Stream Handlers: Application-specific Message Services on Attached Network Processors

Ada Gavrilovska, Kenneth Mackenzie, Karsten Schwan, and Austen McDonald
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, 30332, USA
{ada, kenmac, schwan, austen}@cc.gatech.edu

Abstract

This paper presents a software architecture that enables the application-specific processing of messages on network processors attached to cluster machines. Such processing is performed by stream handlers *executed on such* attached network processors *(ANPs) and able to manipulate both message headers and their data content. Handler execution can be associated with the ANPs' receive side, its transmit side, or both. Using Intel's IXP1200 boards as sample ANPs, the paper evaluates performance advantages and tradeoffs in stream handler execution. Results indicate that while receive-side stream customization is useful for simple stream handlers, it becomes a bottleneck and results in degraded performance with increased handler complexity or with increased amounts of data manipulated by handlers. In comparison, transmit-side handler execution exhibits more constant and acceptable performance levels and can therefore, support richer ANP functionality.*

1. Introduction and Motivation

An emerging class of programmable network processors [7, 10] has made it possible to implement application-specific services that execute 'close' to the network. Services known to be useful include software routing, payload caching, synchronization, zero-copy access to an application's communication buffers, reliable management of application data via replication, packet classification, load-balancing, and support for network monitoring [2, 5, 14, 15, 13, 18]. Furthermore, for data streaming applications that include multimedia, visualization, or operational information systems, research has demonstrated the utility of placing services like data mirroring [6] or scheduling [11] onto network processors and/or at various host nodes along a communication path [4, 16].

Targeting data streaming applications, our research concerns the movement of selected stream processing actions – *stream handlers* – from host nodes onto network processors associated with them. The services placed onto such *attached network processors* (ANPs) (1) address large data applications by filtering data as per current end user needs, an example being filtering to transmit only those data items that are actually currently viewed in a remote scientific visualization [9, 12]; (2) they address multimedia applications, by executing scheduling algorithms that reduce data by discarding packets based on loss tolerance and deadline information [11, 17]; and (3) they concern execution of codes for large-scale operational information systems (OISs) used by companies that include Delta Air Lines [6], where data mirroring services improve both the response times and the reliability of the information captured, processed, and distributed on a 24/7 basis.

Newer generation network processors like Intel's IXP1200 chip [10] are quite capable of running application-specific services. Specifically, since their hardware is designed to efficiently move high volumes of network packets between its ports, such packet movement can be enriched to execute many application services at link speeds. Experimental results presented in this paper validate this statement, even under high load conditions and particularly for services that filter or reduce message traffic.

Previous work with IXP boards [5, 15] and with other network processors [2, 11, 14, 18] has already demonstrated the benefits to applications of offering a richer set of ANP-resident services. IXP-based improvements for wide area applications are attained by enabling packet header-based customization of an incoming data stream, thereby offering services such as software routing, network monitoring, etc. On ANPs used with cluster machines, improvements are attained by computationally lightweight but communication-rich application services. These services run more efficiently on ANPs than on host nodes, because the ANP is 'closer' to the network than the host node with which it is associated, allowing it to operate on data streams with lower latency and higher achieved throughput [13]. Finally, the use of ANPs can offload host nodes both directly, by removing from them application actions like data filtering and stream customization and indirectly, by removing loads from their CPUs, I/O busses, and memory via elimination of unnecessary communications. As a result, host resources can be dedicated to the application's core functionality, while the ANPs execute the low-latency and high-bandwidth application-specific tasks for which they are best suited.

We are developing and evaluating a software architecture that allows applications to place custom processing of headers and data content into an ANP. Such stream handlers can be placed on the receiving or transmitting sides of an ANP, and they operate on application-level messages, thereby generalizing the receive-side MAC layer services developed elsewhere for the IXP [15]. The applications considered exhibit continuous data flows from one or multiple sources to a set of cluster nodes that process such data on behalf of end users. Node-based processing actions include business logic executed in OISs, data rendering for graphical display or visualization-specific computations like the computation of isosurfaces, and others. In contrast to and complementing such host-level actions, ANP-resident processing by stream handlers includes data striping across machines, data filtering and mirroring, and other 'lightweight' operations on both data content and headers.

In this paper's experimental evaluations, a single ANP (an Intel IXP1200 network processor) acts as a stream preprocessing engine for a set of cluster nodes that perform the higher-level processing required by each data stream. Applications can place onto the ANP stream handlers that operate on application-level data units, and such handlers can touch both data headers and data content, thereby permitting them to operate on data content in the same fashion as host nodes. Toward these ends, (1) we have developed an efficient, ANP-resident implementation of packet fragmentation and reassembly, and (2) we utilize low-level meta-information (i.e., the PBIO data formats described in [3]) to describe the format and layout of the data contained in application-level messages. ANP-level handlers are realized as microcode that operates on data streams when data is received, when application-level messages have been constructed in ANP memory, and/or when data is sent out.

Experimental results demonstrate that our approach achieves throughput rates of up to 99.97% of attainable network bandwidth (using multiple 100Mbps ethernet links on the IXP1200 board), even for larger numbers of streams or when processing the data contents of large messages. The benefits of transmit-side processing include the ability to implement more complex and resource-demanding

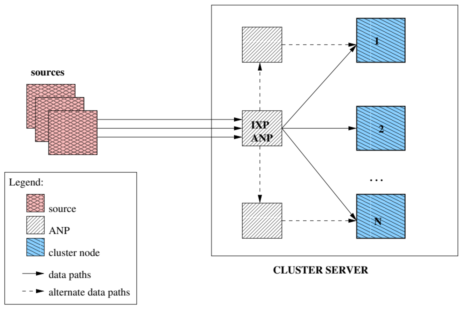

Figure 1. System Architecture.

stream handlers, and the attainment of potentially significant application-level performance gains by use of such handlers. For instance, for the specific handlers used in this paper, we demonstrate performance advantages of over 20% for transmit- vs. receive-side handler execution, even under medium processing and memory loads. This gap is expected to widen as these loads increase.

2. Customization Architecture: Design and Implementation

Attached Network Processors and Computational Clusters. We focus on applications that require cluster-based server solutions for their continuous data streams, an example being the Delta Airline OIS described in more detail in [6]. Here, multiple cluster nodes apply business logic to continuous input streams comprised of FAA flight position updates and Delta-specific flight information. The results of such processing are then sent to a large number of clients, who receive continuously derived system-state updates and/or responses to explicit update requests. The large number of clients, the complexity of the business logic being applied and its working set size of hundreds of gigabytes, and a 24/7 uptime requirement dictate a cluster-based solution to the stream processing done by a business server of this nature. Other examples of such cluster-based streaming servers include sensor processing, graphics and visualization servers, transaction processing engines, and 'fresh information' database servers.

We use the term 'attached' network processor (ANP), since the functionality of the network processor associated with each server cluster (or even with each machine in the server cluster) is strongly affected by the processing tasks undertaken by the cluster. In other words, for the sample Intel IXP1200 ANP acting as a cluster-attached network processor (see Figure 1), the data filtering and content-based routing or load-balancing tasks being performed are determined by the specific needs of the OIS implemented by the cluster server. The intent, of course, is to deliver exactly and only the application-level messages currently needed by each cluster machine, using application-level information about message formats and data layout. The implementation of application-level processing on the IXP-based ANP used in this paper represents all code run on the ANP as *stream handlers* defined by application programs and realized as microcode running on the IXP's micro-engines. The formats of application-level messages are described with the lightweight PBIO binary data formats [3], and they are made available to the ANP's handlers at the time an active network connection is established[1]. Application-level messages are assembled and operated upon in IXP memory, prior to being sent out to their destinations.

Intel IXP1200 Network Interconnect Boards. The Intel IXP1200 [10] is a multiprocessor on a chip containing a StrongArm (SA) core and six microengines with four thread contexts each (see Figure 2), all operating at 232MHz. Each microengine has a 2Kword instruction store loaded by the SA and shared among the four contexts. The chip integrates memory controllers for up to 256MB of SDRAM, 8MB of external SRAM and 4KB of internal scratch memory. Ethernet or other MACs connect externally through a proprietary 64-bit/66-MHz IX bus accessed through on-chip transmit and receive FIFOs. Our work targets the IXP1200 chip on Radisys ENP2505 boards, which have the maximum amount of memory and four 100T ethernet MACs. We expect to migrate to a version with two gigabit ethernet MACs in the near future.

Reliable Communication. Since our target applications require data transfers across a wide area network, an underlying reliable protocol is necessary in order to guarantee in-order, lossless communication. Initial experiences with TCP suggest that the protocol is too heavyweight to be efficiently implemented on the IXP1200 [15]. Therefore, we selected to implement RUDP [1], a reliable protocol built on top of the MAC layer's transfer of raw Ethernet IP frames. The IXP implementation of RUDP is split between the microengines and the SA, with the microengines' receive (Rx) and transmit (Tx) threads executing the protocol's fast path, and the SA handling special cases, such as connection setup and tear-down, exceptions, etc. The Rx threads receive the incoming data stream, one 64-byte MAC-layer packet at a time, classify the packets based on their header fields, and determine the memory addresses to which they must be copied. The Tx threads copy the data from memory, compute the appropriate header fields' updates, and transmit the data on the outgoing port(s). In addition, both threads share and maintain the per-stream state necessary for guaranteeing reliable communication. The same communication protocol is used by the data sources and the destination cluster nodes.

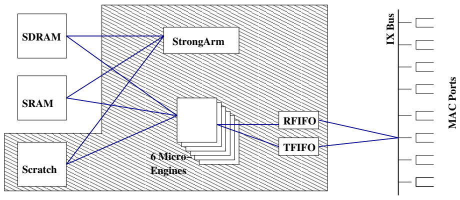

Figure 2. IXP1200 block diagram.

Stream Handlers: Design and Implementation. The RUDP protocol and its IXP implementation are not the focus of this paper. Instead, we evaluate how certain application-level functionality is best associated with the processing of RUDP-based or MAC-layer messages on the IXP, and we consider the performance tradeoffs or advantages that exist for such processing. As with other designs [5, 8, 15], our framework for stream handlers separates the data, the control, and the management planes. The data plane is where 99% of the packet processing occurs, and all such processing actions are executed by the 24 microengine contexts available on the IXP1200. Connection control and management, and the initialization of the runtime are left to the StrongArm. The basic implementation of stream handlers requires their application to application-level messages, whereas the ANP's hardware supports only the receipt and transmission of Ethernet packets handled as a sequence of 64B MAC-layer packets, with higher level protocol frames starting on Ethernet frame boundaries. Given these constraints, all data plane operations involve the following steps:

1. 64B MAC-packet is received;
2. packet is classified based on header information and/(or) a memory address for the application-level message is computed;
3. packet is copied into appropriate location in memory buffer and/or stream handler is applied to the packet (i.e., to the message fragment) – receive-side message processing;
4. if entire application-level packet is in memory, the stream handler is signaled;
5. handler processing on the application-level packet is performed, and if necessary, result is copied into memory (ANP-resident message processing);
6. application-level packet is transmitted as a sequence of 64B packets (transmit-side ANP message processing).

These steps are designated as separate tasks to different microengine threads. Receive threads (Rx) perform steps 1-4, i.e. buffer management and the receipt and classification of packets, and transmit threads (Tx) perform the necessary updates in the protocol header fields and transmit the

[1]Our current implementation uses hardcoded IXP-resident handlers and formats. We are now developing the general mechanisms for connection-based format and handler 'caching' on the IXP, where formats are available from attached hosts.

packets (step 6). Basically, the core functionality of the Rx and Tx threads is the execution of the RUDP fast-path code. For ANP-resident message processing, separate transform threads (X) execute the application-specific handlers, either by operating directly on the memory locations in which the Rx threads have placed the data, or by storing the resulting data in separate buffers, from which the Tx threads then send it out. The stream handler code executed by the X threads is written by the application programmer, and currently, issues such as security, resource management, etc., remain the programmer's responsibility. Explicit data format representations (see [3] for a detailed description of these efficient binary formats) are used to obtain the correct field offsets, and to access the stream data as necessary, in a similar manner as other approaches use fixed header definitions to support header-based processing.

For applications that require complex stream handlers, it is necessary to dedicate separate X threads to the execution of the handler's code. This is due to (1) the physical limitation of the instruction store associated with the microengines, and (2) the fact that a greater degree of concurrency on the critical path increases the sustainable rate of throughput. However, for most applications, message-processing code can be combined with the receive or the transmit code, resulting in a more scalable message processing architecture through improved threads utilization. Receive-side processing (RxX) is useful when the processing actions are based upon information contained in header fields, or in just a few MAC-layer packets, so that the number of accesses into the stream data unit, and the state maintained for each data unit, are as low as possible. Examples of Rx-side processing include header-based packet classification, stream differentiation or filtering based on small number of comparisons, etc. One such example is the software-based router described in [15]: the routing/discarding decision is computed based solely on the header contained in the first 64B-packet of each Ethernet frame, and the remaining packets are either directly placed on the appropriate transmitting queue, or simply discarded.

As the complexity of the stream handler and the amount of state information it requires increase, Rx-side processing is no longer feasible. An alternative to dedicating a separate thread to execute the stream handler is to combine the execution of the stream handler with the transmission process in transform-transmit threads (XTx). Such transmit-side processing is useful whenever all or most of the data unit needs to be received before the handler can be executed. It is particularly efficient when data can be modified as it is being sent out, which reduces the loads imposed on ANP memory by avoiding double-copying. Again, explicit format representations are used to access the desired portions of the stream data unit. Furthermore, such Tx-side execution of the stream handler, allows for efficient implementation of multicast customized on a per client basis. Finally, based on the application-specific data formats, further optimizations can be performed on the data path, in the sense that a handler can be invoked as soon as the necessary data fields are received. In this manner, potentially unnecessary memory accesses can be avoided, per application-level data unit fast-path execution can be sped up, and higher throughput can be achieved.

Current Implementation. We currently statically allocate stream handler computations to microengine threads. Therefore, we do not support the dynamic reconfiguration of the execution environment, where threads would switch between the different tasks of receiving and transmitting packets, or of executing application-specific handlers. Furthermore, handlers and format descriptions are created at system boot time, rather than when connections are established. When the application-specific processing can be combined with the transmit code, or when limited state updates are performed per application-level data unit, the IXP1200 is configured with 16 Rx/RxX threads and 8 XTx/Tx threads. For handlers that perform more extensive transformations on the original data stream, the IXP1200 is configured with 8 Rx, 8 X, and 8 Tx threads. While these configurations are adequate for the specific experiments conducted in this paper, in general, compiler-based solutions to such micro-engine resource management should be sought; we are currently planning such solutions.

3. Performance Evaluation

The initial experimental evaluation of stream handlers and their utility and performance is encouraging. Experiments described below use the IXP1200 simulation package SDK2.0, claimed to be cycle-accurate to within 2%, and the aforementioned Radisys boards. Data streams are generated using the simulator's network traffic generator, and each data item consists of few descriptors followed by a variable length array of integers. The stream handlers used in the experiments differ in their complexity, in the portion of the original data on which they operate, and in the amount of memory load they generate. Handler f1, the simple handler, performs a comparison only on the first descriptor in the data items, which is contained in the first 64B MAC-packet. Handler f2 is more complex, as it inspects every integer in the array and therefore, 'touches' almost every byte in the data item and generates a large number of memory accesses. When acting as filters, these handlers filter out 20% of the stream based on the values of the data they 'touch'.

The first set of experiments does not take any stream processing into account. It simply analyzes the scalability of our approach by looking at the achieved throughput as a function of application-level data sizes and the number of streams serviced (see top graph in Figure 3). The

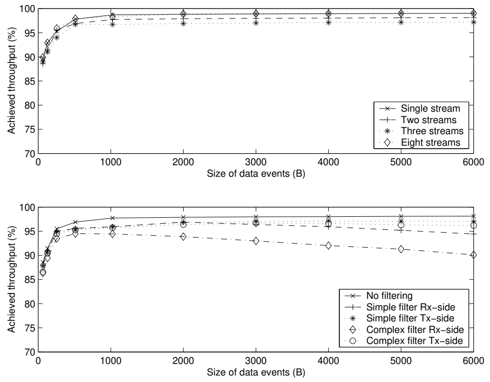

Figure 3. Top: Normalized achieved throughput for different numbers of incoming streams; Bottom: Performance impact of Tx- vs. Rx-side execution of two handlers with different complexity.

vertical axis represents the sustained throughput as a percentage of the cumulative incoming data rates. The experimental setup is such that the sources are delivering data to the IXP at maximum rates of 100Mbps each. From these measurements, we observe that even in the worst case when each stream consist of minimum size data packets of 64B, the basic implementation of stream handlers sustains high throughput rates. As the data sizes and the number of streams increase, the measured throughput reaches 99.97% of the cumulative incoming link rates, since the concurrency existing in the IXP architecture is exploited better, including overlap in threads' transmit and receive processing.

Table 1 presents the impact of adding stream handlers with different complexity on the achieved throughput. We observe that for larger messages split across many min-sized 64B packets, the combined amount of available cycles can be used, and more processing can be supported, with minimal degradation in the sustained throughput. Implementing the stream handler in a separate X thread allows for more complex transformations to be performed on the incoming data stream, but it increases memory load in the system, and in general, results in decreased scalability. In future work, we are performing a more detailed analysis of the possible handler instruction mixes and their effects on communication performance.

For certain applications, knowing the data format of the incoming stream allows us to invoke the stream handler as early as the necessary data fields become available in the IXP. Previous approaches, which implement additional functionality by operating on packet headers, exploit such Rx-side processing by executing the handler's code on each first 64B packet of a new Ethernet frame. The results in Table 1 demonstrate the feasibility of such Rx-side processing

\	null	f1 (Tx)	f1 (Rx)	f2 (Tx)	f2 (Rx)
64B	91.03	90.96	93.66	88.12	86.66
2048B	98.55	98.45	98.98	94.02	84.14

Table 1. Effect of handlers on throughput for different sizes of stream data items.

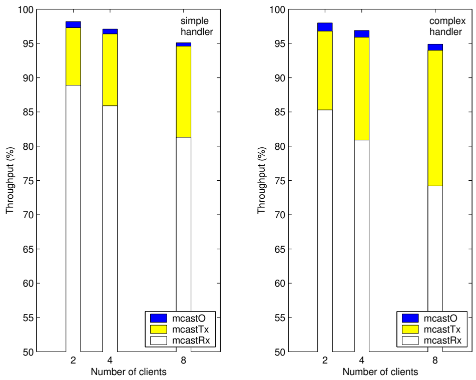

Figure 4. Client-customized multicast.

in our implementation (Rx columns), particularly for simple handlers, such as filtering stream handlers, which consist of few memory accesses and comparisons, and which require a limited amount of state to interpret the data format and maintain limited filtering parameters and state.

However, as the complexity of Rx-side handlers increases, or as the data structures in the incoming data stream become more complex, and/or as the evaluation of filtering parameters requires multiple accesses in the packet stream and the maintenance of additional state, a point is reached at which Rx-side filtering is no longer feasible. Specifically, Rx-side processing has limited use when the transformations of the incoming stream depend on values of data fields embedded in subsequent MAC-packets. The lower graph in Figure 3 demonstrates these limitations, by comparing runs in which the same filter is executed on the receive vs. on the transmit side. From these figures, it is apparent that Rx-side filtering is appropriate only for simple handlers, which operate on small portions of the application data. In comparison, high throughput levels are attained with Tx-side filtering, even for complex filters that operate on an entire data message. These limitations motivate our design in which stream handler processing is separated from packet receipt (Rx), and/or combined with message transmission (Tx).

The last set of experiments demonstrates how important it is to enable transmit-side processing for stream handlers, by evaluating an implementation of a per-client customized multicast. The graphs in Figure 4 represent the achieved throughput when the original stream is sent to n clients without any modifications (mcastO), when Rx-side handlers perform the n transformations needed to customize the single stream for each client and thereby, generate n customized copies of the original data (mcastRx), or when the Tx-side handlers simply customize the outgoing stream as it is being copied from memory and sent out (mcastTx). Note that such non-memory-intensive stream customization is possible only with Tx-side handler execution. Repeating this evaluation for two customizations that generate different levels of memory load, we find that the use of Tx-side stream handlers results in a per-client customizable multicast solution with efficiency within 10% of a non-customized multicast. The use of Rx-side handlers offering the same functionality, on the other hand, results in substantial degradation in the achieved throughput levels, offering performance that varies significantly based on the number of multicast clients.

4. Conclusion and Future Work

This paper presents *stream handlers*, which are lightweight, application-specific computations placed into network processors associated with cluster machines. The software architecture implementing stream handlers for *attached network processors* (ANPs) allows applications to place custom processing of headers and data content into an ANP, on both its receive and transmit sides. Experimental evaluations demonstrate that while receive side processing is useful in some cases, it becomes a bottleneck and results in degraded performance when the complexity of the application-specific handlers or the amount of data they 'touch' increase. At the same time, with transmit-side processing, we obtain more constant and acceptable performance levels, and can support richer functionality in the ANP.

We are in the process of moving our implementation to the IXP hardware, where we plan to continue to evaluate and further extend our design. Additional future work will further formalize the relationships between message sizes, handler complexity, and IXP resource management. In addition, we are automating format and handler placement onto the ANP, with the intent of presenting to applications a simple interface for 'programming' the ANPs associated with their machines.

References

[1] T. Bova and T. Krivoruchka. Reliable UDP Protocol - Internet Draft, Feb. 1999. draft-ietf-sigtran-reliable-udp-00.txt.

[2] F. Braun, J. Lockwood, and M. Waldvogel. Protocol wrappers for layered network packet processing in reconfigurable networks. *IEEE Micro*, 22(1):66–74, Jan./Feb. 2002.

[3] F. Bustamante, G. Eisenhauer, K. Schwan, and P. Widener. Efficient Wire Formats for High Performance Computing. In *Proc. of Supercomputing 2000*, Dallas, TX, Nov. 2000.

[4] F. Bustamante, G. Eisenhauer, P. Widener, K. Schwan, and C. Pu. Active Streams: An Approach to Adaptive Distributed Systems. In *Proc. of 8th Workshop HotOS-VIII*, Elmau/Oberbayern, Germany, May 2001.

[5] A. T. Campbell, S. Chou, M. E. Kounavis, V. D. Stachtos, and J. B. Vicente. NetBind: A Binding Tool for Constructing Data Paths in Network Processor-based Routers. In *Proc. of IEEE OPENARCH' 02*, New York City, NY, June 2002.

[6] A. Gavrilovska, K. Schwan, and V. Oleson. Practical Approach for Zero Downtime in an Operational Information System. In *Proc. of 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, Vienna, Austria, July 2002.

[7] I2O Special Interest Group. http://www.intelligent-io.com/.

[8] Intel Corporation. *Intel IXA SDA ACE Programming Framework*, 2001.

[9] C. Isert and K. Schwan. ACDS: Adapting Computational Data Streams for High Performance. In *Proc. of International Parallel and Distributed Processing Symposium (IPDPS)*, May 2000.

[10] Intel IXP1200 Network Processor Family. http://developer.intel.com/design/network/producs/ npfamily/ixp1200.htm.

[11] R. Krishnamurthy, K. Schwan, and M. Rosu. A Network Co-Processor-Based Approach to Scalable Media Streaming in Servers. In *Proc. of International Conference on Parallel Processing (ICPP)*, Aug. 2000.

[12] B. Plale and K. Schwan. Optimizations Enabled by Relational Data Model View to Querying Data Streams. In *Proc. of International Parallel and Distributed Processing Symposium (IPDPS)*, May 2001.

[13] M.-C. Rosu and K. Schwan. Support for Recoverable Memory in the Distributed Virtual Communication Machine. In *Proc. of International Parallel and Distributed Processing Symposium (IPDPS)*, May 2000.

[14] M. Sanders, M. Keaton, S. Bhattacharjee, K. Calvert, S. Zabele, and E. Zegura. Active Reliable Multicast on CANEs: A Case Study. In *Proc. of IEEE OpenArch 2001*, Anchorage, Alaska, Apr. 2001.

[15] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a Robust Software-Based Router Using Network Processors. In *Proc. of 18th SOSP'01*, Chateau Lake Louise, Banff, Canada, Oct. 2001.

[16] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. of ACM SIGCOMM 2001*, San Diego, CA, Aug. 2001.

[17] R. West and K. Schwan. Dynamic Window-Constrained Scheduling for Multimedia Applications. In *Proc. of 6th International Conference on Multimedia Computing and Systems (ICMCS'99)*, Florence, Italy, June 1999.

[18] K. Yocum and J. Chase. Payload Caching: High-Speed Data Forwarding for Network Intermediaries. In *Proc. of USENIX Technical Conference (USENIX'01)*, Boston, Massachusetts, June 2001.