An Ethernet to FireWire Bridge for Real-Time Control of the da Vinci Research Kit (dVRK)

Long Qian*†, Zihan Chen* and Peter Kazanzides*
* Johns Hopkins University, Baltimore, MD USA, email: pkaz@jhu.edu
† Tsinghua University, Beijing, China

***Abstract*—In this paper, a real-time control network based on Ethernet and FireWire is presented, where Ethernet provides a convenient, cross-platform interface between a central control PC and a FireWire subnetwork that contains multiple distributed nodes (I/O boards). Real-time performance is achieved because this architecture limits the number of Ethernet transactions on the host PC, benefits from the availability of real-time Ethernet drivers, and uses the broadcast and peer-to-peer capabilities of FireWire to efficiently transfer data among the distributed nodes. This approach and resulting benefits are comparable to EtherCAT, but preserves existing investments in FireWire-based controllers and relies only on conventional, vendor-neutral network hardware and protocols. The system performance is demonstrated on the da Vinci Research Kit (dVRK), which consists of 8 FireWire nodes that control 2 Master Tool Manipulators (MTMs) and 2 Patient Side Manipulators (PSMs), for a total of 28 axes. This approach is generally applicable to interface existing FireWire-based systems to new control PCs via Ethernet or to serve as an open-source alternative to EtherCAT for new designs.**

I. Introduction

Robot systems require high-frequency (usually greater than 1 kHz), and preferably hard real-time, periodic computation for low-level control of joint positions, velocities, or torques. Traditionally, low-level control is implemented on one or more embedded systems that each contain a microprocessor and I/O devices to achieve fast I/O and limit data communication, which can be called a *distributed computation and I/O architecture*. Recently, there has been growing interest in the research community for a *centralized computation and distributed I/O architecture* (Fig. 1), which performs all computations, including low-level control, on a PC and connects to I/O devices via a high-speed fieldbus network. This architecture provides a more familiar development environment (e.g., a Linux PC), thereby facilitating rapid prototyping and reconfiguration.

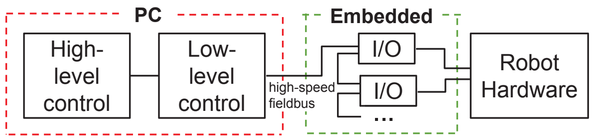


Fig. 1. Centralized computation and distributed I/O architecture

Given that I/O communication occurs within the low-level control loop, the control performance is bounded by the I/O network bandwidth and latency, which is difficult to minimize in conventional networks such as Ethernet and USB, especially when multiple bus transactions are needed to communicate with multiple distributed nodes. Controller Area Network (CAN) is well accepted in the real-time community, but has a maximum rate of 1 Mbps and thus is not fast enough to support a high-frequency control loop on the PC. For higher performance, FireWire (IEEE-1394) has traditionally been a viable option and has been incorporated into several commercial products. FireWire also forms the basis for SAE AS5643 [1], which is designed for military and avionics applications and focuses on bus determinism and safety features. Various real-time Ethernet solutions have been proposed both by vendors and academia and real-time Ethernet drivers, such as RTnet, are promising developments [2]. Some Ethernet-based solutions adopt custom hardware, at least for the embedded devices. Of these, the most popular is EtherCAT (www.ethercat.org), depicted in Fig. 2, which uses conventional Ethernet hardware and custom software drivers on the master PC and custom hardware on the embedded (slave) nodes. The main advantage of EtherCAT is that all the slave nodes can be daisy-chained together and the master PC can perform I/O with the entire network by sending and receiving just one Ethernet frame, thereby limiting software overhead. Each slave node receives the frame on its *in* port and extracts/inserts data from/into the frame on-the-fly while forwarding it to the *out* port.

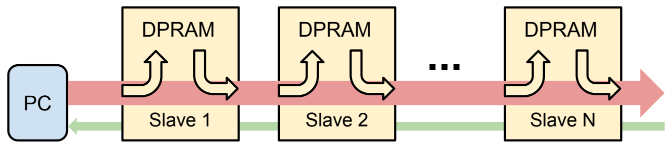


Fig. 2. EtherCAT system with 1 master and N slave nodes

We investigated and implemented a FireWire-based design in 2008 [3] and later proposed a broadcast communication protocol on the same system [4], achieving a four-fold performance improvement. Although designed for general-purpose use, these FireWire-based controllers have primarily been used to control the da Vinci® Research Kit (dVRK) at Johns Hopkins University and more than 15 other institutions that have replicated this setup [5]. However, FireWire today is less prevalent than in the past and even the real-time PC driver stack, RT-Firewire [6], is no longer maintained. Thus, although we achieved multi-kilohertz control using conventional FireWire drivers on a generic Linux kernel, this solution suffers from occasional timing outliers. Furthermore, while the new broadcast protocol can achieve up to 6 kHz control rates [4], we have found that it does not work reliably with some PC FireWire chipsets/drivers. Finally, FireWire interfaces are not as common as Ethernet on modern computers and laptops, and use of *libraw1394* primarily restricts the system to Linux and its real-time variants. Although a Windows version of *libraw1394* has been reported [7], this would only be suitable for non-real-time applications. More importantly, some dVRK sites have invested in other real-time platforms, such as Matlab Simulink Real-Time (formerly called Matlab xPC), which supports Ethernet and EtherCAT but not FireWire. Thus, while one could invest in developing real-time FireWire drivers for the different platforms and require all control computers to have FireWire interfaces, it is more practical to leverage the existing hardware and software support for Ethernet as a real-time control fieldbus.

We considered two approaches to leverage Ethernet-based technology for our multi-node distributed control system: (1) replace the FireWire interfaces on each node with EtherCAT, or (2) introduce an Ethernet-to-FireWire bridge between the PC and FireWire subnetwork. The first approach has the potential advantage that the cables are conventional unshielded twisted pair (UTP) and can be high-flex, longer, and more easily routed inside robotic structures. Cabling is not an issue for the dVRK, however, and this approach would require a substantial retrofit of existing systems, so we adopted the second approach, which is illustrated in Fig. 3. This paper presents the Ethernet-to-FireWire bridge design and the results of experiments, including those with the actual dVRK hardware, to demonstrate that with the appropriate software, it provides hard-real-time performance for multi-kiloherz centralized control of a large number of distributed robot axes.

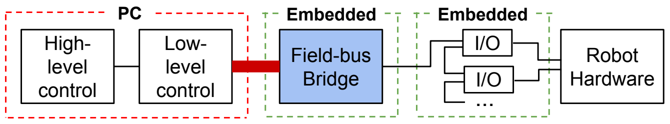


Fig. 3. Centralized computation and distributed I/O with fieldbus-bridge

II. Background

This section gives an overview of the dVRK system architecture and reviews the FireWire-based communication protocols.

A. *Open Source Mechatronics*

A FireWire-based controller [8] (Fig. 4) has been designed and is available open source at http://jhu-cisst.github.io/mechatronics, including the schematics, PCB layout, and FPGA firmware (Verilog). The controller features a separate FireWire/FPGA board and a Quad Linear Amplifier (QLA) I/O board, which provides the flexibility to design a new FPGA board that introduces a different communication network (e.g., EtherCAT) or to design a new I/O board to interface to different hardware. Although this is a general-purpose mechatronics system, its primary application is to control the da Vinci Research Kit (dVRK). As illustrated in Fig. 5, multiple FPGA/QLA board sets are daisy-chained together and connect to a Linux control PC.

B. *FireWire Communication Protocol*

IEEE-1394 is well suited for real-time control applications due to its high-speed, peer-to-peer, full-duplex characteristics. The maximum transmission rate of the IEEE-1394a physical medium is 400Mbits/sec. Up to 64 nodes are supported in a FireWire fieldbus network.

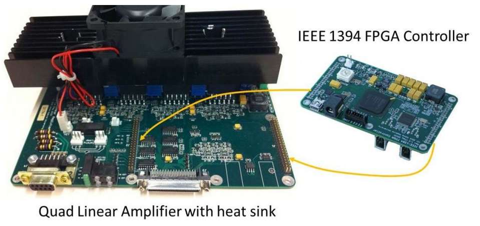

IEEE 1394 FPGA Controller
Quad Linear Amplifier with heat sink

Fig. 4. FireWire/FPGA board and Quad Linear Amplifier (QLA)

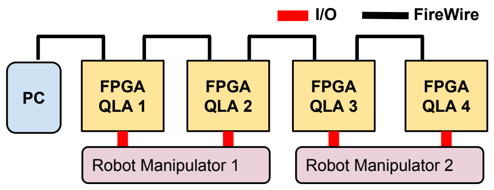
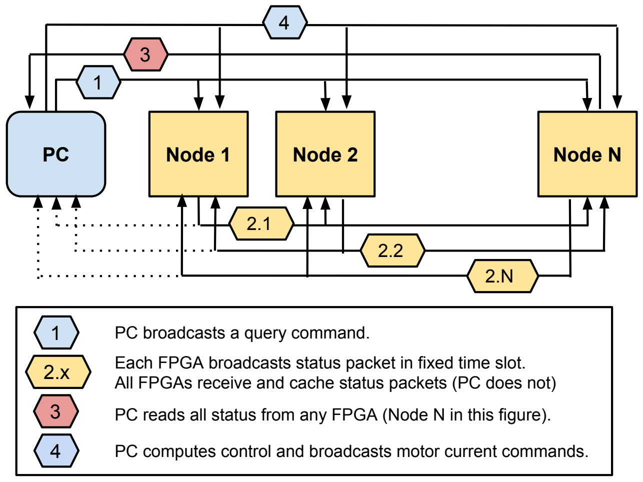


Fig. 5. Hardware architecture: one control PC and 4 FireWire FPGA/QLA board sets controlling 2 da Vinci manipulators (7 DOF each); a complete dVRK system would use 8 FPGA/QLA board sets to control 4 manipulators.

We initially used a sequential FireWire protocol to communicate to the slave nodes. In the control loop, the PC software sequentially queried each FPGA node status (individual block reads), computed the control commands, and sent them to each board in separate FireWire block write requests. This protocol required two transactions per node and thus the I/O time increased linearly and quickly became a bottleneck for high performance control of multiple degrees-of-freedom.

Recently, we introduced a broadcast-based protocol (Fig. 6) that was designed to take advantage of FireWire's peer-to-peer capability [4]. With this protocol, the control cycle starts with the PC broadcasting a query packet to all nodes (FPGA boards). After sending this packet, the PC software sleeps for 5 x N μs (N = number of nodes). Each FPGA board transmits its status using an asynchronous broadcast block write packet 5 x NodeID μs after it received the query packet (NodeID is the FireWire node-id, which is assigned during the bus initialization). Each FPGA node has a unique node-id, which always starts at 0 and is sequential. All broadcast packets are received and cached by each FPGA node, so that all nodes maintain a copy of the entire robot status feedback. Upon awakening, the PC software reads the status of all boards from any FPGA node. This is a Time Division Multiple Access (TDMA) method, scheduled with respect to the query command. It is similar to the FireWire isochronous transfer mode, except that it can have an arbitrarily specified frequency. After the PC computes the control commands, it transmits a broadcast packet that is received by all nodes.

This system was shown to provide fast I/O times on average, enabling closed-loop control up to 6 kHz for a full dVRK system [4]. But, several limitations were revealed: (1) while the average I/O time was fast (116 μs), outliers were up to five times the average time (562 μs); (2) the broadcast protocol did not work reliably on all PC chipsets/drivers; (3) many computers do not have a FireWire interface; and (4) many operating systems do not provide a low-level FireWire packet interface such as *libraw1394*. The first limitation could be overcome by using a real-time FireWire driver, such as RT-FireWire, but this driver is no longer maintained. These issues



4
3
1
PC
Node 1
Node 2
Node N
2.1
2.2
2.N
1 PC broadcasts a query command.
2.x Each FPGA broadcasts status packet in fixed time slot. All FPGAs receive and cache status packets (PC does not)
3 PC reads all status from any FPGA (Node N in this figure).
4 PC computes control and broadcasts motor current commands.

Fig. 6. FireWire-based broadcast protocol.

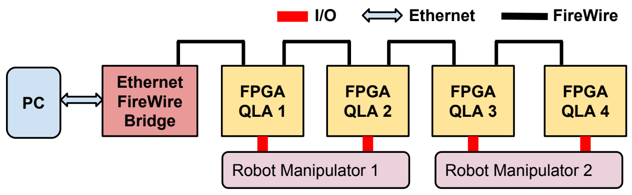


Fig. 7. Hardware architecture with Ethernet to FireWire bridge

motivated us to develop an Ethernet-to-FireWire Bridge, which is described in the next section.

III. Ethernet-to-FireWire Bridge Design

Compared with a fully FireWire-based system (Fig. 5), Fig. 7 introduces a new system hierarchy for dVRK with an embedded bridge between the PC and FireWire-based control network. All the communications between the PC and slave nodes are done via the bridge node. While the bridge talks to the rest of control network using the FireWire-based broadcast protocol, the connection between the PC and bridge node is point-to-point, thereby eliminating the need for a complicated media access control protocol.

This section describes the bridge design, frame transmission protocol and status control process implemented on the firmware of the bridge node.

A. Bridge Design

The prototype bridge consists of our custom FPGA board, which contains a Xilinx Spartan-6 XC6SLX45 FPGA and IEEE-1394a physical layer chip, coupled with an off-the-shelf Ethernet PHY and MAC controller board (KSZ8851-16mll-EVAL). The Ethernet chip manufactured by Micrel is a single-port controller chip with a non-PCI Interface and is available in 8-bit and 16-bit bus designs. We utilized the 16-bit bus in our design for better efficiency in Ethernet data I/O. We designed a custom connector board to interface these two boards, as shown in Fig. 8. The final version of this design could consist of two boards: the existing FPGA board and a custom mating board that contains the Ethernet chip and physical interface.

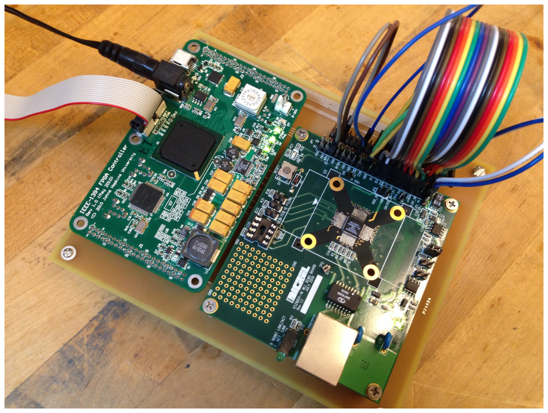


Fig. 8. Prototype Ethernet-to-FireWire Bridge

B. Frame Transmission Protocol

The main functionality of the bridge node is to convert an Ethernet packet from the PC into a FireWire packet, perform the FireWire transaction, and convert the FireWire response to an Ethernet packet for the PC. To simplify the development of the bridge node FPGA firmware and to maximize system efficiency, the FireWire packet construction and parsing is implemented in the PC software.

Frames transmitted from the PC to the bridge node include *quadlet read/write, block read/write*, the previously defined *broadcast-based write/query*, and *system synchronization*, each with an appropriate Ethernet header and checksum. The frame structure is presented in Fig. 9. *Quadlet read/write* and *block read/write* are defined in the IEEE-1394 standard, enabling basic read and write transactions of variable length between two individual nodes within a FireWire network. As demonstrated previously, the *broadcast-based write/query* accelerates the control system by eliminating multiple requests to each separate node from the PC. The *system synchronization* frame is used to inform the bridge of the number of active nodes, N, in the FireWire network. The PC controller is authorized to add or remove an existing board to or from the list of current active boards. After a broadcast query is transmitted, the bridge will serve as a hub, collecting N responses before sending the combined packet back to the PC.

Upon receiving an Ethernet packet, a parallel validation checking process based on the frame format is activated. The validated frame is passed to the FireWire network with the Ethernet header and checksum removed. If a *system synchronization* packet is received, the local parameter N is updated; it is not necessary to relay that frame to the FireWire subsystem.

In the reverse direction, *quadlet/block read response*, *write*

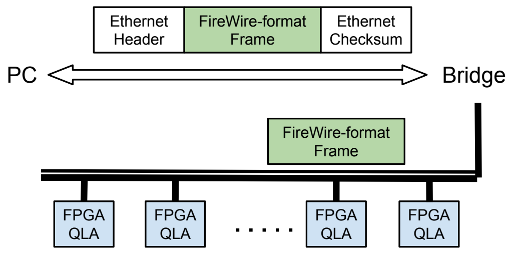


Fig. 9. Frame Structure

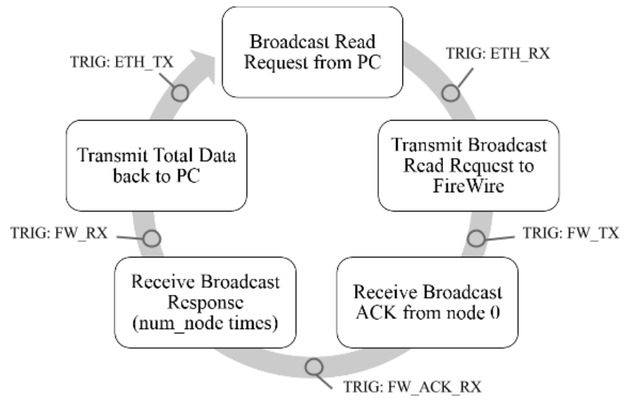

Broadcast Read Request from PC
TRIG: ETH_RX
Transmit Broadcast Read Request to FireWire
TRIG: FW_TX
Receive Broadcast ACK from node 0
TRIG: FW_ACK_RX
Receive Broadcast Response (num_node times)
TRIG: FW_RX
Transmit Total Data back to PC
TRIG: ETH_TX

Fig. 10. Finite State Machine for control loop

acknowledge frame, and *broadcast query response* are transmitted through the FireWire field-bus. The first two frame types are defined in the IEEE-1394 standard as responses to *quadlet/block read/write* requests. When *quadlet/block read responses* are received by the bridge node, they are passed to the Ethernet network with a specific Ethernet header and a correct checksum. The *write acknowledge frame* triggers the switch of state in the bridge. The PC controller is not acknowledged because the overhead of transmitting an Ethernet frame is comparatively high. The *broadcast query response* is initiated by individual slave nodes in the distributed I/O subsystem, and provides the feedback information for the PC to perform closed-loop control. The bridge gathers N feedback frames and then transmits them in one Ethernet packet with a predefined Ethernet header.

With this design, we successfully inherit the advantages of the FireWire-based approach and at the same time benefit from the ubiquity of Ethernet hardware and its well-maintained real-time driver stack for the PC. Robustness is also improved because the FireWire broadcast protocol no longer involves the FireWire chipset on the PC, which was problematic on some systems.

C. Status Control

Featuring its parallel operation, hard real-time capability and plentiful I/O extensions, the FPGA is more suitable than the PC to implement the finite state machine of the control loop. Sequential jumping is predefined in the firmware of the bridge, along with the signals triggering status shift. Though the PC is responsible for initiating read or write commands, its request does not act as an interrupt for the bridge node. Instead, commands are buffered until the bridge reaches the status of fetching a PC request. A timeout is set in order to avoid unnecessary waiting caused by errors. When the timeout requirement is met, the bridge switches back to the *Broadcast Read Request from PC* state, where it constantly polls for the trigger representing the arrival of a request initiated by the PC controller. In a complete control cycle, the finite state machine (FSM) is implemented as illustrated in Fig. 10. Five triggers are utilized to complement the FSM architecture.

D. Ethernet Software

The software of the bridge-based da Vinci Research Kit is arranged into several functional layers which remain unmodified compared to the FireWire-based dVRK: hardware interface, PID-based low-level control, high-level customized control, teleoperation and application layer [5].

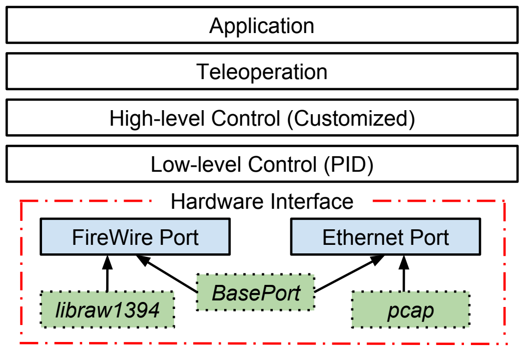


Fig. 11. Software Architecture

The introduction of the Ethernet-to-FireWire bridge requires an Ethernet interface instead of FireWire in the hardware interface level; this Ethernet interface is provided by the C++ library *pcap*. An *Eth1394Port* is created to represent the node in the bridge-based design. An abstract base class *BasePort* is introduced so that both *Eth1394Port* and *FirewirePort* can inherit from it and provide the same functionality. Class *AmpIO*, which represents an FPGA/QLA node, is unchanged, thereby keeping the upper software layers intact.

The Ethernet interface library *pcap* is directly available for Linux and OS X. For Windows, a slightly modified library, *winpcap*, is utilized, which provides the same methods for Ethernet port operation. Portability between different operating systems is guaranteed by the cross-platform support of the *pcap* library, which is a significant improvement compared to the FireWire-based dVRK, which required the *libraw1394* library that is only readily available on Linux. To achieve best performance of the system, a real-time environment composed of a real-time operating system and real-time Ethernet driver is required. Our software can be quickly ported to such platforms with the ubiquitous support for the *pcap* library; for example, Xenomai with the RTnet driver or Matlab Simulink Real-Time.

IV. Experiments

System performance experiments are conducted in the following three aspects. First, we measure the round-trip time of the standard FireWire protocol *quadlet read*, using a real-time operating system and real-time Ethernet driver. Timing characteristics of the bridge-based data transmission are compared with the FireWire-only transmission. Following that, control loop performance of both systems is tested and discussed. Furthermore, the cross-platform support for an Ethernet-based design is demonstrated.

A. FireWire Transaction

As presented in the *System Overview*, the previous FireWire-based design achieves a good average timing performance, however, the system reliability suffers from lack of support for a real-time PC FireWire driver. The introduction of the Ethernet-to-FireWire bridge aims to improve the system performance by taking advantage of the prevalent real-time Ethernet driver. First, the round-trip time of the basic *quadlet*

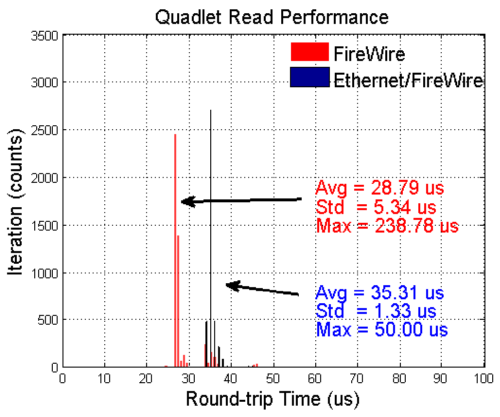


Fig. 12. Quadlet read transaction times, tested on Xenomai real-time operating system. FireWire uses standard (non-real-time) FireWire driver, whereas Ethernet/FireWire uses RTnet real-time driver.

read transaction for both system configurations is measured by averaging over 5000 transactions, as shown in Fig. 12. An Ethernet sniffer (tcpdump) based on the *pcap* library is used to capture the timestamp of Ethernet frames. A round-trip of a quadlet read transaction includes an Ethernet quadlet read request initiated by the PC controller and a corresponding response transmitted from the slave node. For both designs, the testing environment is set up on a real-time operating system (Xenomai 2.6.3 real-time framework based on Linux). The bridge design uses a real-time Ethernet driver (RTnet 0.9.12), while the FireWire-based design uses the standard (non-real-time) FireWire driver.

The average execution times for the FireWire-only and Ethernet-to-FireWire Bridge designs are 28.79 μs and 35.31 μs, respectively. The average time for the bridge-based design is longer due to the two additional Ethernet transmissions. But, the maximum time for the *quadlet read* transaction is significantly reduced from 238.78 μs for the FireWire-based design to 50 μs for the bridge-based design. This is primarily due to the use of the real-time Ethernet driver.

B. System Performance

We measured the control loop performance of the bridge-based design and compared it with the FireWire-only design, as shown in Fig. 13. The system performance includes the I/O time of Ethernet and FireWire in an 8-node system, which is typical for the control of a da Vinci Surgical Robot. Broadcast transfers are employed in both systems to maximize control efficiency. The test environment is the same as for the quadlet read, except that the FireWire-only design is tested with the generic Linux kernel rather than with Xenomai. In our experience, Xenomai and Linux-generic produce similar I/O times, since the primary cause of timing variations appears to be the non-real-time FireWire driver used in both cases.

The Ethernet I/O costs about 47.76 μs in the system loop, which is higher than the Ethernet I/O time for a *quadlet read* transaction due to the larger payload. As expected, the bridge-based design has more consistent timing measurements than the FireWire-only design. The standard deviation of the bridge-based design is 1.47 μs, and the maximum time cost is 175.00 μs, which is less than one-third of the FireWire-only design. With PC computation time added, the complete control loop for the bridge-based design is less than 200 μs,

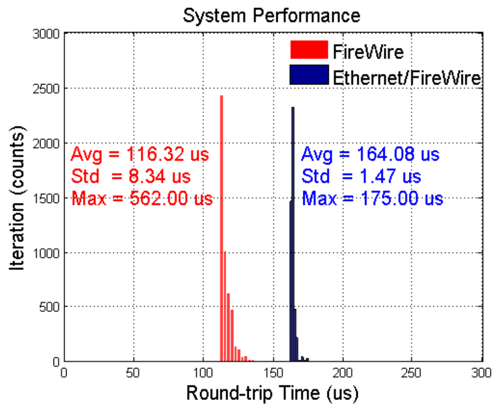


Fig. 13. I/O Time of FireWire and Ethernet/FireWire bridge in an 8 FPGA-QLA board system (standard dVRK setup); FireWire tested on generic Linux, whereas Ethernet/FireWire tested on Xenomai with RTnet driver.

which is sufficient for 5 kHz control. Though the control frequency of the FireWire-based design is higher on average, the performance is not as deterministic due to the lack of a real-time FireWire driver.

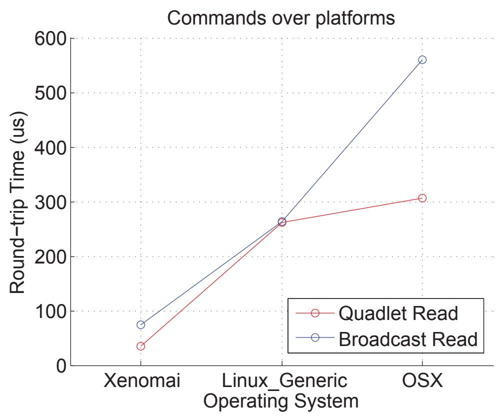
The introduction of the Ethernet-to-FireWire bridge separates the FireWire subsytem, which can then be implemented entirely on the FPGA. This makes it easier to support our custom broadcast protocol and avoid problems that we faced with some PC FireWire chipsets and drivers. On the PC, this design benefits from the availability of a real-time Ethernet driver.

C. Cross-platform Capability

The prevalence of Ethernet ports and software support (e.g., *pcap* library) renders the bridge-based design as a cross-platform solution. This test measures the timing performance of *quadlet read* and *broadcast read* for an 8-node system using different operating systems, as shown in Fig. 14. Xenomai is a real-time framework for Linux; real-time drivers such as RTnet are supported on the Xenomai platform. System performance is less satisfactory on non-real-time platforms such as Linux Generic and Apple OS X. For the Windows operating system, an echo test reveals that it takes approximately 2.34 ms for two Windows controllers to communicate through the raw Ethernet protocol. These tests verified the cross-platform capability of our design, but also demonstrated the importance of a real-time platform for a control system. Thus, cross-platform capability is more important for different real-time operating systems, such as Xenomai, Matlab Simulink Real-Time, and QNX. Support for the latter two real-time systems is the subject of future work.

D. Analytical Comparison with EtherCAT

The performance of the Ethernet/FireWire bridge architecture can be compared based on analytical models. Prytz [9] presented a model for the round-trip time on EtherCAT. This model assumes a master forwarding time based on the packet size and Ethernet bandwidth (e.g., 100 Mbit/s), a maximum delay of the master PHY (expected to be less than 0.5 μs), and a 1 μs forwarding time per slave node (this includes cable and slave PHY delays). The model does not, however, consider software overhead. We add a constant term, T_{sw}, to model



		Xenomai	Linux	OS X
Quadlet Read (μs)	Avg	35.3088	260.8400	306.7500
	Max	50.0	744.0	414.0
	Std	1.3337	16.0609	30.7173
Broadcast Read (μs)	Avg	73.5246	262.7728	559.6167
	Max	84.0	660.0	636.0
	Std	1.5074	14.2807	36.5718

Fig. 14. Quadlet Read/Broadcast Read Timing data on Xenomai, Linux and OS X

software overhead. Essentially, we assume that the software overhead due to the increase in packet size (as the number of nodes is increased) is negligible compared to the total software overhead. This is a reasonable assumption because the amount of time required to copy packet data from one memory location to another is small compared to other tasks done by the operating system and driver, such as context switches. Thus, we model the round-trip EtherCAT time as:

$$T_{ecat} = T_{sw} + N\frac{8\left(S_w + S_r\right)}{BW_e} + NT_{se} \tag{1}$$

where N is the number of slave nodes, S_w is the number of bytes written from the PC to each slave, S_r is the number of bytes sent by each slave to the PC, BW_e is the Ethernet bandwidth in Mbits/s (100), and T_{se} is the forwarding time of each EtherCAT slave node (1 μs according to [9]). This equation assumes the use of a single packet, which is reasonable given the number of bytes required in our application. It also does not include the overhead for the standard Ethernet header and CRC, which are the same for all Ethernet-based protocols, or the overhead for the EtherCAT header, which is assumed to be negligible.

To estimate a value for the software overhead, T_{sw}, we note that since both EtherCAT and the Ethernet/Firewire bridge use a standard Ethernet port on the PC, any software driver used for one could be used for the other. Thus, we estimate T_{sw} from the experimental results obtained with our Ethernet/Firewire bridge. To do this, we develop a similar model for the round-trip bridge time as:

$$T_{bridge} = T_{sw} + T_b + N\frac{8\left(S_w + S_r\right)}{BW_e} + N\frac{8S_w}{BW_f} + NT_{sf} \tag{2}$$

where BW_f is the FireWire bandwidth (400 Mbits/s for IEEE-1394a), T_b is the bridge delay, and T_{sf} is the time required for each slave to broadcast its data to the bridge node (5 μs). We assume that the bridge delay is constant because it can immediately begin processing an incoming (Ethernet or FireWire) packet and start transmitting the outgoing (FireWire or Ethernet) packet. Furthermore, because it is implemented in an FPGA and uses a 25 MHz 16-bit parallel interface to the Ethernet MAC, we assume that T_b is negligible.

For the da Vinci research kit, the packet sizes (not including headers, checksums, etc.) are $S_w = 20$ bytes and $S_r = 68$ bytes. Using these values, and the other values presented above, produces the following linear equation:

$$T_{bridge} = T_{sw} + 12.44N \qquad (\mu s) \tag{3}$$

We measured the round-trip time for the Ethernet/FireWire bridge for setups with 4, 5, 6, 7, and 8 nodes. Performing a linear regression yielded a slope of 16.08 μs/node and an intercept of 35.12 μs. Thus, we estimate $T_{sw} = 35.12\mu s$. We note that the measured slope (16.08) is larger than the computed slope (12.44), which indicates that our model may be missing some sources of delay or that some of our parameter estimates may be inaccurate. Our future work includes development of an improved model.

Applying the estimated value of T_{sw} to the EtherCAT implementation produces the following model:

$$T_{ecat} = 35.12 + 8.04N \qquad (\mu s) \tag{4}$$

To make a fair comparison, we evaluate the EtherCAT analytical model to the Bridge analytical model, rather than the measured data, because it is likely that measurements on an actual EtherCAT system would not agree exactly with the model. The models predict that the Ethernet/FireWire Bridge approach does not scale as efficiently as the EtherCAT approach; specifically, the overhead per node is about 50% higher. But, it is also important to note that the difference of about 4.4 μs/node is relatively small compared to the software overhead of 35.12 μs, which is a best-case scenario since it was obtained using the RTnet driver on Xenomai. For example, on an 8-node dVRK system, the Ethernet/FireWire bridge would require 135 μs, compared to 100 μs for EtherCAT.

V. Discussion and Conclusions

We developed an Ethernet-to-FireWire bridge that enables real-time control of a distributed system from a central PC. Real-time control is possible because the number of Ethernet transactions can be limited to two or three (depending on the protocol), regardless of the number of distributed nodes. In this manner, our system offers benefits similar to EtherCAT but utilizes only commodity network protocols (Ethernet and FireWire) and thus our complete hardware/software design is available open source.

The addition of the Ethernet-to-FireWire bridge node and associated real-time driver improves I/O performance, significantly reducing the maximum I/O time at the cost of a slight increase in the average I/O time (48 μs for the dVRK

System). From a systematic level, the bridge acts as a buffer or switch between two fieldbuses, Ethernet and FireWire. FireWire is designed as a real-time control fieldbus, however, real-time performance is only guaranteed within the embedded subsystem. Ethernet is not intrinsically designed as a real-time transmission media, but has a wide range of real-time support benefitting from its ubiquitous applications. The bridge approach leverages the strengths of two different transmission media (Ethernet and FireWire), while compensating for the drawbacks of each to achieve high bandwidth hard-real-time control performance. Although demonstrated on the da Vinci Research Kit, this approach is generally applicable to other systems.

Acknowledgment

This work is supported by NSF NRI 1208540.

References

[1] 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.

[2] J. Kiszka, B. Wagner, Y. Zhang, and J. Broenink, "RTnet-a flexible hard real-time networking framework," in *IEEE Intl. Conf. on Emerging Technologies and Factory Automation (ETFA)*, Catania, Italy, Sep 2005.

[3] 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.

[4] Z. Chen and P. Kazanzides, "Multi-kilohertz control of multiple robots via IEEE-1394 (FireWire)," in *IEEE Intl. Conf. on Technologies for Practical Robot Applications (TePRA)*, April 2014, pp. 1–6.

[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] Y. Zhang, B. Orlic, P. Visser, and J. Broenink, "Hard real-time networking on Firewire," in *7th Real-Time Linux Workshop*, Lille, France, Nov 2005, pp. 1–8.

[7] M. A. Tsegaye, "A comparative study of the Linux and Windows device driver architectures with a focus on IEEE1394 (high speed serial bus) drivers," Master's thesis, Dept. of Computer Science, Rhodes University, Dec 2002.

[8] 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.

[9] 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.