# AN EMPIRICAL ANALYSIS OF THE IEEE-1394 SERIAL BUS PROTOCOL

THE STATISTICS COLLECTOR AND ANALYZER RECORDS, DISPLAYS, AND ANALYZES PERFORMANCE MEASUREMENTS FROM AN ACTIVE IEEE-1394 BUS IN REAL TIME. AN EMPIRICAL ANALYSIS USING SCA EXPOSES THE UNIQUE, COMPLEX ARBITRATION MECHANISMS USED BY IEEE-1394 NODES AND THEIR EFFECT ON THE PERFORMANCE OF HIGHER LEVEL PROTOCOLS.

Dan Steinberg
Yitzhak Birk
Technion - Israel Institute of Technology

Recently, we have begun to see a convergence of traditional network and bus technologies. The IEEE-1394 Serial Bus Protocol (FireWire)[1] is a good example of this convergence, targeting the interconnection of consumer electronics, computers, and peripheral devices.[2] IEEE 1394 provides for a high-speed, plug-and-play, peer-to-peer interconnect supporting both asynchronous and isochronous traffic.

In this work, we investigate the actual performance of various 1394 topologies in different configurations to gain a deeper insight into the efficiency, fairness, and robustness of the protocol. To this end, we have designed and implemented the Statistics Collector and Analyzer. SCA is a powerful traffic analyzer that is built on top of existing hardware called the CATC FireInspector.[3] We designed the SCA to record, display, and analyze performance of an active 1394 bus in real time.

To demonstrate SCA's usefulness, we investigate the ability of the protocol to interleave asynchronous and isochronous traffic. We also look closely at the arbitration mechanism for ensuring fairness, and identify and demonstrate cases where unfairness exists. In addition, we examine the effect that gap count optimization has on native and higher level protocols, and expose some of the subtleties of the 1394 specification, including possible effects of a gap count mismatch in a poorly managed topology.

## IEEE 1394

The IEEE-1394-1995 and 1394A specifications[4,5] detail a high-performance serial bus with many advanced features. IEEE 1394 targets the convergence of consumer electronics and high-speed computer peripherals. A 1394 topology can support up to 63 devices per bus. The 1394.1 specification currently defines bridges that will allow the interconnection of multiple buses, thereby creating even larger topologies. Data rates of 100, 200, and 400 Mbps are presently supported, with future plans for rates of 800, 1,600, and 3,200 Mbps. Device speeds are scalable, permitting a single topology to be composed of devices having different transfer rates.

A 1394 topology targets home use with plug-and-play automatic node identification and topology configuration. Devices and computers can easily be added or removed at any time with hot plugging. In addition, 1394 provides guaranteed latency and band-



0272-1732/00/$10.00 © 2000 IEEE

width for real-time applications such as high-resolution digital video and audio. It also supports split transactions, so that multiple request packets can be sent to one or more devices with response packets returned in any sequence. Finally, 1394 provides true peer-to-peer connectivity with no need for host intervention. For example, a 1394 scanner could send an image directly to a 1394 printer with no need for a host computer in the middle.

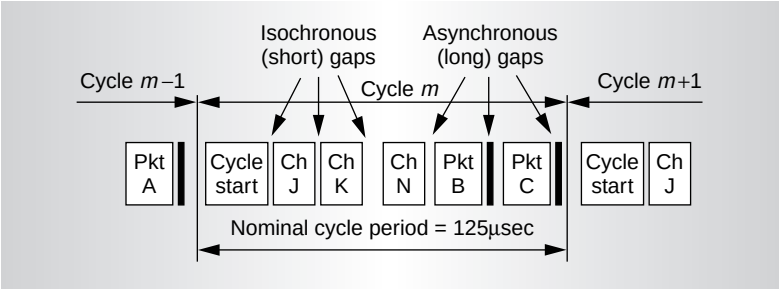


Figure 1. IEEE 1394 cycle structure. Ch: channel.

### Topology configuration

The physical topology of a 1394 network consists of up to 63 devices connected by serial cables with no loops allowed. The protocol implements a scheme for automatic configuration of the logical topology. The configuration process for a 1394 topology consists of several stages: bus initialization, tree identification, and self-identification.

Bus initialization or bus reset primarily results from power-on initialization, device connection or removal, or software initiation. Following a bus reset, all nodes on the bus initiate the tree identification process, which results in the determination of a root node, branch nodes, and leaf nodes in a nondeterministic manner. Loops are not allowed. Their use results in a nonfunctioning bus until the loop connection is removed.

Upon completion of the tree identification phase, the root node initiates the deterministic process of self-identification that assigns a unique node ID to each device in the topology and determines the isochronous resource manager or bus manager. This node ID becomes a source or destination identifier for all ensuing packets transferred on the bus. Every time the physical topology changes or software initiates a bus reset, the selection of the root node and some or all of the node IDs may change.

To cope with dynamically changing node IDs, 1394 devices must implement a core group of control and status registers defined by the CSR Architecture Specification.[6] This specification details locations of various registers including a 24-bit vendor ID and a 64-bit unique node ID. By accessing the CSR registers, devices can match a node's 6-bit ID assigned after a bus reset with the device's unique 64-bit node ID that does not change.

### Normal arbitration

Arguably, the most interesting aspect of the 1394 protocol is its unique arbitration mechanism. Once the topology configuration is complete, normal arbitration begins. In general, time on the bus is divided into 125 microsecond cycles, as depicted in Figure 1. Nominally, the root node broadcasts a cycle start packet at the beginning of each cycle to indicate to all nodes that a new cycle has begun. Nodes that wish to transmit isochronous data must reserve both an isochronous channel and the desired bandwidth. According to the specification, up to 80% of the total bandwidth for each cycle may be allocated for isochronous traffic. Once all of the isochronous traffic is complete, the rest of the cycle may be used for asynchronous transmission.

Nodes that wish to transmit asynchronous packets arbitrate for the bus by signaling requests up toward the root and signaling a denial to other neighboring nodes farther from the root. These requests work their way up to the root node that grants the bus to the first request that reaches it and signals a denial to nodes located on other branches of the tree. The node that wins arbitration then transmits the packet. The node that receives the packet will immediately respond to the incoming packet with an acknowledge code, and then the arbitration process may begin again until the cycle completes.

The arbitration mechanism in 1394 defines a fairness interval for asynchronous traffic that is meant to ensure that all nodes receive an equal opportunity to access the bus and avoid starvation situations. Arbitration between successive packets within a fairness interval is separated by a minimum amount of idle time called a subaction gap. This idle time is an indication that the previously transmitted

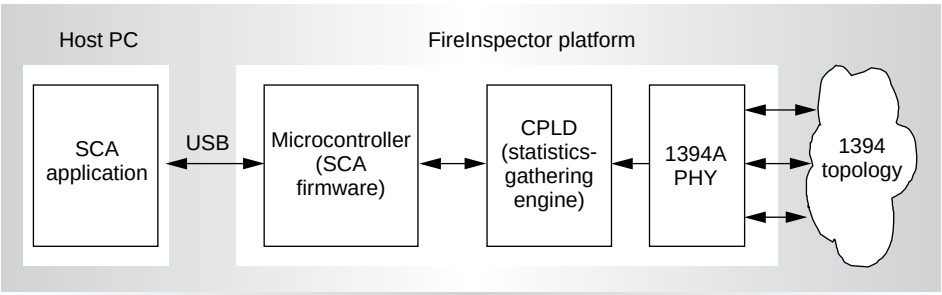


Figure 2. SCA system architecture.

packet is complete and nodes may begin to arbitrate without fear of corrupting the packet data. Each node is allowed to transmit exactly one asynchronous packet per fairness interval. The order of the transmitted packets will tend to favor nodes that are closer to the root. Once a node has transmitted its asynchronous packet, it must wait until all other nodes that wish to transmit complete their transmission and an arbitration reset gap is detected. This gap, consisting of a considerably larger amount of idle time than a subaction gap, indicates that the current fairness interval has ended, and a new one has begun. In theory, this mechanism should ensure equal access to the channel, but we will see that in certain situations "unfairness" exists.

### Gap count optimization

The gap count is a parameter that indicates the maximum propagation delay in the topology and determines the corresponding lengths of both subaction and arbitration reset gaps. This parameter is maintained individually by each node, but ideally should be equal for all nodes on the bus. The gap count parameter takes on a value in the range of 0 to 63, defaulting to its maximum value. As the value of the gap count increases, the detection time for the various types of gaps increases according to formulas listed in the 1394-1995 specification. The protocol provides a mechanism for the bus manager to optimize the gap count at all nodes. Optimization is performed by selecting the minimal value possible based upon the size and configuration of the current topology. A smaller gap count value reduces the size of the gaps detected between packets and can potentially improve overall performance. The gap count may be optimized based upon the maximum number of hops in the topology. In a later section, we investigate the effect that gap count optimization has on higher level protocols such as the Serial Bus Protocol 2 (SBP-2) and the Internet protocol over Ethernet over 1394. Also, we will investigate what can happen when the gap count parameter is not identical at all nodes.

## SCA architecture

As depicted in Figure 2, the SCA runs on top of the FireInspector hardware platform. It is composed of a statistics-gathering engine, embedded SCA firmware, and the SCA application. The FireInspector platform has a 1394A compliant PHY, a high-density CPLD, and an embedded microcontroller. We loaded the CPLD with our statistics-gathering engine, which implements the real-time event decoding and counting logic. We loaded the microcontroller with our SCA firmware, which configures and periodically samples the counters inside the statistics-gathering engine. The sample data is then transmitted over a USB channel to our SCA application running on a host PC. The SCA application can configure, record, play back, and analyze 1394 performance data in real time. To create many of the desired test elements, we needed a controlled traffic generator that is both powerful and flexible. To this end, we developed the Traffic application that also uses the FireInspector platform.

### Statistics-gathering engine

Perhaps the most challenging portion of the overall SCA design is the real-time decoding and analysis of the 1394 traffic within the statistics-gathering engine. The statistics-gathering engine decodes the incoming 1394 traffic data from the PHY, which is an active participant in the 1394 topology. The SCA does not initiate the transmission of packet data, but the PHY does take part in the normal tree identification, self-identification, and arbitration processes. Ideally, an analyzer should be completely passive, but this minimal gain does not warrant the greatly increased complexity of building custom hardware to do so.

Some of the biggest challenges came in trying to optimize the logic resources used within the CPLD and to meet speed requirements. The number of gates and logic elements inside the CPLD is finite, and our desires to add features were unbounded. Currently, we have used 80% of the logic available, which translates to about 80,000 logic gates. In addition, we had to incorporate complicated pipelining techniques into the design of the statistics-gathering engine to meet the 50-MHz timing requirements for register-to-register performance. The statistics-gathering engine can essentially be thought of as an application-specific link layer IC.

The inputs to the statistics-gathering engine are the control and data lines of the PHY-link interface, as specified in 1394A. Depending upon the packet transmission rate used on the cable, the PHY converts the high-speed serial data into two, four, or eight parallel data streams, each operating at 49.152 MHz. These streams are sent to the PHY decode logic, which converts them into status words, data quadlets, and acknowledge octets. Hardware counters keep track of basic events such as the total number of quadlets and acknowledgments. Event-detection logic performs a more in-depth analysis of the data and status received. The incoming quadlets are sorted into packets consisting of header quadlets and data quadlets. Counters record the number and type of packets. Status words are decoded into bus resets, subaction gaps, and arbitration reset gaps that increment respective counters. In addition to regular traffic, error conditions—such as bad header and data CRCs, bad ACK parity, PHY interrupts, reserved transaction codes, and other data—are recorded. Finally, user-configurable counters keep track of 1394 traffic for specific source-destination node pairs.

To make an intelligent analysis of the traffic on the 1394 bus, we must have an up-to-date view of the logical topology. In this vein, upon detection of a bus reset, the statistics-gathering engine saves and counts the self-ID packets in an internal memory block. The microcontroller periodically extracts the data and sends it to the application. The application performs the complex task of accurately extracting, re-creating, and displaying the current topology from the self-ID packets. Knowledge of the current topology proves to be an extremely useful feature when collecting performance data.

### SCA firmware

The firmware can configure and collect counter data from the statistics-gathering engine and pass them to the application over the USB channel. The application initiates sample collection at a rate that may be configured within the application. Currently, the polling rate may be as fast as 50 times per second. To achieve this rate, the firmware sends data samples in bulk transfers of the maximum USB packet size. Along with each set of data samples, the firmware reads an accurate time stamp from the statistics-gathering engine and adds it to the status information regarding the current node ID of the SCA.

### SCA application

The SCA application primarily provides the user interface to configure and record statistical performance data from the active 1394 topology under test. The application, consisting of over 5,000 lines of C/C++ code, is implemented with multiple threads to provide for real-time data capture, analysis, and graphical display. Currently, the application tracks over 50 different statistics per sample. They include the overall bus throughput and utilization, throughput for specific source and destination nodes, error rates, breakdown of different packet types and speeds, and monitoring of specific isochronous channels.

Although data speeds over 1394 can approach 400 Mbps, the hardware analysis and data reduction performed in the statistics-gathering engine allow the application to record data for hours or days with minimal storage requirements. In addition, the up-to-date bus topology is displayed, including the tree configuration, SCA location, maximum speed capability, and gap count for each node as well as which nodes contended to be bus manager. Every time a bus reset occurs, the new topology is drawn.

Once the performance data is recorded, the SCA application can perform an in-depth data file analysis, tabulating totals, and detect trends that may not have been noticeable during the recording. The application also has the ability to open previously recorded files and

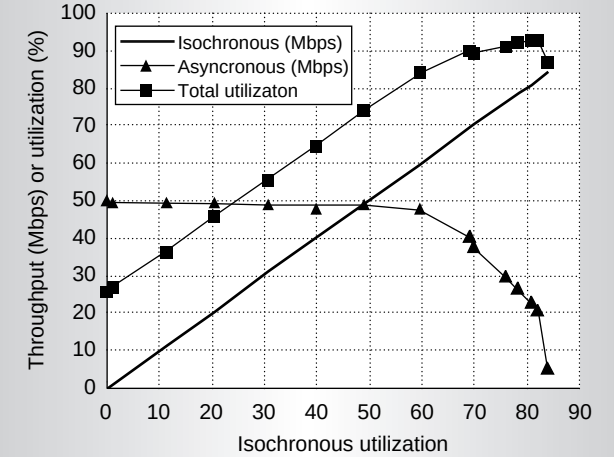


Figure 3. Effect of isochronous traffic on asynchronous traffic.

play them back, re-creating the graphical display and analysis.

### Controlled traffic generator

We developed the controlled traffic generator to generate configurable and repeatable traffic stimuli as well as help in configuring the topology. The generator consists of a dialog application that interfaces to a FireInspector hardware unit. To generate packet traffic with low latency and high bandwidth, the packet generator must work in hardware with the application responsible only for configuration. Special support for this capability is built into the existing SCA firmware, which is downloaded into the FireInspector prior to generation.

The application allows the user to configure the type, length, speed, and destination of the packet to be generated. The desired packet may be sent continuously, periodically, or in bursts. In addition, an interactive mode enables two traffic generators to communicate using a simple stop-and-wait protocol complete with acknowledgments. Finally, the traffic generator may initiate bus resets, force a specific node to become a root, and change the gap count value.

## Performance analysis

The SCA lets us collect meaningful statistics that will provide insight into various performance aspects of the 1394 protocol.

### Mixing asynchronous and isochronous

The first group of test measurements focuses on how well the protocol supports interleaving isochronous and asynchronous traffic. Specifically, we verify that as the bus saturates, only the nodes that wish to transmit asynchronous traffic are affected and isochronous traffic proceeds smoothly.

In the first experiment, the setup consisted of the following nodes: two host controller cards for isochronous traffic generation, two Ethernet emulations over 1394 (FireNet 200) cards for asynchronous traffic generation, and an SCA for performance monitoring. Substantial asynchronous traffic was generated using the NetTraffic test application provided by Unibrain. This application performs repeated file transfers between two host PCs using the Internet protocol over Ethernet over the 1394.

The offered asynchronous traffic remained constant at about 50 Mbps, or 25% utilization of the bus, and data points were collected as the offered isochronous load was increased. Each asynchronous packet was transmitted at a bus rate of 200 Mbps, whereas each isochronous packet transmitted at a 100-Mbps bus rate. We increased the lengths of the isochronous packets with each data point, resulting in a larger isochronous bandwidth. Figure 3 plots the results. The curve for the isochronous traffic is linear, indicating that it was unaffected by the background asynchronous traffic. The asynchronous throughput was also relatively unaffected by the real-time traffic until the total bus utilization increased above 85%. As the bus reaches saturation, the isochronous data continued without incident, but the asynchronous throughput decreased rapidly. Note that even with more than 80% of the cycle allocated for real-time data, the asynchronous file transfers were able to complete successfully.

We conducted a second experiment with a different offered asynchronous load to verify that the attainable utilization does not depend upon the working point. The offered isochronous load was the same as before, but this time we used two controlled traffic generators to generate the asynchronous traffic. Each generator was composed of a FireInspector unit running our Traffic application, which simply sends a fixed-length packet repeatedly to a specified node. The selected packet length and speed correspond to roughly 168 Mbps,

or 42% bus use. Again, we collected data points as the offered isochronous load was increased. Our results indicate that the isochronous throughput was linear and unaffected by the asynchronous traffic. The asynchronous traffic, however, began to decrease as the bus saturated, and overall bus utilization rose above 80%.

The primary conclusion that can be drawn from these tests is that asynchronous and real-time traffic can be interleaved successfully in 1394. As long as the overall bus use stays below 80%, both traffic types are virtually unaffected by each other. Above 80% use, however, the asynchronous throughput is dramatically affected whereas the isochronous traffic continues to flow smoothly.

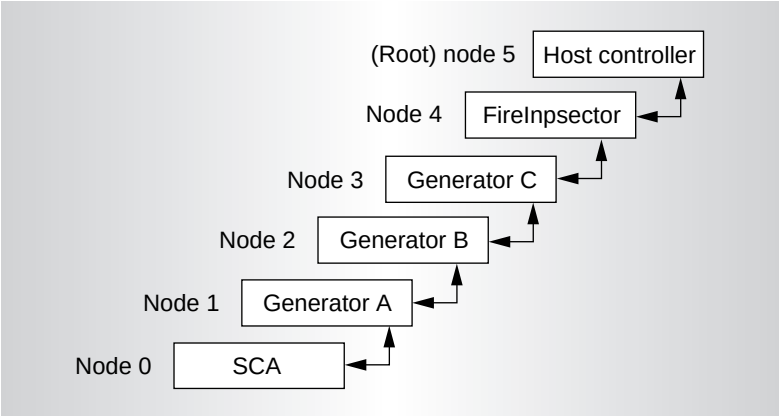

Figure 4. Topology for fairness tests.

### Fairness protocol

As stated previously, the 1394 arbitration mechanism uses the concept of a fairness interval to ensure that nodes on the bus wishing to transmit asynchronous packets will each have equal opportunities to access the bus. While conducting the experiments, we noticed a specific situation in which both controlled traffic generators did not seem to gain fair access to the channel, and began to investigate further.

Upon close examination of the arbitration protocol, we clearly see that cases of unfairness exist. The problem stems from a relatively minor detail in the specification. The result is that nodes farther from the root to, on average, have a longer latency from the time they wish to transmit until the time they are granted bus ownership. At first glance this scheme seems fair since the longer initial arbitration time should be offset by each node's guaranteed ability to transmit one asynchronous packet during each fairness interval. In some cases, a transmitting node may be modeled as a system with a fixed amount of processing time that begins upon completion of a packet transmission and ends when the next packet is ready to be sent to the PHY. If this is the case, two identical nodes located at different distances from the root will have unequal success transmitting if the total time required by each node to process, arbitrate, and send a packet is different.

To verify this, we created the topology shown in Figure 4. It is composed of the SCA, three controlled traffic generators, a FireInspector, and a host controller. We used generator B and the host controller to generate background asynchronous and isochronous traffic. Nodes A and C generated bursts of asynchronous packets separated by a fixed amount of processing time. Furthermore, they were both running identical firmware code with identical parameters. The only difference among them was that generator C was physically located closer to the root node. We configured the SCA to monitor performance of generators A and C in particular and the overall bus in general.

To measure relative unfairness, we defined the following:

- $Q_A$ equals the total quadlets transmitted from node A;
- $Q_C$ equals the total quadlets transmitted from node C;
- $R$ equals the percentage of relative unfairness; and
- $R = | Q_C - Q_A | / Q_A * 100$.

After adjusting the various parameters, we attained a value for $R$ as high as 19%. In other words, node C achieved 19% higher throughput generating asynchronous packets than node A even though they are identical. To verify that this phenomenon occurs with real-world applications and not just in synthetic tests, we created a similar setup, replacing the controlled traffic generators with FireNet cards. We measured performance for two topologies that differed only in the location of the FireNet cards. Again, we calculated the rel-

ative unfairness, yielding a result slightly greater than 1%. Although this difference is smaller than the synthetic tests, it does demonstrate that this minor flaw in the 1394 protocol will affect even real-world applications. To minimize or avoid this phenomenon altogether, 1394 devices must perform packet processing and transmission functions independently, and not remain idle while waiting to access the bus.

### Gap count effect on performance

The gap count parameter is an essential part of the arbitration mechanisms in 1394, and we investigated its effect on performance. In the first set of tests, we looked at the effect on higher level protocols such as SBP-2 and the Internet protocol over 1394. In a configuration with two FireNet cards and the SCA, we varied the gap count over the full range of possible values and noted that the throughput between the FireNet cards remains unchanged.

We ran a similar test with a host controller and an SBP-2 disk drive. Again, the gap count value varied over the full range, and the asynchronous throughput from the drive to the host did not change. In both cases, delays between back-to-back packets are shorter when the gap count is smaller, but significant processing delays remained the same yielding no overall improvement. The conclusion here is that the bottleneck is in the processing of packets at the host and not in the 1394 channel. For this reason, gap count optimization may be unnecessary in many topologies.

A set of tests using the controlled traffic generator as a source and varying the gap count resulted in an almost linear increase in performance as we decreased the gap count. These results indicate that the optimization of the gap count has a severe impact on native or lean protocols with low latencies.

In a bus that is poorly managed or unmanaged, a gap count mismatch may occur. In practice, we have observed this situation quite frequently, and decided to investigate what may happen when nodes disagree on the value of this global parameter. We set up a topology with two controlled traffic generators and the SCA. Generator B had a gap count configured to a value of 4, which is the minimum value recommended for a topology with two hops. We connected it to the SCA and generator A, which both had the maximum gap count value of 63. When we configured only generator A to generate packets continuously at maximal speed, the result was a throughput of 119 Mbps. Generator B alone generated a throughput of 188 Mbps. This difference indicates that the gap count will indeed have a significant effect upon native 1394 protocols with little or no latency.

When both generators attempt to transmit identical packets continuously, the combined throughput reaches 189 Mbps. Using the source-destination match feature of the SCA, we could see that the breakdown of the traffic among the generating nodes greatly favored the node with the smaller gap count. Specifically, node A only succeeded in transmitting 36,206 quadlets in the same amount of time that node B succeeded in transmitting 3,095,201 quadlets. The resulting ratio favored node B by more than 85:1! Note that this ratio is even higher than the gap count ratio of 63:4, which translates roughly to 16:1.

The explanation for this result is that node B, in general, will begin arbitrating for the channel very soon after transmitting a packet, while node A waits. Node A will time a longer gap before it recognizes its right to arbitrate. When node A finally does arbitrate for packet transmission, it will likely already have lost the arbitration and have to wait for the next gap. In summary, a mismatch in the gap count parameter can, under certain circumstances, have a dramatic effect on the arbitration mechanisms, resulting in unfairness or even starvation.

Experimental data indicates that the 1394 channel is successful in interleaving both isochronous and asynchronous traffic simultaneously until the bus saturates. The fairness protocol for asynchronous packet transmission, however, has certain flaws may result in unequal access to the bus. The gap count parameter can play a significant role in overall system performance, especially when a bus is poorly managed. Two identical tested nodes that disagree about the value of the gap count had unequal success in their asynchronous arbitration attempts. The node with the smaller gap count was approximately 85 times more successful than the node with the larger gap count! MICRO

## References

1. D. Anderson, *FireWire System Architecture,* Addison Wesley, Reading, Mass., 1999.
2. Unibrain, Inc.; http://www.unibrain.com.
3. Computer Access Technology Corporation; http://www.catc.com (10 Jan. 2000).
4. *IEEE Std. 1394-1995, IEEE Standard for a High Performance Serial Bus,* IEEE Press, Piscataway, N.J. Aug. 1996.
5. *IEEE Draft Standard for a High Performance Serial Bus (Supplement), P1394a Draft 2.0,* IEEE Press, Mar. 1998.
6. *IEEE Std. 1212, Control and Status Register (CSR) Architecture for Microcomputer Buses,* IEEE Press, Oct. 1994.



Dan Steinberg is currently employed by Integrated Device Technology (IDT) in Silicon Valley, where he is designing an advanced network processor (system on a chip). He recently received his MSc in electrical engineering at the Technion University in Haifa, Israel, where he participated in the work in this article. He completed his research in the area of high-speed serial interconnects, specifically exploring the performance of the IEEE-1394 protocol. Steinberg received his BSc in electrical engineering from the University of Maryland. In between studies, he worked as a design engineer at Stanford Telecom, where he designed both hardware and software, integrating cellular phone, global positioning system (GPS), and car alarm technologies.

Yitzhak Birk is on the faculty of the Technion's Electrical Engineering Department and heads its Parallel Systems Laboratory. Previously, he was a research staff member at IBM's Almaden Research Center. He also consulted to Hewlett-Packard Labs in the areas of storage systems and video servers. His research interests include computer systems and communication networks. Much of his recent work is on communication-intensive storage systems such as stream servers, as well as the judicious exploitation of redundancy for performance enhancement. Birk received the BSc and MSc degrees from the Technion and the PhD degree from Stanford University, all in electrical engineering. He is a member of the IEEE Computer and Communications societies.

Direct comments about this article to Dan Steinberg at DanSteinberg@attglobal.net.