Practice of Parallelizing Network Applications on Multi-core Architectures

[1]Junchang Wang, [2]Haipeng Cheng, [3]Bei Hua
School of Computer Science and Technology
University of Science and Technology of China
Hefei, Anhui, 230027, China

Suzhou Institute for Advanced Study
University of Science and Technology of China
Suzhou, Jiangsu, 215123, China

{[1]wangjc, [2]hpcheng}@mail.ustc.edu.cn
[3]bhua@ustc.edu.cn

Xinan Tang
Intel Compiler Lab
SC12, 3600 Juliette Lane
Santa Clara, California, 95054, USA

xinan.tang@intel.com

Abstract

The industry wide shift to multi-core architectures arouses great interests in parallelizing sequential applications. However, it is very difficult to parallelize fine-grained applications for multi-core architectures due to insufficient hardware support of fast communication and synchronization. Fortunately, network applications can be decomposed into pipelined structures that are amenable to streaming based parallel processing. To realize the potential of pipelining on multi-core architectures, it requires reevaluating the basic tradeoffs in parallel processing, including the ones between load balance and data locality and between general lock mechanisms and special lock-free data structures. This paper presents the practice of building a high-performance multi-core based network processing platform in which connection-affinity and lock-free design principles are applied effectively for better data locality and faster core-to-core synchronization and communication.

We parallelize a complete Layer 2 to Layer 7 (L2-L7) network processing system on an Intel Core 2 Quad processor, including a TCP/IP stack based on Libnids (L2-L4) and a port-independent protocol identification engine by deep packet inspection (L7+). Furthermore, we develop a compiling method to transform sequential network applications to parallel ones to enable those applications to run on multi-core architectures. Our experience suggests that (1) fine-grained pipelining can be a good software solution for parallelizing network applications on multi-core architectures if connection-affinity and lock-free are used as the first design principles; (2) a delicate partitioning scheme is required to map pipelined structures onto specific multi-core architecture; (3) an automatic parallelization approach can work if domain knowledge is considered in the parallelizing process. Our multi-core based network processing platform can deliver not only 6Gbps processing speed for large packet sizes but also more challenging 2Gbps speed for smaller packets.

ICS'09, June 8–12, 2009, York Town Heights, New York, USA.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architecture; C.2.2 [**Network Protocols**]: Applications; D.1.3 [**Programming Languages**]: Concurrent Programming – *parallel programming*; D.3.4 [**Processors**]: Compilers and run-time environments.

General Terms

Performance, Algorithms, Experimentation.

Keywords

lock-free data structures, TCP/IP protocol processing, deep content inspection, multi-core parallelization, pipelining implementation, application-level protocol processing.

1. Introduction

Previously special-purpose programmable network processors (NPUs) [21] have dramatically reduced both the cost and time to develop a network system, and have been successfully used in routers and switches [19][20]. However, programming these NPUs is very challenging since low-level hardware details are exposed to the programmers, which prevents NPUs from being widely accepted by the industry. For example, the size of each microengine on an Intel IXP 2800 NPU is maximal 8K words and there are 16 engines in total, It is difficult to partition code to fit exactly into each microengine, and it is impossible to run an application if its code space is over 128K words.

When multi-core commodity processors emerge as mainstream, they become a promising candidate for building high-performance network systems that support complicated Layer 2(L2) through Layer 7(L7) processing at the Gbps speeds. The most obvious advantages of multi-core processors are, to name a few, familiar programming environments for programmers and abundant third-party software and tools available for system development. Furthermore, multi-core processors usually have more resources, hence are more powerful than NPUs. For example, the on-chip L1 and L2 caches allow fast access to the memory while putting no restrictions on instruction space. Therefore, multi-core processors are sometimes the only solution to efficiently implementing L7+ network applications since both ASIC and NPU approaches failed

to handle the complexity appeared in the application-level protocols.

The industry wide shift to multi-core architectures [1] arouses great interests in parallelizing sequential applications. However, unlike high-performance scientific computation, network applications are time-sensitive applications. Even though coarse-grained applications in which a thread runs 10,000 cycles or more can be successfully parallelized [22][32], fine-grained network applications in which a packet must be processed within a few thousand cycles are very difficult to be parallelized because the fast communication and synchronization mechanisms that are needed for fine-grained applications are not efficiently supported on existing multi-core platforms. For example, a Pthread lock/unlock operation can easily take more than 1,000 cycles to execute while a fast-path TCP processing takes only about 2,000 cycles. For such fine-grained network applications, a lock/unlock operation becomes a new performance bottleneck, and it should hardly be used in practice.

It has been demonstrated that 10Gbps line-rate processing speed can be achieved for a single networking algorithm [16][17][34] or 1Gbps line-rate can be achieved with a dummy application using three CPU cores [3]. However, achieving 1Gbps line-rate for a complete L2-L7 network application is still very challenging, and thus the efforts were spent on parallelizing those applications on multi-core platforms [2][9][29]. Unfortunately, neither of them could reach 1Gbps line-rate in the worst case and the speeds are far below the line-rate requirement for smaller packets. We parallelize a complete TCP/IP stack and a port-number independent protocol identification engine (L7+) based on deep packet inspection (DPI) on an Intel Core 2 Quad processor. Experiments show that the system can deliver not only 6Gbps processing speed [1] for large packet sizes but also more challenging 2Gbps speed for smaller packets using only three CPU cores.

This paper presents the practice of parallelizing legacy sequential network applications on multi-core architectures by exploiting application domain knowledge and multi-core architecture features. Network applications have two inherently features that are suitable for parallelization: 1) they have naturally layered structures that can be organized into a functional pipeline; and 2) packets belonging to different flows can be processed in parallel. However, it is still very challenging to implement a software pipeline on multi-core architectures. First, network applications are inherently memory and I/O intensive and thus they may further exacerbate the disparity between computing power and memory latencies of multi-core architectures. Second, inter-core synchronization and communication must be handled by software, which in general is much slower than the mechanisms employed in NPUs. Particularly an X86 multi-core processor doesn't have efficient hardware FIFO to support fast core-to-core communication. Therefore, new design principles need to be carefully sought in parallelizing network applications on multi-core architectures.

This paper makes the following main contributions:

- A fast core-to-core FIFO is implemented to support fine-grained pipelining execution model on multi-core architectures, which is a foundation for parallelizing any fine-grained applications.
- A multi-core based network application parallelizing framework is built by employing network domain knowledge, concurrent lock-free data structures, and functional pipelining. This practice provides valuable experience on studying effective parallelizing principles to build a high-performance network system on the commodity multi-core processors.
- A prototype source-to-source compiler is implemented to facilitate porting sequential network applications written in C onto the parallel framework.

To the best of our knowledge, the system we built is the first one capable of delivering stable 2Gbps line-rate processing speed for a complete L2-L7 network application using only three CPU cores. Since only 75% of the CPU power of a quad-core processor is utilized, enough headroom is reserved for other advanced L7 applications. Our experimental results show that general-purpose multi-core CPUs are a viable alternative to NPUs or ASICs in building network processing systems, especially for complete L2-L7 network applications.

The remainder of this paper is organized as follows. Section 2 discusses design principles and system design space. Section 3 presents runtime system implementation applying the parallel design principles. Section 4 presents the evaluation results and performance analysis. Section 5 discusses related work. We conclude in Section 6.

2. System Design Space

This section reviews the basic and most important design principles for parallelizing network applications on multi-core platforms.

2.1 Design Principles

1) To exploit flow-level parallelism in a network application, flow-pinning [2] can be used to ensure that all the packets belonging to the same TCP flow are processed by a single CPU core. Since each TCP connection consists of two TCP uni-flows, we assign (bind) the two flows belonging to the same TCP connection to one CPU core. Therefore no synchronization is needed to access the connection table belonging to the same connection, in which one connection entry consists of two flow table entries. In addition, partitioning the connection table makes the size of each table smaller, which will significantly increase the L1/L2 cache locality, hence the cache hit rate. Therefore, *connection-affinity* is one of the most valuable design principles to develop networking software on multi-core processors.

2) A load-balancing approach is needed to dispatch packets among CPU cores. Due to the nature of fine-grained parallelism, we prefer the static load-balancing by trading the imbalance of workload with the much less runtime overhead. In general, a packet-classifying-hash function is used to dispatch packets among CPU cores. There are two types of hash functions: *asymmetric* and *symmetric*. Even though asymmetric hash function distributes packets more evenly among CPU cores, it

[1] Some network equipment vendors claim to handle 10Gbps line-rate for large packet sizes but this speed is achieved for stateless UDP packets only.

cannot guarantee connection-affinity. Therefore, we use a symmetric hash function to distribute packets among CPU cores, since on multi-core platforms cache locality is more important than load balance.

3) A balanced workload partitioning scheme is required to divide the workload evenly among pipeline stages. To make an optimal decision, the execution time of each pipeline stage needs to be accurately measured or predicted. We tried different partitioning schemes and derived a 2-stage pipelining in which one CPU core does packet capturing and L2 processing, and the other three CPU cores do the L3 above protocol processing work.

4) A high-efficient FIFO implementation is needed to provide fast core-to-core communication on multi-core platforms. For example, on a 2.3GHz Intel Core 2 Quad processor, a Pthread lock/unlock pair cost at least 250ns per operation, whereas a packet's processing time should take less than 672ns to meet 1Gbps line-rate requirement. We develop a cache-friendly concurrent lock-free FIFO queue by aggregating read/write operations based on cache line access to reduce the bus traffic used for maintaining the shared cache coherence and thus to make access time more stable.

5) Packet buffer management needs to be carefully studied. It is often unnoticeable that the reentrant library (Libc) calls contain hidden lock operations that may become performance bottlenecks in multi-core architectures. Malloc() that is often used for allocating a buffer whenever a new packet arrives contains locks. Therefore, eliminating the malloc usage is preferable in fine-grained multi-core network applications.

6) Sequential Libnids and Flex generated C code must be modified to run on the multi-core platform. As shared memory model is used in multi-core programming development, a naive way to make sequential code run in parallel is to change every global data access into atomic one; however the resulting parallel program may run slower than the original sequential program due to the overhead involved in atomicity enforcement. Since connection-affinity based parallelizing approach divides global connection state table into independent sub-tables, and each sub-table is private to its corresponding CPU core, it makes the sequential code one step closer to run in parallel by transforming a global table access with an extra pointer based dereferences.

7) In general, automatically parallelizing a sequential application is a hard problem with limited success [23][24]. We believe that domain knowledge could help lead to a viable solution in automatically parallelizing a sequential application. Network applications are amenable to pipelined processing due to their naturally layered stack structure. For example, HTTP (L7) is built on top of TCP (L4) which is built on top of IP (L3) which is built on top of Ethernet (L2). In principle, each layer can be regarded as one pipeline stage. Network applications also have inherent flow-level parallelism. Each flow is identified by the source and destination IP addresses, and source and destination port numbers. Such natural flow-level parallelism can be effectively used to parallelize network applications by running different flow processing in parallel. We rely on this domain knowledge to identify inherent parallelism, and then use the above parallelizing principles to utilize the parallelism.

In summary, domain knowledge guided parallelization, connection affinity, and lock-free data structures are the main parallel design principles used in building our parallel framework.

2.2 System Building Blocks

We make use of several open source software to implement our parallel runtime system in user space to make the system more portable. The system consists of the following modules:

1. Pf_ring [13] to improve the packet capture speed of the NIC and optimized Pcap[10] to get packets from trace files.
2. A cache-friendly lock-free FIFO to link pipeline stages together.
3. A TCP/IP stack implemented in Libnids[11], including IP defragmentation, TCP stream assembly, and TCP port scan detection for preventing denial-of-services (DOS) attacks.
4. A Flex [15] based protocol identification engine using L7-filter patterns [12].

We build a test bed based on Pf_ring for real traffic and optimized Pcap for stored packet traces, where a ring of buffers is added between Pf_ring/Pcap and the rest of L2-L7 network processing to allow them run in parallel. The system uses *mmap()* to reduce the memory copy between kernel space and user space.

Four cores are organized into a functional pipeline, with one core forming the first pipeline stage (Input), and the other three cores forming the second stage (Application) shown in Figure 1. Each pair of neighboring cores is linked with a cache-friendly lock-free FIFO. Original Libnids and Flex generated C code are parallelized using our compiler to make the sequential code run in a multi-core environment.

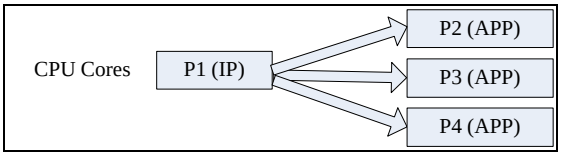

Figure 1. A tree-like data-parallel pipelines

3. Important Implementation Details

This section presents important implementation details on how to parallelize a complete L2-L7 network application system by applying connection-affinity and lock-free parallel design principles to achieve high speed.

We use ***m***-stage-***n***-way to describe a pipelining scheme in which *m* is the number of pipeline stages and *n* is the number of pipelines. The more pipeline stages the higher system throughput for a single packet. The more pipelines the more flow-level parallelism is exploited. The actual *m* and *n* need to be determined by profiling the application to make the pipeline execution in a balanced way.

3.1 Optimized Concurrent Lock-Free Queue

For the single-producer/single-consumer case, lock-free FIFO has been proposed in [3][6]. We improved the FIFO on speed and stability by:

1. Aggregating read/write operations within the same cache line and only modifying a cache line once per operation, i.e., the unit of read/write to the FIFO is a single cache line instead of one data item.
2. Introducing a timer to handle the extreme timeout case.

As shown in Figure 2, a local buffer *temp* is introduced in producer. If *temp* is not full, the producer puts *data* in *temp* (Line 5) without touching the global area *queue*, and returns immediately. To handle the extreme timeout case, a timer is introduced into the queue. Only when the local buffer is full or the timer times out (Line 7), the producer copies *temp* into *queue*. Similarly, the data transfer in the consumer side incurs only one transfer per cache line instead of one per data item.

```
1  FIFO_ELEM temp[ELEM_PER_CACHELINE];
2  int enqueue_aggregation(FIFO_ELEM * data) {
3     if ( NULL != queue[head] )
4        return FALSE;       /* the queue is full */
5     temp[current] = *data;     /* put data in local buffer. */
6     current ++;
7     if ((current == ELEM_PER_CACHELINE) ||
           (timeout == TRUE)) {   /* write back local buffer temp */
8        memcpy( queue[head], temp,
              sizeof(FIFO_ELEM) * ELEM_PER_CACHELINE );
9        head = NEXT( head, CACHELINE_SIZE);
10       current = 0;
11    }
12    return TRUE;
13 }
```

Figure 2. Aggregation of queue write operations

For a single packet, it seems that tiny delay is introduced due to aggregation operation. However, the producer saves lots of global operations, which results in less traffic in enforcing the cache coherence protocol, therefore performance of the whole system increases, especially for systems dealing with realistic network workload. The aggregation based FIFO works steadily in practice with each read/write operation taking around 45ns/op, 36% faster than the FastForward FIFO [3]. More performance details are shown in Section 4.5.

3.2 Pipeline Organization

Figure 1 depicts a 2-stage-3-way pipelined system to be mapped onto an Intel Core 2 Quad processor, in which a tree-like three data-parallel pipelines are used and each pipeline consists of two stages, *Input* (IP) and *Application* (APP). Since it lacks of hardware FIFO support on a multi-core commodity processor, the efficient lock-free FIFO described in section 3.1 is the key to enabling such a pipelined execution model to be implemented on a multi-core platform.

The IP core uses a symmetric packet-classifying-hash algorithm, discussed in section 2.1, to evenly distribute packets among the three APP cores. Specifically, the *checksum* on source/destination IP addresses and source/destination port numbers is used as the hash function. Since all the packets belonging to the same TCP connection are dispatched to the same APP core, each connection sub-table is accessed only by its APP core, therefore the access can be done in totally lock-free manner as if the connection sub-table was private.

3.3 Pipeline Mapping

To map the L2-L7 network stack on the pipelined model, a protocol stack processing is divided into seven functional pipeline stages, shown in Figure 3(a). An input packet is read from Pf_ring or Pcap, then memory is allocated by malloc() and the packet is copied. Next, the system checks the packet's integrity and reassembles IP fragments if necessary. Then the TCP process starts and TCP state tracking and detection of DOS attacks are done. Finally, L7 deep packet inspection (DPI) is performed by inspecting the packet's payload using pattern matching.

To select a proper cutting point that evenly divides the workload between IP and APP stages, we do the mapping based on accurate profiling on execution time of each pipeline stage. In general, to reach 1Gbps line-rate for the minimal 64-byte Ethernet packet input, a system must handle 1,488,095 frames per second (including the frame gap). This means that a new packet arrives every 672ns or 1545 cycles for a 2.3GHz CPU. So, on a 2-stage-1-way pipelined implementation, both the IP and APP cores should finish their processing in (1545-100) cycles assuming that the FIFO takes 100 cycles on average. On a 2-stage-2-way pipelined module, APP core's workload can be almost double that of the IP core, i.e., the APP's workload can be as high as (1545*2-100) cycles. In general, if FIFO communication takes C_{fifo} cycles, the IP core takes C_{ip} cycles, and the APP core takes C_{app} cycles in a 2-stage-N-way pipelined model, on a F-GHz CPU 1Gbps line-rate can be reached if

$$\max \{C_{ip}, C_{app}/N\} + C_{fifo} <= 672*F$$

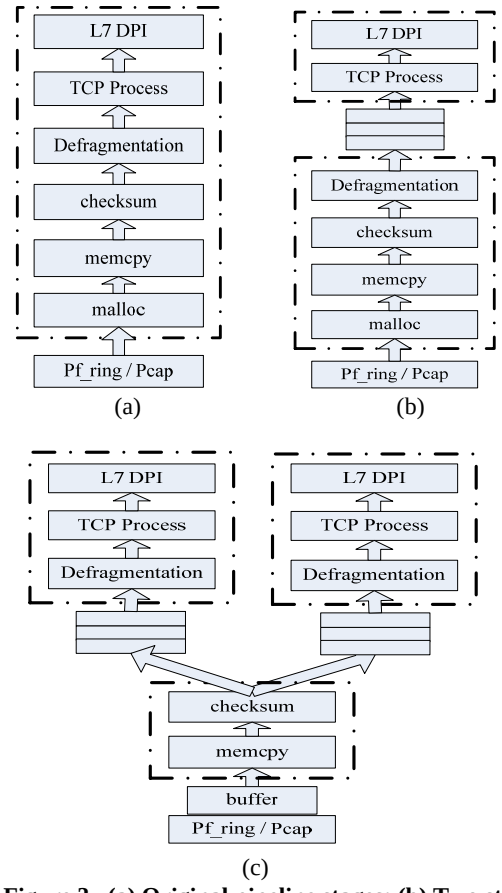

Figure 3. (a) Original pipeline stages; (b) Two staged pipeline; (c) 2-stage-2-way data-parallel pipeline

Figure 3(b) shows a partitioning scheme by dividing the workload just before TCP stage. This scheme favors network applications with heavy L4-L7 workload. However, our

experimental results (shown in Section 4) reveal that the pipeline stages with this cutting point are unbalanced, and splitting between Defragmentation and Checksum is the best for 2-stage-1-way pipeline model. Figure 3(c) depicts an improved cutting scheme for 2-stage-2-way pipeline model, in which the APP phase is duplicated. This scheme permits each APP phase to consume twice the inter-frame arrival rate by doubling the system's throughput. With the 2-way data-parallel model, L3-L7 layers can do more work than the original pipeline model allows.

3.4 Eliminate Lock/Unlock Operations

Since malloc() contains locks and it should be used as minimal as possible for fine-grained network applications which only have a budget of a few thousands of CPU cycles. We replace a malloc() by pre-allocating a bulk of memory space to the core responsible for the input packets, and then the subsequent buffer allocation is done locally on the core. Section 4.4 reports how eliminating malloc boosts system performance dramatically.

Besides, there is another hidden lock between Pcap and the upper layer network stack. In general, Pcap gets a packet once a time and then forwards it to the upper layer stack. Pcap does not handle the next packet until the current packet is being completely processed. We implement a ring of buffers shown in Figure 3(c), which is inserted between Pcap and the L2 layer to (1) decouple the Pcap interface and upper-layer functions; (2) eliminate the malloc operation.

3.5 Fast Memcpy

After eliminating unnecessary malloc, the performance of the pipeline execution shown in Figure 3(c) depends on a fast implementation of library function *memcpy*. To optimize it, the following methods are applied:

1. The SSE load/store instructions are used to access memory in 128 bits.
2. The loop unrolling technique is used to make full use of the SSE registers to
    a. Prefetch more data;
    b. Overlap latencies of the memory loads and stores by manually applying instruction scheduling technique.

3.6 Source-to-Source Parallelizing Compiler

For a parallelizing compiler, it must (1) detect parallelism, (2) utilize parallelism. Sometimes detecting parallelism is much harder than utilizing parallelism, especially for sequential legacy code. For example, without accurate points-to analysis [22], it is hard to identify all global variables accessed in a function. Without a good shape analysis [22], it is hard to know if two link lists collide.

The connection-affinity parallelizing approach allows to sequentially process packets on each CPU core while running multiple threads of such sequential processing on different cores in parallel. The idea is to keep the control-flow of original sequential code unchanged but to make global data accesses *atomic*. For example, accessing the TCP connection table needs to be modified to access each entry atomically. One way to guarantee the atomic property is to partition the connection table into independent sub-tables so that each one can be accessed in parallel on each CPU core. Since packets belonging to different TCP connections can be processed independently, assigning packets based on TCP connection-affinity guarantees the atomic access of each sub-table assigned to different CPU cores. This parallelization approach simplifies the task of parallelizing sequential code due to the sequential semantic maintained in each CPU core. Therefore, we rely on domain knowledge and the programmers to identify parallelism. The role of the compiler is to perform source-to-source translation under the domain knowledge guidance to make global data accesses atomic.

For example, the function listed in Figure 4(a) cannot run in parallel due to a global variable *timenow*. With traditional lock-based multithreaded programming, any operation on shared data that is susceptible to race conditions must be made atomic by locking and unlocking with a mutex. Through the connection-affinity analysis, each core can have a localized version of *timenow*, and the compiler can use the connection-affinity principle to infer that each local access is atomic without applying any lock. Figure 4(b) shows the parallelized code that accesses a local copy directly without any lock operation.

```
1  static int timenow = 0;
       /* timenow cannot be used in parallelized code directly. */
2  static int jiffies() {
3      if( timenow )
4          return timenow;
5      timenow = ...;
6      return timenow;
7  }
```

(a)

```
1  static int
2  jiffies(IP_THREAD_LOCAL_P ip_thread_local_p) {
       /* timenow is a private data of each thread. */
3      if( ip_thread_local_p->timenow )
4          return ip_thread_local_p->timenow;
5      ip_thread_local_p->timenow = ...;
6      return ip_thread_local_p->timenow;
7  }
```

(b)

Figure 4. (a) sequential legacy code; (b) parallelized code

The parallelizing transformation is as follows. For each functional pipeline stage corresponding to each layer of the protocol stack, a local storage (C structure) is pre-allocated for each CPU core, in which each global variable used in that layer takes one field position in the local structure. A function that accesses those global variables then has one extra parameter, a pointer to the local structure (ip_thread_local_p), and a read/write to a global variable is transformed into the pointer deference to the corresponding field of the local structure. In the example shown in Figure 4(b), pointer *ip_thread_local_p* is for the IP functional pipeline stage. By referencing the local copy of variable *timenow*, function *jiffies* can run in parallel.

Similarly, for any pointer dereferenced global data, its enclosing function will add an extra parameter, a pointer to the

pre-allocated local storage. A pointer dereference will be transformed into an indirect pointer dereference. For example, in the TCP stage the following line

tcp_oldest->nids_state = TIMED_OUT;

is transformed into

tcp_thread_local_p->tcp_oldest->nids_state= TIMED_OUT;

in which *tcp_thread_local_p* is a pointer to the local storage pre-allocated for the TCP layer.

Adding a level of pointer deference for each function may introduce the runtime overhead. However, its cost is neglect compared to a lock/unlock pair. In addition, the compiler performs function in-lining as much as possible to reduce the overload caused by passing an extra parameter.

Finally, a runtime system including thread initialization, buffer management, packet distribution, lock-free FIFO and linked lists, and time-out handling is implemented to facilitate parallelizing effort. The code for initialization, buffer assignment, and selection of cutting point using FIFO for m-stage-n-way pipeline is generated based on pragma line indication.

3.7 L7 Deep Packet Inspection

Based on Flex [15], we implement a deep packet inspection engine, which uses regular expression based pattern match to inspect packet payload. Firstly, L7-filter patterns [12] are translated into Flex ones by a pre-processor. For example, a pattern starting with a wild character * is replaced by the corresponding pattern starting with ^, indicating the beginning of TCP payload. Such rewriting can dramatically cut down the number of DFA states. Then, the Flex generated C code is fed into the parallelizing compiler to generate multi-core code as described in the previous subsection.

The protocol automatic identification engine identifies protocols according to protocol patterns instead of TCP port numbers. This engine is much more processor and memory intensive than that only checks for port numbers, however it is much more powerful because it can match any protocols that use unpredictable ports (e.g. P2P file sharing), non-standard ports (e.g. HTTP on port 1000) and the same ports (e.g. P2P file sharing using port 80).

4. Experimental Results and Performance Analysis

In this section we demonstrate that our system is capable of processing packets at the much higher line-rate speed. Specifically, our framework can finish the workload of prevention of DOS attacks (L2-L4) and deep packet inspection (L7+) up to 6Gbps speed for large packets.

The X86 64-bit time stamp counter (TSC) is used to measure execution time of each pipeline stage. The TSC measures elapsed cycles since the system is started and is accurate within a few cycles. Firstly, we make only Pcap stage run in the system. Then, we add other following pipeline stages one by one to derive the execution time of each pipeline stage.

4.1 Evaluation Platform

We run experiments on one Dell server equipped with Intel Core 2 Quad processors. DELL PowerEdge 2900 has a dual Xeon E5410 processors running at 2.3GHz, and it has two 6MB L2 caches with 64B cache-line size and 1333MHz FSB. The two Gigabit Ethernet cards (Intel 9400PT and Broadcom BCM5721) are connected by a cross cable (back-to-back). The system is configured to run the 64-bit Linux 2.6.x kernel and the code is compiled by the GCC 4.1.2.

The characteristics of the packet trace files used in the experiments are described in Table 1. File-7 and File-8 come from the 1998-1999 DARPA intrusion detection evaluation at MIT Lincoln Lab [30]. File-9 and File-10 come from the Defcon 9 Capture the Flag contest [31]. The other trace files are collected from the gateway of a university campus. In Table 1, column *#Conn.* denotes the number of total connections, column *#Packet Len* lists the average packet size, and column *#Conn. Rate* gives the number of connections per 1,000 packets. The higher the connection rate is, the more difficult it is for the L4-L7 applications to process.

Table 1. Characteristics of trace files

FILE	#Packets	#Conn.	#Conn. Rate	#Packet Len. (Byte)
File-1	646,703	9,191	13.9	186
File-2	297,024	4,384	14.8	201
File-3	594,064	8,768	14.5	205
File-4	2,376,256	35,072	14.7	204
File-5	9,496,620	140,194	14.7	202
File-6	17,092,300	253,371	14.8	208
File-7	3,393,924	111,777	32.9	307
File-8	3,201,341	103,578	32.3	243
File-9	3,960,205	12,736	3.2	194
File-10	1,050,364	1,043	1.0	851

4.2 Packet Input

In our experiments, we use two methods to generate input packets:

1. One NIC (Broadcom BCM5721) injects packets into the cable, and then another (Intel 9400PT) captures the packets and delivers them to the applications by assistance of Pf_ring. The packet traces are injected into the Gigabit link using Tcpreplay [14]. To achieve high speed traffic, we had to "speed up" the traffic by setting "--topspeed" to send packets as soon as possible. We assume that this would not affect the correctness of our experiments. Since Tcpreplay can play back the trace files at 410Kpps rate in our testbed, 1Gbps is reached if the average packet length is larger than 304 bytes.

2. Since the recipient is not the performance bottleneck [13], for higher speed testing the input packets must come from tcpdump files because it is very challenging to use software approach to generate traffic at a speed greater than 1Gbps without heavy investment on hardware testing equipment. On the other hand, our work focuses on the protocol processing. Therefore, we rely on input files to feed packets to measure higher processing speeds.

4.3 System Performance

In this section, we evaluate the system performance for the entire L2-L7 application. Lincoln Lab traces (File-7 and File-8) are simulations of large military networks generated during an online evaluation of IDSes. The Defcon traces (File-9 and File-10) are logs from a contest in which hackers attempt to attack and defend vulnerable systems. These traces contain a huge amount of attacks and anomalous traffic, representing a sort of pathological cases for network processing system. Table 2 lists main identified threats, protocols, and applications after the L7 processing. Row *Alerts* shows the number of TCP port scan attempts that are detected by our system. Row *HTTP*, *POP3*, *SMTP*, *IMAP*, *TELNET* and *BITTO*. show the number of HTTP, POP3, SMTP, IMAP, TELNET and BITTORRENT flows respectively, and row *Others* counts the number of other flows recognized by our system, including SSH, EDONKEY, IRC, X11, AIM, and Unknown. Although the number of protocols that can be recognized by our system is more than 60, only a few widely used protocols are contained in the trace files.

Table 2. Results of L7 Content Processing

	File-4	File-7	File-8	File-9	File-10
Alerts	21,270	1,407	12,064	94,579	483
HTTP	1,504	110,103	67,621	3,934	765
POP3	0	39	7,407	1	0
SMTP	4	1,265	3,254	11	0
IMAP	0	0	2,533	0	0
TELNET	0	390	3,347	55	0
BITTO.	18,000	0	0	0	0
Others	32	201	190	692	139
PI-per-C	6	6	5	3	3

For each new connection, L7 DPI is called to inspect the payload of first few packets to identify the protocol type. Row *PI-per-C* shows the average number of packets that are inspected to recognize the protocol type for each new connection. Since File-1 to File-6 all come from the campus, only File-4 is listed in this table. Based on L7 DPI, our system checks the packet payload instead of simply port numbers. That is why HTTP like P2P BitTorrent protocol can be detected correctly.

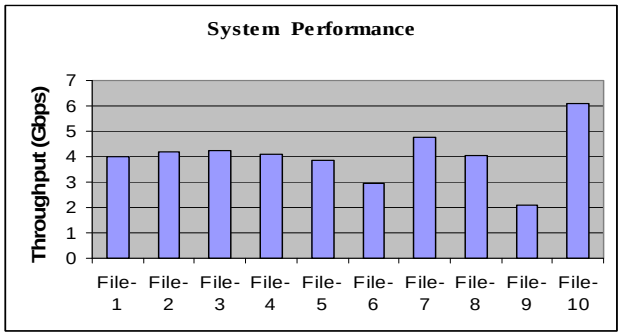

Figure 5. System performance achieved with trace files

Careful reader may find that the sum of per column in Table 2 doesn't match the corresponding #Conn. entry listed in Table 1. This is because a connection may triger multiple alters while some anomalous traffics are not categorized by the L7 DPI engine. However, in Table 2 only the sum of column File-8 is slightly smaller than the corresponding #Conn. entry in Table 1, which indicates that the protocol identifier can detect most of the protocols correctly.

Figure 5 shows the system performance achieved with the ten trace files listed in Table 1 after all the optimization techniques are applied. The system has the lowest throughput of 2Gbps with File-9, which contains many small alert packets, and the highest throughput of 6Gbps with File-10, which has the largest packet size. To sum up, even deployed in the vicious environment (File-9) that has many small attack packets, our system can still achieve 2Gbps network processing speed.

As shown in Figure 5, the system performance measured in data rate (Gbps) varies greatly with the average packet size, and the throughput discrepancy between large-packet traffic and small-packet traffic could be very large. Since many protocol processing operations require a fixed amount of CPU time per packet, the number of packets processed is more important than the data rate. In addition, our system can easily achieve higher 1Gbps speed. Therefore, in the following discussion we will use the more challenging metric, packet processing rate (Million Packet Per Second, MPPS), to measure the system performance in the worst case (i.e., even with small packets). The maximum packet rate required to support 1Gbps data rate with minimum 64-byte frames is about 1.49Mpps, and we will use this lower bound no matter what the packet size is. We encourage other research results to be measured in MPPS to have a fair comparison in the future.

4.4 Pipeline Partitioning and Mapping

This section presents the test results on trace File-1 to demonstrate the effects of connection-affinity and lock-free design principles described in Section 3.

Table 3. Execution time of each pipeline stage for File-1

Cycles	Mode 1	Mode 2	Mode 3	Mode 4	Mode 5
Pcap	450	450	450	300	300
Malloc	450	700	700	---	---
Memcpy	440	440	440	440	440
FIFO	---	100	100	100	100
Defrag.	250	250	250	250	250
TCP	500	500	500	500	500
L7 DPI	400	400	400	400	400
Per-frame	2490	1940	1690	1250	840
Speed (mpps)	0.97	1.18	1.36	1.83	2.73
Imp. (%)	-----	21.6	15.3	34.6	49.2

Table 3 shows the execution time of each pipeline stage with different pipelining schemes. *Mode 1* (Figure 3(a)) corresponds to the sequential pipeline in which all stages run on the same CPU core, therefore the FIFO overhead is zero. The average execution time per packet is 2490 cycles, which is greater than 1,545 cycles required for reaching 1Gbps line-rate speed. *Mode 2* (Figure 3(b)) corresponds to the 2-stage-1-way pipeline in which the cutting point is between L3 and L4, and *Mode 3* moves the cutting point of *Mode 2* to between L2 and L3, and *Mode 4* further uses pre-allocated Pcap buffers to link the two pipeline stages of *Mode 3*. *Mode 5* (Figure 3(c)) corresponds to the 2-stage-2-way pipelining

scheme. The 2-stage-3-way pipelining scheme (Figure 1) was also evaluated in our experiments. However, for 2-stage-n-way pipelining schemes with n>2, IP core becomes the bottleneck and adding more APP cores has little effect on lifting performance. So the results of 2-stage-3-way pipelining scheme are not listed except in section 4.6. To achieve higher speed, the workload of IP core can be transferred to a hardware based solution [35].

Row *Pcap, Malloc, Memcpy, FIFO, Defrag(Defragmentation), TCP* and *L7 DPI* display the average execution time of each pipeline stage. The reason TCP stage takes only 500 cycles is that after connection-affinity based parallelization is applied, the lock-free TCP pipeline stage can execute much faster. *L7 DPI* is calculated by averaging the total number of cycles spent on L7 DPI stage over the total number of packets processed by the system. Therefore, although a single L7 DPI operation takes around 4,000 cycles, the average L7 DPI workload is not so high, since only the first few packets of each connection are inspected (Table 2 PI-per-C rows).

Row *Per-frame* shows the average frame (packet) processing time with different pipelining schemes by calculating the larger number between (C_{ip}+C_{fifo}) and (C_{app}/N+C_{fifo}) (Section 3.2). For example, for the pipelining scheme in *Mode 3*, the IP core takes 1,690 cycles in all (450 for Pcap, 700 for Malloc, 440 for Memcpy, and 100 for FIFO), and the APP core takes 1250 cycles in all (250 for Defragmentation, 500 for TCP, 400 for L7 DPI and 100 for FIFO). Therefore, on average each packet takes 1,690 cycles to be processed, and the equivalent packet processing rate is 1.36Mpps.

Comparing the *speed* of *Mode 2* with that of *Mode 1*, we can see that adding one more CPU core does not necessarily double the packet rate if an unbalanced pipelining scheme (*Mode 2*) is used. However, if the workload is more evenly partitioned and the malloc function is eliminated (*Mode 4*), the packet rate can reach 1.83Mpps (that is above the 1Gbps line-rate requirement) with only two CPU cores. By adding one more CPU core to form a 2-stage-2-way pipeline in *Mode 5* to apportion the heavy workload of APP stage of *Mode 4*, the packet rate increases to 2.73Mpps that is 49.2% improvement over *Mode 4*, and almost triple of the speed of the single-core solution (*Mode 1*). Even if taking into account the worst behavior of hash function that the *Mode 5* almost degenerates to *Mode 4* (2-stage-1-way pipeline), the system can still achieve line-rate (>1.49Mpps).

The experimental results in Table 3 also reveal the performance impact of malloc on multi-core based systems. The overhead of malloc increases 35.7% from 450 cycles in the single-core environment (*Mode 1*) to 700 cycles in the two-core environment (*Mode 2*), and *Mode 4* achieves 34.6% performance improvement over *Mode 3* by only eliminating the malloc function call.

4.5 Performance of Optimized FIFO

This section evaluates the performance of our optimized FIFO with FastForward [3] as the baseline of the evaluation. Experimentally, we find out that if the consumer has dummy workload or no workload, both FIFOs can achieve high and stable performance. For example, if the consumer has no workload, both FIFOs can finish a put/get operation within 35ns. If the consumer takes spin loop as workload, FastForword takes 70ns and our optimized FIFO takes 45ns to finish a put/get operation.

However, if the producer and consumer have realistic network processing workload, the performance of both FIFOs decreases due to unbalanced stages. Table 4 lists the system performance achieved with different FIFOs as stage-to-stage communication mechanism. Row *FastForward* and *Optimized* display the system performance with FastForward and optimized FIFO, respectively. Row *Inc* shows the performance improvement of Optimized FIFO over FastForward. Since File-1 to File-6 all come from the campus, we list File-4 only in this table. Table 4 indicates that the Optimized FIFO is more robust and the performance improvement overall can be as high as 9%, and 6.6% on average. The optimized FIFO is more robust and practical, and as a result, it can be used as a foundation for parallelizing in multi-core commodity processors that lack hardware queue support.

Table 4. System performance with two different FIFOs

(Gbps)	File-4	File-7	File-8	File-9	File-10
FastForward	3.60	4.57	3.92	2.06	5.68
Optimized	3.93	4.74	4.13	2.21	6.12
Inc.	9.2%	3.7%	5.4%	7.3%	7.4%

4.6 TCP Connection Scalability

In our system, each TCP connection is tracked with the state of connection stored in a C structure. For each TCP packet, function *find_tcp_stream*() is called to determine whether the connection has been established. If the connection already exists, the packet is processed and the state stored in the C structure is modified; otherwise a new connection is created. As we introduced pre-allocated local storage to this data structure, we'd like to explore how well the system performance is when the number of TCP connections grows big. I.e., how many of TCP connections can be opened in the system while the 1Gbps processing speed is still maintained?

Table 5. System performance with increased number of TCP connections

(Mpps)	File-2	File-3	File-4	File-5	File-6
1 Core	1.35	1.34	1.25	1.04	0.97
Dec.	--	-0.7%	-7.5%	-16.8%	-6.7%
2 Cores	1.82	1.81	1.72	1.32	1.22
Dec.	--	-0.5%	-4.9%	-23.3%	-7.5%
3 Cores	2.68	2.68	2.59	2.39	1.72
Dec.	--	0.0%	-3.4%	-7.7%	-28.0%

Table 5 shows the system performance with five trace files that have large number of TCP connections. The row *1-Core, 2-Cores*, and *3-Cores* show the packet processing speed with 1-way, 2-way, and 3-way pipeline models respectively. Rows *Dec* shows the performance decrease when the number of connections increases column by column.

For an Intel Quad-core processor with 2x6MB L2 cache, the size of connection C structure is 280 bytes on a 64-bit machine; therefore maximum 2*6*1024*1024/280 = 44,940 entries can be stored in the L2 cache, assuming that no cache collision occurs. That is why the system performance incurs only single-digit loss when the connection count increases but not exceeds the capacity of L2 cache (File-2 to File-4). Significant performance decrease (-

23.3%) is observed on File-5 whose connection count (140K) exceeds the capacity of L2 cache, but situation gets better when a 3-way pipeline is used since both L2 caches are fully used. File-6 has much larger connection count, and even the 3-way pipeline is used the performance still drops by 28.0%. Therefore, the number of TCP connections opened simultaneously has dramatic impacts on performance.

To analyze the cache behavior under different number of TCP connections, we use Valgrind to analyze the L2 cache miss rate. As Valgrind does not support machine equipped with 6MB L2 cache, we run Valgrind with 2x4MB L2 cache that can maximally maintain 2*4*1024*1024/280 = 30,000 TCP connections in L2 cache.

Table 6. L2 cache miss rates with increased number of TCP connections

File	#Reference	#Miss	#Miss Rate	Inc.
File-2	14,243,536	63,369	0.44%	--
File-3	28,515,072	177,805	0.62%	41%
File-4	75,837,429	629,647	0.83%	34%
File-5	328,858,387	9,747,387	2.96%	256%
File-6	575,212,699	29,246,911	5.08%	72%

Table 6 shows the L2 cache miss rate reported by Valgrind. Column *#Reference* shows the number of references to the connection table, and column *#Miss* shows the total number of L2 cache misses. Column *#Miss Rate* gives the L2 cache miss rate, and the last column calculates the increase of miss rate over the previous row. The greatest miss-rate increase occurs when the connection count increases from 35K to 140K, which explains the great performance drop between column *File-4* and column *File-5* in Table 5.

In summary, even with as large as 253K TCP connections our system can still maintain 1.72Mpps, which demonstrates the potential of the commodity multi-core processors. Furthermore, L2 cache is the key to boost system performance. Table 6 shows that if the miss rate increases to almost 3% the system performance can decrease dramatically.

5. Related Work

Prior work on using commodity multi-core processors on networking have been reported in [2][3][25][29]. Unlike previous work, our system achieves 2Gbps speed even under small packets input and handling complicated L2-L7 network applications.

Using a Cell SPE to speed up TCP processing is discussed in [33]. However, the results were measured against 128 TCP flows that are too small to be used in practice.

FastForward[3][7] is a single producer/single consumer concurrent lock-free queue for supporting pipeline parallelism. It uses the similar idea proposed in [27] by retrying a data access if a conflict in terms of the same cache line access occurs. However, its performance is only verified with dummy workload. We proposed a new lock-free FIFO in which aggregated read/write is applied on the cache-line boundary. The new FIFO is more robust and efficient in practice.

There is some work focusing on improving the performance of malloc() to benefit the performance of any sophisticated multi-threaded application [4][5][18]. We focus on eliminating malloc operations as much as possible with pre-allocated memory area.

Aspen [25] is a language designed to help capture flow-locality in network applications. However, it only supports L5 above layers (socket level). Whether such a language is useful for L2-L4 network applications needs to be proven.

Snort is an open-source network intrusion prevention and detection system, which attracts much research interest in porting it onto the multi-core architecture [2][26][29]. However, the previous porting efforts are far short of reaching the 1Gbps line-rate speed in the worst case. We believe that the combination of connection-affinity, lock-free, and multiple pipelines as the first-priority parallel design principles dramatically increases the performance of our system.

Our parallelizing approach is unlike other general automatic compiling approaches. We rely on domain knowledge to help solve the hard problems such as parallelism detection and load balance. The compiler performs source-to-source translation on global data accesses to support the connection-affinity based pipeline execution model.

Intel's ETA[8] is designed to accelerate the processing of packets by dedicating a processor to act as a TCP/IP onloading engine [9]. On the other hand, Receive-Side Scaling (RSS) [28] is a driver approach to integrate the hashing function in network adapters. However, RSS is unsuitable for an inline-device.

6. Conclusions and Future Work

This paper presented a high performance connection-affinity based lock-free multi-core network processing system, a software-only framework that can achieve multiple Gbps network processing speed while finishing complicated tasks. Our experiments suggest that (1) connection-affinity is an effective means in parallelizing sequential network applications onto the multi-core architectures; (2) it is the same important to eliminate lock operations in the user space; specifically, we try to build a non-blocking system that has almost no lock at all; (3) a delicate partitioning scheme is required to map the pipelined structure onto a multi-core architecture.

Our system implementation indicates that lock-free data structures are very useful in parallelizing network applications. We will continue to study lock-free data structures and explore their effective usage in network applications.

7. Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments. This work was supported by the National Natural Science Foundation of China under Grant No.60673173, and the Fund for Foreign Scholars in University Research and Teaching Programs under Grant No.B07033.

8. References

[1] CNET News.com. Intel pledges 80 cores in five years. http://news.com.com/2100-1006_3-6119618.html, Sep. 2006.

[2] Intel Corporation, Supra-linear Packet Processing Performance with Intel Multi-core. http://www.intel.com/technology/advanced_comm/311566.htm, 2006.

[3] J. Giacomoni, T. Moseley, and M. Vachharajani. Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In *PPoPP'08*, New York, NY, USA, February 2008. ACM Press, 2008.

[4] Lever C, Borheham D. malloc() Performance in a Multithreaded Linux Environment. In *USENIX 2000 Annual Technical Conference*: FREENIX Track, May 2000.

[5] Dave Dice and Alex Garthwaite. Mostly Lock-Free Malloc. *Proceedings of the 3rd international sysposium on Memory management*, Berlin, Germany. 2002.

[6] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190-222, 1983.

[7] John Giacomoni and John K. Bennett et. al. Frame Shared Memory: Line-Rate Networking on Commodity Hardware. In *ANCS'07*, Orlando, Florida, USA, 2005. ACM Press.

[8] G. Regnier, D. Minturn, G. et al., ETA: Experience with an Intel Xeon processor as a packet processing engine. *IEEE Micro*, Jan./Feb. 2004, pp. 24-31.

[9] G. Regnier, S. Makineni, et. al. TCP onloading for data center servers. *Special issue of IEEE Computer on Internet data centers*, Nov 2004.

[10] Libpcap, http://www.tcpdump.org .

[11] Libnids, http://libnids.sourceforge.net .

[12] L7-filter, http://l7-filter.sourceforge.net .

[13] Pf_ring, http://www.ntop.org/PF_RING.html .

[14] Tcpreplay, http://tcpreplay.synfin.net/trac/.

[15] Flex, http://flex.sourceforge.net/.

[16] Haipeng Cheng, Zheng Chen, Bei Hua, Xinan Tang. Scalable Packet Classification Using Interpreting—A Cross-platform Multi-core Solution. In *PPoPP'08*, Salt Lake City, USA, Feb.20-23, 2008.

[17] Xianghui Hu, Xinan Tang and Bei Hua. High-Performance IPv6 Forwarding Algorithm for Multi-core and Multithreaded Network Processor. In *PPoPP'06*, New York , USA , Mar. 29-31, 2006.

[18] Maged M. Michael, Scalable Lock-Free Dynamic Memory Allocation. *ACM SIGPLAN Notices*, pp.35-46, 2004.

[19] Ron Wilson. Cisco taps processor array architecture for NPU. http://www.eetimes.com/showArticle.jhtml?articleID=26806315, Aug. 9th. 2004.

[20] Huawei Technologies Co.. Huawei Launches NetEngine80 Core Router At Networld Interop 2001 Exhibition in US. http://www.huawei.com/news/view.do?id=88&cid=-1001 , 2001.

[21] Intel Corporation. IXP2XXX Network Processors. http://www.intel.com/design/network/products/npfamily/ixp2xxx.htm.

[22] Xinan Tang, Rakesh Ghiya, Laurie J. Hendren and Guang R. Gao, Heap Analysis and Optimizations for Threaded Programs. In *Proc. of PACT'97*, Sanfracisco, CA, Nov., 1997.

[23] Xinan Tang and Guang R. Gao, Automatically Partitioning Threads for Multithreaded Architectures. In *Journal of Parallel Distributed Computing*, 58(2) pp.159-189, 1999.

[24] Xinan Tang and Guang R. Gao. How hard is thread partitioning and how bad is a list scheduling based partitioning algorithm? *Proc. of the tenth annual ACM symposium on Parallel Algorithms and Architectures*, pp. 159-189, 1998.

[25] Gautam Upadhyaya, Vijay Pai, and Samuel Midkiff. Expressing and Exploiting Concurrency in Networked Applications with Aspen. In *PPoPP'07*, San Jose, USA, Mar. 14-17, 2007.

[26] Aaron Kunze, Stephen Goglin, and Erik Johnson. Symerton - Using Virtualization to Accelerate Packet Processing. In *ANCS'06*, San Jose, CA, USA, Dec. 4-5, 2006.

[27] Keir Fraser and Tim Harris. Concurrent programming without locks. In *ACM Transactions on Computer Systems*, Vol. 25 (2), May 2007.

[28] Microsoft, Scalable Networking with RSS. http://www.microsoft.com/whdc/device/network/ndis_rss.mspx, 2005.

[29] Derek L. Schuff, Yung Ryn Choe and Vijay S. Pai. Conservative vs. Optimistic Parallelization of Stateful Network Intrusion Detection. In *ISPASS'08*, Austin, Texas, April 20-22, 2008.

[30] J. W. Haines, R. P. Lippmann, D. J. Fried, E. Tran, S. Boswell, and M. A. Zissman. 1999 DARPA Intrusion Detection System Evaluation: Design and Procedures. Technical Report 1062, MIT Lincoln Laboratory, 2001.

[31] Shmoo Group. Defcon 9 Capture the Flag Data, Sept. 2001.

[32] M. Kulkarni, P. Carribault, K. Pingali, et. al. Scheduling strategies for optimistic parallel execution of irregular programs. In *SPAA'08*, Munich, Germany, June 14-16, 2008.

[33] Yuji Kawamurra, Takeshi Yamazaki, Tatsuya Ishiwata et. al. Network Processing on an SPE Core in Cell Broadband Engine. In *Proc.of 16th IEEE Symposium on High Performance Interconnects*. 2008.

[34] Duo Liu, Zheng Chen, Bei Hua, Nenghai Yu, Xinan Tang. High-performance Packet Classification Algorithm for Multithreaded IXP Network Processor. In *ACM Transactions on Embedded Computing Systems*, Vol.7, No.2, Article 16, 2008.2.

[35] Livio Ricciulli, et al. Programmable Multifunctional Line Rate Analyzer for 10 Gbps Networks. http://www.force10networks.com.