un-executed events in the list.
- **RECL:** PEF execution will perform reclassification on the current `pkt_t`, which means the packet will go through the IP classifier again. Since there is no classification on the transmit side, this value is used only in receive side.

Moreover, PEF execution guarantees that each event is executed strictly under the mutual exclusive protection of the `squeue_t` perimeter if specified. In order to achieve the goal of networking parallelism, we assign different `squeue_t` perimeters to the different events. Typically, we can split the protocol processing horizontally into three major events, representing the protocol layers from IP to TCP and SOCKFS, and assigning these events each with a different `squeue_t`. When a `pkt_t` traverses through the stack, it will be processed under different protections of `squeue_ts`, since the `squeue_t` is a per-execution unit synchronization mechanism, the `pkt_t` consequently gets processed on different execution units.

3.3 Prototype TCP Event List

In this section, we present a brief introduction to the implementation of the prototype TCP event list that is used as the demonstration of PEF. Our prototype is a coarse-grained event list, that is, the network stack is split horizontally into major layers such that IP, TCP, and SOCKFS owns one single event.

• Non-STREAMS data path

The TCP implementation in OpenSolaris is complicated. one of the reason is the heavy dependency of STREAMS. The traditional STREAMS framework queues the data at the STREAM head, which involves a large amount of code and requires much processing time. Hence, in our demonstrative TCP prototype event list, PEF introduces a non-STREAMS data path for the receiver side between TCP and SOCKFS. The STREAM head read queue has been replaced by a novel data structure called `pef_q_t`, which provides a lighted-weighted mechanism to allow TCP perform STREAMS style semantics such as `canputnext(9F)`, `putnext(9F)`, and `getq(9F)`. Furthermore, a PEF enabled socket will perform flow control based on the information from `pef_q_t`, other than from the STREAM head read queue.

• TCP Fallback

The PEF TCP event is a light-weighted TCP implementation with fallback compatibility to the processing of FireEngine. Briefly, the PEF TCP event performs a series of header predictions based on the status of current TCP data structure, `tcp_t`, and the content of the incoming packet. Then PEF makes a decision whether to fallback to normal TCP data processing or to follow the fast path provided by the prototype TCP event. The fallback code path has been modified to be PEF aware that is capable of pushing the data to the proper upper layer queue, that is, `pef_q_t` or STREAMS head queue.

4. PERFORMANCE EVALUATION

In this section, we provide the detail of software and hardware setup for performance measurements. Also, we will present the analysis to the experiment results. We emphasize on both TCP throughput performance and TCP latency performance, and so far our PEF implementation is based on OpenSolaris build 48, so we use that as baseline and make comparison with it.

4.1 Software Benchmark

4.1.1 Netperf

The netperf micro-benchmark is a socket-based application for measuring TCP and UDP throughput tests, the revision used throughout the tests are 2.4.1. We use various test setups to measure the performance of TCP throughput with send and receive socket buffer sizes set to 64KB. Each test runs 120 seconds and is repeated multiple times.

4.1.2 Iperf

This is a popular socket-based micro benchmark program to measure TCP and UDP throughput. The version we used for measurement is 1.7.0a. In our experiments, both the socket send buffer and receive buffer are set to 64KB, and packet size ranging from 32 bytes to 64KB are addressed. Each test runs for 120 seconds and is repeated multiple times after 120 seconds warm-up.

4.1.3 Ping-Pong

This is a simple but popular socket-based program that measures end-to-end latency. The latency value is represented by the delta time between `sendto(3SOCKET)` and `recvfrom(3SOCKET)`. In our experiments, latency of socket write buffer size ranging from 8 bytes to 64KB are measured. Also, for a given write buffer size, the test is repeated 10000 times and the average time is calculated.

4.2 Test Environment

We use a Sun Kirkwood 10Gigabit Ethernet back-to-back system to perform the experiments. The receiver is a Sun Fire T2000 server, which is a 8x4x1.2GHz (eight cores with four threads per-core) UltraSPARC-T1 with 32 GB RAM. And the sender is a Sun Fire X4200 with two way 2.6GHz dual-core AMD Opteron and 8 GB RAM.

4.3 Experiment Results

Of all the tests, we compared both the TCP throughput and latency performance of PEF and the baseline, OpenSolaris build 48.

Fig 2 demonstrates the single connection TCP throughput tests across various message sizes. It illustrates that

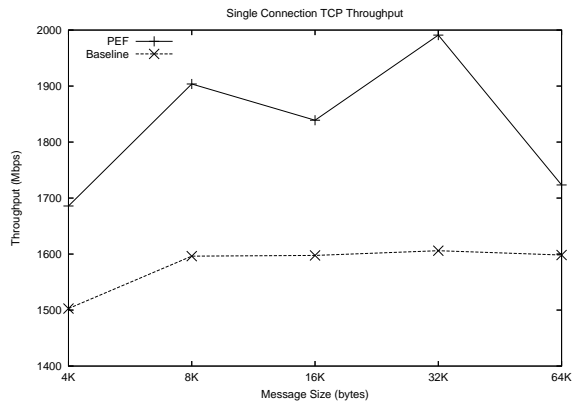


Figure 2: Large TCP Packets on T2000

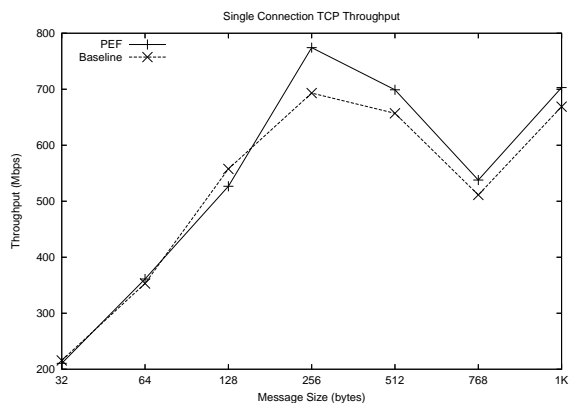


Figure 3: Small TCP Packets on T2000

together with our prototype TCP event list, the implementation of Packet Event Framework provides further performance improvement (approximately 7% to 20%) when compared with OpenSolaris.

Fig 3 illustrates another aspect of single connection TCP throughput performance. When the socket write size is relatively small, that is, ranging from 32 bytes to 1KB, and with TCP_NODELAY option set, PEF outperforms baseline slightly, though some of the tests shows degradations.

We use Iperf to aggregate the TCP throughput performance number with multiple connections. Fig 4 and Fig 5 shows the aggregated TCP throughput performance of PEF and the baseline. The softirq[8] mechanism in OpenSolaris allows the protocol processing of multiple connections to fanout to different execution units. The OpenSolaris network stack provide some kernel tunables to turn softirq fanout on and off. Fig 4 demonstrates the setup of disabling softirq fanout on a Sun T2000, the aggregated connection numbers range from 6 to 192. As we can see from Fig 4 and Fig 5, PEF outperforms baseline both with softirq enabled

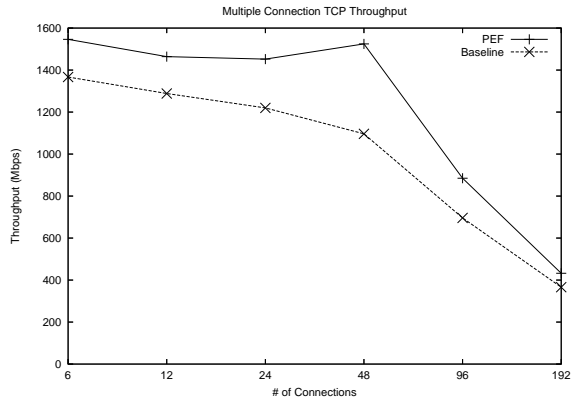


Figure 4: Multiple connections on T2000 without softing

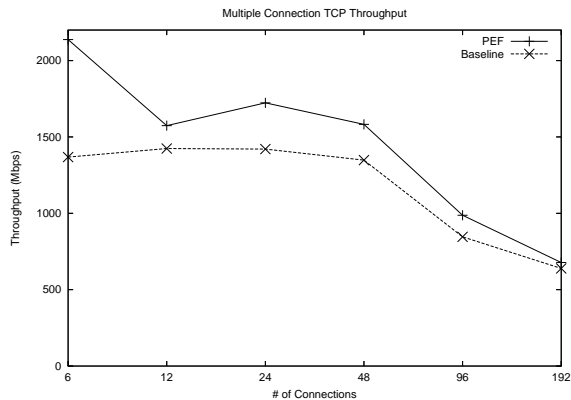


Figure 5: Multiple connections on T2000 with softing

and disabled, the performance improvement is between 6% to 56%.

Fig 6 illustrates that PEF shows significant performance improvement for some certain workloads, that is, by setting both the socket read and write buffer to the same value to emulate small messages delivering with Iperf single connection tests, PEF outperforms baseline significantly at about 60% to 800% improvement.

We use Ping-Pong to collect latency experiment results, and our tests on the T2000 and X4200 pair indicate that latency reductions ranges from 10% to 20%, as Fig 7 illustrated.

5. FUTURE WORK

First of all, we will continue to evaluate the APIs to allow researchers and developers to integrate their own PEF events into the framework. With these APIs, we envision a developer is capable of concentrating on providing protocol specific logic and does not have to

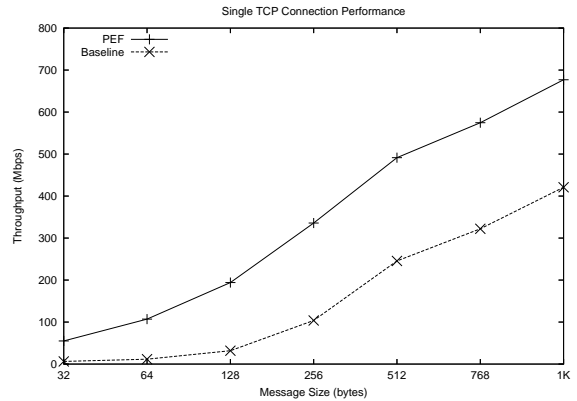


Figure 6: Small TCP Messages Delivery

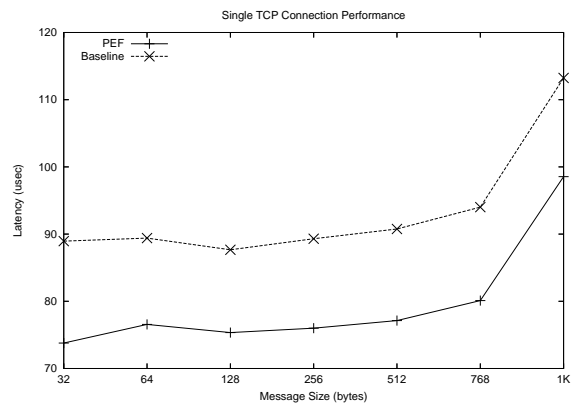


Figure 7: TCP(IPv4) Latency

handle too much low level kernel related works. Also, with such APIs, PEF could be an alternative in terms of providing better networking observability for the OpenSolaris operating system.

Further more, since all of the demonstrations and experiments presented above are based on full-bound TCP connections. However, it would make sense to allow non-bound packets, for instance, the forwarding packets, to utilize the framework. Hence we will further investigate the possibility of making PEF suitable for non-bound packets processing, an obvious approach is to move the PEF classifier, which currently sits at the bottom of IP, to the lower of the networking stack, for instance, the GLD[8] level.

Also, in the implementation side, PEF introduces a novel per-packet data structure called `pkt_t`, and our current implementation places `pkt_t`, `mblk_t`, and the underlying `dblk_t` into different slab caches[4, 5], which leads to different kernel pages. This has been observed to lead to greater number of data TLB misses on several architectures. Therefore, we may modify the current implementation to place the `pkt_t`, and the `mblk_t` to

continuous memory region to achieve better data localities.

Finally, having demonstrated that TCP can be integrated into the Packet Event Framework, we will continue to seek solutions to make the TCP event list more fine-grained, and we will merge more other protocols, such as UDP, into the framework as well.

6. CONCLUSIONS

Networking performance is of important for of all the networked computer systems, while the gap between the bandwidth of modern network interface and the computation ability of single execution unit keeps increasing. Hence it is necessary to address the performance issue, particularly single connection performance on CMT and other computer systems. This paper presents the Packet Event Framework, which provides a framework that allows the networking stack to be split into a fine-grained event list, and allow the parallel processing of the networking packets in a pipeline fashion. PEF provides both flexibility and efficiency for the OpenSolaris networking stack, and in our implementation of a prototype TCP event list, the eliminating of STREAMS data path between TCP and SOCKFS, and the light-weighted TCP event result in significant performance improvement.

7. ACKNOWLEDGMENTS

Many thanks to Mike Cheng and Alex Peng for their hard working on the development and testing of PEF. Also thanks to the management, Markus Flierl and Darin Johnson for their enduring support. We'd also like to thanks Peter Memishian for having thoroughly reviewed this paper. This project is also made possible thanks to the many other contributors from Solaris Networking.

8. REFERENCES

- [1] Opensolaris community homepage at <http://www.opensolaris.org>.
- [2] Packet event framework on opensolaris homepage at <http://www.opensolaris.org/os/project/pef>.
- [3] Ultrasparc iv processor architecture overview. White Paper, Feb. 2004.
- [4] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98, 1994.
- [5] J. Bonwick and J. Adams. Magazines and vmem: Extending the slab allocator to many CPUs and arbitrary resources. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 2001.
- [6] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. F. Towsley. Performance issues in parallelized

network protocols. In *Operating Systems Design and Implementation*, pages 125–137, 1994.

- [7] S. Tripathi. Fireengine - a new networking architecture for the solaris operating system. White Paper, June 2004.
- [8] S. Tripathi. The solaris network stack. *Solaris Internals Second Edition*, pages 885–898, 2006.
- [9] P. Willmann, S. Rixner, and A. L. Cox. An evaluation of network stack parallelization strategies in modern operating systems. In *Proceedings of the USENIX Annual Technical Conference*, June 2006.
- [10] D. Yates. Connection-level parallelism for network protocols on shared-memory multiprocessor servers, 1997.
- [11] D. J. Yates, E. M. Nahum, J. F. Kurose, and D. F. Towsley. Networking support for large scale multiprocessor servers. In *Measurement and Modeling of Computer Systems*, pages 116–125, 1996.