Title

RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators

Journal

ACM TRANSACTIONS ON RECONFIGURABLE TECHNOLOGY AND SYSTEMS, 8(4)

ISSN

1936-7406

Authors

Jacobsen, Matthew
Richmond, Dustin
Hogains, Matthew
et al.

Publication Date

2015

DOI

10.1145/2815631

Peer reviewed

RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators

MATTHEW JACOBSEN, DUSTIN RICHMOND, MATTHEW HOGAINS,
and RYAN KASTNER, University of California, San Diego

We present RIFFA 2.1, a reusable integration framework for Field-Programmable Gate Array (FPGA) accelerators. RIFFA provides communication and synchronization for FPGA accelerated applications using simple interfaces for hardware and software. Our goal is to expand the use of FPGAs as an acceleration platform by releasing, as open source, a framework that easily integrates software running on commodity CPUs with FPGA cores. RIFFA uses PCI Express (PCIe) links to connect FPGAs to a CPU's system bus. RIFFA 2.1 supports FPGAs from Xilinx and Altera, Linux and Windows operating systems, and allows multiple FPGAs to connect to a single host PC system. It has software bindings for C/C++, Java, Python, and Matlab. Tests show that data transfers between hardware and software can reach 97% of the achievable PCIe link bandwidth.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Real-Time and Embedded Systems

General Terms: FPGA, Communication, Framework, Performance

Additional Key Words and Phrases: FPGA, communication, synchronization, integration, framework

ACM Reference Format:
Matthew Jacobsen, Dustin Richmond, Matthew Hogains, and Ryan Kastner. 2015. RIFFA 2.1: A reusable integration framework for FPGA accelerators. ACM Trans. Reconfigurable Technol. Syst. 8, 4, Article 22 (September 2015), 23 pages.
DOI: http://dx.doi.org/10.1145/2815631

1. INTRODUCTION

Field-Programmable Gate Arrays (FPGAs) are being used by an ever widening audience of designers, engineers, and researchers. Tools such as Xilinx Vivado High-Level Synthesis and the Bluespec language are lowering the barriers to entry for FPGA use. Many of these uses will require high-bandwidth input and output between the FPGA and a traditional CPU workstation. When faced with this problem one can either write his/her own interconnection, license an existing solution from a vendor, or use an open source solution.

Writing an interconnect is a significant amount of work. FPGAs are flexible enough to connect to virtually any device. However, this flexibility also makes it challenging to connect to virtually any device. The protocol standards that make workstations interoperable must be implemented from the physical layer on up in order for the FPGA to interface with it. This can be a large obstacle to overcome for most designers. In many cases, implementing the interface logic can match or exceed the effort required for implementing the original application or control logic.

Authors' addresses: M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, Computer Science and Engineering Building (EBU3B), University of California, San Diego, 9500 Gilman Drive, Mail Code 0404, La Jolla, California 92093-0404; emails: {mdjacobs, drichmond}@cs.ucsd.edu, mhogains@ucsd.edu, kastner@cs.ucsd.edu.

DOI: http://dx.doi.org/10.1145/2815631

Several commercial solutions exist that can be licensed from vendors such as Northwest Logic, PLDA, and Xillybus. These solutions are PCI Express (PCIe) based. PCIe based interconnects have become the de facto standard for high-bandwidth FPGA-PC communication because of PCIe's ubiquitous presence, low latency, and scalable performance. These solutions are high performing and available for most modern FPGA devices. However, the licensing costs can be prohibitively high and often tied to vendor-specific hardware. Some offer special pricing for research based licenses, but no source is supplied.

Other commercial solutions such as Maxeler, Convey, and National Instruments provide full development environments along with their communications frameworks. This class of solutions works only with custom vendor hardware and includes custom programming languages. Similarly, the Altera OpenCL HLS solution includes support for PCIe based FPGA communication as part of their Software Development Kit (SDK) for OpenCl. The SDK compiles OpenCL kernels to FPGA primitives and produces an OpenCL execution environment that executes on a FPGA instead of a GPU. All of these frameworks are quite impressive but impose additional programming and runtime constraints, in addition to vendor hardware lock-in.

Open source connectivity solutions exist as well, such as Microsoft Research's Simple Interface for Reconfigurable Computing (SIRC) [Eguro 2010], the Open Component Portablity Infrastructure (OpenCPI) [John M. III 2009], MPRACE, and DyRACT. These solutions offer interconnections over Ethernet or PCIe. SIRC is a full solution with a high-level Application Programming Interface (API). It works out of the box with minimal configuration. Unfortunately, it only runs over 100Mb Ethernet and is only supported on Windows operating systems. This limits the bandwidth and platform. OpenCPI is designed to use either Ethernet or PCIe to connect components such as FPGAs, GPUs, or Digital Signal Processors (DSPs). The APIs are general enough to support a number of different components. This flexibility, however, makes configuring and using OpenCPI difficult and overwhelming. The MPRACE project provides a PCIe based framework that runs on Linux. It provides a low-level buffer management API and a DMA IP controller. This low-level API is usable but not as well suited for transferring data to and from software applications. Moreover, it only supports Linux platforms. Lastly, the DyRACT project provides a platform for FPGA based accelerators needing PCIe connectivity, DDR3, and partial reconfiguration. It is a nice framework for those needing partial reconfiguration, PCIe connectivity, a clock manager, and memory controller.

Despite existing offerings, none provided a no-cost, cross-platform solution without being saddled with additional features, framework restrictions, languages, or vendor hardware restrictions. A suitable, high-bandwidth, minimum-feature solution could not be found. This was the motivation that led to the development of RIFFA [Jacobsen et al. 2012].

RIFFA 2.1 is a reusable interface for FPGA accelerators. It is an open source framework that provides a simple data transfer software API and a streaming FIFO hardware interface. It runs over PCIe and hides the details of the protocol so designers can focus on implementing application logic instead of basic connectivity interfaces. It can be integrated with projects built using a variety of tools and methods. Both Windows and Linux operating systems are supported and allow communication between multiple FPGAs per host. In the 2.1 release, FPGAs from Xilinx and Altera are supported up through PCIe Gen 3.

It should be noted that RIFFA, and the solutions described previously, follow a long list of attempts to integrate FPGAs into traditional software environments. Many applications exist that solve this problem. However, these solutions are typically highly customized and do not port well to other projects without considerable rework. There are also many framework level attempts to bridge the communications divide between

CPUs and FPGA cores. Projects such as Hthreads [Peck et al. 2006], HybridOS [Kelm and Lumetta 2008], and BORPH [Brodersen et al. 2006] all address this problem. However, these solutions utilize custom operating system kernels and often only support CPUs running on the FPGA fabric.

In the sections that follow, we present a detailed description of the RIFFA 2.1 design, comparison with earlier releases, and an analysis of the architecture, and experimental performance results. This article's chief contributions are the following:

—An open source, reusable, integration framework for multi-vendor FPGAs and workstations.
—Improved packet reordering, scatter gather DMA, PCIe Gen 3, and multivendor FPGA support.
—Detailed design for PCIe based scatter gather DMA bus mastering.

2. DESIGN

RIFFA is based on the concept of communication *channels* between software threads on the CPU and user cores on the FPGA. A channel is similar to a network socket in that it must first be opened, can be read and written, and then closed. However, unlike a network socket, reads and writes can happen simultaneously (if using two threads). Additionally, all writes must declare a length so the receiving side knows how much data to expect. Each channel is independent. RIFFA supports up to 12 channels per FPGA. Up to 12 different user cores can be accessed directly by software threads on the CPU, simultaneously. Designs requiring more than 12 cores per FPGA can share channels. This increases the number of effective channels, but requires users to manually multiplex and demultiplex access on a channel.

Before a channel can be accessed, the FPGA must be opened. RIFFA supports multiple FPGAs per system (up to five). This limit is software configurable. Each FPGA is assigned an identifier on system start up. Once opened, all channels on that FPGA can be accessed without any further initialization. Data is read and written directly from and to the channel interface. On the FPGA side, this manifests as a First Word Fall Through (FWFT) style FIFO interface for each direction. On the software side, function calls support sending and receiving data with byte arrays.

Memory read/write requests and software interrupts are used to communicate between the workstation and FPGA. The FPGA exports a configuration space accessible from an operating system device driver. The device driver accesses this address space when prompted by user application function calls or when it receives an interrupt from the FPGA. This model supports low-latency communication in both directions. Only status and control values are sent using this model. Data transfer is accomplished with large payload PCIe transactions issued by the FPGA. The FPGA acts as a bus master scatter gather DMA engine for both upstream and downstream transfers. In this way, multiple FPGAs can operate simultaneously in the same workstation with minimal CPU system load.

The details of the PCIe protocol, device driver, DMA operation, and all hardware addressing are hidden from both the software and hardware. This means some level of flexibility is lost for users to configure custom behaviors. For example, users cannot set up custom PCIe Base Address Register (BAR) address spaces and map them directly to a user core. Nor can they implement quality of service policies for channels or PCIe transaction types. However, we feel any loss is more than offset by the ease of programming and design.

To facilitate ease of use, RIFFA has software bindings for

—C/C++,
—Java 1.4+,

Table I. RIFFA 2.1 Software API (C/C++)

Function Name & Description
`int fpga_list(fpga_info_list * list)` Populates the `fpga_info_list` pointer with info on all FPGAs installed in the system.

`fpga_t * fpga_open(int id)`
Initializes the FPGA specified by ID. Returns a pointer to a `fpga_t` struct or `NULL`.
`void fpga_close(fpga_t * fpga)`
Cleans up memory and resources for the specified FPGA.

`int fpga_send(fpga_t * fpga, int chnl, void * data, int len, int offset, int last, long timeout)` Sends `len` 4-byte words from `data` to FPGA channel `chnl`. The FPGA channel will be sent `len`, `offset`, and `last`. `timeout` defines how long to wait for the transfer. Returns the number of 4-byte words sent.
`int fpga_recv(fpga_t * fpga, int chnl, void * data, int len, long timeout)` Receives up to `len` 4-byte words from the FPGA channel `chnl` to the `data` buffer. The FPGA will specify an offset for where in `data` to start storing received values. `timeout` defines how long to wait for the transfer. Returns the number of 4-byte words received.
`void fpga_reset(fpga_t * fpga)` Resets the FPGA and all transfers across all channels.

—Python 2.7+, and
—Matlab 2008a+.

Both Windows 7 and Linux 2.6+ platforms are supported. RIFFA supports the following FPGA families from Xilinx (left) and Altera (right):

—Spartan 6,
—Virtex 6,
—7 Series,
—Arria II,
—Cyclone IV,
—Stratix IV–V.

RIFFA designs can make use of PCIe data bus widths: 32, 64, and 128. All PCIe Gen 1 and Gen 2 configurations up to $\times 8$ lanes are supported. PCIe Gen 3 up to $\times 4$ lanes are supported for Altera devices.

In the next sections, we describe the software interface, followed by the hardware interface.

2.1. Software Interface

The interface for the original RIFFA release [Jacobsen et al. 2012] was a complicated collection of functions that provided users with an array of data transfer options and threading models. These functions were designed under the assumption that every PC-initiated call to the FPGA would result in a transfer of data in both directions. It also required data transfers in either direction to be initiated by the PC. User IP cores would need to be designed with this paradigm in mind to function properly with RIFFA 1.0. RIFFA 2.1 does not impose such restrictions. The interface on the software side has been distilled down to just a few functions. Moreover, data transfers can be initiated by both sides; PC functions initiate downstream transfers and IP cores initiate upstream transfers. The complete RIFFA 2.1 software interface is listed in Table I (for the C/C++ API). We omit the Java, Python, and Matlab interfaces for brevity.

There are four primary functions in the API: `open`, `close`, `send`, and `receive`. The API supports accessing individual FPGAs and individual channels on each FPGA. There is also a function to list the RIFFA-capable FPGAs installed on the system. A reset function is provided that programmatically triggers the FPGA channel reset signal. This function can be useful when developing and debugging the software application. If installed with debug flags turned on, the RIFFA library and device driver provide

```
char buf[BUF_SIZE];
int chnl = 0;
long t = 0; // Timeout
fpga_t * fpga = fpga_open(0);
int r = read_data("filename", buf, BUF_SIZE);
printf("Read %d bytes from file", r);
int s = fpga_send(fpga, chnl, buf, BUF_SIZE/4, 0, 1, t);
printf("Sent %d words to FPGA", s);
r = fpga_recv(fpga, chnl, buf, BUF_SIZE/4, t);
printf("Received %d words from FPGA", r);
// Process results ...
fpga_close(fpga);
```

Fig. 1. RIFFA 2.1 software example in C.

useful messages about transfer events. The messages will print to the operating system's kernel log. RIFFA includes this functionality because visibility into hardware bus transfers can be very useful when communicating with custom-designed FPGA cores.

The software API has only one function to send data and only one to receive data. This has been intentionally kept as simple as possible. These function calls are synchronous and will block until the transfer has completed. Both take byte arrays as parameters. The byte arrays contain the data to send or serve as the receptacle for receiving data. In the send data function, the `offset` parameter is used as a hint for the FPGA. It specifies an offset for storing data at the destination. This can be used to support bulk memory copies between the PC and memory on the FPGA. The `last` parameter is used to group together multiple transfers. If `last` is 0, the destination should expect more transfers as part of the same group. The final transfer should have `last` set to 1. This grouping is entirely user specified and can be useful in situations where memory limitations require sending multiple partial transfers. Lastly, the `timeout` parameter specifies how many milliseconds to wait between communications during a transfer. Setting this value to an upper bound on computation time will ensure that RIFFA does not return prematurely. Setting a zero timeout value causes the software thread to wait for completion indefinitely.

Figure 1 shows an example C application using RIFFA. In this example, the software reads data from a file into a buffer, sends the data to the FPGA, and then waits for a response. The response is stored back into the same buffer and then processed. In this example, the same buffer is used to store the file data and the FPGA result. This is not a requirement. It just makes for a simpler example.

2.1.1. In Practice. Our experience with this interface is positive, but not without drawbacks. Removing the expectation of function-call style bidirectional data transfer allows users to design systems with more flexibility. The software interface can be easily understood as there exists a single function for each basic operation (open, close, send, receive). The interface also allows users to develop IP cores that perform either stream processing or traditional von Neumann style computing (bulk memory copies to FPGA memory).

The interface, however, does not support some common use cases very well. Specifically, many programming models require some type of read/write capability for FPGA based registers. Strictly speaking, this is supported. But it requires users to write an IP core that maps data transferred with `offset` parameters to an internal register array (for example). These register accesses require a Direct Memory Access (DMA) transfer for every read/write, which is inefficient.

Table II. RIFFA 2.1 Hardware Interface

Signal Name	I/O	Description
CHNL_RX_CLK	O	Clock to read data from the incoming FIFO.
CHNL_RX	I	High signals incoming data transaction. Stays high until all data is in the FIFO.
CHNL_RX_ACK	O	Pulse high to acknowledge the incoming data transaction.
CHNL_RX_LAST	I	High signals this is the last receive transaction in a sequence.
CHNL_RX_LEN[31:0]	I	Length of receive transaction in 4-byte words.
CHNL_RX_OFF[30:0]	I	Offset in 4-byte words of where to start storing received data.
CHNL_RX_DATA[DWIDTH-1:0]	I	FIFO data port.
CHNL_RX_DATA_VALID	I	High if the data on CHNL_RX_DATA is valid.
CHNL_RX_DATA_REN	O	Pulse high to consume value from on CHNL_RX_DATA.
CHNL_TX_CLK	O	Clock to write data to the outgoing FIFO.
CHNL_TX	O	High signals outgoing data transaction. Keep high until all data is consumed.
CHNL_TX_ACK	I	Pulsed high to acknowledge the outgoing data transaction.
CHNL_TX_LAST	O	High signals this is the last send transaction in a sequence.
CHNL_TX_LEN[31:0]	O	Length of send transaction in 4-byte words.
CHNL_TX_OFF[30:0]	O	Offset in 4-byte words of where to start storing sent data in the CPU thread's receive buffer.
CHNL_TX_DATA[DWIDTH-1:0]	O	FIFO data port.
CHNL_TX_DATA_VALID	O	High if the data on CHNL_TX_DATA is valid.
CHNL_TX_DATA_REN	I	High when the value on CHNL_TX_DATA is consumed.

Additionally, the lack of nonblocking function calls makes it cumbersome to perform common stream processing tasks. Consider the example in Figure 1. Because the calls `fpga_send` and `fpga_recv` are blocking, all the data must be transferred to the FPGA via the `fpga_send` call before the call to `fpga_recv` can be made. The upstream transfer cannot begin until the call to `fpga_recv` is made. This arrangement may be a problem if the IP core attached to the channel is designed to start sending a response before it receives all the data. Many streaming-oriented designs will produce output in this fashion and attempt to start an upstream transaction while the downstream transaction is still running. To avoid a deadlock, users will need to use two threads; one for the call to `fpga_send` and one for the call to `fpga_recv`. This allows both calls to execute in a time overlapping manner as the IP core would expect.

RIFFA would benefit from an expanded software API that supports programmed I/O and nonblocking function calls. Future versions may include these features.

2.2. Hardware Interface

The interface on the hardware side is composed of two sets of signals; one for receiving data and one for sending data. These signals are listed in Table II. The ports highlighted in blue are used for handshaking. Those not highlighted are the FIFO ports, which provide first word fall through semantics. The value of `DWIDTH` is 32, 64, or 128, depending on the PCIe link configuration.

Figure 2 shows a Verilog example of an IP core that matches the C example code from Figure 1. In this example, the IP core receives data from the software thread, counts the number of 4-byte words received, and then returns the count.

For upstream transactions, `CHNL_TX` must be set high until all the transaction data is consumed. `CHNL_TX_LEN`, `CHNL_TX_OFF`, and `CHNL_TX_LAST` must have valid values until the `CHNL_TX_ACK` is pulsed. The `CHNL_TX_ACK` pulse indicates that channel has read the parameters and started the transfer. The `CHNL_TX_DATA_OFF` value determines where data will start being written to within the PC's receiving byte array (this is the hardware equivalent to the software `offset` parameter). This is measured in 4-byte words. As described in Section 2.1, `CHNL_TX_LAST` must be 1 to signal the end of a set of transfers. This will unblock the thread waiting in `fpga_recv`. Data values asserted

```
parameter INC = DWIDTH/32;
assign CHNL_RX_ACK = (state == 1);
assign CHNL_RX_DATA_REN = (state == 2 || state == 3);
assign CHNL_TX = (state == 4 || state == 5);
assign CHNL_TX_LAST = 1;
assign CHNL_TX_LEN = 1;
assign CHNL_TX_OFF = 0;
assign CHNL_TX_DATA = count;
assign CHNL_TX_DATA_VALID = (state == 5);
wire data_read = (CHNL_RX_DATA_VALID & CHNL_RX_DATA_REN);

always @ (posedge CLK)
  case(state)
  0: state <= (CHNL_RX ? 1:0);
  1: state <= 2;
  2: state <= (!CHNL_RX ? 3:2);
  3: state <= (!CHNL_RX_DATA_VALID ? 4:3);
  4: state <= (CHNL_TX_ACK ? 5:4);
  5: state <= (CHNL_TX_DATA_REN ? 0:5);
  endcase

always @ (posedge CLK)
  if (state == 0)
    count <= 0;
  else
    count <= (data_read ? count+INC:count);
```

Fig. 2. RIFFA 2.1 hardware example in Verilog.

on CHNL_TX_DATA are consumed when both CHNL_TX_DATA_VALID and CHNL_TX_DATA_REN are high.

The handshaking ports are symmetric for both sets of signals. Thus, the behaviors are reversed with downstream transactions. The user core is notified of a new downstream transaction when CHNL_RX goes high. The user core must acknowledge the transaction by asserting CHNL_RX_ACK high for at least one cycle. The CHNL_RX_ACK pulse indicates that the user core has read the parameters and that data can now be delivered into the FIFO. This barrier serves to separate FIFO data between downstream transfers. Back to back transfers may otherwise keep the FIFO full and there would be no way to delineate transfer boundaries. As with upstream transactions, data will be made available on CHNL_RX_DATA and is valid when CHNL_RX_DATA_VALID is high. Each cycle CHNL_RX_DATA_VALID and CHNL_RX_DATA_REN are high, data present on CHNL_RX_DATA is considered consumed. The channel may present new valid data on that port the following cycle. The values for CHNL_RX_LAST, CHNL_RX_LEN, and CHNL_RX_OFF correspond to those provided via the fpga_send function call on the host PC and should be used as intended.

Timing diagrams for these signals illustrate the upstream and downstream transfers on a cycle by cycle basis. They are available on the RIFFA website: http://riffa.ucsd.edu.

2.2.1. In Practice. The interface requires providing a length value for transfers. This can be problematic in situations where the transfer length is unknown. Consider, for example, a data compression IP core that compresses data received from the host PC. To accommodate this type of situation, the core could buffer the compressed data until it is all generated and then start an upstream transfer. Another solution is to buffer data in chunks and send each in its own transfer. To avoid such workarounds, RIFFA supports early termination on upstream transfers. Designs with unknown upstream

transfer size can set CHNL_TX_LEN to an upper bound value, start the transaction, and send data as it is generated by the core. When the core finishes producing output, it can lower CHNL_TX, regardless of how much data was actually transferred. The RIFFA channel will interpret this behavior as an early termination and complete the transfer with the data sent thus far. As long as CHNL_TX is high, the transaction will continue until CHNL_TX_LEN words have been consumed.

Despite a minimal set of signals in the hardware interface, we have found that the handshaking protocol can be an obstacle in designs. It requires building a state machine to manage. Many designs simply need a simple FWFT FIFO (or AXI-4 Lite style) interface. In these designs, the handshaking and parameter information are not used but require additional logic and understanding to deal with.

2.3. Changes from RIFFA 1.0

RIFFA 2.1 is a complete rewrite of the original release. It supports most modern FPGA devices from Xilinx and Altera across PCIe Gen 1, Gen 2, and Gen 3.[1] The original release only supports the Xilinx Virtex 5 family.

RIFFA 1.0 requires the use of a Xilinx PCIe Processor Local Bus (PLB) Bridge core. Xilinx has since moved away from PLB technology and deprecated this core. The PLB Bridge core limited the PCIe configuration to a Gen 1 $\times$1 link. Additionally, the bridge core did not support overlapping PLB transactions. This did not have an effect on the upstream direction because upstream transactions are one way. Downstream transactions, however, must be sent by the core and serviced by the host PC's root complex. Not being able to overlap transactions on the PLB bus results in only one outstanding downstream PCIe transaction at a time. This limits the maximum throughput for upstream and downstream transfers to 181MB/s and 25MB/s, respectively. The relatively low downstream bandwidth was a chief motivator for improving upon RIFFA 1.0.

RIFFA 1.0 made use of a simple DMA core that uses PLB addressing to transfer data. The hardware interface exposes a set of DMA request signals that must be managed by the user core in order to complete DMA transfers. RIFFA 2.1 exposes no bus addressing or DMA transfer signals in the interface. Data is read and written directly from and to FWFT FIFO interfaces on the hardware end. On the software end, data is read and written from and to byte arrays. The software and hardware interfaces have been significantly simplified since RIFFA 1.0.

On the host PC, contiguous user space memory is typically scattered across many noncontiguous pages in physical memory. This is an artifact of memory virtualization and makes transfer of user space data difficult. Earlier versions of RIFFA had a single packet DMA engine that required physically scattered user space data be copied between a physically contiguous block of memory when being read or written to. Though simpler to implement, this limits transfer bandwidth because of the time required for the CPU to copy data. RIFFA 2.1 supports a scatter gather DMA engine. The scatter gather approach allows data to be read or written to directly from/to the physical page locations without the need to copy data.

RIFFA 1.0 supports only a single FPGA per host PC with C/C++ bindings for Linux. Version 2.1 supports up to 5 FPGAs that can all be addressed simultaneously from different threads. Additionally, RIFFA 2.1 has bindings for C/C++, Java, Python, and Matlab for both Linux and Windows. Lastly, RIFFA 2.1 is capable of reaching 97% maximum achievable PCIe link utilization during transfers. RIFFA 1.0 is not able to exceed more than 77% in the upstream direction or more than 11% in the downstream direction.

[1]PCIe Gen 3 is supported on Altera devices up to $\times$4 lanes.

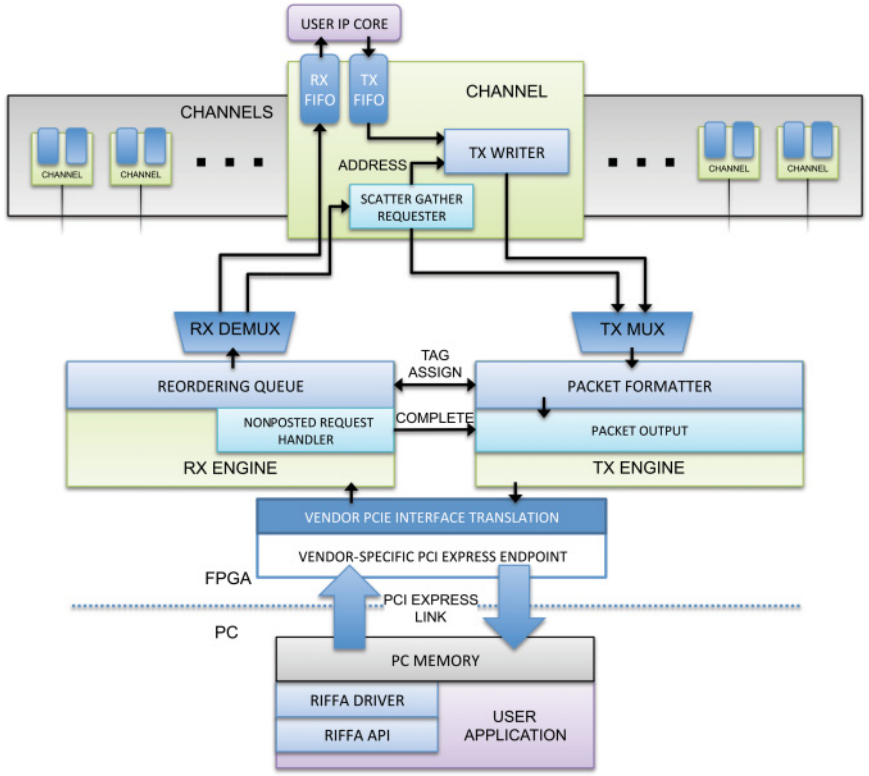

Fig. 3. RIFFA architecture.

3. ARCHITECTURE

A diagram of RIFFA architecture is illustrated in Figure 3. A goal underlying all architectural decisions is to achieve maximum throughput and minimum latency with the least amount of resources. At each level in the architecture, any potential blocking by a channel or component is averted through the use of FIFOs, multiplexing, and overlapping execution. As a result, the architecture can run at line rate, never needing to block or pause, in both upstream and downstream directions.

3.1. Hardware Architecture

RIFFA uses PCIe as the underlying transport protocol. PCIe is an address based packet protocol. Each packet has a three or four word header followed by some number of words of payload (each word is 4bytes). Data is sent using write packets. Requests for data are made by issuing read packets and receiving completion packets. Completion packets contain the requested data as payload. Device end points control the flow of these packets by advertising a credit limit for each type of packet. These credits represent how many of each packet can be accepted and processed at a time.

All these features require careful handling. For example, multiple completion packets may be returned for a single read request. These packets can arrive out of order and overlap with other completion packets. Packet headers use tag numbers as identifiers for correlating requests with completions. The tag number space is limited and must be managed fastidiously to keep the request pipeline full. Credit limits must be monitored to avoid sending too many packets or causing too many packets to be received. Lastly,

care must be taken so that data sent or requested does not cross any physical memory page boundary.

On the FPGA, the RIFFA architecture is a scatter gather bus master DMA design connected to a vendor-specific PCIe Endpoint core. The PCIe Endpoint core drives the gigabit transceivers and exposes a bus interface for PCIe formatted packet data. RIFFA cores use this interface to translate between payload data and PCIe packets. A set of RIFFA channels provide read and write asynchronous FIFOs to user cores that deal exclusively with payload data. This interface is described in Section 2.2.

The RIFFA cores are driven by a clock derived from the PCIe reference clock. This clock's frequency is a product of the PCIe link configuration. It runs fast enough to saturate the PCIe link if data were sent every cycle. User cores do not need to use this clock for their `CHNL_TX_CLK` or `CHNL_RX_CLK`. Any clock can be used by the user core.

The PCIe link configuration also determines the width of the PCIe data bus. This width can be 32, 64, or 128 bits wide. Writing a DMA engine that supports multiple widths requires different logic when extracting and formatting PCIe data. For example, with a 32-bit interface, header packets can be generated one 4-byte word per cycle. Only one word can be sent/received per cycle. Therefore, the DMA engine only needs to process one word at a time, containing either header or payload data. However, with a 128-bit interface, a single cycle presents four words per cycle. This may require processing three header packets and the first word of payload in a single cycle. If the FPGA vendor's PCIe end point supports straddled PCIe transactions, a single cycle on a 128-bit interface can contain data from two different PCIe transactions (completions for example). To achieve the highest performance, the DMA engine must be able to correctly process data for/from those two packets in a single cycle.

It is possible (and simpler) to design a scatter gather DMA engine that does not perform such advanced and flexible processing. However, the result is a much lower-performing system that does not take advantage of the underlying link as efficiently. There are many examples of this in research and industry.

3.1.1. Upstream Transfers. Upstream transfers are initiated by the user core via the `CHNL_TX_*` ports. Data written to the TX FIFO is split into chunks appropriate for individual PCIe write packets. RIFFA will attempt to send the maximum payload per packet. It must also avoid writes that cross physical memory page boundaries, as this is prohibited by the PCIe specification. In order to send the data, the locations in host PC memory need to be retrieved. This comes in the form of scatter gather elements. Each scatter gather element defines a physical memory address and size. These define the memory locations into which the payload data will be written. Therefore, each channel first requests a read of list of scatter gather elements from the host. Once the channel has the scatter gather elements, they issue write packets for each chunk of data. Channels operate independently and share the upstream PCIe direction. The TX Engine provides this multiplexing.

The TX Engine drives the upstream direction of the vendor-specific PCIe Endpoint interface. It multiplexes access to this interface across all channels. Channel requests are serviced in a round-robin fashion. The latency of checking each channel for requests is mitigated by overlapping the search for the next channel during the servicing of the current request. The TX Engine also formats the requests into full PCIe packets and sends them to the vendor-specific PCIe Endpoint. The TX Engine is fully pipelined and can write a new packet every cycle. Throttling on data writes only occurs if the vendor-specific PCIe Endpoint core cannot transmit the data quickly enough. The Endpoint may apply back pressure if it runs out of transmit buffers. As this is a function of the host PC's root complex acknowledgment scheme, it is entirely system dependent.

In practice, PCIe Endpoint back pressure occurs most during transfer of small payload PCIe packets. The vendor cores are configured with a fixed amount of header and payload buffer space. Less than maximum sized PCIe packets increases the header to payload ratio. This consumes more header buffers in the vendor Endpoint cores and may cause them to throttle the upstream transmit rate even though payload buffers exist. RIFFA was designed to send maximum sized payload packets whenever possible. Therefore, this tends to not be a problem in practice.

3.1.2. Downstream Transfers. Downstream transfers are initiated by the host PC via the software APIs and manifest on the `CHNL_RX_*` ports. Once initiated, the channel cores request scatter gather elements for the data to transfer. Afterward, individual PCIe read requests are made for the data at the scatter gather element locations. As with upstream transfers, up to the maximum payload amount is requested and care is taken to avoid requesting data that crosses physical page boundaries.[2] Care is also taken to request data so as to not overflow the RX FIFO. Each channel throttles the read request rate to match the rate at which the RX FIFO is draining. Channel requests are serviced by the TX Engine. When the requested data arrives at the vendor Endpoint, it is forwarded to the RX Engine. There the completion packet data is reordered to match the requested order. Payload data is then provided to the channel.

The RX Engine core is connected to the downstream ports on the vendor-specific PCIe Endpoint. It is responsible for extracting data from received PCIe completions and servicing various RIFFA device driver requests. It also demultiplexes the received data to the correct channel. The RX Engine processes incoming packets at line rate. It therefore never blocks the vendor-specific PCIe Endpoint core. In addition to higher throughput, this also means the number of completion credits can be safely ignored when issuing requests. Data received by the Endpoint will be processed as soon as it is presented to the RX Engine, avoiding the possibility of running out of buffer space. After extracting payload data, the RX Engine uses a Reordering Queue module to ensure the data is forwarded to the channel in the order it was requested.

3.1.3. Reordering Queue. The PCIe specification allows the host PC's root complex to service outstanding read requests in any order it chooses. This can result in requested data returning to the RX Engine out of order. To reorder the data, each PCIe read request is issued with a unique tag in the header. This tag is selected by the Reordering queue at the time the TX Engine sends the request. The channel number making the request is captured and associated with the tag. When PCIe completion packets arrive at the RX Engine, they contain the request tag in their header. The Reordering Queue stores the payload data in on-chip Block RAM (BRAM) in a location corresponding to the tag, and thus the sequence in which it was requested. Tags are issued in increasing order, wrapping at overflow. They are not reused until the received data is consumed by the channel. Received completions are processed in increasing wrapped order as well. In this manner, data is always returned to the channels in the order in which it was requested. This can be visualized as a two pointer circular tag queue.

A naive implementation of the Reordering Queue might partition the tag space between channels and use a separate FIFO for each tag. This approach will work, but can consume considerable resources and limit the throughput of each channel. The Reordering Queue instead uses a single logical BRAM to store the payloads for each tag, across all channels. The BRAM is partitioned into equal amounts, with size equal to the maximum read payload size. For the 32-bit bus width, this design is straightforward

[2]Scatter gather elements may be coalesced by the host operating system if they are adjacent and may span more than one page.

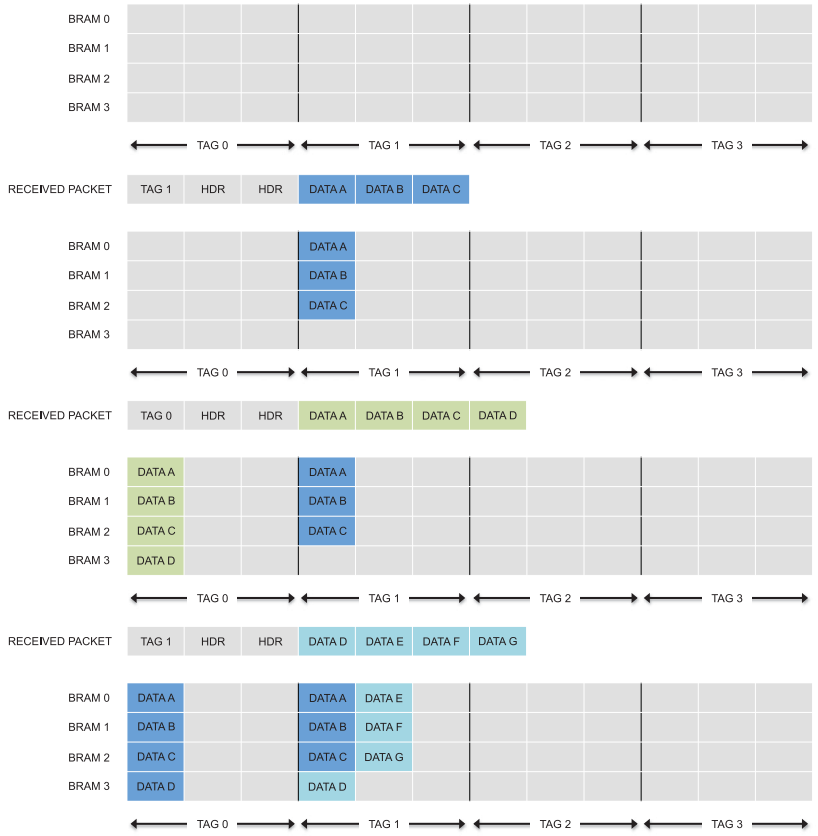

Fig. 4. Reordering Queue filling example for 128-bit-wide bus. BRAM is initially empty. Three packets arrive (sequence is top to bottom), filling the BRAM. The first and last packet are completions for Tag 1. The middle packet is for Tag 0. Data from the last packet must be written to two separate positions within the Tag 1 buffer space. This is why simply using byte/word write enables will not suffice.

as only one 32-bit word of payload can be provided each cycle. However, the 64- and 128-bit bus widths may have varying amounts of payload each cycle.

To make efficient use of the BRAM, each word must be written independently, at potentially different RAM positions and word enables. This requires a pair of BRAM ports for each word of payload provided by the bus, and independent RAM positions for each word.[3] This typically means a single BRAM for each word of bus width. A diagram illustrating the use of the Reordering Queue is displayed in Figure 4. This scheme requires less BRAMs than one that uses one per tag. There are 32 or 64 tags (depending on configuration) but only four words of payload at the widest 128-bit bus configuration.

3.1.4. Multiple Vendor Support. The 2.1 release of RIFFA includes support for Altera FPGAs. Adding support for these devices revealed that both vendors provide nearly the same interface for their PCIe Endpoint cores. Many signal names are different, but map to the same PCIe feature. There are four significant differences that could not easily be accounted for with simple connection assignment. These differences are encapsulated

[3]Data in RIFFA is transferred in 32 bit word increments. Therefore, using words as the level of granularity is sufficient.

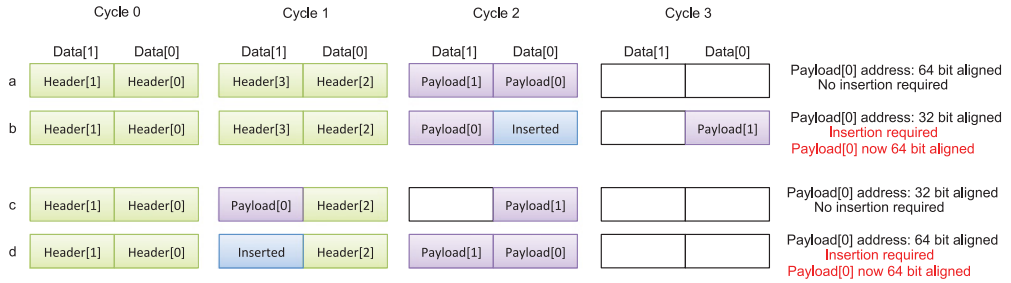

Fig. 5. Quad-word aligned PCIe packets on a 64-bit-wide bus. All examples show packets with two words of payload. Examples *a* and *b* show packets with four word headers. Payload[0] in example *a* is 64 bit aligned and appears in position 0 on the data bus. It needs no adjustment because it is 64 bit aligned. Payload[1] in example *b* is 32 bit aligned (not 64 bit aligned). It cannot appear in position 0 on the data bus, so it must be shifted by 32 bits. Examples *c* and *d* are similar but show packets with three word headers. These examples show how the payload alignment alone cannot determine whether it must be shifted.

within a vendor-specific translation layer module. Design time parameters instantiate the necessary translation layer for Altera or Xilinx. Logic required for one translation layer does not appear in designs for a different vendor.

3.1.5. Data Latency. The first difference between the two vendor interfaces is the data latency. Altera provides downstream data to the RX Engine using a FIFO similar to the Xilinx interface. However, it signals the availability of data n cycles before the actual data arrives. This is intentional to allow the receiving core n cycles to prepare, and therefore meet timing closure more easily. The value of n varies by device class. In contrast, Xilinx provides data the same cycle it signals its availability. These FWFT FIFO semantics are more common but often require data pipelining to meet timing.

3.1.6. Endianness. The second difference is word endianness. The Xilinx PCIe cores perform word endianness swapping. This has been identified in the community as a misinterpretation of the PCIe specification. However, it has permeated all of the cores we are aware of for the Spartan 6, Virtex 6, and 7 Series. Altera leaves the word endianness unchanged. This swapping only affects payload data.

3.1.7. Quad-Word Alignment. The third chief difference is the Altera requirement for quad-word aligned payload data. Altera PCIe Endpoint cores require that payload data sent or received be quad-word aligned according to the memory address on the packet. Here, quad-word refers to 64bits (8bytes). If necessary, data placed into each PCIe packet must be shifted (delayed) by 32bits to force this alignment.

To understand this requirement consider that every PCIe packet includes a memory address that corresponds to the packet's payload. RIFFA requires payloads to be 32bit aligned. This is done because the basic unit of transfer is 32bits and also as a simplification to avoid byte enable logic in the RX and TX Engines. The language bindings and kernel driver enforce this. The Altera PCIe Endpoint cores require that payload data appear on the bus in the position aligned with a 64-bit-wide memory. This is presumably useful for storing data in 64-bit-wide memories without having to shift data. Figure 5 shows a 64-bit-wide PCIe bus with data that has been made quad-word aligned. Depending on the memory address alignment and bus position, this requires shifting the payload. This can result in data being delayed until the next cycle.

It may seem possible to enforce a 64bit alignment for data transfers by altering the language bindings and kernel driver. However, this approach would not be able to account for several conditions that would result in non-quad-word aligned packets. The most difficult of which is the root complex's behavior with respect to completion packet

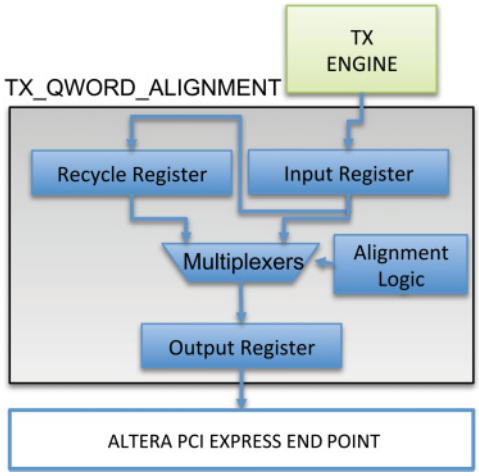

Fig. 6. Diagram of the quad-word alignment pipeline in the Altera PCIe Endpoint translation layer.

splitting. The root complex is free to split completion packets on a read completion boundary that is 64bit aligned. But as read completions have three word headers, this would result in a non-quad-word aligned packet. Therefore, the Altera translation layer module contains special handling for quad-word aligned PCIe packets.

Downstream data arriving in the RX Engine is first buffered and checked for quad-word alignment. The packet type and memory address can identify if the packet payload has been altered to be quad-word aligned. If it has, the inserted data is discarded before it reaches the RX Engine. The RX Engine receives packets corrected back to normal 32bit alignment ordering.

Upstream data going into the PCIe Endpoint requires more careful handling. Data is packed by the TX Engine into PCIe packets for each channel. Upon entering the translation layer, the packets are evaluated to see if they require modification for quad-word alignment. This evaluation determines how the packet is treated in the alignment pipeline.

The alignment pipeline contains five parts: an input register, logic to read the header and determine if an insertion is necessary, multiplexers that insert a blank word by repeating the first four bytes of payload, and a "recycle" register to save part of the payload not transmitted when a blank word is inserted, and finally, a set of output registers. The alignment logic analyzes the packet's format and address field and determines if a blank word should be inserted between the header and the payload. If required, the multiplexers switch, adding a blank word. The multiplexers are only reset when the end of a packet is reached. If the insertion forces the packet into the next cycle, back pressure is applied to the TX Engine. A diagram of this pipeline is provided in Figure 6.

We found the 64bit alignment to be difficult to accommodate, with questionable benefit. It is interesting to note, that Xilinx has included this 64bit alignment as an option in their newer PCIe Endpoint cores supporting Gen3 links. We hope that Altera makes quad-word alignment optional in the future as well.

3.1.8. Buffer Overflow. The last difference requiring handling for Altera devices is for a bug that arises during high-throughput downstream transactions. It stems largely from the quad-word alignment requirement.

During downstream transactions, the Altera Endpoint core reformats PCIe packets received from the root complex before it passes them on to the RX Engine. This reformatting conforms payload data to be quad-word aligned. It is effectively the same procedure as is performed in the alignment pipeline described in Section 3.1.7. One of the artifacts of this adjustment is that inserting a blank word into the packet may cause the packet to "spill" over into the next cycle. The extra cycle is normally not a serious concern. However, during high-throughput downstream transactions, the end point receives mostly three word completions with 64bit aligned completion data. This combination requires an inserted blank word to align the payload. These packets then require an extra cycle to transmit out of the vendor Endpoint core than it took to receive them. We found this spillage compounds under such situations and the difference in ingress and egress rates results in a packet loss and corruption.

Though subject to the same conditions, the Xilinx Endpoint cores do not suffer from this condition. This is chiefly due to the fact that they do not require payload be quad-word aligned. Additionally, their Endpoint cores support packet straddling, which would allow the spilled packets to cooccupy the data interface on the same cycle.

To avoid this problem with the Altera cores we monitor the number of completion header and data credits the core advertises. Then, when issuing read requests we calculate the number of outstanding read requests and maintain a conservative margin so as to not exhaust all the credits. This throttles the downstream rate by about 1% but ensures that the quad-word alignment has enough buffers and cycles to insert blank words. This throttling is not necessary with Xilinx Endpoint cores because the RIFFA RX Engine consumes packets at line rate (i.e., never blocks). The problem is simply that the Altera cores induce a rate difference that cannot be maintained.

3.2. Software Architecture

On the host PC is a kernel device driver and a set of language bindings. The device driver is installed into the operating system and is loaded at system startup. It handles registering all detected FPGAs configured with RIFFA cores. Once registered, a small memory buffer is preallocated from kernel memory. This buffer facilitates sending scatter gather data between the workstation and FPGA.

A user library provides language bindings for user applications to be able to call into the driver. The user library exposes the software interface described in Section 2.1. When an application makes a call into the user library, the thread enters the kernel driver and initiates a transfer. This is accomplished through the use of the `ioctl` function on Linux and with `DeviceIoControl` on Windows.

The `ioctl` and `DeviceIoControl` functions support flexible, unrestricted communication between user space and kernel space. However, they are unstructured and cannot be used directly by users. This is largely why language bindings are provided. We considered exposing the kernel driver via the standard filesystem functions: `read`, `write`, `open`, `close`, etc. This would remove the need for language-specific bindings. Users would be able to use any language and simply call filesystem functions in that language. In fact, the Xillybus driver works this way. The drawback is that any communication not modeled by the filesystem interface will not fit as easily. Functions like `fpga_reset` and `fpga_list` would be difficult to support. Moreover, language bindings provide users additional convenience by performing type conversion and checking.

At runtime, a custom communication protocol is used between the kernel driver and the RX Engine. The protocol is encoded in PCIe payload data and address offset. The protocol consists of single word reads and writes to the FPGA BAR address space. The FPGA communicates with the kernel driver by firing a device interrupt. The driver reads an interrupt status word from the FPGA to identify the conditions of each channel. The conditions communicated include start of a transfer, end of a transfer, and

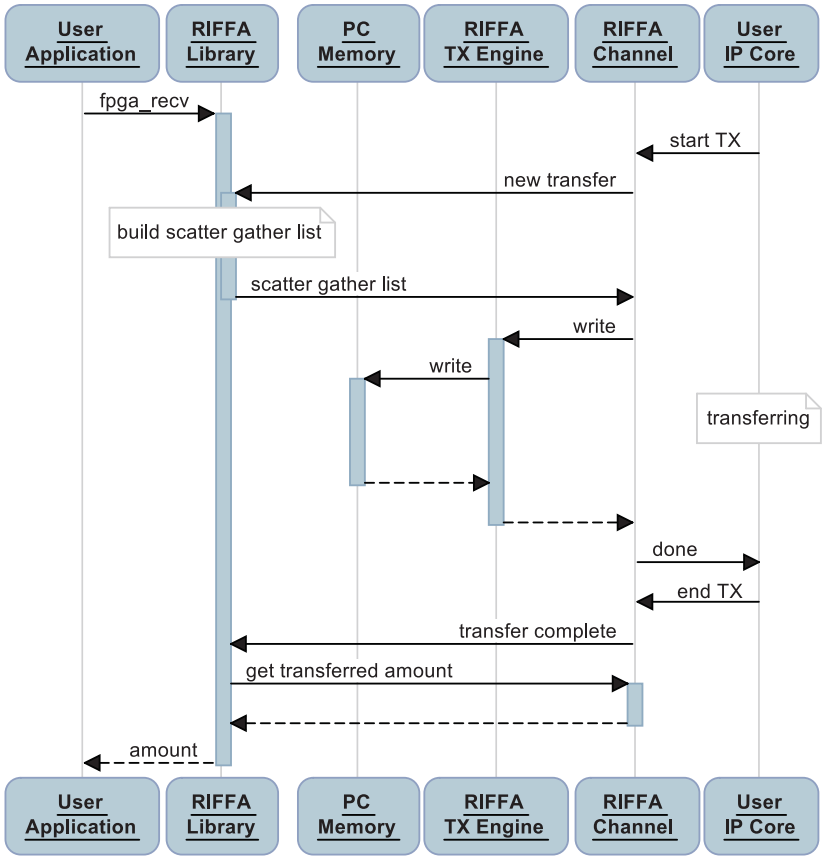

Fig. 7. Upstream transfer sequence diagram.

request for scatter gather elements. The protocol is designed to be as lightweight as possible. For example, a write of three words is all that is needed to start a downstream transfer. Once a transfer starts, the only communication between the driver and RIFFA is to provide additional scatter gather elements or signal transfer completion.

3.2.1. Upstream Transfers. A sequence diagram for an upstream transfer is shown in Figure 7. An upstream transfer is initiated by the FPGA. However, data cannot begin transferring until the user application calls the user library function `fpga_recv`. Upon doing so, the thread enters the kernel driver and begins the pending upstream request. If the upstream request has not yet been received, the thread waits for it to arrive (bounded by the `timeout` parameter). On the diagram, the user library and device driver are represented by the single node labeled "RIFFA Library."

Servicing the request involves building a list of scatter gather elements that identify the pages of physical memory corresponding to the user space byte array. The scatter gather elements are written to a small shared buffer. This buffer location and content length are provided to the FPGA so that it can read the contents. Each page enumerated by the scatter gather list is pinned to memory to avoid costly disk paging. The FPGA reads the scatter gather data, then issues write requests to memory for the upstream data. If more scatter gather elements are needed, the FPGA will request additional

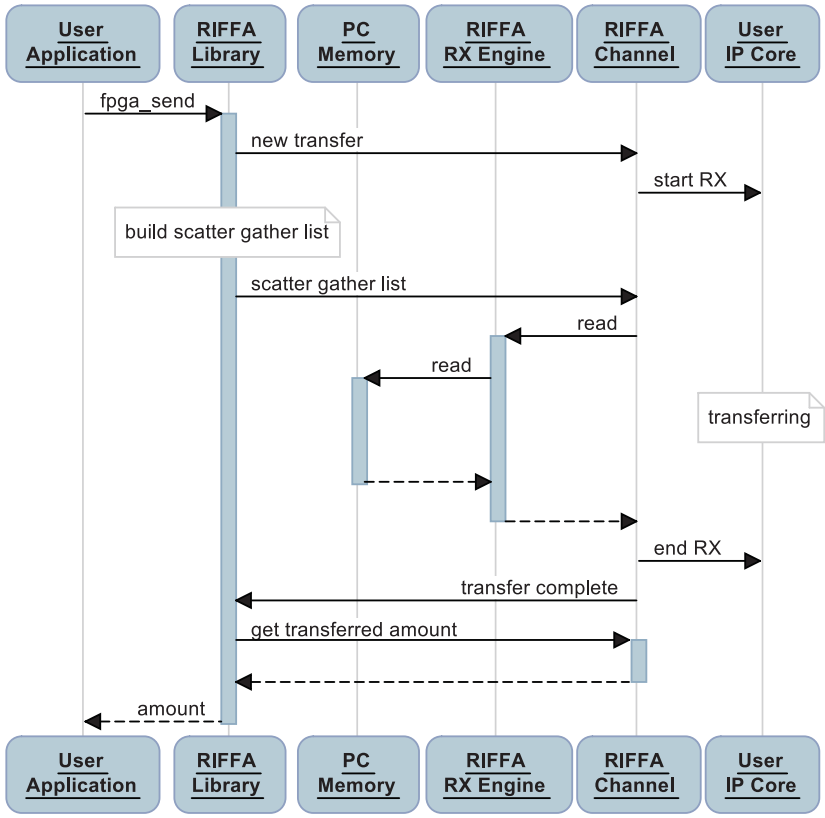

Fig. 8. Downstream transfer sequence diagram.

elements via an interrupt. Otherwise, the kernel driver waits until all the data is written. The FPGA provides this notification, again via an interrupt.

After the upstream transaction is complete, the driver reads the FPGA for a final count of data words written. This is necessary as the scatter gather elements only provide an upper bound on the amount of data that is to be written. This completes the transfer and the function call returns to the application with the final count.

3.2.2. Downstream Transfers. A similar sequence exists for downstream transfers. Figure 8 illustrates this sequence. In this direction, the application initiates the transfer by calling the library function `fpga_send`. The thread enters the kernel driver and writes to the FPGA to initiate the transfer. Again, a scatter gather list is compiled, pages are pinned, and the FPGA reads the scatter gather elements. The elements provide location and length information for FPGA issued read requests. The read requests are serviced and the kernel driver is notified only when more scatter gather elements are needed or when the transfer has completed.

Upon completion, the driver reads the final count read by the FPGA. In error-free operation, this value should always be the length of all the scatter gather elements. This count is returned to the application.

The kernel driver is thread safe and supports multiple threads in multiple transactions simultaneously. For a single channel, an upstream and downstream transaction can be active simultaneously, driven by two different threads. But multiple threads

Table III. RIFFA 2.1 FPGA Board Tested Configurations, Maximum Achieved Bandwidths, Link Utilization, and Normalized Link Utilization for Upstream (Up) and Downstream (Down) Directions. Normalized Link Utilization is Based on Maximum Negotiated Payload Limits. Highest Values for Each Interface Width are Bolded.

FPGA Board and Configuration	Max. (Up/Down) Bandwidth	Link Util.	Normalized Link Util.
AVNet Spartan 6 LX150T **PCIe Gen 1 ×1, 32-bit-wide data path, 62.5MHz**	**226/212MB/s**	**90/85%**	**96/88%**
Xilinx ML605, Virtex 6 LX240T **PCIe Gen 1 ×8, 64-bit-wide data path, 250MHz**	**1,816/1,694MB/s**	**91/85%**	**96/87%**
Xilinx VC707, Virtex 7 VX485T **PCIe Gen 2 ×8, 128-bit-wide data path, 250MHz**	**3,640/3,403MB/s**	**91/85%**	**97/88%**
Terasic DE4, Altera Stratix IV EP4SGX230KF40 **PCIe Gen 1 ×8, 64-bit-wide data path, 250MHz**	1,800/1,593MB/s	90/80%	96/85%
Terasic DE5-Net, Altera Stratix V 5SGXEA7N2F45 **PCIe Gen 2 ×8, 128-bit-wide data path, 250MHz**	3,600/3,192MB/s	90/80%	96/85%
Terasic DE5-Net, Altera Stratix V 5SGXEA7N2F45 **PCIe Gen 3 ×4, 128-bit-wide data path, 250MHz**	3,542/3,139MB/s	90/80%	96/85%

Table IV. RIFFA 2.1 Latencies

Description	Value	
FPGA to host interrupt time	3μs	$\pm$ 0.06
Host read from FPGA round-trip time	1.8μs	$\pm$ 0.09
Host thread wake after interrupt time	10.4μs	$\pm$ 1.16

cannot simultaneously attempt a transaction in the same direction. The data transfer will likely fail as both threads attempt to service each other's transfer events.

4. PERFORMANCE

We have tested RIFFA 2.1 on several different platforms. A listing of the FPGA development boards and their configurations are listed in Table III. This table also lists the maximum achieved transfer rates for each configuration. RIFFA has been installed on Linux kernels 2.6 and 3.1, as well as on Microsoft Windows 7. Our experiments were run on a Linux workstation with six 3.6GHz Intel i7 cores on an Ivy Bridge architecture, using 12 channel RIFFA FPGA designs. The user core on each channel was functionally similar to the module in Figure 2. The software was operationally similar to the example listed in Figure 1. We used PCIe cores packaged with Xilinx Vivado 2013.2, Xilinx ISE 14.1, and Altera Quartus II 13.1.

Latency times of key operations are listed in Table IV. Latencies were measured using cycles counted on the FPGA and are the same across all tested boards and configurations. The interrupt latency is the time from the FPGA signaling of an interrupt until the device driver receives it. The read latency measures the round-trip time of a request from the driver to RIFFA (through the RX and TX Engine cores), and back. The time to resume a user thread after it has been woken by an interrupt is the only value that stands out. At $10.4\mu s$ it is the longest delay and is wholly dependent on the operating system.

Bandwidths for data transfers are shown in Figure 9 (scale is logarithmic to fit all configurations). The figure shows the bandwidth achieved as the transfer size varies for several PCIe link configurations. The maximum bandwidth achieved is 3.64GB/s (using the 128-bit interface). This is 91% utilization of the theoretical link bandwidth. All devices tested reach 89% or higher theoretical bandwidth utilization. This is quite efficient, even among commercial solutions. This is largely due to the low amount of overhead the framework imposes and the fact that RIFFA can run at line rate without blocking. In practice, this can keep the PCIe link busy transferring payload data nearly every cycle.

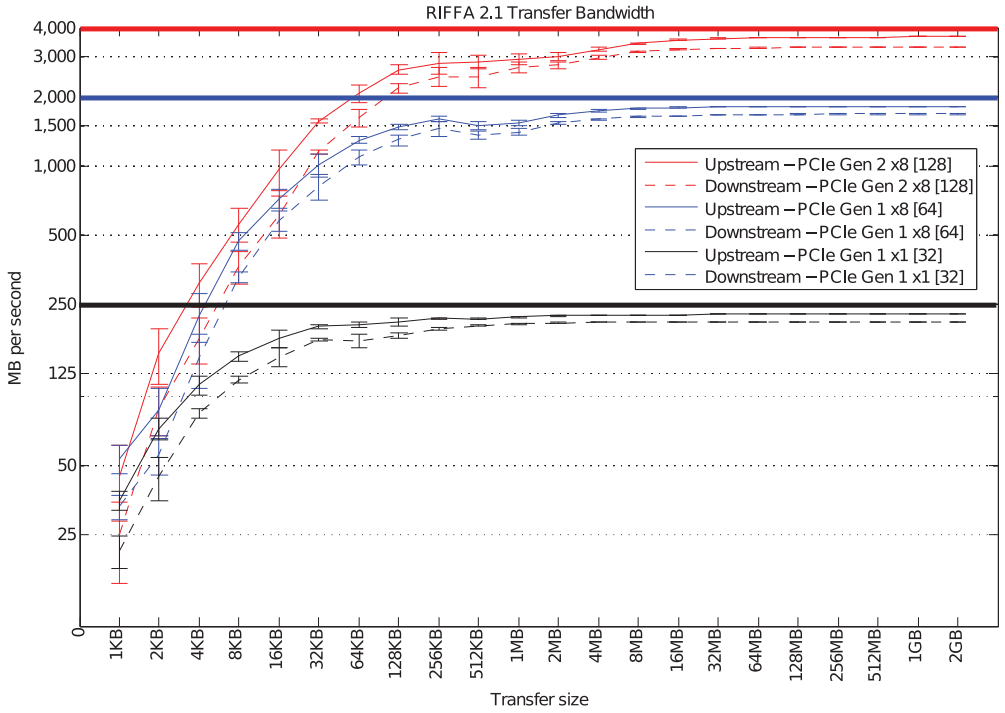

Fig. 9. Transfer bandwidths as a function of transfer size for several FPGA PCIe link configurations. Maximum theoretical bandwidths for each configuration are dawn in the same color. The upstream and downstream configurations are provided in the legend. The value in square brackets is the width of the primary bus interface.

Looking closely at the curves we notice that the upstream direction outperforms the downstream direction. This is due to the fact that the FPGA performs reads for downstream transfers and writes for upstream. Upstream transfers are made by the FPGA issuing back to back PCIe writes to the host PC's root complex. Downstream transfers are made by the FPGA issuing back to back PCIe reads. The FPGA must then wait for the host PC's root complex to respond with the requested data. Despite pipelining, this extra one-way request adds overhead as the root complex must queue and service each read request.

While not shown in Figure 9, RIFFA 1.0 was only able to achieve bandwidths of 181MB/s and 24MB/s for upstream and downstream directions, respectively. This represents a theoretical link utilization of 73% and 10%. On the same hardware and configuration, RIFFA 2.1 achieves 226MB/s and 212MB/s (or 90% and 85%) for the same directions. The relatively poor performance of RIFFA 1.0 was one of the strongest motivators for RIFFA 2.1.

The performance between the Xilinx and Altera devices is nearly evenly matched. From Table III, it appears that the Xilinx devices outperform the Altera equivalents. However, these differences are primarily in the maximum bandwidth achieved. Figure 10 shows the downstream bandwidth for Xilinx and Altera devices configured with a 128-bit interface (scale is linear to show more detail). From this figure, one can see that each device supports a slightly different maximum bandwidth. The difference in maximum bandwidths between the Altera and Xilinx Gen 2 configurations are likely due to packet straddling. On 128-bit-wide PCIe configurations, the Xilinx PCIe

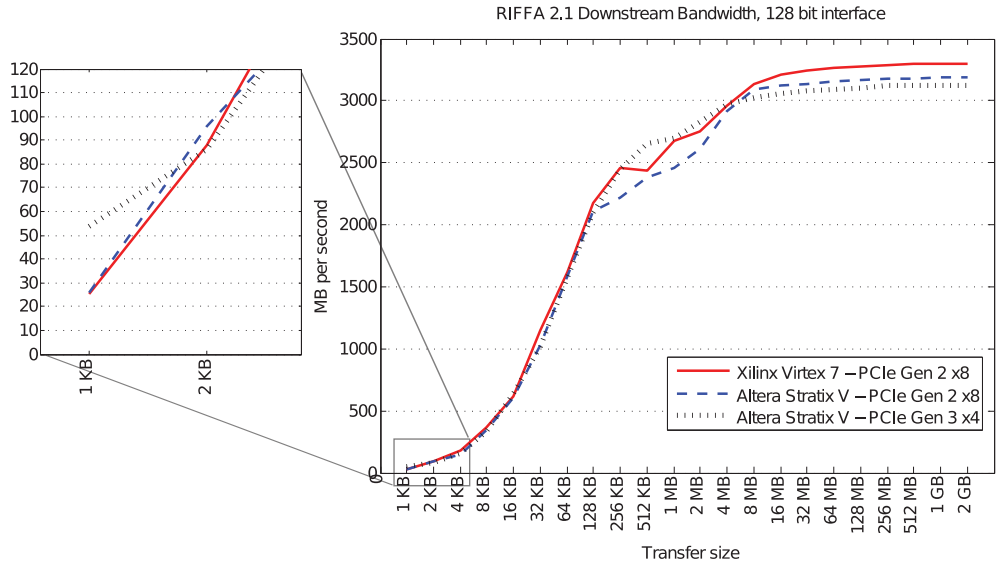

Fig. 10. Downstream bandwidths as a function of transfer size for Altera and Xilinx devices.

Endpoint core will provide multiple packets per cycle in the downstream direction. The Altera core will not, which can reduce throughput when saturated.

Within the 256KB to 2MB transfer range, the three configurations exhibit slightly different performance. The Gen 3 configuration fares best and has the most even growth. This is not surprising given the Ivy Bridge architecture of the host PC. However, the difference in performance between the Altera Gen 2 and Xilinx Gen 2 configurations is a bit puzzling. Neither behavior can be attributed to the number of advertised completion credits. Both are end-point devices and must advertise unlimited completion credits to the root complex. The Xilinx configuration performs better but has half the receive buffer size of the Altera configuration (8KB vs. 16KB). Thus, we can only presume these differences in performance stem from implementation decisions in each vendor's respective PCIe Endpoint core.

Between the Altera Gen 2 and Gen 3 configurations there is a noticeable difference in performance. The difference in maximum bandwidth attained is due to the difference in bandwidths between Gen 2 $\times 8$ and Gen 3 $\times 4$ links. The Gen 3 lanes are not quite twice as fast as the Gen 2 lanes (500MB/s vs. 984MB/s). The maximum theoretical bandwidth for the Gen 3 $\times 4$ is actually a bit less than the Gen 2 $\times 8$. Given this understanding, the Altera devices perform equally well as measured by the utilizations for both configurations.

One notable exception to this performance equivalence is the bandwidth of the Gen 3 $\times 4$ configuration during smaller transfers. Figure 10 shows a scaled view of the lower end of the transfer range with more detail. Here we see that for payloads below 2KB, the use of PCIe Gen 3 can provide double the bandwidth. Moreover, this illustrates that much of the latency in these small transfers is PCIe protocol-specific, not attributed to RIFFA.

4.1. Normalized Utilization

Theoretical bandwidth utilization is a commonly compared metric among communication frameworks as it identifies how much overhead the framework imposes. RIFFA 2.1 performs well using this metric but any utilization less than 100% invites further

Table V. RIFFA 2.1 Resource Utilization on Xilinx Devices

RIFFA (1 channel)	Slice Regs	Slice LUTs	BRAMs	DSP48Es
32bit Endpoint	4,270	3,903	12	0
additional channel	2,430	2,870	7	0
64bit Endpoint	5,350	5,110	10	0
additional channel	3,175	3,584	7	0
128bit Endpoint	7,396	7,489	16	0
additional channel	4,371	5,221	12	0

Table VI. RIFFA 2.1 Resource Utilization on Altera Devices. Altera Devices do not Support a 32-bit Interface

RIFFA (1 channel)	Regs	ALUTs	RAM bits	DSPs
64bit Endpoint	5,432	5,888	406Kb	0
additional channel	2,662	3,098	293Kb	0
128bit Endpoint	7,660	8,475	600Kb	0
additional channel	4,146	4,608	390Kb	0

improvement. A 100% utilization is impossible to achieve because it is calculated by assuming only payload data is sent every cycle. PCIe packets require headers and transfers can require packet exchanges that do not contain payload. Therefore, this is a fundamentally impossible goal.

Identifying and accounting for protocol overheads, for each test case, can be complex and difficult for comparison. However, differences in PCIe configurations between systems can yield considerably different bandwidth utilization values. To account for some of these differences and provide a point of reference for how much they can affect the theoretical utilization metric, we compare against the maximum *achievable* bandwidth. This *achievable* bandwidth is the same as the theoretical bandwidth but also recognizes the existence of packet headers. The header to payload ratio in PCIe packets can be affected significantly by differences in the maximum read/write packet payload. This maximum payload parameter varies across systems. The benefit of using this metric is that it normalizes utilization values across systems, and provides a more system agnostic value.

Using this metric, RIFFA achieves 97% of the maximum achievable bandwidth. This suggests that at most 3% is attributable to RIFFA overhead and link maintenance. In Table III we provide both the theoretical and achievable bandwidth utilizations for the upstream and downstream directions. The achievable bandwidth utilization is referred to as normalized link utilization in the table. We feel using a normalized utilization that does not vary with PCIe configuration payload limits is a comparably better metric.

4.2. Resource Utilization

Resource utilizations for RIFFA, for a single channel, are listed in Tables V and VI. The cost for each additional channel is also listed. Resource values are from the corresponding FPGA devices and configurations listed previously. Wider data bus widths require additional resources for storage and PCIe processing. It is useful to note that single channel designs use 4% or less of the FPGA on all our devices. The utilizations listed do not include resources used by the vendor-specific PCIe Endpoint core. These core utilizations can vary depending on the configuration values specified during generation and whether a hard block IP exists on the device.

4.3. Factors Affecting Performance

Many factors go into attaining maximum throughput. There is enough confusion on the topic that Xilinx has published a whitepaper [Goldhammer and Ayer Jr. 2008]. The

key components affecting RIFFA performance are transfer sizes, maximum payload limits, completion credits, user core clock frequency, and data copying.

As Figure 9 clearly illustrates, sending data in smaller sizes reduces effective throughput. There is overhead in setting up the transfer. Round-trip communication between the Endpoint core and the device driver can take thousands of cycles, during which time, the FPGA can be idle. It is therefore best to send data in as large a transfer size as resources will allow to achieve maximum bandwidth.

When generating the vendor-specific PCIe Endpoint core, it is beneficial to configure the core to have the maximum values for payload size, read request size, and completion credits. This will give RIFFA the highest amount of capacity when sending data.

The payload size defines the maximum payload for a single upstream PCIe transaction. The read request size defines the same for the downstream direction. At system start-up, the PCIe link will negotiate rates that do not exceed these configured maximum values. The larger the payloads, the more payload data each PCIe transaction can carry. This affects effective bandwidth most directly as it reduces the amount of time spent transmitting packet header data. Maximum rates can be reached by configuring the IP cores to the highest setting and letting the system determine the negotiated maximum.

Completion credits and their corresponding buffers are used in the PCIe Endpoint to hold PCIe transaction headers and data for downstream PCIe transaction requests. During downstream transfers, completion credits limit the number of in-flight requests. RIFFA does not throttle memory read requests due to completion credit availability. Because RIFFA processes completions at line rate, no data will ever be lost. However, more in-flight requests provide a greater margin for moving data from the workstation to the user core at maximum bandwidth.

The speed at which a channel's RX FIFO is drained is also a factor. RIFFA will throttle read requests for downstream data to avoid overflowing the channel RX FIFOs (throttled independently). This allows each channel to read received data at whatever rate it chooses. However, to maximize transmission bandwidth, a user core must empty its RX FIFO at the same rate (or faster) than it is filled. Using a clock with a frequency at least as high as that used by RIFFA is recommended to achieve this. Note that the user core can be clocked by any source. It need not be the same clock that drives the RIFFA.

Lastly, end-to-end throughput performance can be diminished by excessive data movement. Making a copy of a data buffer before sending it to the FPGA takes time. RIFFA's software APIs accept byte arrays as data transfer receptacles. Depending on the language bindings, this may manifest as a pointer, reference, or object. However, the bindings have been designed carefully to use data types that can be easily cast as memory address pointers and be written or read contiguously within the virtual address space without needing to be copied into the kernel (pass by reference, not value). Scatter gather DMA operations perform the actual movement in the physical address space during transfer.

5. CONCLUSION

We have presented RIFFA 2.1, a reusable integration framework for FPGA accelerators. RIFFA provides communication and synchronization for FPGA accelerated applications using simple interfaces for hardware and software. It is an open source framework that easily integrates software running on commodity CPUs with FPGA cores. RIFFA supports modern Xilinx and Altera FPGAs. It supports multiple FPGAs in a host system, Linux and Windows operating systems, and software bindings for C/C++, Java, Python, and Matlab. We have also provided a detailed analysis of RIFFA as a FPGA bus master scatter gather DMA design and an analysis of its performance.

Tests show that data transfers can reach 97% of the achievable PCIe link bandwidth. We hope RIFFA will enable designers to focus on application logic instead of building connectivity infrastructure. RIFFA 2.1 can be downloaded from the RIFFA website at http://riffa.ucsd.edu.

REFERENCES

R. Brodersen, A. Tkachenko, and H. Kwok-Hay So. 2006. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. In *CODES+ISSS'06*.

K. Eguro. 2010. SIRC: An extensible reconfigurable computing communication API. In *FCCM*, Ron Sass and Russell Tessier (Eds.). IEEE Computer Society, 135–138.

A. Goldhammer and J. Ayer, Jr. 2008. Understanding performance of PCI express systems. *White Paper: Xilinx Virtex-4 and Virtex-5 FPGAs* (2008).

J. M III. 2009. Open Component Portability Infrastructure (OPENCPI).

M. Jacobsen, Y. Freund, and R. Kastner. 2012. RIFFA: A reusable integration framework for FPGA accelerators. In *Field-Programmable Custom Computing Machines (FCCM'12)*. IEEE, 216–219.

J. H. Kelm and S. S. Lumetta. 2008. HybridOS: Runtime support for reconfigurable accelerators. In *FPGA*. ACM, New York, 212–221.

W. Peck, E. K. Anderson, J. Agron, J. Stevens, F. Baijot, and D. L. Andrews. 2006. Hthreads: A computational model for reconfigurable devices. In *FPL*. IEEE, 1–4.

Received July 2014; revised November 2014; accepted January 2015