2013 IEEE/RSJ International Conference on
Intelligent Robots and Systems (IROS)
November 3-7, 2013. Tokyo, Japan

Unity-Link: A Software-Gateware Interface for Rapid Prototyping of Experimental Robot Controllers on FPGAs

Anders Blaabjerg Lange, Ulrik Pagh Schultz and Anders Stengaard Soerensen

Abstract— **In experimental robotics, we are often faced with differing requirements between projects and as a project evolves, making the initial choice of technology difficult, often requiring a continuous and tedious development of the low-level parts of the robotic system. We propose the use of FPGAs as a flexible solution to these low-level issues; We here address the hitherto unresolved issue of interfacing the FPGA-based controllers to high-level robotics software running on a PC.**

This paper presents the Unity-Link software-gateware stack, which connects high-level software frameworks to our modular, FPGA-based generic hardware. Unity-Link provides simple, unified abstractions for quickly and easily interconnecting PC-based systems with nodes that provide hard real-time control of distributed robotic systems. Unity-Link uses a component-based modular bus structure based on open standards, and interfaces with a library of gateware components, enabling us to create complex applications quickly and efficiently. Automated code generation is used to provide convenient, application-specific interfaces to high-level robotics middleware such as ROS.

Terminology: The bit-file loaded into an FPGA, generated through synthesis of HDL code, will throughout this article be denoted as gateware (GW) in order to distinguish this from actual hardware (HW) or software (SW).

I. Introduction

There are numerous well-proven technologies for interfacing robot controllers to actuators and sensors. Given specific requirements we can choose an appropriate standardized solution, such as a serial bus or simply a TCP/IP-based network if there are no real-time requirements. In experimental robotics, we are however often faced with differing requirements from project to project and as the project evolves over time [1], [2], [3]. This issue is a fundamental challenge: adopting a new interface architecture each time the requirements change takes a significant amount of time, and usually does not contribute directly to the functionality of the robot [1], [4]. The tendency is towards ad-hoc solutions, resulting in an architectural mismatch between the requirements and the technology, as well as an absence of reuse between projects.

To remedy this problem, a hardware-software interface approach is needed that facilitates development and increases reusability [1], can be used without expertise in embedded systems [5], and provides integration to high-level software frameworks such as e.g. ROS [6] or Orocos [7]. We believe that such an approach is most easily achieved using a modular design [8], [9]: Hardware can be modularized into standardized nodes that exhibit a high degree of flexibility [1], [5], and software can be modularized into components that can be combined to implement a solution that satisfies a given set of requirements.

We propose that this approach be based on a single software/hardware framework that provides a unified architecture, such that (1) the same generic technology can be used to quickly and easily provide a wide range of different hardware architectures, and (2) the robot controller interface remains invariant as the hardware architecture of the robot system evolves [10].

This paper presents the Unity-Link software-gateware stack, which connects high-level software frameworks to our modular, FPGA-based generic hardware. Unity-Link provides simple, unified abstractions for quickly and easily interconnecting a wide range of hardware and software technologies. Unity-Link connects PC-based systems with nodes that provide hard real-time control of distributed robotic systems. Unity-Link can be used with both stream- and memory-mapped interfaces, uses a component-based modular bus structure based on open standards, and interfaces with a steadily growing library of gateware components providing commonly used, realtime processing, IO-, data-storage and data-distribution cores, enabling us to create complex applications quickly and efficiently.

Our approach is based on using automated code generation to interface high-level frameworks to the underlying hardware, providing flexible, hard-realtime hardware interfaces using FPGAs, and using a real-time network to transparently distribute control across multiple nodes [1], [2], [5], [11]. The contributions of this paper concern the overall architecture of the Unity-Link generic software-hardware connection framework: a bus-based gateware architecture that exposes the underlying shared-memory model of a realtime network through a proxy model that integrates with a dataflow-based middleware. In short Unity-Link provides a free, open-source and easy-to-use architecture for connecting FPGAs to PCs through a variety of interface technologies, a feature we find to be lacking for FPGAs today. We provide a detailed performance analysis of the runtime characteristics of Unity-Link and evaluate the implementation complexity of Unity-Link compared to traditional, ad-hoc solutions.[1]

Relation to previous work: Unity-Link is the component of the Unity framework that connects the FPGA-based infrastructure to a standard PC; the Unity framework [10] is a unified software/hardware framework based on FPGAs for quickly interfacing high-level software to low-level robotics hardware, and is an evolution of the TosNet framework [1].

A. B. Lange, U. P. Schultz and A. S. Soerensen are with the Maersk McKinney Moeller Institute, University of Southern Denmark, Odense, Denmark (e-mail: {anlan, ups, anss}@mmmi.sdu.dk)

[1]The entire Unity framework is open-source and currently being made available at `https://github.com/Embedix/Unity`

II. Hardware-Software Interfaces for Robotics

A. Fundamental challenges and the role of the FPGA

Interfacing hardware to software is a fundamental challenge in robotics, in particular in those domains where new hardware and control methods are developed through experiments. We are primarily concerned with two key challenges: providing an interface to specific actuator or sensor electronics required for a given application, and bridging the gap between this interface and high-level software frameworks such as ROS that provide reusable functionality for the high-level behaviors of the robotic system. The issue of scalable real-time processing and control is critical here: although not required for all applications, it is often essential for high-performance robots interacting with the physical world.

We propose to solve the interfacing challenge using FPGAs and to bridge the gap to high-level software frameworks using a generic software-gateware infrastructure. We choose FPGAs over microcontrollers (MCUs) due to their superiority in almost all performance areas relevant to experimental robotics, except for price and library support:

- FPGAs can provide deterministic hard real-time performance no matter the complexity or scale of the implemented algorithms [9], [12], [13].
- Logic developed for a small FPGA can be moved directly to a larger device while preserving timing behavior, but code written for one MCU cannot always be moved to another larger MCU because of architectural differences.
- On an FPGA the architecture is controlled by the developer, facilitating interfacing to sensor and actuator electronics, and providing increased flexibility as well as the possibility of tailoring the gateware to reduce the need for costly software abstractions on higher levels [1], [2], [5], [11].
- FPGAs can be reconfigured allowing the architecture and IO system to be updated even after deployment. Using an MCU-based system only the algorithmic functions can be altered, and only to the extent the processor can keep up with the computation requirements; new hardware interfaces, or updates to existing ones, cannot be done without replacing the physical hardware.

B. MCU-based approaches

Many modern robotic systems use a layered architecture with a PC running a high-level robotics framework, connected to MCU-based boards that handle real-time concerns. The connection could be a point-to-point serial connection, a serial bus such as RS-485, a real-time network such as CANBUS or Ethernet PowerLink (EPL), or simply a standard network if there are no real-time requirements.

ROS and OROCOS are two of the most popular frameworks for high-level SW development within robotics. As an example of their use, Smits and Bruyninckx describe how a complex robot control application can be created using a composition of different component based SW frameworks (ROS, OROCOS/RTT, KUKA-FRI and Blender) and data flow integration [7]. ROS is described to have a simple, easy and effective data flow model, but to have inefficient coordination support. OROCOS on the other hand has a more complex data flow model but also good support for coordination through the use of FSMs. Neither of these frameworks however provide generic solutions for interfacing to low-level electronics, but instead rely on the aforementioned layered approach, for example using ROS-serial which in turn requires a custom MCU-based solution to be developed.

C. FPGA-based approaches

We use FPGA-equipped boards as generic infrastructure nodes for distributed control and HW/SW interfacing in robotics [1], [2], [5], [11]. FPGAs have however also been used in various other projects and fields of study. Fernandes et al describe an HDL-based library for data acquisition and real-time algorithms utilizing a PCIexpress interface [13]. Using this library a Transient Recorder and Processor (TRP) unit is developed for use in for example nuclear fusion experiments. The library is built as modular/componentized HDL-code with the goal to ease algorithm development for developers novice to the low-level HW details of FPGAs. Heckerling et al describe a general infrastructure for rapid development of FPGA-based control systems using the Wishbone System-on-chip (SoC) bus coupled with an Ethernet UDP-based interface [14]. The framework allows gigabit transmission rates and provides a remote-DMA (Direct Memory Access) interface to the PC user. Various other implementations of Ethernet-based communication stacks for FPGA to PC interfaces are presented in [9], [15], [16] and [17], the latter which in particular encourages the use of Component Based Software Engineering techniques. Like our work, the Wishbone SoC bus standard is a central technology in the work of Kissler et al detailing a generic framework for rapid prototyping of SoC systems [8].

Common technologies for interfacing FPGAs to PCs are USB [9], [12], [18], Ethernet [14] and PCIexpress [1], [5], [11], [13]. Ethernet-based technologies such as EPL and EtherCat are easy to connect to a PC with medium-to-high (10-1000Mbps) transmission speeds and software development is relatively simple. Alternatively, with simple USB-based interfaces readily available for CAN-based networks, it is easy to interface a PC. Modern USB-based UART interfaces can handle transmission speeds of several Mbps, and USB-FIFO interfaces reach the full potential of the USB 2.0 specification (the limited bandwidth of CAN-bus becomes the limiting factor here). PCIexpress-based interfaces have very high performance in terms of data bandwidth, but are also expensive in terms of the needed hardware infrastructure, are more complex, and more difficult to use because of the needed low-level kernel drivers.

None of these technologies however provide a software-gateware stack that bridges high-level software frameworks, such as those commmonly used in robotic MCU-based systems, to the gateware control logic of an FPGA.

D. Evaluation

FPGAs are not commonly used in robotics, we believe the reason to be partly historical: people stick to what they know,

and a well-established community already provides high-quality open-source software components, frameworks and tools for standard PC and micro-controller based equipment. Moreover, FPGAs suffer from a combination of a lack of good, open-source, vendor-independent HDL-component libraries for programming FPGAs, and a high degree of complexity associated with FPGA programming, since knowledge of low-level, internal FPGA structures is required for advanced work. To make FPGAs more useful for robotics, we believe there is a need for open-source, well-written, and well-documented HDL-component libraries. These libraries must be part of a framework that both provides an easy way to construct the FPGA gateware and provides easy access to modern, medium-to-high performance interface technologies. This is the overall goal of the Unity framework [10]. We will now present Unity-Link, which is the generic interface between our HDL-library components and the PC.

III. UNITY-LINK

We have designed and implemented Unity-Link, a generic PC-FPGA interface protocol that bridges the gap between realtime logic and high-level software without compromising ease-of-use and simplicity. Unity-Link provides direct access to an arbitrarily large shared memory address space hosted by multiple FPGA-based IP-cores, possibly distributed across multiple physical nodes connected by a realtime network. Communication with the underlying hardware is through this shared memory, which thus serves as a software-independent abstraction that can be mapped to higher level software. Unity-Link can connect to a standard PC using a USB interface, which provides reasonable performance and attaches to a standard address/data bus where the hard real-time capabilities of the FPGA can be exploited. The goal of the Unity-Link protocol and interface is to shield the user from the complexities of the underlying hardware interface and to provide a single, simple and unified interface between a high-level software framework, such as ROS running on a PC, and the application-specific gateware in an FPGA or network of FPGA's. In effect Unity-Link bridges the gap between high-level software frameworks and low-level FPGA GW using a generic software-gateware infrastructure.

The Unity-Link gateware protocol has a stream-oriented, layered and componentized design that provides flexibility with regards to interface technologies that are point-to-point (UART, FIFO, USB, etc.) and addressable streams (RS485, Ethernet, etc.). The gateware provides flexible access to the shared-memory model of the underlying real-time network, in our case the TosNet realtime network [1]. The user can interact with the Unity-Link protocol through a serial link using any general-purpose programming language or tools like MatLab. A high-level software stack for Unity-Link, written in Python, facilitates integration into a publish-subscribe computational model, and has been interfaced to the Robot framework ROS. Automated code generation directly supports both a proxy-based object-model implemented in Python and the automatic generation of a complete ROS node that links the Unity shared memory models

```
public scara(ctrl,comm) {
 TOSNET(CTRL=ctrl, COMM=comm, BASE=0);
 j1: joint("joint 1") @(0x80,0x88,0x8C,0x94);
 j2: joint("joint 2") @(0x81,0x89,0x8D,0x95);
 j3: joint("joint 3") @(0x82,0x8A,0x8E,0x96);
 j4: joint("joint 4") @(0x83,0x8B,0x8F,0x97);
}
joint(name): @(spd,cpos,pos_sp,spd_sp) {
 @cpos: signed READ(ID=name+":cpos", PUBLISH(1,10));
 @spd: READ(ID=name+":spd", PUBLISH(1,10));
 @pos_sp: signed WRITE(ID=name+":pos_sp");
 @spd_sp: WRITE(ID=name+":spd_SP");
}
```

Fig. 1. UL-spec for the SCARA case study

to appropriate ROS topics. Configuration of the complete software stack for interfacing ROS to a given hardware configuration can in this way be done using a simple declarative specification, as illustrated later in this section.

The Unity-Link gateware layer is completely deterministic, therefore the setup is capable of real-time performance as long as the selected physical interface and the connected computer are both real-time capable. The goal of Unity is however to provide the user with an easy way of moving the real-time critical parts of a design directly into hardware, ensuring hard real-time performance where appropriate [10]. The Unity-Link gateware directly supports automatic (hard real-time) isochronous sampling and publishing of data.

A. Example: PanaRobo SCARA robot and ROS

As a concrete example of a use-case we have chosen a working robot setup in our robotics lab: a PanaRobo SCARA robot, part of an experiment to measure spray nozzle characteristics [19]. Unity-Link is used to connect a standard PC to a TosNet real-time network of FPGAs that control the individual parts of the robot.

All the software is either generic or automatically generated from the Unity-Link specification (UL-spec) shown in Figure 1. The public interface `scara` defines a TosNet hardware link and four joints named `j1`–`j4`. Each joint is parameterized by a name and the memory space addresses used in the gateware interface. Joints are defined by the `joint` expansion, which also specifies automatic publishing of some of the values. These are automatically published as ROS topics, and ROS topics are automatically created for setting the joint positions. The underlying hardware is based on distributed controller nodes using TosNet. The Unity-Link to TosNet interface is completely generic. The distributed controller nodes that interface the shared memory model to various parts of the robot were developed previously [20].

The system could be extended by adding additional nodes to the TosNet network or by adding additional IP-cores to the Unity-Link address/data bus. In the latter case additional VHDL code would be required, typically in the form of component configuration and connections. We aim to provide generic VHDL components for the most common tasks and to automatically generate the required configuration and connection code based on an overall architectural specification, this is however left as future work.

B. Example: 18-DOF walker

As a simple evaluation of the ease with which software-hardware interfaces can be established, we compare two

implementations of the software-hardware interface for an 18-DOF walker robot. The original interface for this robot was developed using a traditional MCU-based approach, which required the implementation of a communication layer on the PC, a corresponding communication layer on the MCU, and a low-level controller for the MCU written in assembler. Control of the robot was independently reimplemented using the TosNet framework [2], which is now one of the components of Unity. As was the case for the PanaRobo SCARA, the Unity-Link specification language defines the high-level software interface, and generic VHDL components are configured on the gateware side such that they interface to and control the underlying hardware.

For the MCU-based solution, the implementation of the communication comprised a PC-side serial communication library, which at the time was written manually. Had an approach such as ROS-serial been used, the PC-side could have been automatically generated, similarly to the case for Unity-Link, where the complete PC-side software stack is configured and generated based on a 23-line declarative specification. On the MCU, 58 lines of C code are used to implement the communication, whereas hardware control is implemented using 82 lines of mixed C and assembly code. On the FPGA, the communication is completely generic, and the hardware control required 14 lines of configuration code to be written, plus 32 lines of boilerplate code that needed to be copy-pasted to instantiate existing components; both parts of this VHDL code are highly amenable to automated code generation, this is however left as future work.

From a practical point of view, we note that the initial software implementation was a time-consuming task, which was significantly reduced by the use of the TosNet framework, and in the end required less than 20 minutes for the FPGA design, most of which was spent on creating the .ucf file (IO pin specification) and less than 10 minutes to define the Unity-Link specification.

C. Experience

In addition to the PanaRobo SCARA and 18DOF Walker robots, we have also used Unity-Link for interfacing to two Mitsubishi PA10 7DOF robot arms [21], controlling prototypes of rehabilitative robotic training devices in [22], and numerous student projects such as ROS-based field robots and motor controllers.

The overall experience is that the gateware part of Unity-Link works reliably, has high performance, and is easy to interface to many different kinds of controller software due to the simple protocol (described in more detail in Sec. IV). Feedback from users, students and researchers alike, have all been positive, although most users until now only have been using the GW part of the framework, and have therefore had an initial software development overhead in interfacing to the serial protocol. The software stack now eliminates this issue for software written in Python and any software based on ROS, and we also plan to create bindings for other popular tools and general purpose programming languages.

D. Software to gateware interface

The Unity-Link stream-based interface is currently based on a standard FTDI FT232H IC, which can provide a data rate of 64Mbit/s while resembling a standard serial-port on the PC, it can however also support data rates of 320Mbit/s using the FTDI proprietary D2XX drivers. In order to provide the possibility of higher performance and hard real-time guarantees, the Unity framework can easily support the introduction of a memory-mapped interface such as a PCIexpress endpoint that directly integrates with the shared memory model [1] instead of the Unity-Link gateware. Using a PCIexpress interface provides a DMA connection directly from the PC to the target FPGA (the high-level Unity-Link software however does not currently support this feature).

IV. Unity-Link Gateware

Unity-Link is a stream-based interface protocol designed for the generic address/data bus architecture of the Unity framework. To ensure maximum compatibility with various FPGAs it is written entirely in VHDL, without use of vendor-generated Logic/IP-cores. The Unity-Link gateware layer is completely deterministic, capable of hard real-time performance for read/write operations as well as automatic, isochronous sampling and publishing of data.

A. Architecture

The Unity-Link gateware implements a stream-based protocol for communication directly from a PC to the FPGA fabric. Rather than directly interfacing to a specific component, like a real-time network core or a fixed-size memory, it is designed for handling communication between a PC and a generic address/data bus. This approach solves the inflexibility and resource waste problems of our previous TosNet implementation [1], [11], at the expense of being a more general interface without the possibility of dedicated protocol commands for special-purpose actions, like, e.g., commands to commit in/out registers of double-buffered memories, or transmitting asynchronous data. Such actions must instead be implemented at a higher abstraction level using general bus transactions.

Fig. 2 (left) shows a high-level architectural overview of the Unity-Link gateware communication/protocol stack. The Unity-Link gateware is built from three major modules all connected to each other using FIFOs, these modules are the physical layer, the datalink layer and the application layer. This layered and componentized design - using simple FIFO interfaces - facilitates easy change between various datalink and physical layers, enabling security and reliability, as well as connectivity requirements to different physical mediums (e.g. UART, USB-FIFO, RS485 or Ethernet) to be met.

The Unity-Link application layer implements a subscription manager which support up to 256 configurable subscription groups that can publish data over the link. Each subscription group is capable of holding up to 255 addresses, and they can be individually configured for requesting data publishing at any integer division of the subscription manager's base publish frequency. In addition, any subscription group can

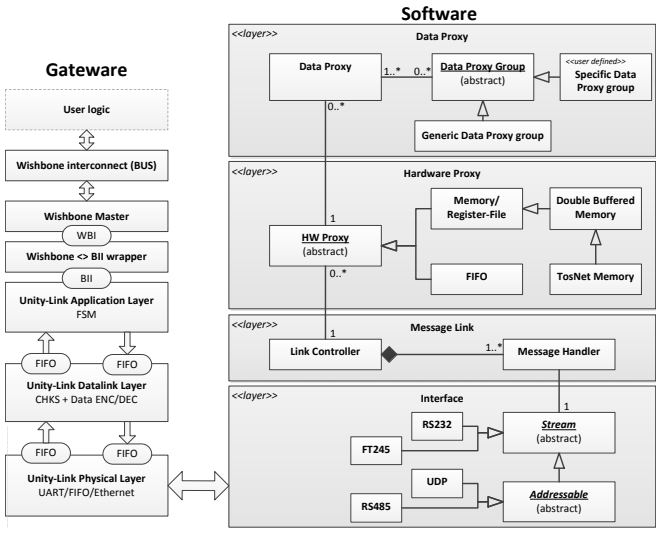

Fig. 2. Unity-Link gateware and software

also be configured to request a publish-event upon an external hardware generated interrupt, enabling publishing of data from FIFOs to happen asynchronously when available.

The datalink layer and application layer has been designed to support an intuitive and human-readable message format. This format allows arbitrarily long commands and data fields in a message, the entire English alphabet can be used in the command part of a message, while the data part is restricted to hexadecimal characters. The datalink layer can easily be extended with support for a pure binary protocol if the overhead of the current ASCII based approach becomes an issue. Moverover, automated code generation makes it feasible to consider application-specific binary protocotols for even higher performance, this is however considered future work. ASCII encoding of the protocol has been selected solely for ease of use.

Communication over Unity-Link follows a simple protocol that gives access to the shared-memory model of the underlying hardware. Read and write commands for specific memory locations can be issued from the PC, for read commands the resulting value is returned over the link in a read result message. Publishing of data can be issued from the FPGA, in messages that identify the data being pushed to the PC.

The protocol has basic commands for reading and writing data to and from the shared-memory model using; single-read (R), multiple-read (RM) which can generate an atomic read of up to 255 data-words in one transaction, and the write (W) commands, all of which can be issued with and without CRC. Apart from these basic IO commands a set of instructions for configuring, enabling/disabling publishing as well a command for reading the configuration of the Unity-Link GW exist. The latter is used for automatic configuration of the Unity-SW stack with regards to data- and address-widths, maximum read-multiple command size, number of subscription-groups, their size, and the base-publish-frequency.

Unity-Link is an interface for accessing a generic bus. In order to actually make practical use of it, it must however be interfaced to a specific System on Chip (SoC) bus. Several standard bus schemes for SoC exist; we have chosen the Wishbone bus as the default bus technology for our work with the Unity framework and Unity-Link. The reason for our choice of Wishbone is due to its inherent simplicity and the availability of free Wishbone-compatible IP-cores through the Open Cores project website. All GW components in the Unity GW library aimed at bus-interfacing are designed for easy integration with any SoC bus architecture.

B. Protocol stack implementation

The Unity-Link stack consists of the physical, datalink and application layers. The datalink layer is optional: a physical layer might not require a datalink layer, in which case it can be omitted. The configurations and experiments reported in this paper however all make use of the datalink layer.

The physical layer implements the logic necessary to interface the higher layers to the interface medium. The interface medium could be any type implementable in the digital logic of an FPGA, e.g., UART, I2C, SPI, JTAG, FIFO, USB or Ethernet interfaces (some of which require external signal conditioning logic).

The datalink layer ensures that only valid messages are passed on to the application layer. This makes it possible to implement the application layer without the need for it to rigidly verify the validity of an entire packet before beginning to process it, hence both enabling increased message processing speed and simplified error handling logic, at the expense of increased latency. The datalink layer provides CRC checksums using a standard CRC-8-CCIT checksum (any CRC checksum could be generated simply by inputting the desired CRC polynomial in binary form before synthesizing the GW).

The application layer mediates interaction between the message-based I/O and the generic address/data bus. This layer is implemented as a finite state machine (FSM). In the standard configuration, the application layer contains a subscription manager holding one or more subscription groups, and a set of states implementing the actual data publishing, as well as states for activating and deactivating the publish service, and configuring the individual subscription groups. The subscription groups can be configured with the addresses of data to be published and the rate of which to publish them (relative to the base publish frequency). The application layer FSM simply waits for a publish request from the subscription group/manager (if present) or the reception of a message header, after which it processes the request and returns any data or generated error messages. The application layer interfaces to a generic address/data bus, in other words a Bus Independent Interface (BII). In order to use the Unity-Link stack with a specific SoC bus, it is necessary to create a wrapper for the application layer BII.

All complex IP-cores in the Unity framework (Unity-Link, TosNet, double buffered memories, FIFOs, ...) are implemented with the IP-core functionality separated from the On-Chip-Bus (OCB) implementation; making it easy to switch OCB technology if desired, as advocated by Kissler [8].

V. Unity-Link Software

The Python-based software stack for Unity-Link is designed to provide a flexible platform that will enable software developers to quickly and painlessly implement high-level access to, and control of, any FPGA-based system using the Unity framework, and to enable easy integration into data-flow oriented frameworks (e.g. ROS). Code generation is used to provide application-specific bindings to Python and ROS, as well as automated configuration of the SW-stack.

Fig. 2 (right) shows a UML class diagram of the main classes and upper layers of the Unity SW stack. The Unity SW stack is divided into four main layers plus an underlying layer for the HW-interface device driver. This layered approach enables the developer to choose the best level of abstraction for the task at hand, simply by connecting to the layer with the desired abstraction-level.

The lowest two layers are the device driver and stream interface layers, direct use of these layers provides complete control over the data transmission, but requires direct management of data and requests sent across the link. The stream devicer-driver layer (**Layer 0**) is an application-specific device-driver interface to for example UART, Ethernet, etc. This layer is used by the stream interface layer (**Layer 1**), which provides a unified, simple, direct and low-level control of read and write requests to the FPGA address-space.

The next two layers provide message-based and proxy-based access models, use of these layers enables direct access to individual data elements, but lacks logical abstractions for grouping data. The message link layer (**Layer-2**) provides a message-centered, FIFO-based interface to the FPGA address-space. It handles both transmission, re-transmission and reception of responses and publish messages. The hardware proxy layer (**Layer-3**) provides high-level proxy classes for direct access to the associated GW-module in the FPGA, simplifying data access to mere relative addressing, as well as providing abstractions over module-specific behaviour that cannot be directly or efficiently mapped to the generic read/write/publish protocol of the lower SW/HW layers.

The uppermost layer is the data proxy layer which provides logical groupings of data and software-based publishing if required. Our code generator uses this layer to provide high-level abstractions to the programmer. The data proxy layer (**Layer-4**) provides classes for a high-level data-centered abstraction layer across all available HW-Proxies, even from differing streams. It is possible to create logical groups of data proxies based on any data proxy object(s), setup automated data-publishing (HW as well as SW controlled), and register user-specified callback functions on both data proxies and data proxy groups.

Apart from the Python-based high-level interface we currently also have simpler interfaces for C, C++ and MatLab.

VI. Experiments

We now describe a series of experiments designed to analyze the performance of the Unity-Link stack. We first describe the experimental setup, then report on a detailed set of read-write experiments and a simple experiment performed using ROS, and last present our analysis of the data.

A. Experimental setup

The Unity-Link GW can be configured in multiple ways and is designed to connect to any stream-capable interface. For the experiments, we have configured the GW for 32-bit data-width, 8-bit address-width, and 50MHz clock. All tests have been conducted using the FTDI FT232H chip as interface between a PC and the FPGA; it has been configured in both RS232-UART mode with baud rates of 1, 6 and 12 MBaud as well as Asynchronous FT245 mode (64Mb/s), whereas Rx- and Tx-FIFO depth of the GW-interface has been set to both 16 bytes and 256 bytes. The FT245 mode with 256-level FIFOs generally had the best performance. The PC is an Intel Core 2 DUO vPro E8400 at 2x3GHz with 2Gb of RAM running Ubuntu 12.04 LTS.

The physical-layer GW has the possibility to force the FT232H-IC to send data "immediately", i.e., to disregard USB timeout and receive-fifo thresholds in order to reduce latency, at the expense of decreased throughput. This Send-Immediate (SI) functionality has been tested for both publish-messages alone and in general for read/write responses.

We have made experiments on three basic levels to determine the performance parameters of Unity-Link: Level-one parameters have been obtained through VHDL-simulations of the GW for the R, RM, W commands as well as for the GW controlled publishing in the various configurations possible for the GW. Level-two parameters are obtained through small test-scripts written in Python, with direct access to the serial-port interface presented by the FTDI FT232H hardware. Level-three data are obtained through test-programs run directly on top of the auto-generated wrapper for the Unity-SW stack.

B. Detailed read-write experiments

The performance for a single read command at the different levels can be seen from Table I, subtables Read and Read:SI. The FTDI row is shown for reference of the expected throughput: its latency is the ideal-case latency consisting of transmission-time of request+response plus the GW-delay. For average values the standard deviation is written in parentheses and denoted "sd:". Latency describes the time from a request is sent until its response is available. Throughputs are given - relative to the PC - as input "(i)", output "(o)", input+output "(i+o)", and input-or-output "(i,o)" depending on what is most appropriate for the level and access-method used. An asterix "*" indicates an asynchronous test, meaning requests have been sent in large bursts from one thread and a second independent thread has been used to receive and store the responses.

If the GW does not use the SI functionality, the latency of a synchronous read is measured to be, on average, 974μs from a low-level python test-program, and from the Unity-SW stack to be 1072μs. If on the other hand the SI functionality is utilized, the latency drops to 223.6μs and 652μs respectively. This increase in performance however comes at the price

Read	Latency [s]	Throughput [b/s]
Unity GW	1.14	85.7×10^6 (i)
		200.0×10^6 (o)
FTDI FT245	3.524 (ideal)	67.1×10^6 (i,o)
Low-Level python	974 (sd: 120)	164.3×10^3 (i+o)
Low-Level python*		889.8×10^3 (i)*
Unity-SW	1072 (sd: 17.5)	149.2×10^3 (i+o)

Read:SI	Latency [s]	Throughput [b/s]
Unity GW	1.14	85.7e6 (i)
		200.0e6 (o)
FTDI FT245	3.524 (ideal)	67.1e6 (i,o)
Low-Level python	223.6 (sd: 9.06)	715.6e3 (i+o)
Low-Level python*		639.1e3 (i)*
Unity-SW	652 (sd: 11.7)	245.6e3 (i+o)

Write	Latency [s]		Throughput [b/s]
	req→rsp	req→effect	
Unity GW	1.26	approx. 0.84	151.3e6 (i) 54.1e6 (o)
FTDI FT245	-	-	67.1e6 (i,o)
Low-Level python	971 (sd: 76.81)	19.9 (sd: 3.77)	156.5e3 (i+o)
Low-Level python*		1.315 (burst)*	56.6e6 (o)* (30xWrite)*
Unity-SW	1058 (sd: 16.64)	-	143.7e3 (i+o)

Write:SI	Latency [s]		Throughput [b/s]
	req→rsp	req→effect	
Unity GW	1.26	approx. 0.84	151.3e6 (i) 54.1e6 (o)
FTDI FT245	-	-	67.1e6 (i,o)
Low-Level python	221.74 (sd: 5.89)	19.03 (sd: 3.07)	685.5e3 (i+o)
Low-Level python*		1.159 (burst)*	62.5e6 (o)* (30xWrite)
Unity-SW	553 (sd: 34.86)	-	274.9e3 (i+o)

TABLE I

PERFORMANCE MEASUREMENTS FOR READ-WRITE OPERATIONS

of reduced throughput for large/bulk asynchronous transfers, where the throughput of the received data drops from roughly 890 Kb/s to 640 Kb/s. The maximum size of an asynchronous transfer also drops from 200+ requests in a single transfer to just 18. The throughput for asynchronous transfers is calculated as the average time between reception of each response in the asynchronous request, divided by the response size. These throughputs seem very low, and one could suspect the USB-interface hardware and the GW of being much slower than expected. But by letting the FPGA GW timestamp the generated responses with a high resolution (50MHz) timer, it can however be seen that the asynchronous requests are actually handled at very high throughputs in the GW. For the large asynchronous transfer of 200+ requests, a mean response-data-rate of 35.3×10^6 b/s is generated, with a periodic slow-down after roughly each 36 messages (roughly every 62.14μs). The 36 messages in each period are sent at an average data rate of 64.2×10^6 b/s. The reason for this periodic slow-down is believed to be due to the output FIFOs running full, because of the periodic nature of the USB-interface. For the GW using the SI function, the maximum size deteriorates to 18 messages, but it can be seen from the GW timestamps that the responses for these 18 messages are generated with an average data rate of 121.8×10^6 b/s, which can only occur as the output FIFO in the GW before the FTDI interface is 255 bytes deep, just enough for the 252 bytes generated by the 18 requests.

Table I also shows the performance parameters for a write command (subtables Write and Write:SI). The latencies are either the time from a request is sent until a response is received (req→rsp), or the time from a request is sent to its effect on the FPGA output-pins (req→effect). From Table I it can be seen that the stateful (synchronous) write performance is very low, but that it can be drastically improved by using the SI functionality. A write has a latency of roughly 20μs (on average) from the request is sent until it takes effect on the FPGA output, subsequent writes if performed as a burst are effectuated with a period of 1.16-1.32μs depending on whether SI is used or not. Writes performed asynchronously as a burst of, e.g., 30 writes has an effective (output) throughput of 56.6 Mb/s for non-SI writes and 62.4 Mb/s

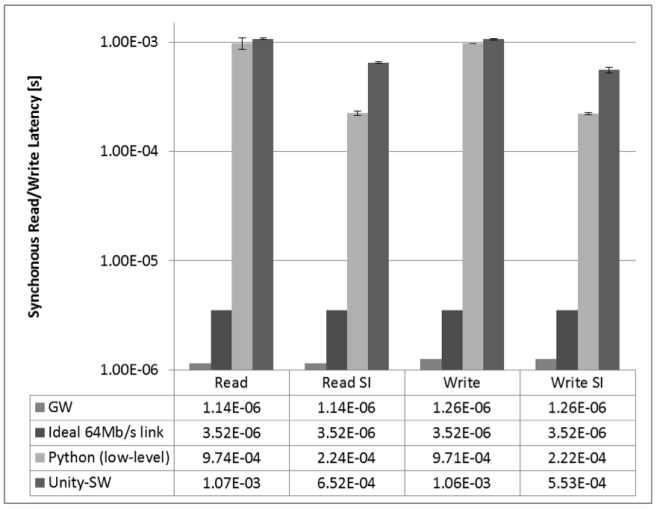

Fig. 3. Average latencies

for SI-writes. In essence SI seems to have purely a positive effect on writes, which might be because of the smaller size, and hence much lower data rate of the generated responses.

Figure 3 shows the average latencies and their standard deviation of synchronous Reads and Write from different levels (GW, ideal 64Mb/s link, low-level Python test code and Unity-SW), with and without the usage of SI.

Figure 4 shows the publish-frequency-limit versus the publish-message-size. The theoretical limit set by the GW (running at 50Mhz) as well as the lower limit set by an ideal 64Mb/s link are both plotted together with the measured burst and sustained performance data from the low-level Python test-code and the Unity-SW. It can easily be seen that the Unity-SW is much more inefficient than the low-level Python test-code, this indicates that the current Unity-SW implementation has a performance issue that needs solving.

C. ROS experiments

A detailed analysis of the performance of ROS with Unity-Link is out of the scope of this paper. Since the ROS interface is a thin wrapper around the highest level of the Unity-Link software stack, the performance is given by the performance of ROS itself, the ROS Python bindings, and the Unity-Link stack. To provide an overall idea of the performance, we have measured the rate at which data can be published to

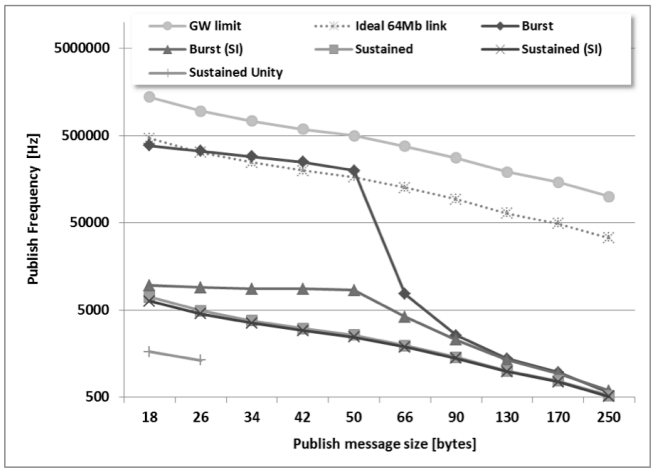

Fig. 4. Publish-frequency-limit versus publish-message-size

and from Unity-Link. The rate is simply measured using "rostopic" for a single-integer message type. Data can be published from Unity-Link to ROS at a sustained rate of roughly 1.8KHz whereas publishing data from a Python script to Unity-Link can only be done at a significantly lower sustained rate of 0.3kHz. Since ROS subscriptions are currently implemented using synchronous Unity-Link writes, the performance observed correlates with the performance of the highest levels of the Unity-Link software stack.

D. Analysis

Based on the experimental results, we conclude that the current implementation of the Unity-link protocol needs to be extended. First, support for stateless Writes would allow the PC application to write data at the full bandwidth of the link without generating responses. Second, support for an optional message-ID field should also be added, in order for the Unity-SW to be able to properly support asynchronous transmission and reception of data for increased utilization of the link bandwidth. Last, support for variable bit-width datatypes, such as booleans, bytes, word and dwords etc. is desirable on layer-4 of the SW stack for flexibility.

The GW and USB-link can support high bandwidths, but the latency of the USB link is significant when used for control applications. We believe that this can be mitigated by using publishing of all timing-critical data, using SI on publish messages to reduce their latency, and providing support for stateless writes. An additional performance enhancement would be to make use of the proprietary D2XX driver from FTDI instead of the generic virtual-com-port (VCP) driver.

VII. Conclusion & Future Work

In this paper we have presented Unity-Link, a uniquely flexible software-gateware framework for connecting high-level robotics frameworks to low-level FPGA-based electronics. The performance analysis demonstrates good overall performance, very high performance of the gateware stack, and indicates directions for improving the performance of the software stack. In terms of future work, to improve overall performance, the Unity-SW stack should probably be reimplemented in Java or C++. The use of Python has been (and remains) highly convenient for prototyping, making the initial research and development of the SW architecture much easier, at the cost of performance.

References

[1] S. Falsig and A. Soerensen, "An FPGA based approach to increased flexibility, modularity and integration of low level control in robotics research," in *IEEE/RSJ IROS 2010*, Oct. 2010, pp. 6119–6124.

[2] R. Ugilt, A. S. Soerensen, and S. Falsig, *A step toward 'plug and play' robotics with SoC technology*. World Scientific, 2010, ch. 52, pp. 415–422.

[3] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback, "Orca: A component model and repository," in *Software Engineering for Experimental Robotics. STAR 30*, D. Brugali, Ed. Springer-Verlag Berlin Heidelberg, 2007, pp. 231–251.

[4] A. Bonarini, M. Matteucci, M. Migliavacca, and D. Rizzi, "R2P: an Open Source Modular Architecture for Rapid Prototyping of Robotics Applications," in *Proc. of 1st IFAC CESCIT'12*, April 2012, pp. 68–73.

[5] A. Soerensen and S. Falsig, "A system on chip approach to enhanced learning in interdisciplinary robotics," in *IEEE/RSJ IROS 2010*, Oct. 2010, pp. 4050–4056.

[6] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[7] R. Smits and H. Bruyninckx, "Composition of complex robot applications via data flow integration," in *IEEE ICRA 2011*, may 2011.

[8] D. Kissler, A. Kupriyanov, F. Hannig, D. Koch, and J. Teich, "A generic framework for rapid prototyping of system-on-chip designs," in *CDES*, H. R. Arabnia and M. M. Eshaghian-Wilner, Eds. CSREA Press, 2006, pp. 189–195.

[9] M. Pordel, N. Khalilzad, F. Yekeh, and L. Asplund, "A component based architecture to improve testability, targeted FPGA-based vision systems," in *IEEE ICCSN 2011*, May 2011, pp. 601–605.

[10] A. B. Lange, U. P. Schultz, and A. S. Soerensen, "Unity: A Unified Software/Hardware Framework for Rapid Prototyping of Experimental Robot Controllers using FPGAs," in *SDIR-VIII Workshop at IEEE ICRA 2013*, May 2013.

[11] S. Falsig and A. Soerensen, "Tosnet: An easy-to-use, real-time communications protocol for modular, distributed robot controllers," in *ROBOCOMM 2009*, March 31–april 2 2009, pp. 1–6.

[12] S. Toscher, T. Reinemann, and R. Kasper, "An adaptive FPGA-based mechatronic control system supporting partial reconfiguration of controller functionalities," in *First NASA/ESA Conf. on Adaptive Hardware and Systems 2006*, June 2006, pp. 225–228.

[13] A. Fernandes, R. Pereira, J. Sousa, A. Batista, A. Combo, B. Carvalho, C. Correia, and C. Varandas, "HDL Based FPGA Interface Library for Data Acquisition and Multipurpose Real Time Algorithms," *Nuclear Science, IEEE Trans.*, vol. 58, no. 4, pp. 1526–1530, Aug. 2011.

[14] A. Heckerling, T. Anderson, H. Nguyen, G. Price, S. Siegal, and J. Thomas, "An ethernet-accessible control infrastructure for rapid FPGA development," *High Performance Embedded Computing Workshop 2008*, pp. 1 – 2, sep. 2008.

[15] A. Lofgren, L. Lodesten, S. Sjoholm, and H. Hansson, "An analysis of FPGA-based udp/ip stack parallelism for embedded ethernet connectivity," in *NORCHIP 2005*, nov. 2005, pp. 94 – 97.

[16] N. Alachiotis, S. Berger, and A. Stamatakis, "Efficient PC-FPGA communication over gigabit ethernet," in *IEEE CIT 2010*, 29 2010-july 1 2010, pp. 1727 –1734.

[17] T. Uchida, "Hardware-based TCP processor for gigabit ethernet," *Nuclear Science, IEEE Trans.*, vol. 55, no. 3, pp. 1631–1637, 2008.

[18] F. Jolfaei, N. Mohammadizadeh, M. Sadri, and F. FaniSani, "High speed USB 2.0 interface for FPGA based embedded systems," in *EMCom 2009*, dec. 2009, pp. 1 –6.

[19] I. Lund, J. Cortsen, and D. Solvason, "Development of a fully automated high-resolution mechanical spray patternator," *Aspects Of Applied Biology 114, 2012, page 243*, no. 114, p. 243, 2012.

[20] S. Falsig, "Interaction framework for loosely-coupled controllers," Ph.D. dissertation, University of Southern Denmark, 2012.

[21] H. Gylfason, "FPGA Based Real Time Network Bridge & Robot Control Platform," Master's thesis, University of Southern Denmark, Faculty of Engineering, The Maersk Mckinney Moller Institute, 2012.

[22] A. Soerensen, S. Chreitch, H. Olsen, and G. Sjoegaard, "Robot assisted training for rehabilitation after traffic accident a case report," *Dansk Biomekanisk Selskab*, 2011.