THE TEJA SOFTWARE PLATFORM FOR NETWORK PROCESSORS

Illustrated Using a Case Study for Traffic Flow Accounting

Akash Deshpande, Kevin Crozier, and Mandeep Baines
Teja Technologies, Inc.
2 West Santa Clara Street 6th Floor
San Jose, CA 95113 USA

18 September 2001

ABSTRACT

Network bandwidth and packet processing needs are growing faster than processor speed. Even with the doubling of processing power every eighteen months, next generation network systems will require between one and two orders of magnitude more processors than today's systems. They must rely on scalable, parallel and distributed architectures both in hardware and software. Because the traditional ASIC approach limits flexibility for adaptation to rapidly changing system architectures and delays time to market for the introduction of new functions, silicon vendors have introduced high-performance, programmable network processors. System software for network processors is a critical success factor in next generation networks. This paper describes the Teja software platform for network processors and illustrates its use and performance with a traffic flow accounting case study.

KEYWORDS

Multiprocessors, API, Data Forwarding, Quality of Service, Network Operating System

1 INTRODUCTION

Network bandwidth is projected to continue its historic growth at twice the rate of processor speed growth. Additionally, packet processing needs are growing from about 200 instructions per packet for basic forwarding to over 2000 instructions per packet for advanced functions such as quality of service and security. Within three years network systems will require between one and two orders of magnitude more processors than today's systems, forcing them to rely on scalable, parallel and distributed architectures both in hardware and software. Because the traditional ASIC approach limits flexibility for adaptation to rapidly changing system architectures and delays time to market for the introduction of new functions, silicon vendors have introduced high-performance, programmable network processors.

Deb (Deb2000) explains the nature of network processors by considering the time required to process a packet. For example, at 1 Gbps, the network processor has 360 nanoseconds to process a minimum size packet. In this time, it must receive the packet, parse it, look up one or more policies for forwarding, modification and queuing, and transmit the packet. Using typical SRAM memories with 10 ns access speeds, there is time for only 36 memory accesses. For higher bandwidths such as 10 Gbps, there is time for just 3 memory accesses. The only effective method for handling high data rates is to split the function and execute the processing in a pipelined, parallel, multi-threaded architecture. Spalink, Karlin and Peterson (Spalink2000) also note the use of parallel computing in network processors to hide memory latency.

Building on their work, Spalink *et al* (Spalink2001) explain the architecture and performance of a software-based router implemented using network processors. Their experience indicates that network processors are not easy to program. They report encountering many false starts in the

overall approach to managing the parallel resources. They were also hampered by having to explore the design space through low-level assembly programming. But they are not convinced that a C compiler will achieve the required performance. Further, they note that programming some of the functionality such as DMA state machines in procedural languages is error prone. Cravotta (Cravotta2001) also discusses the "software breakdown" in network processor-based system design, where the fundamental problem is shown to be the lack of an appropriate programming model. Network processor software must be written with several dimensions in mind – whether the code is "fast" or "slow," whether it is symmetric (i.e., multiple threads run the same code on different packets) or asymmetric (i.e., different threads perform different functions on the same packet), and whether the code is sequential or parallel. In all cases, the main problem with the programming model is shown to be the shoe-horning of sequential code into a multi-threaded environment without optimized scheduling.

Herity (Herity2001) suggests several lines along which solutions could be explored for network processor software. Isolation of fast and slow paths is suggested as the first step. This isolation is to be rendered efficient and portable by a carefully designed and standardized API that enables communication between the two. In his view, software must be developed in different programming styles, such as specialized pattern matching languages, assembly programs and object-oriented C++ programs, for different functional elements. And finally software written for the network processor must exploit the multiprocessor architecture. Spalink et al (Spalink2001) rely on a static mapping of resources in their implementation, but concede that the static approach requires the software to be redesigned for different applications, thereby restricting reusability, and acknowledge that the problems of both, optimization and reuse are complicated by the assembly code implementation which is hard to change. They conclude that the best (i.e., optimal, reusable and extensible) approach would be to construct the software from building block components supported by a domain-specific compiler.

Deb (Deb2000) predicts that eventually the industry will standardize on a "software set" that includes operating system software, tools, protocols and management applications, and that system developers will put systems together around such a plug-and-play architecture. For this vision to be realized, we believe that network processor software needs to be developed around a programming model that exploits the multithreaded nature of network processors and unifies the apparently irreconcilable needs such as forwarding *vs*. control processing, sequential *vs*. parallel code, special *vs*. general languages, and static *vs*. dynamic resource assignment.

It is our belief that these complex issues are resolved by a solution based on three major considerations:

- State machine-based behavior description
- Efficient, extensible and portable data structure description, and
- Multiprocessor scheduling of state machine contexts communicating structured data.

State machines are a well-understood, mathematically defined and extensively analyzed formalism. They are commonly used in the design of both ASICs (which the network processor software replaces) and network protocols (which the network processor software implements). State machines have been used in a wide range of distributed, reactive system description languages such as CSP and Esterel. Finite state machines are known to have the same expressive power as regular expressions (which are used for pattern matching in packet classification), and state machines augmented with suitable data variables are known to correspond in expressive power to the various classes of the Chomsky hierarchy. Finally, even traditional compilers for languages such as C store and analyze the program in a state machine format before applying optimizations and generating code. We have chosen a state machine description format that is close to the interme-

diate representation used by compilers, thereby enhancing the potential for generating efficient code for the distributed application.

Object-oriented data structure description, with specially designed data types for space optimization and memory layout for portability, form the second element of the programming model. The inheritance of member variables and functions enables the construction of an extensible framework of components that can be reused, extended and combined in different ways.

The components of the application are distributed to the various processing resources and access the different data structures placed in the various memory banks. The multiprocessor scheduling environment ensures the smooth execution of the different state machine components.

Based on these three principles, the Teja software platform simplifies and standardizes network processor-based system development. Its core underlying technology is a high-performance, parallel and distributed software execution platform for network processors and multiprocessor systems. The accompanying graphical development environment supports a sound system design methodology. The platform incorporates an extensible framework of network application building blocks for data forwarding, with integrated control and management interfaces.

The Teja platform meets the following objectives:

- Deliver network services at wire speed in a complex, high-performance, multiprocessor environment,
- Provide an open, reliable, extensible framework,
- Enable multiple independent software vendors to plug their network applications into a common, integrated system, and
- Enhance the maintainability, reusability, and portability of applications.

To achieve the first objective, the platform is optimized for the supported target network processor. To achieve the second objective, it provides a scalable, modular software architecture. To achieve the third objective, it provides standard application programming interfaces. To achieve the fourth objective, it provides robust graphical development and code generation tools with debugging, testing, and simulation capabilities for system programming.

The following guidelines were followed in designing and implementing the Teja platform:

- Do not sacrifice performance
  The platform's hand-coded NPOS, a runtime system of network middleware, is optimized for the target network processor in order to preserve packet throughput performance and enhance scalability of flow contexts in the control plane.

- Reduce complexity
  The platform's graphical development environment and integrated C/C++ and assembly code generation provide a complete, easy-to-use, system-level implementation tool chain with heterogeneous processor support.

- Provide extensibility, reuse and portability
  The platform's component class framework enables the network application building blocks to be combined and extended in different ways to develop different applications and different target hardware configurations.

- Use as few new concepts as possible
  The platform uses familiar and standard network system design techniques and implements them for robust, optimized execution.

2 TEJA SOFTWARE PLATFORM

The platform has three components: system platform, tools, and a framework of network application building blocks.

2.1 System Platform

The system platform consists of a parallel, distributed execution (PDX) platform for scheduling state machine-based application logic. PDX provides two types of schedulers: a concurrent execution unit (CXU) that schedules a variable number of state machine contexts within a single thread of execution, and a sequential execution unit (SXU) that schedules exactly one state machine context within a single thread. The system consists of multiple CXUs and SXUs operating asynchronously, and in parallel. The execution units (also known as servers) provide a scheduler for event-driven and time-driven state machine execution. Additionally, shared memory and message passing primitives are available for synchronization between servers.

In the data plane, the SXU server exploits hardware concurrency in the data path processor to schedule a single state machine component per hardware-supported thread. These data path processors can be programmed to execute the packet processing functions in a pipeline, with a different state machine component for each stage of the pipeline. They can also be used in a symmetric multi-processing (SMP) architecture for parallel execution of identical functions.

In the control plane, the CXU server schedules a dynamically changing network of multiple event-driven and time-driven state machine components. Each state machine component is typically used to handle a separate active protocol context. Additional components provide the framework for protocol dispatching and packet input-output functions such as OS pseudo-drivers.

The system platform provides operating system and hardware abstraction layers that implement the concepts of threads, shared memory, inter-thread communication, interrupt delivery, input-output devices, and timers.

The system platform APIs are expected to be compliant with industry standards such as Network Processing Forum's Software API (NPF SWAPI) specification when these standards are finalized.

2.2 Tools

The Teja platform tools provide a graphical development environment for:

- System architecture design,
- Application logic design,
- Mapping and code generation, and
- Simulation, testing, and debugging.

The system architecture design tool enables developers to describe their system both logically, using CXU and SXU threads that communicate using shared memory and a messaging network, and physically, using processors and memory banks of different types.

The application logic design tool enables developers to describe their application in a portable, component-based manner using object-oriented data structures, structured state machines, and structured event (synchronous) and alert (asynchronous) messages.

The mapping and code generation tool enables developers to map their application logic to the logical system architecture, and to map the logical system architecture to the physical platform. Based on this mapping, the tool generates optimized code for the entire application (C/C++ as well as assembly) targeting the application programming interfaces provided by the system platform.

The simulation, testing and debugging tool enables developers to verify the logic of their application and improve the performance and quality of their system by running the application interactively on both the development host and the target hardware. Test cases can be built and run as a part of the application. Execution can be controlled through logic level breakpoints and step-by-step debugging with state inspection.

2.3 Network Application Framework

The Teja software platform provides network application building blocks and a framework for integrating them across the data plane, control plane, and management plane.

The data plane packet processing and forwarding engine is designed as a network of packet queues served by service modules. The queues are implemented as hardware-accelerated auxiliary data structures in shared memory. The service modules are implemented as state machine packet processing components such as receive and transmit drivers, input scheduling, classification, metering, marking, forwarding, and output scheduling.

The control plane protocol processing engine is designed as a dispatcher component that classifies control packets and manages the protocol execution by creating new protocol contexts, delivering control messages to contexts, and destroying contexts. Each context is governed by one or more specialized state machine components that handle specific control messages.

The management plane consists of agent interface modules accessed using web and other interfaces. These interfaces provide a message abstraction layer for converting messages between the external management interface format and the internal alert format.

Using theses concepts, the Teja platform provides common, extensible building blocks such as layer 3 forwarding, packet classification and quality of service.

2.4 Platform Concepts

The Teja platform implements a small set of powerful system design concepts. They are divided into system architecture concepts and class framework concepts.

2.4.1 System Architecture

System architecture concepts provided by the platform are threads, memory spaces, and a network. A Teja application is a collection of threads and memory spaces. Multiple threads may connect to a memory space, thereby sharing the data stored in that space, and a thread may connect to multiple memory spaces. The platform also provides a universal network (implemented over TCP).

Threads communicate asynchronous, structured messages called alerts. This communication can occur over a network by copying the alert or over a shared memory space by copying a pointer to the alert from the sender to the receiver(s). The platform provides mechanisms for managing network connections and shared alert queues. Threads also communicate using traditional operating system primitives such as interrupts and semaphores.

A thread can be a server or an agent interface. As described before, server threads provide a scheduler for the execution of state machine components. CXUs can schedule multiple concurrent state machines while SXUs can schedule a single state machine. (This restriction enables optimization and produces a guarantee on the data and code space used.) Agent interface threads provide connectivity to external applications such as legacy management and control programs.

The core of developing applications using the Teja platform consists of programming server threads. Server threads provide an event-driven and time-driven scheduling engine. The server schedules run-to-completion callback functions called actions that are executed in the context of a data structure called a component. The actions are triggered by logic, timing, synchronization, or interrupt conditions called guards. A guard condition paired with an action function is called a transition. The component's behavior is specified as a state machine – that is, a directed graph whose nodes are states and links are transitions. Multiple state machines execute in parallel across multiple servers, and the servers schedule their transitions based on alert inputs, timer inputs, and transition guards.

Memory spaces provide application-specific memory management using configurable memory pools. A memory space may have zero or more memory pools. Each pool has multiple identical memory segments, called nodes. The pool specifies a list of application classes whose instances can be stored in a node. The Teja platform provides memory pool API functions for getting and putting nodes, and memory space API functions for instantiating and deleting instances of application classes. In addition to the dynamically created instances, memory spaces can also contain static instances of the application's auxiliary data structures. Each thread attached to a memory space indicates whether it can receive alerts in that space. If so, an alert input queue for that thread is defined in the memory space.

2.4.2 Class Framework

The object-oriented class framework provided by the Teja platform includes the Aux, Event, Alert, Component, Mutex and Queue classes. The Aux class is the root of the class hierarchy for the entire application. It prescribes a format for declaring portable and efficient data structures and provides accessor functions for them. The Event class inherits from the Aux class. Instances of Event and its subclasses are used to exchange structured messages between components within a single CXU in a synchronous manner. The Alert class inherits from the Event class. Instances of Alert and its subclasses are used to exchange structured messages between components in different servers in an asynchronous manner.

The Component class inherits from the Aux class. In addition to data structure descriptions permitted for Aux subclasses, subclasses of the Component class specify state machine behaviors that are scheduled by server threads. These state machines are a collection of continuous and discrete states joined by transitions. Discrete states are useful for event-driven finite state machine implementations. Continuous states are useful for time-driven computations. The state transitions define the active part of the state machine. All user code execution occurs during the state transitions. Transitions are triggered by time or event activity, and they in turn may trigger additional events. Components within a server communicate synchronous events. Components across servers communicate asynchronous alerts.

The Mutex class inherits from the Aux class. It provides blocking as well as non-blocking lock and unlock mechanisms for mutual exclusion in the access of shared resources by different threads. The Queue class inherits from the Aux class. It provides enqueue and dequeue API functions for storing and accessing Aux instances in a first-in-first-out queue in order to facilitate producer-consumer synchronization between different threads.

3 FLOW ACCOUNTING CASE STUDY

The use and performance of the Teja platform is illustrated with the flow accounting case study implemented on the Intel IXP1200 network processor. The IXP1200 combines the StrongARM microprocessor with six independent 32-bit RISC data engines, called micro-engines, each with four hardware-supported threads. The flow accounting case study is a part of the Teja packet classification building block. It was chosen as a relatively simple, well-understood example of network applications that nevertheless demonstrates the methodology of using the Teja platform.

3.1 Functional Description

The function of the flow accounting module is to sniff the network in promiscuous mode, classify packets and maintain billing-related information. Each packet is classified based on source and destination IP addresses, source and destination port numbers and the protocol (TCP, UDP, IP) to which the packet belongs. A record, maintained for each flow entry, accounts for the number of packets, number of bytes and duration of the flow. When a flow is idle for one second, the accounting information is communicated to a billing application and the flow entry is deleted. This module supports two gigabit Ethernet interfaces at line rate for packet sizes of 64 bytes (60 byte data and 4 byte CRC) and higher.

3.2 Architecture and Design

3.2.1 Application Architecture

This section describes the overall system architecture illustrated in Figure 1. The application can be partitioned into two sections: packet classification pipeline and flow data maintenance. In the first stage of the packet classification pipeline, two micro-engines are used to receive packets from the two gigabit Ethernet ports and place the received packets in software queues. In the second stage of the packet classification pipeline, two additional micro-engines are used to classify the received packets and update the accounting data structures. Flow data maintenance is performed by one micro-engine and the StrongARM core. The micro-engine is used to age out and send the accounting data to the StrongARM core. Upon receipt of this data, the StrongARM core performs some additional processing and saves the data to a permanent storage medium. It should be noted that one micro-engine in this application is left completely idle. This excess computing capacity could be used for more complex packet classification or data structure maintenance.

The memory banks in the IXP1200 are partitioned based on the size and access frequency of the data structures stored in them. The packet queues and statistic counters reside in the small, but extremely fast scratch pad. Packet descriptors and the first level hash table structure reside in the limited, but quick SRAM. The accounting data structures reside in the slower, but plentiful SDRAM memory bank. A detailed description of the data structures follows in the next section.

3.2.2 Application Data Structures

Three main data structures are used in the application and are shown in Figure 2. A packet descriptor is created for each packet received in the system. This packet descriptor contains all the

vital information about the packet including: source IP address, destination IP address, source port, destination port, protocol, and the packet size. This information is parsed from the packet header by the receive logic and is used to update the accounting structures. The accounting data in the system is tracked using a first level hash table and linked lists. Source-destination IP address pairs are tracked in a hash table of IPConnection nodes. The IPConnection node consists of the source IP address, destination IP address, and a hash collision list pointer. A parallel data structure to the IPConnection node resides in SDRAM and is found through a simple address mapping. This structure contains a flow node linked list header for each supported protocol (IP, UDP, TCP). The actual accounting data is maintained in a flow node for each source-destination port pair. The flow node contains the following fields: source port, destination port, number of bytes for the flow, and a time stamp of the last update. These nodes are maintained in a sorted singly linked list for each protocol associated with an IPConnection. These data structures are manipulated and updated by the logical components of the application described in the next section.

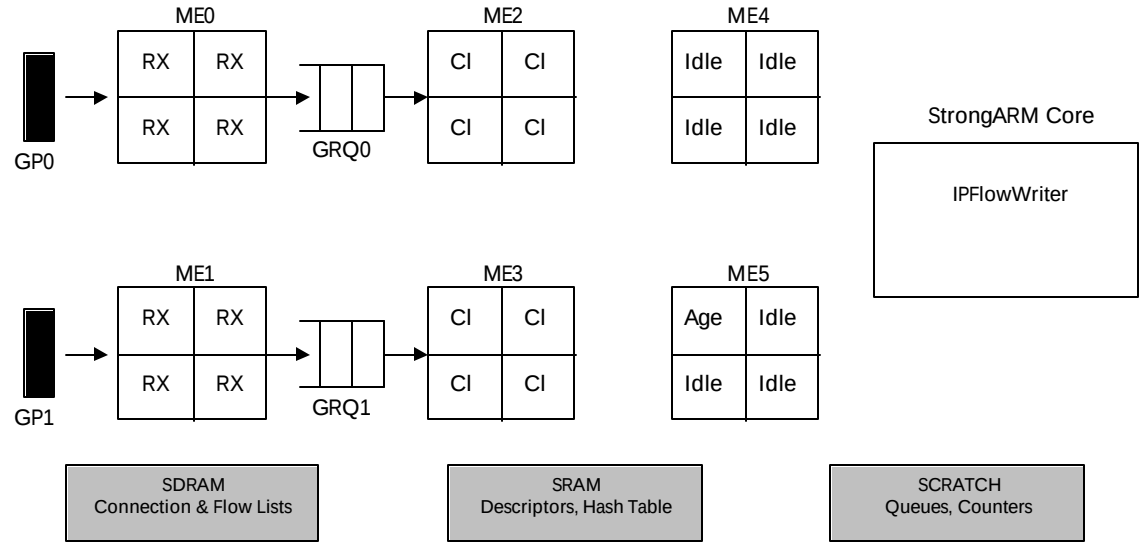

Figure 1. System architecture of the flow accounting application.

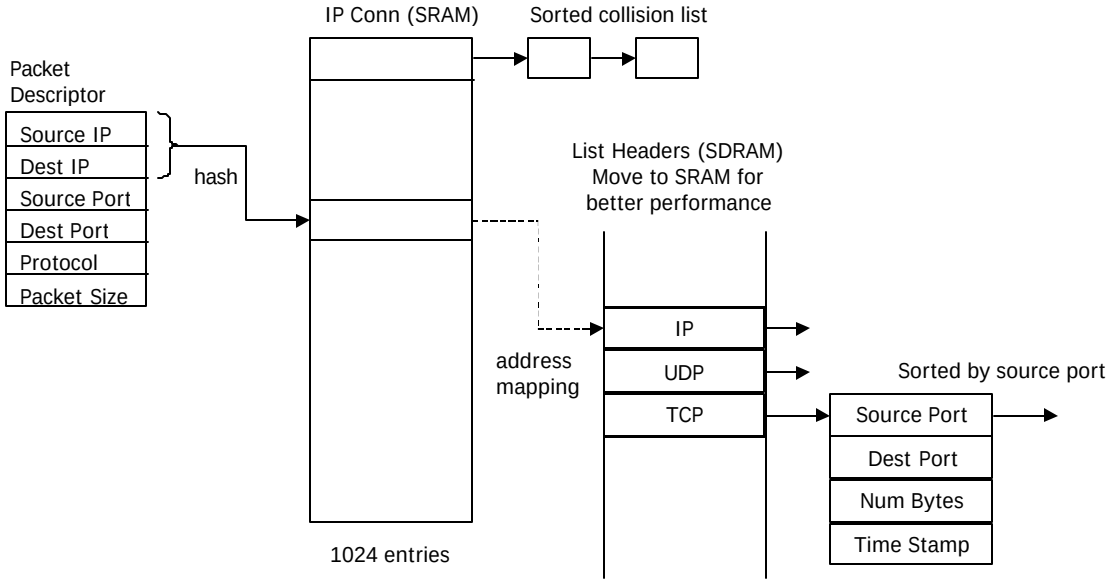

Figure 2. Data structures of the flow accounting application.

3.2.3 Application Logic

This section describes the four main components (state machines) that comprise the application.

3.2.3.1 Receive Driver Component

The receive state-machine receives packets from the MAC, creates a packet descriptor, and stores the header in a queue which will be processed by the Classifier component.

The receive driver component is MAC device-specific. The FastPortRx component of the Teja software platform is responsible for receiving packets from the eight 10/100 Ethernet ports on the IXF440 MAC. The GigPortRx component is responsible for receiving packets from the two gigabit Ethernet ports on the IXF1002 MAC. Both of these component classes are sub-classes of the TejaRx component class, which contains member variables and functions that are common to and required by all receive components. This application uses the GigPortRx component only.

The receive driver performs the following sequence of steps in order to process a single packet:

1. Allocate a packet descriptor from the packet descriptor memory pool in SRAM. This is where we will store the packet descriptor.
2. Check the availability of an MPKT (64 bytes from the Ethernet packet that can be transferred in a single request from the MACs into the IXP1200 receive FIFOs) from the MAC.
3. Initialize MPKT transfers from MACs into the receive FIFOs. If it is not the first MPKT of an Ethernet frame, skip to step 5.
4. Write the header information into the packet descriptor.
5. If it is the last MPKT, compute the packet size and enqueue the packet descriptor so that it can be processed by the receive thread. Continue from step 1.

3.2.3.2 Classifier Component

The classifier component is the second stage in the packet processing pipeline that resides on the data plane. The classifier attempts to dequeue a packet descriptor from the received packets queue. Upon successfully dequeuing a valid packet descriptor the component searches for the hash table entry corresponding to the source-destination IP address pair, also called an IPConnection. If an IPConnection entry is not found a new one is created. The IPConnection entry is locked to ensure that it is atomically updated by the processing element. Next, the protocol is parsed from the packet descriptor and the corresponding flow list is selected. (Flow lists are maintained for the IP, TCP and UDP protocols.) The selected flow list is then traversed searching for a flow element with the same source-destination ports as contained in the packet descriptor. Upon finding a matching flow element, the number of bytes and time stamp fields in the element are updated. If a matching flow element is not found a new element is created. To minimize search time, the flow lists are maintained in sorted order by source port. Upon completing the update, the IPConnection entry is unlocked. Finally, the classifier attempts to dequeue another packet descriptor and the process is repeated.

3.2.3.3 IPFlowAger Component

The IPFlowAger component is the third component that resides on the data plane and works independently from the packet receiving and classifying components. The IPFlowAger periodically scans the hash table of the IPConnections (source-destination IP pairs) looking for valid entries. Upon finding a valid entry, the entry is locked and each of the protocol flow lists for that entry is traversed searching for flow entries that have been idle for approximately one second. When an

idle entry has been found the component sends an asynchronous message containing the flow information to the IPFlowWriter component that resides on the StrongARM core. The Teja execution unit handles the actual transmission of the message, relieving the developer of this somewhat tedious and complicated task. Then, the flow entry node is deallocated by returning it to the free list for future reuse. If all of the protocol flow lists for a given IPConnection are empty, the IPConnection is removed from the hash table. Finally, the entry is unlocked when scanning is complete.

3.2.3.4 IPFlowWriter Component

The IPFlowWriter component resides on the StrongARM core of the IXP1200. This component processes and records the expired flow data. The component receives the expired flow data sent from the IPFlowAger component in the form of asynchronous messages. Upon receiving a message the Teja execution unit "wakes up" the IPFlowWriter, so that it may process the message. In this simple case study, the flow data (source IP address, destination IP address, source port, destination port, number of bytes, and time stamp) are parsed from the message and recorded to memory. In most cases, more advanced processing of this information will be desired and can be performed in this component.

3.3 Performance

3.3.1 Packet Throughput

The application was tested using repeated bursts of 48 million 60-byte packets across 48 unique flows on each of the two gigabit Ethernet ports. Line rate performance was sustained without dropping any packets. The performance was "tight" – i.e., packets will be dropped for smaller packet sizes or significantly longer flow lists. However, the latter degradation can be combated to some extent by introducing hash tables or tries for the source and destination ports and protocol fields, and by moving them into SRAM. Given the performance for minimum sized packets, line rate for larger packet sizes is assured and was verified through actual testing.

3.3.2 Code Size

The total generated code size (in lines, excluding comments) is shown in the following table.

C source code (.c)	17,723
C header code (.h)	9,140
IXP1200 micro-engine assembly source code (.uc)	6,084
IXP1200 micro-engine assembly header code (.h)	249

In addition to code generated for the IPFlowWriter component, the C code contains initialization logic for memory management, message passing and other system functions. It should be noted that the entire application consists of the generated code compiled and linked with the Teja system platform.

The total hand-written code size is shown in the following table. This code is written within the Teja platform tools in code boxes. All other generated code was specified using graphical constructs. The generated code is executed without modification.

C code	3
Teja API calls in code boxes	540
IXP assembly code	496

Even though the Teja API calls were made in hand-written code within the code box constructs, they are portable across all supported target platforms. With further improvements in the graphical tools, these API calls can be removed from the hand-written code and replaced with templates accessed using menu items. This implies that only about 1-2% of the generated code, and less

than 10% of the generated assembly code is non-portable, hand-written assembly code, suggesting that the tools are likely to enhance productivity while at the same time supporting hand-tuning of performance-critical code.

The loaded data plane code size (in words) is shown in the following table.

Receive micro-engine	213
Classification micro-engine	264
Flow aging micro-engine	650

The receive and classification code is significantly smaller than the flow aging code. This is because the latter incorporates two features of the Teja system platform not used by the other elements: timer variables for introducing real-time delays in processing and asynchronous message passing between the data plane and control plane processors initiated by the micro-engine.

3.3.3 Productivity

Using the Teja platform, initial implementation of the case study without regard to performance was accomplished within three person-weeks. This implementation achieved slightly over 70% of line rate performance. Subsequently, one week was spent on improving the performance to 100% of line rate. These improvements resulted from the following optimizations.

- Teja-specific optimizations (9%)
  - Usage of single class memory pools for dynamic allocation (5%)
  - Automatic state machine unrolling (2%)
  - Optimization and re-ordering of transition guard conditions (2%)
- Application-specific optimizations: combine multiple accessors of one class instance into one memory access (20%). For example, the following four accesses require approximately 120 cycles.

  IPFlow_hal_set_num_pkts (hal_this, hal_this_membank_id, num_pkts)
  IPFlow_hal_set_num_bytes (hal_this, hal_this_membank_id, num_bytes)
  IPFlow_hal_set_created_time (hal_this, hal_this_membank_id, time)
  IPFlow_hal_set_updated_time (hal_this, hal_this_membank_id, time)

  They can be combined into the following single memory access requiring approximately 35 cycles.

  Teja_hal_set_4_longwords (hal_this, hal_this_membank_id, offset, num_pkts, num_bytes, time, time)

  This requires laying out application data structures according to their locality of reference. In such a case, the code generator recognizes the locality and compresses memory accesses as appropriate.

The Teja IXP1200 code generator automatically carries out these optimizations. The application only required minor hand tuning to insure that all the optimization opportunities were fully exploited.

The same application functionality was implemented by hand (without using the Teja platform) by a different, but expert, group of firmware engineers from another organization. That implementation achieved the same packet throughput performance. In comparison of the two approaches, it was found that the use of the Teja platform yielded a 3:1 productivity advantage.

4 CONCLUSION

Network processors provide the advantages of flexibility and performance in packet processing but present serious challenges for software development. The current approach of fitting sequential code into a multi-threaded environment without optimized scheduling leads to both perform-

ance and design bottlenecks. Network processor software needs to be developed around a programming model that exploits the multithreaded nature of network processors and unifies the different needs such as forwarding *vs*. control processing, sequential *vs*. parallel code, symmetric *vs*. asymmetric processing, special *vs*. general languages, and static *vs*. dynamic resource assignment.

These complex issues are resolved by a solution based on three major considerations: state machine-based behavior descriptions, object-oriented data structure descriptions and event-driven multiprocessor scheduling. Based on these solution principles, the Teja software platform simplifies and standardizes network processor-based system development. Its core underlying technology is a high-performance, parallel and distributed software execution platform for network processors and multiprocessor systems. The accompanying graphical development environment supports a sound system design methodology of using the platform. The platform incorporates an extensible framework of network application building blocks for data forwarding, with integrated control and management interfaces.

In this paper, we have described the Teja software platform for network processors and illustrated its use and performance with a typical network application case study. Based on measurements of system performance and development productivity, we believe that the use of the Teja platform significantly accelerates program development without compromising – indeed, often enhancing – system performance. Future work on the platform is expected to expand its support for multiple network processors, increase the tools capabilities with respect to usability, debugging and verification, and extend the library of network application building blocks to meet the needs of a wider range of edge and core applications.

REFERENCES

Deb2000. Alak Deb. Building a network processor-based system. ICD.

Spalink2000. Tammo Spalink, Scott Karlin, Larry Peterson. Evaluating network processors in IP forwarding. Princeton University Technical Report TR-626-00. November 15, 2000.

Spalink2001. Tammo Spalink, Scott Karlin, Larry Peterson, Ytzchak Gottlieb. Building a robust software-based router using network processors.

Cravotta2001. Nicholas Cravotta. Network processing: with fragmentation comes opportunity. EDN. April 12, 2001.

Herity2001. Dominic Herity. Network processor programming. Internet Appliance Design. August 27-30, 2001.