Multi-Kilohertz Control of Multiple Robots via IEEE-1394 (Firewire)

Zihan Chen[1] and Peter Kazanzides[1]

Abstract—**We present a specialized FireWire protocol that takes advantage of broadcast messages and peer-to-peer transfers to minimize the number of transactions, and thus the software overhead, on the control PC, thereby enabling fast real-time control. We provide an open source Verilog implementation of a Link Layer Controller (LLC) that supports this design on an FPGA-based motor controller. Performance is measured on a da Vinci® Research Kit that contains 8 of these controllers to drive 28 axes. Compared to a conventional asynchronous transfer-based solution, this protocol decreases the I/O time by more than a factor of 4. This performance gain can be used to increase the control frequency to 6kHz, scale to a larger number of axes, or provide greater tolerance to timing variations due to a non-real-time operating system, such as the standard Linux kernel.**

I. Introduction

Robot control can be divided into three main actions: (1) getting data (input), (2) processing data (computation) and (3) delivering data (output); this implies two fundamental needs: I/O and computation. It is often implemented as a hierarchical multi-rate control architecture, where a (possibly distributed) low-level controller implements high-frequency control of the position, velocity, or torque of individual joints. There are one or more higher levels of control that introduce more sophisticated capabilities and ultimately produce a stream of setpoints to the low-level controller. This paper focuses on the I/O performance between the low-level controller and the electronics that interface to the physical world.

The fastest I/O is achieved when these devices are interfaced directly with the microprocessor that performs the computation. Thus, it is common to find a *distributed computation and I/O* architecture (Fig. 1a), where the low-level control is performed on an embedded system that contains a microprocessor and I/O devices. The high-level control is typically performed on a conventional computation platform, such as a PC, that is connected to one or more embedded controllers via a high-speed network, such as Ethernet, USB, IEEE-1394, or CAN. An alternative approach is to perform all control computations on a PC, often with a real-time operating system. In this case, one can adopt a *centralized computation and I/O* architecture (Fig. 1b), where the electrical interfaces are provided by I/O boards that are installed inside the PC and interface via its high-speed internal bus (originally ISA, now PCI or PCIe). One advantage of this architecture is that the entire control system can be implemented on a familiar development platform (PC), rather than requiring expertise in embedded systems programming. The disadvantage, however, is that a significant amount of cabling is required to connect the robot sensors and actuators to the electrical interfaces inside the PC. This can reduce reliability, increase signal noise, and make reconfiguration difficult, especially if it is necessary to open the PC chassis.

*This work was supported by NASA NNX10AD17A and NSF NRI 1208540

[1]Zihan Chen and Peter Kazanzides are with the Dept. of Computer Science, Johns Hopkins University, Baltimore, MD, USA. P. Kazanzides can be reached at pkaz@jhu.edu.

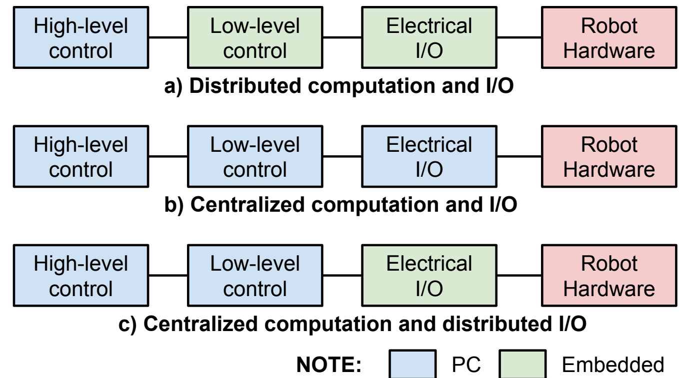


Fig. 1. Three common architectures for robot control. This paper focuses on *centralized computation and distributed I/O* using IEEE-1394 (FireWire) between the low-level control and electrical I/O.

While *distributed computation and I/O* remains an excellent choice, within the research community there is increased interest in a *centralized computation and distributed I/O* architecture (Fig. 1c) because it provides the benefits of a familiar development environment (all control is implemented on the PC) as well as robust cabling (I/O is performed by hardware connected to the PC via a high-speed network) [1]. Because the network is inside the low-level control loop, it must provide high bandwidth and low latency. This has been difficult to achieve with conventional networks such as Ethernet and USB, so in 2006 we began to investigate alternatives and selected IEEE-1394 (FireWire). In 2008, we reported results from our first FireWire-based design, which included an FPGA implementation of the link layer[1]. We note that others have developed FireWire-based protocols for specific application areas. For example, SAE AS5643 [2] is designed for military and avionic applications and focuses on bus determinism and safety features, rather than high update rates (its typical update rate is around 100 Hz). Today, the *centralized computation and distributed I/O* architecture is most commonly implemented using EtherCAT [3], which was first demonstrated in 2003. We did not select EtherCAT because, at that time (2006-2008), it was not clear whether it would be widely accepted (considering also competing efforts, such as PowerLink) and we determined that it was much easier to implement the FireWire link layer in an FPGA, especially given the ample documentation [4].

We contend, however, that IEEE-1394 remains a viable alternative to EtherCAT and, in this paper, introduce a communication protocol that provides similar performance. We demonstrate this in an “open source mechatronics” system, where we achieve a closed-loop control rate of 6 kHz for the 28 axes of a da Vinci Research Kit [5] (see Fig. 2). We further highlight some of the relative advantages of disadvantages of IEEE-1394 and EtherCAT.

978-1-4799-4605-1/14/$31.00 ©2014 IEEE

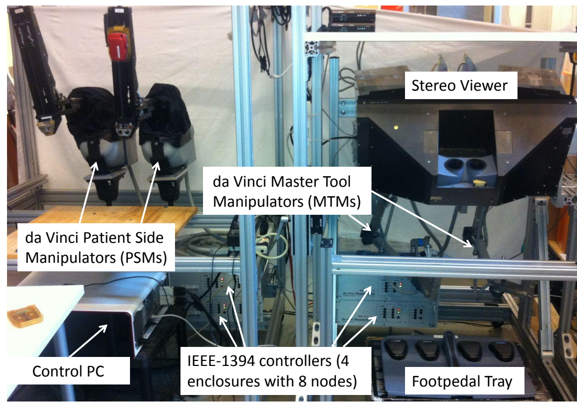


Fig. 2. Research da Vinci System at JHU: two 7-dof Master Tool Manipulators (MTMs) and two 7-dof Patient Side Manipulators (PSMs), for a total of 28 axes, controlled by eight IEEE-1394 nodes (packaged in 4 enclosures), each consisting of an IEEE-1394 FPGA board mated with a Quad Linear Amplifier (QLA).

II. SYSTEM OVERVIEW

This section gives an overview of the system architecture, reviews the FireWire transaction types, analyzes their performance, and shows the performance bottleneck that can result from scaling to a robotic system with 28 axes.

A. Open Source Mechatronics

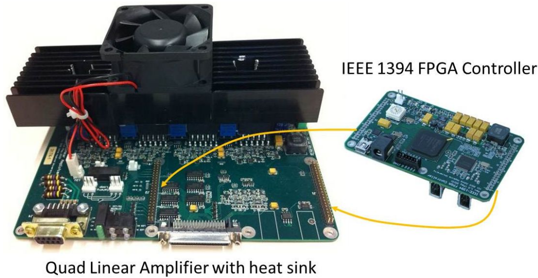


Fig. 3. IEEE-1394 FPGA board and Quad Linear Amplifier (QLA)

Our FireWire-based controller, shown in Fig. 3, is a complete open source design, with the schematics, layout, and FPGA firmware (Verilog) available at http://jhu-cisst.github.io/mechatronics. The controller consists of two boards, an IEEE-1394 FPGA board and a Quad Linear Amplifier (QLA), that are mated via two 44-pin connectors. Most of the 88 signals are connected directly to I/O pins on the Xilinx Spartan-6 XC6SLX45 FPGA; the rest are used for power (+3.3V, +5V) and ground. This design allows researchers to create alternate I/O boards (to replace the QLA) to satisfy different hardware requirements, or to design a new FPGA board to introduce a different communication network. This is a general-purpose mechatronics system, but currently its primary application is to control research systems based on the mechanical components of the first-generation da Vinci® Surgical System [6], [5], as shown in Fig. 2. The low-level control software is implemented on a Linux PC, which is connected via a daisy-chain to several FPGA-QLA board sets, as illustrated in Fig. 4. This controller has been replicated at 10 institutions, producing a research community around a common hardware and software platform.

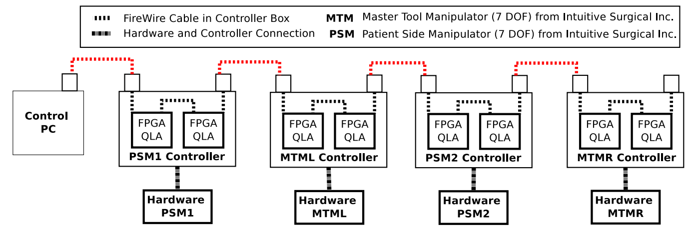


Fig. 4. Hardware architecture: one control PC and 8 IEEE-1394 FPGA/QLA board sets controlling the 4 da Vinci manipulators (7 DOF each).

B. Introduction to IEEE 1394

The IEEE-1394 interface is [7] a high-speed peer-to-peer, full-duplex fieldbus with low overhead that is well suited for real-time control applications. It is a Control and Status Register (CSR) architecture with a tree-like topology that supports up to 64 nodes on a single bus. The IEEE-1394a physical medium transmits data at a speed up to 400 Mbits/sec. In later specifications (IEEE-1394b), the bus can support data transfers of 800 Mbits/sec and even up to 3.2 Gbits/sec.

FireWire supports two types of transactions: asynchronous and isochronous. It operates based on a 125 μs bus cycle (8 kHz), which is triggered by a cycle start packet followed by an isochronous period and then an asynchronous period. An isochronous transaction running at 8 kHz has a reserved bandwidth, and can only happen within the isochronous period. It uses a channel number to address its target nodes and requires no acknowledgement or response packet. Despite its high frequency, isochronous transfer has no guarantee of data delivery and can suffer from cycle start packet time drifting. This makes it a natural choice for video and audio streaming applications, rather than for real time control. Asynchronous transactions can only occur in the asynchronous phase after isochronous transactions have completed. Unlike the isochronous transactions, asynchronous transactions use a 64-bit address for data transfer. The whole FireWire bus network can be mapped into the 64-bit address space, with 10 bits for the bus number, 6 bits for the node number and 48 bits for the node address. An asynchronous transaction is designed to be error free by requiring an acknowledgment packet for each data transmission and a response packet for every asynchronous request. Often it is split into two subtransactions: a request and a response, allowing other transmissions in between. Asynchronous transactions are typically used for control commands and reliable message transmission.

For our control applications, we chose to use asynchronous transmissions, initiated by the PC, to fetch and send data from and to the FPGA boards. Furthermore, for efficiency considerations, asynchronous transactions are implemented as concatenated transactions, where the acknowlegment packet and response packet (if a read transaction) are sent back to the requesting node without releasing the bus. Compared to split transactions, this eliminates the need for the responding node to wait for the subaction gap (at least 5 to 10 μs) and negotiate for bus access. This design has been implemented, tested and used for several robot systems, including a snake-like robot [8] and the da Vinci Research Kit [6], [5]. The following section presents experimental data to assess the performance of the FireWire transaction types, which guides the design of a protocol to achieve higher control performance.

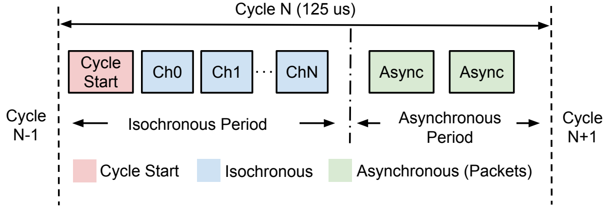


Fig. 5. IEEE-1394 cycle with isochronous and asynchronous transactions

C. System Performance

This subsection presents the performance measurement of concatenated asynchronous read/write transactions, analyzes I/O versus computation ratio in a servo control loop, and reveals the bottleneck of achieving better control timing performance. All the data is collected on a Linux PC (FireWire chip *Ricoh R5C832*) with a *3.2.0-49-generic* kernel, *Juju* FireWire driver stack, and *libraw1394* API library. Timing data is queried using the *gettimeofday* function.

Fig. 6 shows the time required for asynchronous block read and write transactions initiated by the PC software, each based on 5,000 iterations. The read and write payload sizes are 48 and 16 bytes, respectively, which match the payloads used for the FPGA-QLA board set. Mean read and write times are 31.99 and 33.74 μs, with standard deviations of 12.02 and 8.56 μs, respectively. Thanks to our concatenated implementation, the measured data is only half the value (around 60 μs, depending on the kernel) reported in [9].

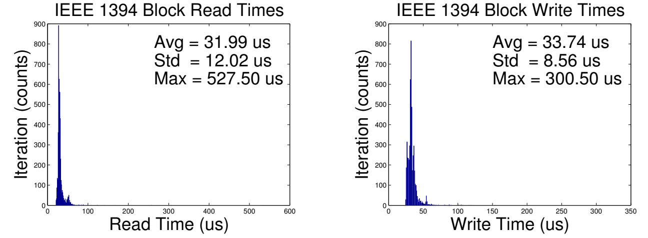
The most straightforward protocol is to perform one asynchronous read (to obtain feedback data) and one asynchronous write (to send control output) to each FPGA board in each servo control loop. In this case, the total mean I/O time for a robot system with N_{boards} FPGA boards would be:

$$T_{I/O} = (T_r + T_w) \times N_{boards}, \tag{1}$$

where T_r and T_w are the mean asynchronous read and write times, respectively. For a da Vinci Research Kit with eight FPGA boards, the computed I/O time cost is $(32.22+34.13) \times 8 = 531.12$ μs. This number is consistent with data collected experimentally, which has a mean time of 495.98 μs and standard deviation of 75.45 μs. Because servo loop computation time T_C is very low (less than 40 μs for an 8-board system), I/O time often takes over 90% of the minimum control period $(T_c + T_{I/O})$ and more than 50% of a 1 kHz control loop. This means that I/O performance is the bottleneck and would make it difficult to: (1) control a more advanced system (e.g., a system with 16 FPGA boards) with a 1 kHz servo loop, or (2) control an 8 board da Vinci system at frequencies greater than about 1.8 kHz.

D. Analysis

Figure 7 shows each step in an asynchronous read transaction, starting from the call to the *libraw1394* API function *raw1394_read* to the return of this function call. The average 32 μs transaction time is comprised of two operating system calls, two data transmission times, and data processing time on the FPGA. We are able to measure the data transmission and data processing time in the FPGA, which is less than 5 μs. This implies that the latency is mainly due to software overhead in the operating system. An obvious inference is that in order to improve the I/O performance, the best approach is to reduce the total number of transactions initiated by the control PC. This insight guides us to use an asynchronous broadcast transaction-based solution, where we compensate for the lack of acknowledgment packets by embedding an acknowledgment (actually, a sequence number) in the packets sent from the FPGAs to the PC.



(a) Asynchronous Block Read (b) Asynchronous Block Write

Fig. 6. Asynchronous block read and write times (400 Mbps)

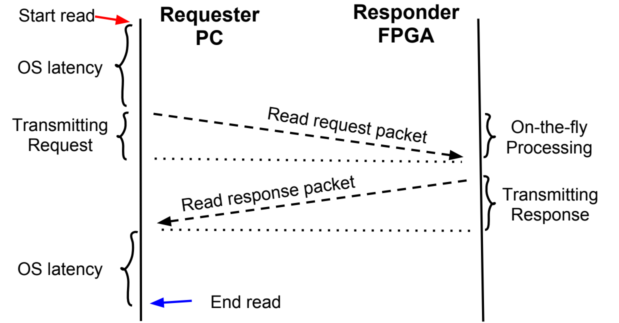


Fig. 7. IEEE-1394 asynchronous block read includes two operating system (OS) calls, data transmission time, and data processing time

III. Broadcast Communication Protocol

This section presents the newly designed high-performance communication model, including several optimizations to further improve performance, and discusses system characteristics, including determinism and system integrity.

A. Transmission model

As shown in Fig. 8, a servo control cycle starts with an asynchronous broadcast quadlet write packet from the PC, serving as query (or sync) command to all FPGA boards. After sending this packet, the PC software sleeps for $5\mu s \times N_{boards}$. Upon receipt of this packet, each FPGA board will wait for a predefined offset $(5\mu s \times NodeID)$, then transmit its status data using an asynchronous broadcast block write packet. This is a Time Division Multiple Access (TDMA) method, similar to the isochronous transfers already present in the IEEE-1394 specification, but scheduled with respect to the query command, which can have an arbitrarily specified frequency. All broadcast packets are received and cached by every FPGA node, so that all nodes maintain a copy of the entire robot status feedback. Upon awakening, the PC sends one asynchronous block read request to any node to fetch this information. This node can be called a *hub node*, though it is important to note that any FPGA node can serve this function. The PC software then performs the control computations and broadcasts new

command data for all FPGA boards. This completes a servo control cycle. The key ideas behind this design are to reduce operating system overhead by cutting the total number of transactions initiated from the control PC, and to use broadcast packets to minimize the number of data packets on the bus. In fact, the number of PC-initiated transactions (3 transactions) is now independent of the number of FPGA boards (nodes) on the bus. The I/O time for the broadcast protocol is:

$$T_{I/O_bc} = T_Q + 5\mu s \times N_{boards} + T_R + T_W, \quad (2)$$

where T_Q, T_R and T_W are time for query, block read and command write transactions, respectively.

In theory, the PC could serve as the hub node, but we have found that this is not a reliable solution. In our experiments, we detected a 2% packet loss when attempting to use the PC as a hub node. We hypothesize that this is due to the use of a software driver to handle asynchronous requests, which is inherently slower than a hardware-based (FPGA) solution. We therefore introduced the hub node concept to solve this problem. By design, all FPGA boards are hub capable and the PC can read complete status feedback from any FPGA board on the bus. We note, however, that a real-time kernel, such as Xenomai, with a real-time FireWire driver [10] could be another solution to prevent dropped packets.

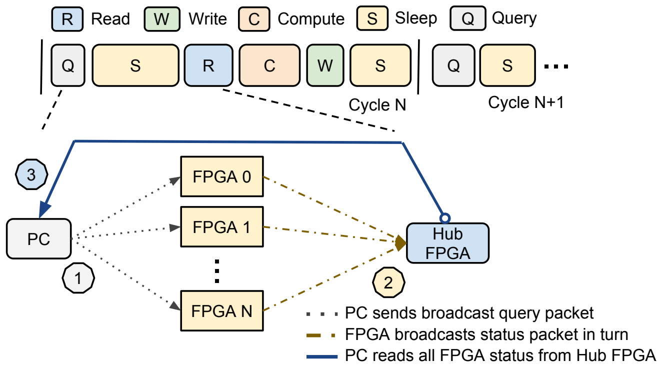


Fig. 8. Asynchronous broadcast based communication model

B. Bus optimizations

While the above protocol greatly improves the performance of the system, the design incorporates several other optimizations, as detailed in this section. These particular optimizations are feasible in a closed system (e.g., where there are no other nodes on the FireWire bus) and could be omitted if necessary.

Bus arbitration acceleration: Whenever the link layer controller wants to transmit data to the FireWire bus, it sends an arbitration request to the physical layer chip and the physical layer will in turn arbitrate for bus ownership. In the FireWire specification, the link layer controller can only issue *priority* or *fair* requests for an asynchronous subaction. For these two types of requests, the physical layer chip starts bus arbitration after it detects a subaction gap, which nominally is 10 μs [4]. This subaction gap time limits the overall performance. But, because the link layer is implemented in an FPGA, we are able to improve performance by issuing an *isochronous* bus request to the physical layer chip, even though we intend to send an *asynchronous* packet. In this case, the physical layer only waits for a 0.04 μs isochronous gap before starting to arbitrate for the bus. In a standard FireWire system, this is possible because an isochronous bus request is only issued when the bus is performing isochronous transactions. In our design, it works because the time each node starts transmitting is deterministic (e.g., based on the TDMA method described above). This mechanism accelerates the bus arbitration process, improves bus bandwidth usage, and breaks the limitation of using regular asynchronous transactions. But, it assumes that we have complete control over the FireWire bus and can prevent an "outside" node (e.g., a FireWire camera or hard drive) to interfere with this protocol.

Disable cycle start packet: The cycle start packet is transmitted from the cycle master node (i.e., PC) on the bus at 8 kHz to synchronize isochronous data transfers. Because we do not use isochronous transactions and, more importantly, to avoid interfering with the broadcast write packets, the cycle master node capability is disabled and no cycle start packet is issued on the bus. It is also reported [11] that this optimization can increase asynchronous transaction performance by 5%.

Full speed broadcast packets: In the standard Linux kernel, the Juju FireWire driver sets the asynchronous broadcast speed to 100 Mbps. This can be changed to 400 Mbps to shorten the data transmision time from the PC and yield about a 4 μs performance gain, at the cost of having to modify and recompile the FireWire driver source code.

C. System Characteristics

Besides high performance, our design has other system characteristics that favor a network-connected centralized processing and distributed I/O control architecture.

1) Determinism: In a networked control architecture, determinism is beneficial and sometimes even required. This means that given a certain bus state, the next bus state is completely determined. This feature is extremely important when doing control at an extremely high frequency, such as 5 kHz. In our design, the determinism is guaranteed by bringing optimizations on top of the IEEE-1394a specification and by not implementing certain functionality in the FPGA FireWire module. The determinism of the system includes using a fixed root node (the PC), data transmission synchronization via a broadcast write packet from the control PC, and pre-configured bandwidth and offset.

By not implementing the bus manager layer on the FPGA nodes and not allowing other types of FireWire nodes on the bus, we can be assured that the control PC is the only node that can be bus manager, isochronous resource manager, and cycle master and is therefore forced (by the IEEE-1394 specification) to be the root node on the bus. This determinism also simplifies the procedure to stop cycle start packets. By not initiating asynchronous transactions (the broadcast asynchronous write packet is considered a "response" to the packet from PC), the software running on the PC has complete control over what data, at what time, is on the FireWire bus.

2) Error tolerance: Data integrity is crucial in a robot control application. This is especially true for a medical robot that is designed to operate on patients. This is also the reason we favored regular asynchronous read and write over fast isochronous transactions in our previous design. For the same

reason, we include three mechanisms to ensure data integrity, even when using broadcast packets for which there is no acknowledgment packet. The basic feature, a Cyclic Redundancy Check (CRC), is compulsory as it is specified in the IEEE-1394 standard. This provides a basic error detection mechanism. A more important feature is to include data integrity information and a sequence number (16 bits) from the PC write packet in the "response" broadcast packet from each FPGA. This feature is a remedy for the lack of an acknowledgment packet for asynchronous broadcast write packets. In a situation where the packet from the PC is corrupted or the data is incorrect, the sequence number in the FPGA packet is set to 0xFFFF; otherwise the received sequence number is returned. If the PC software receives a response with sequence number 0xFFFF, it triggers a software error handling mechanism. Finally, the FPGA firmware includes a watchdog that needs to be refreshed by an asynchronous broadcast write packet from the control PC. This guarantees that in extreme cases (e.g., a software crash on the PC), the FPGA board will disable the amplifiers and ensure that there is no power to the robot system.

3) Backward compatibility: The new design greatly improves communication performance between the control PC and FPGA and retains the support for asynchronous read/write transactions, thereby remaining backward compatible.

IV. EXPERIMENTS

This section experimentally examines the performance of the broadcast communication protocol using both the FPGA board and PC software. The measurement data is compared to a prior asynchronous protocol described in Section II-C.

A. FPGA hardware-based measurement

With a soft JTAG tool, we captured the data transmission on the FireWire bus in one complete servo cycle, as shown in Fig. 9. The blue section of the data bus indicates that its value is changing and the LLC is either receiving (does not mean the data is targeted at the FireWire node) or transmitting data from or to the PHY chip. The counts at the top show the number of time cycles (clock is 49.125 MHz, $1\mu s = 49.125$ cycles). The cycle starts with a broadcast quadlet packet with less than 10 cycles. The total time for 8 FPGA boards to finish data transmission is around 2,000 cycles (40.71 μs), with each board taking 250 cycles on average (5.1 μs). After these transactions, an asynchronous read request, indicating the start of the third phase, has triggered the asynchronous block read response packet from the hub FPGA node. This asynchronous read, including the final ACK packet from the control PC, takes 20 μs. However, this number does not include operating system latency before the asynchronous read request is sent out and the latency after the data packet has physically arrived at the PC hardware. The time between the asynchronous block read (Async hub packet) and the broadcast block write (PC Command packet) is PC computation time. Finally, the broadcast command packet from the PC transmitting at 400 Mbps takes 160 cycles (3.26 μs).

B. PC software-based measurement

While the measurement on the FPGA board is more accurate, it does not include latencies introduced by the PC software, such as the operating system scheduling delay. Thus, timing data measured from the PC is presented here. The asynchronous broadcast request is sent out to the FireWire bus by calling the *raw1394_start_write* function. The time cost of this function call has been measured with a payload of 144 bytes (8 FPGA boards payload). Fig. 10(a) shows that the average time cost is 2.51 μs with a standard deviation of 2.71 μs. Note that this value is only the time cost for calling the function and does not include, or not fully include, the transmission time. Compared with a regular asynchronous block write with a 16 byte payload, the time cost is 90% less, which is not surprising since the broadcast does not require an acknowledgment packet and thus saves the time of waiting for a response. The measured overall I/O time cost for an 8 board robot system is 115.77 μs on average, with 14.51 μs standard deviation (Fig. 10(b)). The I/O time is only 23% of the straightforward protocol described in Section II-C, which uses individual asynchronous read and write transactions to each FPGA node.

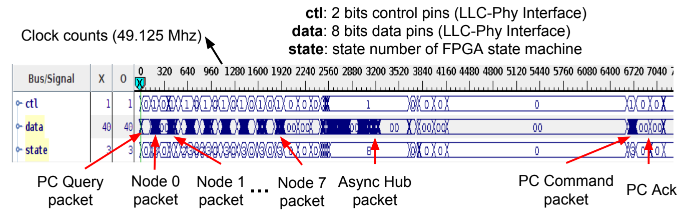
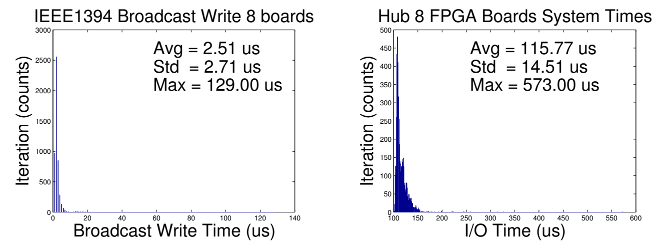


Fig. 9. Waveform of control, data and state bus within one I/O cycle



(a) Asynchronous Broadcast Block Write (128 Bytes)

(b) I/O cycle time for new design

Fig. 10. Broadcast write transaction time and I/O time for 8 board system using new communication model

C. Measurements with da Vinci Research Kit

We experimentally compared the broadcast protocol and the prior asynchronous protocol using test setups with 2 boards, 4 boards and 8 boards. As shown in Fig. 11, the broadcast protocol shows a huge performance increase, especially for a system with many nodes. Using the broadcast protocol, we are able to run a da Vinci system with 8 boards at 6 kHz.

V. DISCUSSION: FIREWIRE VS. ETHERCAT

We selected FireWire in 2006, but the obvious question is whether this is still a good choice today, given the wider deployment of other fieldbus systems, particularly EtherCAT [12]. Similar to the FireWire broadcast protocol, EtherCAT is a link layer level protocol and requires special slave controllers (any Ethernet port, with special software, can be used for the master). Fig. 12 shows an example EtherCAT system with one master node and several slave nodes. The EtherCAT frame

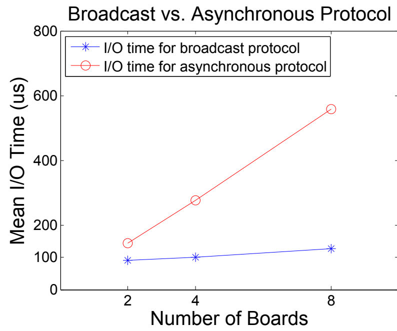


Fig. 11. Mean I/O time comparison between Broadcast and Asynchronous protocols on system with 2, 4 and 8 boards

initiated by the master is passed through the next slave node, which processes data on-the-fly, until it reaches the end of the chain and is sent back to the master. This protocol also minimizes the number of transactions on the master node, which typically is a conventional computing platform and therefore subject to software-induced latency.

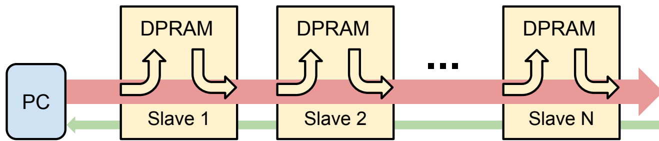


Fig. 12. An example EtherCAT system

There are several obvious benefits to using EtherCAT: (1) while many computers have a FireWire port, it is not as ubiquitous as an Ethernet port; (2) EtherCAT uses standard Ethernet cables, which are more easily routed inside a robot mechanism (with an option for high-flex cables) and can be longer than FireWire cables; and (3) there are more vendors providing control components with EtherCAT interfaces.

Both FireWire and EtherCAT can provide sufficient performance for high-rate control. FireWire has a higher bus bandwith (400 Mbits/s for IEEE-1394a and typically 800 Mbits/s for IEEE-1394b) compared to EtherCAT, which is limited to 100 Mbits/s, but this is not likely to be the limiting factor. Since we do not have access to EtherCAT hardware, we estimated the I/O cycle time based on [13]. For an 8 board system, with a payload of 64 bytes per board, the estimated time would be 50 μs. However, this is an ideal time that does not consider operating system overhead and other implementation issues. Under these ideal conditions the proposed FireWire protocol would provide similar performance. For a real EtherCAT system, Potra et al. [14] reported a 200 μs cycle time for controlling 32 digital signals, which is comparable to the real system times we have measured.

At this point, we can identify a few advantages of our FireWire system: (1) it is easier to dynamically reconfigure FireWire systems by connecting and disconnecting nodes, as compared to the configuration tools and files required for EtherCAT systems; and (2) it is completely open source and therefore simple and inexpensive for researchers to implement custom slave nodes or to customize the protocol. As an example, since we use broadcast to transmit status data from the FPGA, all FPGA nodes can receive these packets and have information about the whole robot system. Besides allowing any FPGA node to act as the Hub, this is potentially valuable in a multivariable control system. Another use of this information is to provide an extra safety feature (e.g., power shutdown if another FPGA node fails).

VI. CONCLUSION

Our analysis of the original control system I/O performance revealed that latency due to PC operating system overhead was the primary cause of the I/O performance bottleneck, which led us to a series of bus optimizations and a new communication protocol. This new approach reduces the number of PC-initiated transactions to three by using broadcast packets and enabling all FPGA controller boards to cache status packets from all other controller boards, thereby enabling any of them to act as a Hub node. Experimental results show that the new approach cuts the average I/O cycle time to 115.77 μs. This indicates that for an eight board system, a control rate of 6 kHz is achievable. The new design also shows very good scalability because adding one FPGA board only requires an additional 5 μs of I/O time. This IEEE-1394 based communication model provides a solution to control multiple robots at multi-kilohertz servo loop frequencies.

REFERENCES

[1] P. Kazanzides and P. Thienphrapa, “Centralized processing and distributed I/O for robot control,” in *Technologies for Practical Robot Applications (TePRA)*, Woburn, MA, Nov 2008, pp. 84–88.

[2] H. Bai, “Analysis of a SAE AS5643 Mil-1394b based high-speed avionics network architecture for space and defense applications,” in *IEEE Aerospace Conf.*, Big Sky, MT, Mar 2007, pp. 1–9.

[3] EtherCAT Technology Group, http://www.ethercat.org/.

[4] D. Anderson, *FireWire System Architecture, 2nd Edition*. MindShare, Inc., Addison-Wesley, 1999.

[5] P. Kazanzides, Z. Chen, A. Deguet, G. S. Fischer, R. H. Taylor, and S. DiMaio, “An open-source research kit for the da Vinci® surgical robot,” in *IEEE Intl. Conf. on Robotics and Auto. (ICRA)*, May 2014.

[6] Z. Chen, A. Deguet, R. Taylor, S. DiMaio, G. Fischer, and P. Kazanzides, “An open-source hardware and software platform for telesurgical robot research,” in *MICCAI Workshop on Systems and Arch. for Computer Assisted Interventions*, Sep 2013.

[7] IEEE-1394 Working Group, “IEEE Standard for a High Performance Serial Bus and Amendments,” *IEEE Std 1394-1995*, 1996.

[8] P. Thienphrapa and P. Kazanzides, “A distributed I/O low-level controller for highly-dexterous snake robots,” in *IEEE Biomedical Circuits and Systems Conf. (BioCAS)*, Baltimore, MD, Nov 2008, pp. 9–12.

[9] M. Sarker, C. Kim, J. Cho, and B. You, “Development of a network-based real-time robot control system over IEEE 1394: using open source software platform,” in *IEEE Intl. Conf. on Mechatronics*, Jul 2006, pp. 563–568.

[10] Y. Zhang, B. Orlic, P. Visser, and J. Broenink, “Hard real-time networking on firewire,” in *7th Real-Time Linux Workshop*, P. Marquet, N. McGuire, and P. Wurmsdobler, Eds. Eindhoven, the Netherlands: IOP Press, 2005, pp. 1–8.

[11] Trade Association and others, “Firewire™ reference tutorial,” 1394 Trade Association, Tech. Rep., 2010.

[12] S. G. Robertz, R. Henriksson, K. Nilsson, A. Blomdell, and I. Tarasov, “Using real-time Java for industrial robot control,” in *Proc. 5th ACM Intl. Workshop on Java technologies for real-time and embedded systems (JTRES)*, Vienna, Austria, 2007, pp. 104–110.

[13] G. Prytz, “A performance analysis of EtherCAT and PROFINET IRT,” in *IEEE Intl. Conf. on Emerging Technologies and Factory Automation (ETFA)*, Hamburg, Germany, Sep 2008, pp. 408–415.

[14] S. Potra and G. Sebestyen, “EtherCAT protocol implementation issues on an embedded Linux platform,” in *IEEE Intl. Conf. on Auto., Quality and Testing, Robotics*, Los Alamitos, CA, May 2006, pp. 420–425.