ARCHITECTURE FOR A HARDWARE-BASED, TCP/IP CONTENT-PROCESSING SYSTEM

A NEW ARCHITECTURE PERFORMS CONTENT SCANNING OF TCP FLOWS IN HIGH-SPEED NETWORKS. COMBINING A TCP PROCESSING ENGINE, A PER-FLOW STATE STORE, AND A CONTENT-SCANNING ENGINE, THIS ARCHITECTURE PERMITS COMPLETE PAYLOAD INSPECTIONS ON 8 MILLION TCP FLOWS AT 2.5 GBPS.

David V. Schuehler
James Moscola
John W. Lockwood
Washington University in St. Louis

The Transmission Control Protocol is the workhorse protocol of the Internet. Most of the data passing through the Internet transits the network using TCP layered atop the Internet Protocol (IP). Monitoring, capturing, filtering, and blocking traffic on high-speed Internet links requires the ability to directly process TCP packets in hardware. Because TCP is a stream-oriented protocol that operates above an unreliable datagram network, there are complexities in reconstructing the underlying data flow.

High-speed network intrusion detection and prevention systems guard against several types of threats (see the "Related work" sidebar). When used in backbone networks, these content-scanning systems must not inhibit network throughput. Gilder's law predicts that the need for bandwidth will grow at least three times as fast as computing power.[1] As the gap between network bandwidth and computing power widens, improved microelectronic architectures are needed to monitor and filter network traffic without limiting throughput. To address these issues, we've designed a hardware-based TCP/IP content-processing system that supports content scanning and flow blocking for millions of flows at gigabit line rates.

TCP splitter

The TCP splitter[2] technology was previously developed to monitor TCP data streams, sending a consistent byte stream of data to a client application for every TCP data flow passing through the circuit. The TCP splitter accomplishes this task by tracking the TCP sequence number along with the current flow state. Out-of-order packets are dropped to ensure that the client application receives the full TCP data stream without the need for large stream reassembly buffers.

Dropping packets to maintain an ordered packet flow throughout the network can adversely affect the network's overall throughput. Jaiswal et al. analyzed out-of-sequence packets in tier-1 IP backbones.[3] They noted that approximately 95 percent of all TCP packets on Internet backbone links were in proper sequence. Network-induced packet reordering accounted for a small fraction of out-of-sequence packets, with most resulting from retransmissions due to data loss. More than 86 percent of all observed TCP flows contained no out-of-sequence packets.

Earlier research at the Washington University Applied Research Laboratory led to the development of a reconfigurable hardware

Published by the IEEE Computer Society

Related work

By their very nature, intrusion detection systems (IDSs) and intrusion prevention systems must perform deep packet inspections on all traffic traversing the network. This task is difficult when data rates are high and the system must track many simultaneous flows. Software IDS solutions, such as Snort,[1] work well only when aggregate bandwidth rates are low.

Implementing an external monitor that can track a Transmission Control Protocol (TCP) connection state is difficult. Bhargavan et al. discuss the complexities associated with tracking various properties of a protocol using language recognition techniques.[2] General solutions to this problem can vary greatly.

Monitoring and reassembling flows—tasks required for an IDS—become even more complicated by direct attempts to evade detection. Handley et al. expound on this topic.[3] One such technique for evading detection would be to modify an end-system protocol stack such that TCP retransmissions contain different content than original data transmissions.

A recently developed passive monitoring system can capture and accurately time stamp packets at data rates of up to OC-48 (2.5 Gbps).[4] Highly accurate time stamps correlate data captured by multiple monitoring systems in a wide area network. Optical splitters deliver a copy of the network traffic to the monitoring station. The system stores the first 44 bytes of each packet and the analysis of the captured data occurs out of band.

Network World Fusion tested six commercially available gigabit IDSs by sending 28 attacks along with 970 Mbps of background traffic.[5] After system tuning, only one system detected all 28 of their attacks while processing data on a gigabit Ethernet link. In general, software-based systems are incapable of matching regular expressions at gigabit rates.

Previous work also exists in the area of string matching on field-programmable gate arrays. Sidhu and Prasanna were primarily concerned with minimizing the time and space required to construct nondeterministic finite automatons (NFAs).[6] They run their NFA construction algorithm in hardware instead of software. To perform string matching, Hutchings, Franklin, and Carver followed with an analysis of this approach for the large set of regular expressions found in a Snort database.[1,7]

References

1. M. Roesch, Snort: Lightweight Intrusion Detection for Networks," *Proc. 13th Systems Administration Conf.* (LISA 99), Usenix Assoc., 1999, pp. 229-238.
2. K. Bhargavan et al., "What Packets May Come: Automata for Network Monitoring," *Proc. 28th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, ACM Press, 2001, pp. 206-219.
3. M. Handley, V. Paxson, and C. Kreibich, "Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics," *Proc. 10th Usenix Security Symp.*, Usenix Assoc., 2001, pp. 115-131.
4. C. Fraleigh et al., "Design and Deployment of a Passive Monitoring Infrastructure," *Proc. Int'l Workshop Digital Comm.*, *Lecture Notes in Computer Science*, vol. 2170, Springer, 2001, pp. 556-575.
5. B. Yocom, R. Birdsall, and D. Poletti-Metzel, "Gigabit Intrusion-Detection Systems," *Network World*, 4 Nov. 2002; NetworkWorld Fusion, http://www.nwfusion.com/reviews/2002/1104rev.html.
6. R. Sidhu and V.K. Prasanna, "Fast Regular Expression Matching Using FPGAs," *Proc. 9th Ann. IEEE Symp. Field-Programmable Custom Computing Machines* (FCCM 01), IEEE CS Press, 2001.
7. B.L. Hutchings, R. Franklin, and D. Carver, "Assisting Network Intrusion Detection with Reconfigurable Hardware," *Proc. 10th Ann. IEEE Symp. Field-Programmable Custom Computing Machines* (FCCM 02), IEEE CS Press, 2002, pp. 111-120.

platform called the Field-Programmable Port Extender (FPX). The FPX operates on packets[4] using customized hardware circuits that are dynamically programmed into a Xilinx Virtex 2000E field programmable gate array (FPGA). The FPX platform operates on a 32-bit-wide data path at 80 MHz to process data at OC-48 (2.5 Gbps) line rates.

A suite of layered protocol wrappers processes network and transport protocols in reconfigurable hardware.[5] The wrappers include an asynchronous transfer mode cell wrapper, an ATM adaptation layer type 5 (AAL5) frame wrapper, and an IP wrapper.

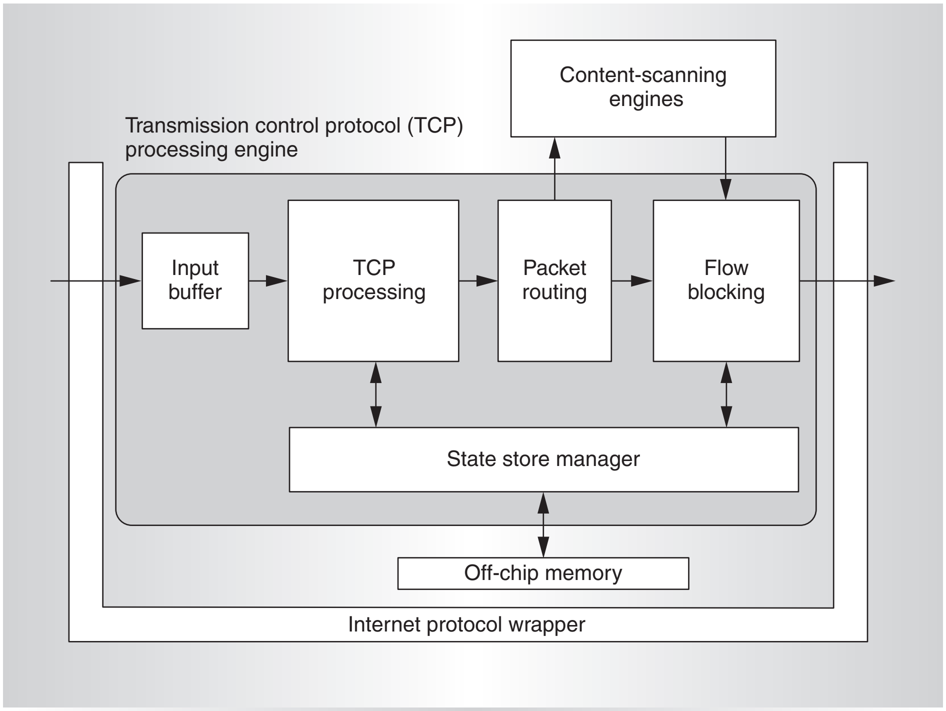


Figure 1. System overview.

These wrappers provide lower-layer protocol processing for our TCP architecture.

Content-scanning engine

The content-scanning engine can scan the payload of packets for a set of regular expressions.[6] To do so, this hardware module employs a set of deterministic finite automata, each searching in parallel for one of the targeted regular expressions. Upon matching a network data packet's payload with any of these regular expressions, the content-scanning engine can either let the data pass or drop the packet. This engine can also send an alert message to a log server when it detects a match in a packet. The alert message contains the matching packet's source and destination addresses along with a list of regular expressions found in the packet. The content-scanning engine, when implemented with four parallel search engines, provides a throughput of 2.5 Gbps.

TCP-based content-scanning engine

The new TCP-based content-scanning engine integrates and extends the capabilities of the TCP splitter and the old content-scanning engine. Figure 1 shows a diagram of a typical system. In the figure, data flows from left to right. IP packets travel to the TCP processing engine from the lower-layer-protocol wrappers. An input packet buffer provides a limited amount of packet buffering for downstream processing delays. The TCP processing engine validates packets and classifies them as part of a flow. This engine then sends packets along with their associated flow state information to the packet-routing module, which routes the packets to one of several content-scanning engines or the flow-blocking module. Multiple content-scanning engines evaluate regular expressions against the TCP data stream. Packets returning from these engines go to the flow-blocking module, which stores application-specific state information and enforces flow blocking. This module sends unblocked packets back to the network switch, which forwards them to their final destination.

Design requirements

Hash tables are used to index memory that stores each flow's state. Gracefully handling hash table collisions is difficult for real-time systems. To ensure proper monitoring of all flows, the state store manager can chain a linked list of flow state records off of the appropriate hash entry. Although this approach allows complete monitoring of all flows, the time required to traverse a long linked list of hash bucket entries can be excessive. Delays from retrieving flow state information can adversely affect system throughput and lead to data loss. Another drawback of linked entries in the state store is the need for buffer management operations, which induces additional processing overhead into a system. Our state store manager limits this linked-list chain length to a constant number of entries, bounding the amount of time required to perform a state retrieval operation.

A hashing algorithm that produces an even distribution across all hash buckets is important to the circuit's overall efficiency. We performed initial analysis of the flow-classification hashing algorithm for this system against packet traces available from the National Laboratory for Applied Network Research. With 26,452 flow identifiers hashed into a table of 8 million entries, a hash collision occurred in less than 0.3 percent of the flows.

We've added features to the TCP processing circuit to support the following services:

- *Flow blocking.* This will let the system block a flow at a particular byte offset within the TCP data stream.
- *Flow unblocking.* The system can re-enable a previously disabled flow so that data for a particular flow can once again pass through the circuit.
- *Flow termination.* This mechanism will shut down a selected flow by generating a TCP FIN (finish) packet.
- *Flow modification.* We will provide the ability to sanitize selected data contained within a TCP stream.

Flow state store

To support millions of TCP flows, the TCP processing engine uses one 512-Mbyte, off-chip, synchronous dynamic random access memory (SDRAM) module. The interface to this module has a 64-bit-wide data path and supports a burst length of eight memory operations. By matching our per-flow memory requirements with the burst width of the memory module, we can optimize use of memory bandwidth. Storing 64 bytes of state information for each flow lets the memory interface match the amount of per-flow state information with the amount of data in a burst transfer to memory. This configuration supports 8 million simultaneous flows. Assuming $50 as the purchase price for a 512-Mbyte SDRAM memory module, the cost to store context for 8 million flows is only 0.000625 cents per flow, or 1,600 flows per penny.

Of the 64 bytes of data stored for each flow, the TCP processing engine uses 32 bytes to maintain flow state and memory management overhead. The additional 32 bytes of state store for each flow can hold the application-specific data for each flow context. Figure 2 shows the layout of a single entry for a given flow.

The hash algorithm contained within the TCP processing engine hashes the source and destination IP addresses and TCP ports into a 22-bit value. This hash value serves as a direct index to the first entry in a hash bucket. The record's format lets the hash table contain 4 million records at fixed locations, and an additional 4 million records to form a linked list of records for hash collisions. Using linked-list records enables the storing of state informa-

63 32	31 0
Flow ID	Hash value
Flags next flow ID	Source IP address
Sequence number	Data IP address
Blocking sequence number	Ports
Application data	Application data
Application data	Application data
Application data	Application data
Application data	Application data

Figure 2. Flow state record for one entry, for a given flow. Each box represents 32 bits; two adjacent boxes collectively represent 64 bits, which the state store manager can read from SDRAM in one clock cycle. For example, the hash value is located at bits 31 to 0, and the flow ID at bits 63 to 32, of the first memory location. Because the memory device supports burst read and write operations, the state store manager retrieves all data (8 rows, 64 bits each) in a single memory operation. The state store manager maintains one of these records for every flow that the content-scanning engine processes.

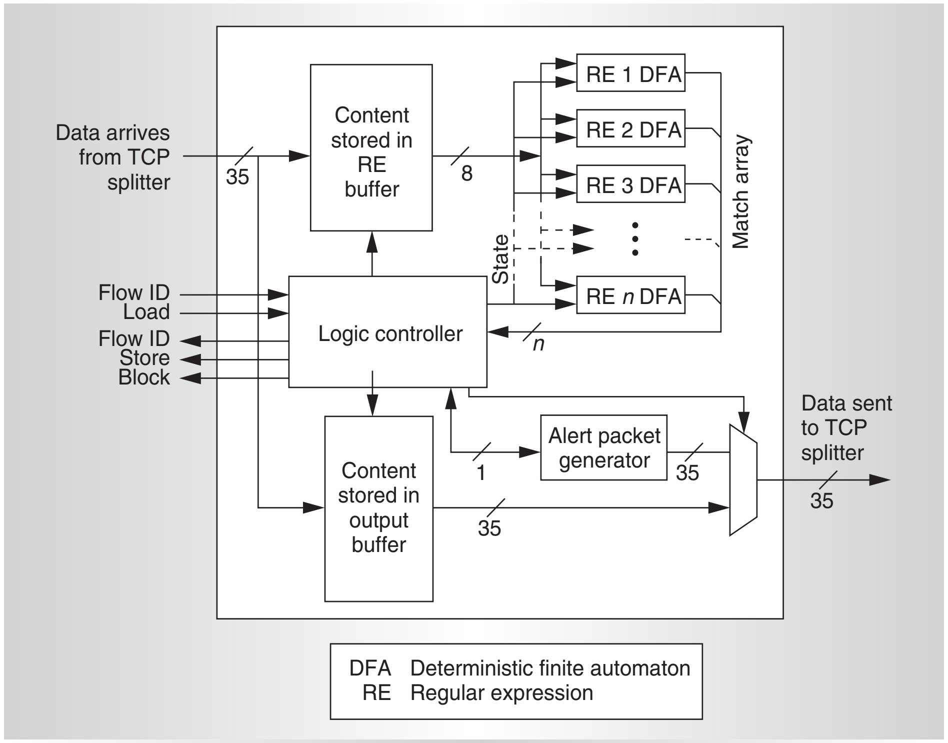

Data arrives from TCP splitter
35
Content stored in RE buffer
8
RE 1 DFA
RE 2 DFA
RE 3 DFA
RE n DFA
Match array
State
Flow ID
Load
Flow ID
Store
Block
Logic controller
n
Alert packet generator
1
35
Data sent to TCP splitter
35
Content stored in output buffer
35
DFA Deterministic finite automaton
RE Regular expression

Figure 3. Block diagram of the content-scanning engine.

tion for multiple flows that hash to the same bucket. To ensure that the system can maintain real-time behavior, we constrain the number of link traversals to a constant value.

The state store manager can cache state information using on-chip block RAM memory. This provides faster access to state information for the most recently accessed flows. A write-back cache design improves performance.

Stream-based content scanning

The content-scanning engine processes TCP data streams from the TCP processing engine, which lets the content-scanning engine match data that spans across multiple packets. The content-scanning engine must perform regular-expression-based scans on many active TCP flows. To process interleaved flows, it must perform a context switch to save and restore per-flow context information. When a packet reaches the content-scanning engine through some flow, the content-scanning engine must restore the last known matching state for that flow before starting the matching operation on that packet. When it has finished processing the packet, the content-scanning engine must save the flow's new matching state by using the TCP processing circuit's state store resources.

Figure 3 shows the content-scanning engine along with the interface signals to the TCP processing circuit. Packets are passed to the content-scanning engine along with a flow identifier and context information. Once the content-scanning engine loads the current state for that flow, it can process the packet. The content-scanning engine contains several regular-expression engines, which use deterministic finite automatons (DFAs) to search for content in TCP data flows. If the content-scanning engine finds no matches in the packet, then the packet can pass through the module. If it discovers a match, then it communicates with the flow-blocking module to block the flow, terminate the connection, or let the data pass. It can also send out alert messages in response to content matches. The alert message's format is a user datagram protocol (UDP) packet; besides using this format for generating alert messages, the system also uses UDP packets for logging system events and processing control information.

Each content-scanning engine processes data one byte at a time. The TCP processing circuit uses a 4-byte-wide data path, so the content-scanning engine must perform a 4-to-1 slowdown when processing packet data. Having four content-scanning engines in parallel and processing four flows concurrently, as Figure 4 shows, can maintain the system's overall throughput. The system dispatches incoming packets to one of the scanning engines based on a hash of the flow ID provided by the TCP splitter. Dispatching packets in this way eliminates the possibility of hazards that could occur if two content-scanning engines were simultaneously processing packets from the same flow.

TCP processing

The architecture receives data through the IP wrappers.[3] As the left side of Figure 1 shows, this data passes into an input buffer.

From the input buffer, IP frames go to the TCP processing engine. An input state machine tracks the processing state within a single packet. The input state machine then forwards the data to

- a first-in, first-out (FIFO) frame buffer, which stores the packet;
- a checksum engine, which validates the TCP checksum; and
- a flow classifier, which computes a hash value for the packet.

The flow classification hash value is passed to the state store manager, which retrieves the state information associated with the particular flow. Results are written to a control FIFO buffer, and the state store is updated with the current state of the flow. An output state machine reads data from the frame and control FIFO buffers and passes it to the packet-routing engine. Most traffic flows through the content-scanning engines, which scan the data. Packet retransmissions bypass these engines and go directly to the flow-blocking module.

Data returning from the content-scanning engines also goes to the flow-blocking module. This stage updates the per-flow state store with the latest application-specific state information. If a content-scanning engine has enabled blocking for a flow, the flow-blocking module now enforces it. This module compares the packet's sequence number with those sequence numbers for which flow blocking should take place. If the packet meets the blocking criteria, the flow-blocking module drops it from the network. Any remaining packets go to the outbound protocol wrapper.

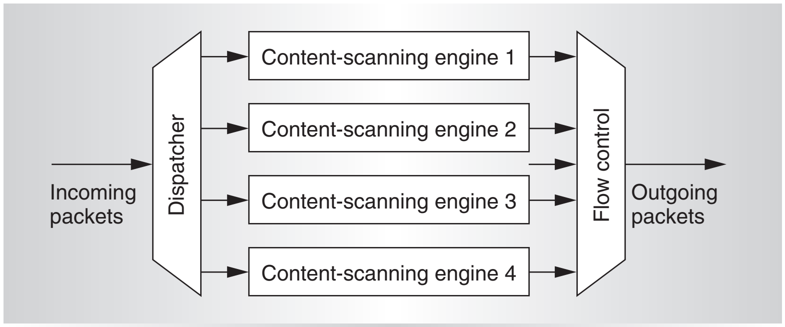
The state store manager is responsible for processing requests to read and write flow state records. It also handles all interactions with SDRAM memory, and it caches recently accessed flow state information. The SDRAM controller exposes three memory-access interfaces: a read-write, a write only, and a read only. The controller prioritizes requests in that order, with the read-write interface having the highest priority.

Figure 5 shows the state store manager's layout along with its interactions to the memory controller and other modules in the TCP processing engine. The flow classifier computes a flow identifier hash value and initiates a record retrieval operation by communicating with the state store manager. The state store manager uses the memory controller's read-only interface to retrieve the flow's current state information and returns this information to the TCP processing engine. If the packet is valid and the engine accepts it, the state store manager performs an update operation to store the new flow state. The flow-blocking module also performs a SDRAM read operation to determine the current flow-blocking state. If the flow-blocking state has changed, or if there's an update to the application-specific state information, the flow-blocking module performs a write operation to update the flow's saved state information.

In a worst-case scenario in which there's no more than one entry per hash bucket, each packet requires a total of two read and two write operations to the SDRAM:

- an 8-word read to retrieve flow state,
- an 8-word write to initialize a new flow record,
- a 4-word read to retrieve flow-blocking information, and
- a 5-word write to update application-specific flow state and blocking information.

Memory accesses aren't necessary for TCP acknowledgment packets containing no data. Analysis indicates that all read and write operations can occur during packet processing if the average TCP packet contains more than 120 bytes of data. If the TCP packets contain less than this amount, there might not be enough time to complete all memory operations during packet processing. In that case,


Incoming packets
Dispatcher
Content-scanning engine 1
Content-scanning engine 2
Content-scanning engine 3
Content-scanning engine 4
Flow control
Outgoing packets
Figure 4. Arrangement of parallel content-scanning engines.

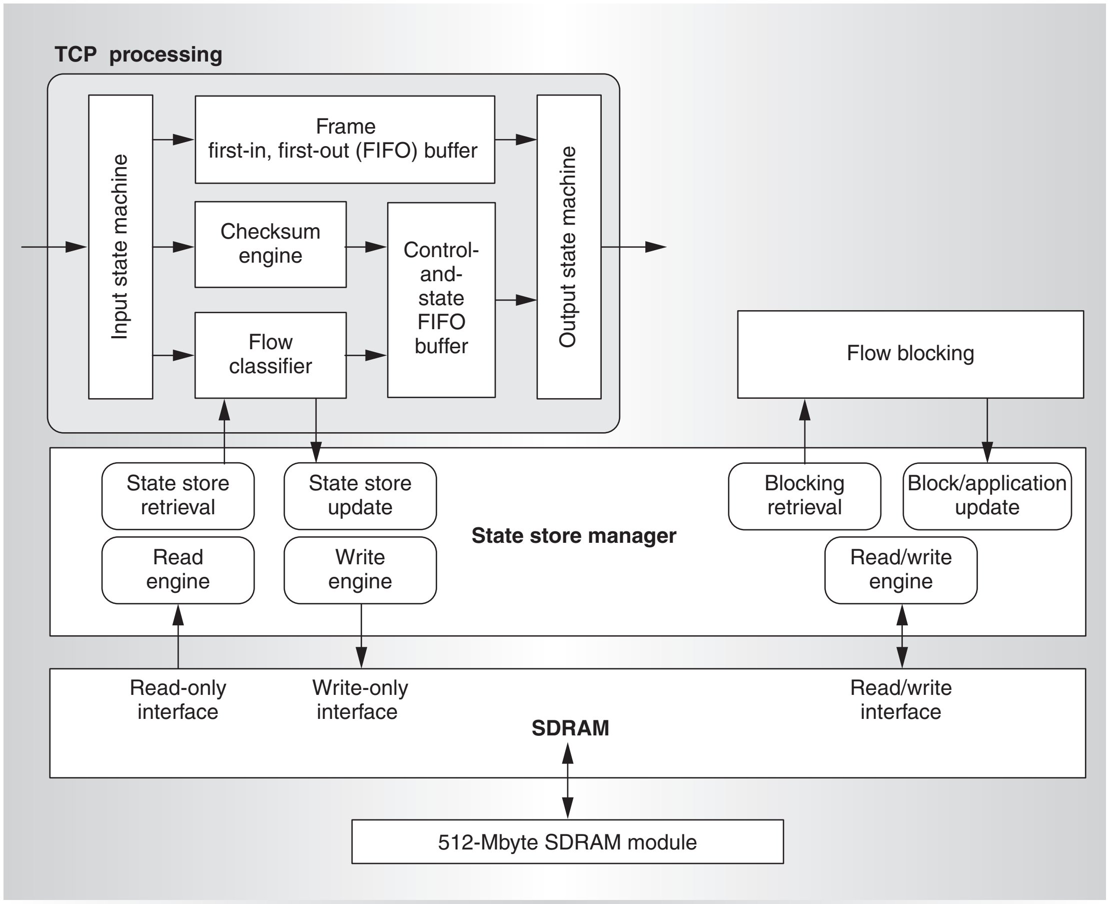


Figure 5. TCP processing engine and state store manager.

the packet could stall while waiting for a memory operation to complete.

The average TCP packet size on the Internet is about 300 bytes.[7] Given that a percentage of all TCP packets are 0-length acknowledgments, the average size of a packet requiring memory operations to the state store will be larger than this 300-byte average. Processing larger packets decreases the likelihood of the packet's stalling to wait for memory access latencies. On average, the system will have more than twice the memory bandwidth required to process a packet when operating at OC-48 (2.5 Gbps) rates.

New FPGA devices are available that have four times the number of logic gates and operate at over twice the clock rate of the XVC2000E used on the FPX platform. Using the higher gate densities, it's possible to instantiate multiple copies of the TCP processing engine to increase the system's throughput. In addition, the latest memory modules have higher clock frequencies and offer double-data-rate transfer speeds, increasing the memory bandwidth. Using these new devices, the TCP-based content-scanning engine could achieve OC-192 (10 Gbps) data rates without requiring major modifications.

Future work will include enhancing the TCP processing engine to support passive monitoring by buffering out-of-order packets and reinjecting them into the system when missing packets are retransmitted. There is also work to integrate the TCP processing engine with other types of high-speed content-scanning technologies. MICRO

Acknowledgments

This research was supported by a grant from Global Velocity. John Lockwood is a cofounder and consultant for Global Velocity and an assistant professor at Washington University in St. Louis. The authors of this article have equity and may receive royalty from a license of this technology to Global Velocity. Packet trace data used to evaluate flow classification hashing algorithms was obtained from the National Laboratory for Applied Network Research, an organization sponsored by the National Science Foundation Cooperative Agreement number ANI-9807479.

References

1. E.P. Markatos, "Speeding up TCP/IP: Faster Processors Are Not Enough," *Proc. 21st IEEE Int'l Performance, Computing, and Communications Conf.*, IEEE Press, 2002, pp. 341-345.
2. S. Jaiswal et al., "Measurement and Classification of Out-of-Sequence Packets in a Tier-1 IP Backbone," *Proc. Internet Measurement Workshop*, ACM Press, 2002, pp. 113-114.
3. D.V. Schuehler and J. Lockwood, "TCP-Splitter: A TCP/IP Flow Monitor in Reconfigurable Hardware," *Proc. 10th Symp. High Performance Interconnects* (Hot Interconnects X), IEEE Press, 2002, pp. 127-131.
4. J.W. Lockwood, "An Open Platform for Development of Network Processing Modules in Reprogrammable Hardware," *Proc. IEC DesignCon*, Int'l Eng. Consortium, 2001, pp. WB-19.
5. F. Braun, J.W. Lockwood, and M. Waldvogel, "Layered Protocol Wrappers for Internet Packet Processing in Reconfigurable Hardware," *Proc. 10th Symp. High-Performance Interconnects* (Hot Interconnects IX), IEEE CS Press, 2001, pp. 93-98.
6. J. Moscola et al., "Implementation of a Content-Scanning Module for an Internet Firewall," *Proc. 11th Ann. IEEE Symp. Field-Programmable Custom Computing Machines* (FCCM 03), IEEE CS Press, 2003, pp. 31-38.
7. K. Thompson, G.J. Miller, and F. Wilder, "Wide-Area Internet Traffic Patterns and Characteristics," *IEEE Network Magazine*, vol. 11, no. 6, Nov.-Dec. 1997, pp. 10-23.

David V. Schuehler is a PhD candidate in the Applied Research Laboratory at Washington University in St. Louis. He is also vice president of research and development for Reuters. His research interests include real-time processing, embedded systems, and high-speed networking. Schuehler has a BS in aeronautical and astronautical engineering from Ohio State University and an MS in computer science from the University of Missouri-Rolla. He is a member of the IEEE and the ACM.

James Moscola is a PhD candidate at Washington University in St. Louis. His research interests include digital content protection and deep packet inspection. Moscola has a BS in physical science from Muhlenberg College; and a BS in computer engineering and an MS in computer science, both from Washington University in St. Louis.

John W. Lockwood is an assistant professor at Washington University in St. Louis. His research interests include designing and implementing networking systems in reconfigurable hardware, and he developed the Field-Programmable Port Extender to enable rapid prototyping of extensible network modules. Lockwood has a BS, an MS, and a PhD in electrical and computer engineering from the University of Illinois. He is a member of the IEEE, the ACM, Tau Beta Pi, and Eta Kappa Nu.

Direct questions and comments about this article to David V. Schuehler, Washington University in St. Louis, Applied Research Laboratory, Campus Box 1045, One Brookings Dr., St. Louis, MO 63130; dvs1@arl.wustl.edu.

Other information related to this project is online at http://www.arl.wustl.edu/arl/projects/fpx/reconfig.htm.