USURP: A Standard for Design Portability in Reconfigurable Computing

James Greco, Brian M. Holland, Ian A. Troxel, Gabe Barfield, Vikas Aggarwal, and Alan D. George  
{greco, holland, troxel, barfield, aggarwal, george}@hcs.ufl.edu  
High-performance Computing and Simulation (HCS) Research Laboratory  
Department of Electrical and Computer Engineering, University of Florida  
Gainesville, FL 32611-6200

Abstract—The proprietary nature of FPGA platforms has been a hindrance to developer and user productivity since the inception of reconfigurable computing. Numerous solutions such as high-level languages, intelligent compilers, and run-time schedulers have had varying success at boosting efficiency by creating standardized methods for accessing and controlling hardware resources. However, significantly less attention has been given to compile-time unification efforts which create design portability by wrapping FPGA resources and exposing them in a standardized fashion. By normalizing software APIs and hardware interconnections under the USURP framework, application developers can focus exclusively on the intended algorithm. End users can concentrate on control flow with little or no required knowledge of underlying hardware or scheduling mechanisms. The increased productivity of such a standardized framework will be critical for continued success in application development for high-performance reconfigurable systems.

Index Terms—reconfigurable computing, FPGA compile-time tools, FPGA run-time services, hardware-accelerated applications, HPC, CARMA, USURP

I. INTRODUCTION

For maximum productivity, reconfigurable hardware design should be centered on the target application and not the physical platform. The difficulties associated with creating FPGA applications have been due to both the peculiarities of hardware description languages (HDLs) and the additional systems and services necessary to make the algorithm run on physical platforms. High-level languages have generally received the most attention due to their ability to generate hardware-independent code. Regardless of the FPGA architecture, synthesizers convert the HDL into an appropriate netlist format. The maturity of HDLs has removed many of the remaining clumsy, tedious, and technical aspects of FPGA programming. Consequently, development of the targeted algorithm itself has been removed as a bottleneck to hardware-accelerated application development. Instead, the existence of non-standard hardware/software interfaces and the lack of a unified run-time support framework have become a major hurdle to the proliferation of reconfigurable computing (RC).

Design portability is another major problem handicapping the RC field. Moving a hardware-accelerated core from one architecture to another may require a few hours to a few days of wasted effort depending on the engineer’s level of expertise, differences in the hardware platforms, and complexity of the vendor’s API. First, the software interface must be rewritten for the new API. This process can be very involved because RC platforms are not intended to be feature compatible. Second, the hardware core must be modified to fit the new architecture. This process is usually far more arduous than translating the software components of the algorithm as the interconnect between the FPGA and processor can be different (e.g. Cray XD1 uses the proprietary RapidArray, Alpha Data ADM-XRC-II uses PCI), the FPGA resources can be different (e.g. Xilinx Virtex-4s feature DSP blocks), and external resources may not be available (e.g. Nallatech BenNUEY-PCI-4e has a dedicated Ethernet interface).

If the core’s interface is not designed in a modular fashion, portions will need to be rewritten for the new platform; this includes the interface to the host processor and memory controller which are often unique to the platform. Revisions to the core’s interface are undesirable as further simulations or trial and error are required to verify correct operation. If the application fails to correctly run, often it is unknown if unstable hardware and/or software are at fault. The complexity of the transition is compounded if the hardware engineer has no experience with the new platform as there is typically a steep learning curve. Having a standard interface also has the advantage of allowing accelerated cores to be simulated under a common test harness.

Platform-specific code is unacceptable in the software field, which has largely left the details of low-level hardware behind with the introduction of high-level languages. When designing software that uses a language standard (e.g. ANSI-C), it is expected the application will seamlessly port between different operating systems and instruction set architectures by using common libraries to interface with the underlying hardware. The reconfigurable computing industry does not have the infrastructure in place so applications are typically designed with a limited number of platforms in mind. There will always be a minority of users with performance demands requiring them to work at a low-level coding level (much like assembly programming), but the majority of users are willing to work at an abstracted level to achieve higher productivity and portability. Like the early computer industry, the RC industry is currently fragmented and proprietary. RC application development is still missing the feature-rich standard libraries, a fully-featured runtime environment, and reliable middleware that power software development.

The USURP (USURP’s Standard for Unified Reconfigurable Platforms) framework’s primary goal is to provide the middleware layers necessary to achieve effortless portability. Hardware-accelerated applications need only be written once and can then be run on any USURP-supported platform. The developer can distribute applications as black box IP cores without needing to provide any additional information because the communication scheme of USURP has a standardized format. All reconfigurable platforms that have provided USURP Interoper-
ability can run the application unmodified (after a completely automated one-time compilation process). Cross-compatibility between RC platforms is achieved by mandating all applications (designed to be portable) use a standard yet richly featured communication protocol; wrappers mask the FPGA board so that it operates in the predefined manner. An extensive software API is provided to facilitate the creation of generic hardware control programs. Other USURP systems and services manage the run-time operation.

In summary, the USURP framework provides a standard for users and developers to interact with hardware. Such a unified approach is only possible due to the high-level similarities in the intent of reconfigurable platforms, which is independent of the vast differences of proprietary implementations. The Streams methodology [1] showed that intra- (and some external) FPGA communications could be expressed as a fixed size, constant rate flow of data, regardless of the application. Similarly, USURP capitalizes on the high degrees of commonality in FPGA control and resource utilization that exist independent of algorithm or platform. Much like Streams methodology, trade-offs must occasionally be made, but the end result is a universally applicable system.

The remaining sections of this paper are organized as follows. Section II describes previous and related work relevant to USURP. Section III describes the hardware/software interface and run-time communication standard developed to support the USURP framework. Low-level performance metrics and application case studies used to demonstrate the performance and flexibility of the framework are described and analyzed in Section IV. Section V summarizes the paper’s contributions and future work.

II. PREVIOUS AND RELATED WORK

To date, there does not exist a unified solution that links high-level development tools with low-level hardware. High-level language tools (e.g., C-based mappers) and graphical tools (e.g., Simulink, Viva, CoreFire) are renowned for reducing development time, but lack the standardization to be deployed on a wide variety of FPGA vendor hardware. Some top-down solutions such as SRC Computers’ Carte allow for a more seamless transition of high-level designs to hardware, but they are proprietary and only work on a handful of platforms. Standardization efforts have continued to remain isolated with no particular system gaining widespread acceptance.

OpenFPGA is a relatively new organization intended to “foster shared and open efforts to address challenges of portability, interoperability and intra-application communication for FPGA and reconfigurable applications in high performance and enterprise computing environments” [2]. This community caters to researchers, FPGA developers, hardware programmers, and software end-users who are actively seeking and contributing ideas and methodologies for reducing the complexity of reconfigurable computing. OpenFPGA hopes to promote a less proprietary and more open-source RC computing model. Despite differing employment backgrounds, research interests, and opinions about future directions for reconfigurable computing, the general consensus remains that the RC community is not at a stage of full maturity with lack of portability being the chief obstacle. The rest of this section presents a brief survey of prior systems, each attempting to standardize particular niches of the reconfigurable computing field.

The Adaptive Computing System (ACS) [3] was one of the first attempts at unifying the fragmented and proprietary FPGA landscape. Particular attention was given to compile-time standardization in order to facilitate both hardware and software portability. Communication, the major obstacle to unification, is handled through FIFO queues much like the streams methodology used in the Streams-C language [4]. The monolithic design approach was optimal during active ACS work with the SLAAC-1 and SLAAC-2 RC platforms. However, the increased deployment of vendor hardware infrastructures and drivers have decreased reliance on custom channels. Vendor communication schemes (though proprietary) provide a stable underlying medium and consequently form the basis of most USURP wrappers. Thinly veiled vendor software APIs are also used in USURP over ACS’s object-oriented models to increase performance and retain a measure of consistency with current C-language bindings for FPGA control.

IGOL (Imaging and Graphics Operator Libraries) [5] provides a compile-time and run-time framework for reconfigurable data processing applications. The primary emphasis of the IGOL system is to abstract the middleware layers of hardware-software reconfigurable systems from developers. The belief is that hardware designs should be platform (and to a degree resource) independent and software end-users should be able to execute hardware applications without knowledge of which underlying FPGA is in use. However, IGOL has several key differences from USURP. First, IGOL uses the Microsoft Component Object Model (COM) system. USURP does not require a rigid software structure but instead provides common control commands for use in larger programming paradigms (primarily C). IGOL also capitalizes on Celoxica’s RC-1000 boards and Handel-C while USURP is built upon VHDL and currently supports multiple vendor platforms.

The subject of portability has not been limited to systems supporting cross-platform development. Instead, researchers begin with a popular application or suite of algorithms and then define a modularization scheme for deployment in a variety of FPGA platforms. A version of the Basic Local Alignment Search Tool (BLAST) [6] seeks to improve the means of deploying this algorithm on arbitrary reconfigurable platforms. The ability to adapt the algorithm to an arbitrary ‘pile of reconfigurable computers’ in order to gain greater access to accelerated BLAST computation is highly desirable, particularly if the cross-platform issues can be mitigated. Unfortunately, the FPGA chip is not the only component that must be standardized for multi-platform support. The RC-BLAST authors define “A portable implementation [as] one where the nonrecurring engineering costs associated with moving the implementation from one FPGA board to another is limited to interfacing issues and do not involve reworking the design.” The hardware/software interface is standardized in the USURP framework to further reduce non-recurring engineering costs.

Dynamic reconfiguration has been an emerging area of reconfigurable computing with similarities to USURP’s run-time services. The Janus framework [7] for reconfiguration sup-
ports both FPGA resource management and also partial reconfiguration scheduling. This system recognizes the need for platform independent applications and suggests a simultaneous hardware/software design flow using Java and JHDL. In contrast, USURP’s design standards provide enough flexibility to allow separate FPGA application development and software control programming. Similarly, the RAGE reconfiguration management system [8] provides FPGA run-time support but often relies on expensive translation operations to achieve dynamic reconfiguration. Using USURP-type standardization instead could create more interchangeable applications for better partial reconfiguration.

In addition to the code portability challenge USURP directly addresses, other efforts are underway to provide the feature-rich run-time environment HPC users have come to expect. Most applications executing on HPC systems are characterized by long execution times and large data sets with hundreds or thousands of coordinating processors, data storage units, input capture instruments, and other devices. To reduce the effects of failures and resource contention that plague such large-scale systems, numerous run-time tools such as distributed operating systems and Job Management Services (JMS) have been developed to coordinate jobs in order to improve their reliability and ultimately availability. A few of the many services JMS provide include a multitasking and multi-user environment with robust job scheduling, data staging and checkpointing, performance monitoring and debug among other management features. As FPGA accelerators augment traditional resources in large-scale HPC systems there exists a need to revamp these run-time services to support FPGAs.

In a recent project, researchers at George Washington University and George Mason University developed an extension to a commercial JMS, namely the Load Sharing Facility (LSF), to include an additional “hook” for FPGAs to be considered in scheduling decisions. Effectively, a semaphore was added for each FPGA resource to tell the LSF scheduler which resources are available at any given period of time [9]. While an important first step, including FPGAs in scheduling decisions relegates their role in the system to that of black-box peripheral components. As such, only a very limited subset of the services provided by traditional JMS can be made available to jobs using the FPGA resources. The addition of FPGA accelerators to such systems should not reduce the effectiveness of typical robust yet transparent run-time infrastructure and middleware. By contrast, FPGA accelerators should be viewed as another prominent system resource and users should be provided with as full a feature set as possible no matter what the resource.

Recent research at the University of Wisconsin [10] seeks to provide a more advanced scheduling techniques such as the Multi-Constraint Knapsack Problem (MCKP) to adjust FPGA usage based on run-time metrics such as speed, area, power, etc. This approach begins to address the scheduling limits found in the previous LSF work by including FPGA-centric performance metrics in scheduling decisions. However, the performance of any scheduler will be limited by its flexibility to assign a particular algorithm to an arbitrary reconfigurable resource. The obvious solution is to ensure the run-time portability of a hardware application across all platforms. The University of Wisconsin system proposes the usage of high-level language compilers and the distribution of both hardware and software implementations of an algorithm to facilitate the creation of all necessary versions of a particular application prior to execution. The remaining obstacle then is to reduce the required effort to create all instances of the algorithm, a chief goal of the USURP framework. In addition, the scalability of this approach has yet to be tested beyond single-node implementations and the need for additional JMS features beyond job scheduling and deployment must be addressed.

A scalable, fault-tolerant run-time management service for FPGA-accelerated HPC systems is sorely lacking and a research project parallel to the USURP research has been underway at the University of Florida to address this need. The Comprehensive Approach to Reconfigurable Management Architecture (CARMA) framework is a fully-featured JMS infrastructure and middleware for FPGA-accelerated HPC systems and seeks to ease the transition from traditional-processor to multi-paradigm systems [11]. CARMA seeks to increase RC device and system usability and utilization that today’s platforms often lack. The CARMA framework is composed of numerous independent software agents that frequently communicate to schedule and execute jobs, configure reconfigurable devices [12], and gather and share system performance information among other management duties. CARMA is specifically designed for FPGA-accelerated systems, providing numerous management features for processors and FPGAs alike. CARMA does not incorporate a specific design-capture tool, programming language, bitstream generator, etc. allowing users to design and build applications in any manner they see fit. CARMA simply provides the infrastructure and corresponding services upon which any type of application can be executed while simultaneously incorporating many of the features found in traditional JMS tools. A more detailed description of CARMA is beyond the scope of this paper.

CARMA’s support for heterogeneous systems has been limited to date by the number of different configuration files that must be created to support run-time decisions. CARMA distributions prior to the development of the USURP framework left the task of developing unique configurations for each system board up to the user without regard for the level of difficulty this task entails. However, the USURP framework provides a straightforward method to produce these configurations with its “write once, run anywhere” paradigm. In addition, USURP often requires performance data to support its run-time features (e.g. API call dereferencing) which the USURP framework in and of itself does not implement. Therefore, the USURP framework will be linked to CARMA’s monitoring mechanism among other services. Also, some features of the USURP run-time API are effectively improved versions of CARMA’s Board Interface Module (BIM) which provides a hardware abstraction layer for FPGA multitasking and security among other actions, lending further credence to the need to integrate the two frameworks. The combination of USURP’s portability and CARMA’s run-time JMS features should make for a powerful infrastructure for FPGA-accelerated HPC systems.
III. USURP SYSTEM ARCHITECTURE

The USURP framework is a unified solution for multi-platform FPGA development. To support the framework, a compile-time hardware/software interface and a run-time communication standard was developed. The compile-time hardware/software interface is responsible for unifying vendor software APIs, standardizing the hardware interface to external components and the communications bus, organization of data for the user application core, and exposing the developer to common FPGA resources. The run-time communication standard handles determining whether the resources meet the application’s requirements, configuring the FPGA, detecting/handling hardware faults and interrupts, and transferring data between the host PC and FPGA. The rest of this section is a top-down description of the components of USURP framework, shown in Fig. 1.

A. Hardware Abstraction API

The Hardware Abstraction API is an optional component of the USURP framework that is designed to completely abstract the FPGA from software developers. In this model, the FPGA becomes merely another computing resource. The application developer can be completely oblivious to reconfigurable computing and still take advantage of hardware-accelerated kernels. Abstracting the hardware away from the software developer is similar to the goals of the IGOL [5] and RC-BLAST [6] frameworks. Work in this area is being pursued by researchers at the University of Florida and Honeywell on the Dependable Multiprocessor (DM) project [13]. The DM project is responsible for fundamental research on enabling high-performance computing in space, especially through FPGA acceleration of applications. Earth and space scientists, with no experience in RC development, are able to take advantage of the DM platform’s FPGA resources. The USURP framework is encapsulated into a library of linear algebra and signal processing kernels that is familiar to the scientific community. The DM project has chosen the GNU Scientific Language (GSL) as the foundation for the Hardware Abstraction API, but another framework could just as easily be substituted to better suit the intended audience. While analysis of the Hardware Abstraction API is beyond the scope of this paper, it is the subject of [14].

B. Universal FPGA Software API

The Universal FPGA Software API is a unified programming model for hardware-accelerated application development. The API provides a common software interface to FPGA vendor hardware, enabling developers to concentrate on algorithm development and not vendor-specific interfaces. The API is designed to unify setup, configuration, and data transfers between the host PC and FPGA. The API can be further extended to support vendor-specific features with the Extended FPGA Software API. The API is briefly discussed in the following section.

While similar in purpose, FPGA vendor APIs are incompatible with each other. There exists no widely used standard; developing an application for an RC platform requires using that company’s proprietary API. Moving the application between architectures is not always a simple global search and replace, as there are often subtle differences in memory organization and levels of feature support.

In designing the API, we have attempted to keep the explicit library calls universal to all FPGA platforms. The API’s functionality includes but is not limited to obtaining the handle of the FPGA accelerator, configuration, acquiring pointers to memory locations inside the FPGA, directly accessing memory inside the FPGA, transferring large amounts of data through DMA, and releasing FPGA resources. As an example, a code fragment from an FFT core that one of the authors developed under the USURP framework is provided below:

```c
/* acquire handle to FPGA resource */
fpga_handle = USURP_Init();
/* load FPGA bitfile */
USURP_Load(fpga_handle, fft_bitfile);

/* acquire register file pointer */
reg = USURP_Get_reg_pointer(fpga_handle);
/* acquire FPGA SRAM pointer */
ram = USURP_Get_ram_pointer(fpga_handle);

/* set FFT size */
reg[reg_N] = 1024;
/* copy array p to FPGA SRAM */
USURP_Memcpy(fpga_handle, ram, p, 1024);
```
/* Use the register file to communicate the start of the FFT computation and to wait until completion */

/* copy FPGA SRAM to array p */
USURP_Memcpy(fpga_handle, p, ram, 1024);

/* release FPGA resources */
USURP_Finalize()

The variables reg and ram are pointers to a block of registers and SRAM respectively memory mapped inside the FPGA. The variable p is a pointer to an array that holds the data used by the FFT algorithm. Similar to software development, the details of the memory organization (internal block RAM, external SDRAM, multiple levels of cache, etc.) and the target FPGA are unnecessary at this level of design. At the FPGA Software API level we are primarily concerned with compatibility between FPGA platforms; the Extended FPGA Software API can be used to tune performance for a specific platform.

With a unified programming model, cores developed under the USURP framework can be seamlessly moved between different RC platforms. The previous FFT example could be run on any vendor’s platform without modification. The API is designed so that it is unnecessary to target a specific hardware platform at compile-time; vendor-specific API references can be dynamically linked at run-time. Applications do not need to be modified or even recompiled (if implemented on the same instruction set architecture) when porting to another RC platform. As an extension, distributed programs can concurrently take advantage of different FPGA vendor resources without needing to separately redesign the software.

The mapping of the Universal FPGA Software API to an FPGA vendor’s API is accomplished through a software wrapper. The wrapper unifies different vendor API implementations under a common architecture. This mapping ranges in complexity from a direct implementation to a more complex sequence that emulates an unavailable feature.

C. Extended FPGA Software API

Until now, all discussions of the USURP framework have concerned the standardization of common elements featured in most FPGA-based systems. In practice, these universal components have all been related to communication between the reconfigurable platform and a software control processor where many issues preventing portability arise. However, the Universal FPGA Software API will not satisfy the needs of all application developers. Despite the overwhelming similarities that exist between two arbitrary RC platforms, there will be instances where standardization is either detrimental to performance or physically impossible. Generally, the existence of proprietary hardware/software interconnects or specialized resources (not necessary for hardware/software communication) will force designs to contain some non-standard elements. By expanding the Universal API with the Extended FPGA Software API, the USURP framework can continue to reduce design overhead while allowing developers to optimize performance for particular platforms. Careful addendums to the USURP framework can allow later platforms to reuse these extensions to quickly integrate their specialized components, thereby creating class-specific portability.

The Cray XD1 system is a candidate for the extended standard. The system features a unique and powerful RapidArray Interconnect between the 12 Opteron processors and up to 6 Virtex II Pro FPGAs per chassis. This low-latency communication channel greatly alters the typical methodology behind FPGA programming by integrating multiple FPGAs, MPI, and other high-performance features. However, the communication channel itself is not the roadblock to standardization. The challenge is that the RapidArray interconnect permeates the entire system, allowing all hardware/software blades to communicate. Designers have the freedom (and subsequently the expectation) of using arbitrary processor to processor, processor to FPGA, and FPGA to FPGA communication. The Universal API can continue to be used for supporting standard processor to FPGA communication on the XD1 platform. Facilitating FPGA to FPGA communication would require use of the Extended FPGA Software API and Extended Hardware Wrapper.

D. Software Fault Manager

The final component of the USURP’s software architecture is the Software Fault Manager. The USURP framework is designed for a wide variety of platforms that do not have identical hardware components. This generality demands the USURP framework notify the user if a hardware or software exception occurs. Exception handling must be built into discovery (e.g. Does the target platform have USURP compatible hardware?), configuration (e.g. Is the bitfile designed for the targeted FPGA?), and data addressing (e.g. How are page faults handled?).

Applications should be designed (but as with software applications, this is not a requirement) to have flexible memory requirements. An exception occurs if the user tries to access an area of memory that does not exist on the RC platform (e.g. external SRAM). Preferably this exception originates from the Universal Hardware Wrapper through an interrupt but can fallback on software look-up tables if necessary. It is left to the user to handle the exception, although exception handling could be automated with a virtual map in the future. Alternatively, a cluster management system (e.g. CARMA) could use the exception information to retask the application to a more appropriate RC board.

E. Universal Hardware Wrapper

The Universal Hardware Wrapper (Fig. 2) is the hardware companion to the Universal FPGA Software API. The wrapper provides a unified interface to the host PC, internal components, and external hardware. There are a wide variety of PC/FPGA communication interfaces available (e.g. PCI, RapidArray, and Ethernet) and an even larger number of implementations (e.g. Nallatech uses a PCI core while Alpha Data uses a dedicated bridge chip). Abstracting the details of the communication bus is important for modularity in core design. In light of these concerns, the USURP framework eliminates direct access to the communication bus from the hardware core. All data access...
Fig. 2. The Universal Hardware Wrapper is designed to give core developers a wide degree of freedom with access to 32 registers, internal block RAM, external SRAM (if available), interrupts, and Demand-mode DMA. Other platform-specific features can be supported through the Extended Hardware Wrapper.

Table 1: Fundamental latencies for an application core designed with the USURP framework

<table>
<thead>
<tr>
<th>Interface</th>
<th>Latency (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core-Register Read/Write</td>
<td>0/1</td>
</tr>
<tr>
<td>PC-Register Read/Write</td>
<td>5/4</td>
</tr>
<tr>
<td>Core-BRAM Read/Write</td>
<td>2/1</td>
</tr>
<tr>
<td>PC-BRAM Read/Write</td>
<td>5/4</td>
</tr>
<tr>
<td>Core-Ext. RAM Read/Write</td>
<td>3+/2+</td>
</tr>
<tr>
<td>PC-Ext. RAM Read/Write</td>
<td>7+/6+</td>
</tr>
<tr>
<td>Core-DDMA Read/Write</td>
<td>2/1</td>
</tr>
<tr>
<td>PC-DDMA Read/Write</td>
<td>3/2</td>
</tr>
</tbody>
</table>

is handled through a register file, internal/external RAM, and input/output FIFOs.

Data transfers are primarily handled through registers and RAM. The Universal Hardware Wrapper has 32 dedicated registers with a user-configurable bit width of 32 or 64 bits. To give maximum flexibility to the core designer, all 32 registers can be read and written in a single clock cycle. A larger amount of memory (typically 4 kB, but this can be user-defined) is made available through internal block RAM. Additional registers and block RAM can be integrated into the core as needed, but cannot be directly accessed by the host PC. External memory can also be interfaced by the PC and core through a dual-port memory controller depending on hardware availability. The Universal Hardware Wrapper organizes the different memory components into a unified memory map that can be modified by the PC.

There can be a loss of performance if the core does not have direct access to the communication bus, as all data must first be transferred to registers or memory. To alleviate this concern, the USURP framework supports Demand-mode DMA (DDMA) transfers. DDMA transfers allow the core to initiate a data burst transfer or request a data burst transfer from the PC. A 2 kB input FIFO and 2 kB output FIFO (one block RAM each), attached to the communication bus, can be used in streaming applications to reduce latency caused from transferring an entire block of data to RAM. DDMA is hardware-dependent so this feature is emulated in platforms that do not directly support it.

Table 1 summarizes the fundamental latencies an application core written under the USURP framework will incur. The internal (BRAM, DDMA, register file) latencies are consistent between the two platforms that the USURP framework has been implemented under, but a core should not be explicitly written to expect a specific timing delay. The external SRAM latencies are listed for ZBT Synchronous SRAM; SDRAM or other types of external memory will incur further penalties.

It should be noted that in the description of the USURP system architecture we often refer to a host PC controlling the FPGA as a coprocessor. This distinction is completely arbitrary and only used to better understand USURP’s compile-time and run-time software services. The Universal Hardware Wrapper can interface an external sensor (e.g. a camera) or an external communications device (e.g. Ethernet) to receive data and commands just as easily as a PCI bus. To the application core, the details of the memory and register interfaces are abstracted.

F. Extended Hardware Wrapper

The Extended Hardware Wrapper is the compliment to the Extended FPGA Software API. The extended wrapper allows the developer to take advantage of non-standard features in FPGA platforms under the USURP framework. The accelerated core has several options for interfacing an extended hardware device, the core can be given a direct interface to the hardware resource, the device’s registers and memory can be integrated in the wrapper’s memory map, and/or streaming devices can be interfaced through the DDMA FIFOs. The first option eliminates the write once, run anywhere paradigm, but is often easier to integrate with an application core. The last two options will not guarantee correct operation on every platform, but should
not require a redesign of the core’s interface to the Universal Hardware Wrapper.

An example of a feature that the Extended Hardware Wrapper can take advantage of is Ethernet. Ethernet is a common, albeit non-universal, feature for reconfigurable platforms. The Nallatech BenNUEY-PCI-4e platform features four Ethernet ports in addition to the standard PCI communication bus. The Ethernet controller can be integrated with the DDMA FIFO queues to maintain the same interface to the Universal Hardware Wrapper and prevent unnecessary redesign efforts. Alternatively, the core can be given direct access to the Ethernet controller which will necessitate the redesign of the wrapper’s interface to the core. More complicated systems involving on-chip PowerPCs and embedded coprocessors are being considered as future extensions to the Extended Hardware Wrapper.

IV. EXPERIMENTAL RESULTS

In the first iteration of the USURP architecture we have targeted two popular reconfigurable computing platforms, the Nallatech BenNUEY-PCI and Alpha Data ADM-XRC-II. At a high-level, these platforms are very similar. Both feature Xilinx Virtex II FPGAs and are connected to the host processor through a PCI interconnect. However, several subtle differences make these platforms difficult to unify under a single framework. The ADM-XRC-II platform uses a PLX 9656 PCI bridge and the BenNUEY-PCI uses a Spartan-II with PCI firmware to interface with the PCI bus; the PCI bridges present different interfaces to the FPGA and are incompatible with each other. The software that the FPGA can take advantage of is Ethernet. Ethernet is a common, albeit non-universal, feature for reconfigurable platforms. The Nallatech instead provides a GUI (DIMEtalk) to layout various pre-built hardware components necessary to support their proprietary wrapper standard. A 4 kB BRAM and a 32-bit register file which can be partially disabled at compile-time. In addition, the ADM-XRC-II’s memory controller is larger than most as it features six independent external SRAM banks. The block RAM can also be partially or completely disabled at compile-time if an application does not require addressable SRAM or DMA transfers.

The USURP hardware/software interface is designed to have a small impact on DMA and direct slave throughput. To categorize data throughput, we compared the USURP framework to hardware cores, provided by the RC vendors, that allow the PC to interface internal and external SRAM. On the ADM-XRC-II platform, we used the zbt core provided in the Alpha Data SDK. The zbt core allows the host PC to access the FPGA’s external ZBT SRAM through a DMA transfer and internal registers through a direct slave transfer. An SDK program to measure throughput was not available for the BenNUEY-PCI. Nallatech instead provides a GUI (DIMEtalk) to layout various pre-built hardware components necessary to support their proprietary wrapper standard. A 4 kB BRAM and a 32-bit register were interfaced to the PCI bus through a router component in DIMEtalk.

Table 3 describes the direct slave throughput for the ADM-XRC-II and BenNUEY-PCI platforms. The USURP framework and the Alpha Data SDK have identical write throughput. On the ADM-XRC-II platform, the write throughput is completely

| Table 2: Hardware resources dedicated to the USURP framework on the ADM-XRC-II (X2V3000) and BenNUEY-PCI (X2V6000) |
|------------------|------------------|------------------|
| Resource         | ADM-XRC-II | % Chip | BenNUEY-PCI | % Chip |
| Block RAM        | 8 kB       | 4.2    | 8 kB        | 2.8    |
| Addressable RAM  | 4 kB       | 2.1    | 4 kB        | 1.4    |
| DDMA Input FIFO  | 2 kB       | 1.1    | 2 kB        | 0.7    |
| DDMA Output FIFO | 2 kB       | 1.1    | 2 kB        | 0.7    |
| Flip-Flops       | 1640       | 5.7    | 1451        | 2.1    |
| Register File    | 1024       | 3.6    | 1024        | 1.5    |
| Memory Controller| 560        | 1.9    | 95          | 0.4    |
| LUT              | 1216       | 4.2    | 1197        | 1.8    |

A. Low-level Performance Metrics

The USURP framework is designed to expose the hardware developer to common FPGA resources in a unified fashion. The Universal Hardware Wrapper’s objective is to provide the core with compatibility between RC platforms and a fast interconnect to the host PC and memory. The objectives are listed in order of importance with compatibility being the main thrust of this paper. However, the authors understand a universal framework is not going to be adopted if it severely impacts performance or limits the size of application cores. Three performance metrics, namely area, latency, and throughput, are measured for the Alpha Data ADM-XRC-II and Nallatech BenNUEY-PCI platforms in this section.

The Universal Hardware Wrapper occupies a very small hardware footprint. Table 2 lists the maximum resources dedicated to the wrapper on the ADM-XRC-II and BenNUEY-PCI platform. The majority of the flip-flops are taken up by a 32x32 register file which can be partially disabled at compile-time. In addition, the ADM-XRC-II’s memory controller is larger than most as it features six independent external SRAM banks. The block RAM can also be partially or completely disabled at compile-time if an application does not require addressable SRAM or DMA transfers.

Table 3: Direct slave throughput on the Alpha Data ADM-XRC-II and Nallatech BenNUEY-PCI (PCI bus at 66 MHz)

<table>
<thead>
<tr>
<th></th>
<th>Read (MB/s)</th>
<th>Write (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADM-XRC-II USURP</td>
<td>3.73</td>
<td>25.2</td>
</tr>
<tr>
<td>ADM-XRC-II SDK</td>
<td>3.92</td>
<td>25.2</td>
</tr>
<tr>
<td>BenNUEY-PCI USURP</td>
<td>0.38</td>
<td>0.39</td>
</tr>
<tr>
<td>BenNUEY-PCI DIMEtalk</td>
<td>0.15</td>
<td>0.51</td>
</tr>
</tbody>
</table>
dependent on the speed of the communication bus; a confirmation from the FPGA does not have to be sent to the PC to acknowledge the write. The USURP framework’s read throughput is 4.8% slower than the Alpha Data SDK because there is additional address decoding done to allow for portability. On the BenNUEY-PCI platform, USURP and DIMEtalk produce very different results. USURP’s direct slave read throughput is more than twice as fast as the DIMEtalk implementation. The Universal Hardware Wrapper has the advantage of removing a lot of the unnecessary clutter that is generated through the DIMEtalk GUI. However, the direct slave writes are 23.5% slower. This is due to an unoptimized write engine being implemented on the Nallatech platform and will be fixed in future generations of the Universal Hardware Wrapper. Despite the differences in implementation between the platforms, feature compatibility remains.

Figs. 3 and 4 describe the DMA throughput for transfer sizes ranging from 64 bytes to 1 MB on the ADM-XRC-II and BenNUEY-PCI platforms, respectively. For small data frames, the USURP framework is slower than the ADM-XRC-II and BenNUEY-PCI baselines. The transfer size is too small to make up for the USURP framework’s additional software overhead and hardware address decoding. For data frames larger than 16 kB on the Alpha Data platform and 4 kB on the Nallatech platform there is very little performance hit (less than 3%). The Universal Hardware Wrapper for the BenNUEY-PCI even shows a slight improvement over the DIMEtalk interface for transfer sizes 64 kB and larger (4.8%). While the performance penalty for small data frames will be improved as the USURP system matures, it is unlikely many applications will need to take advantage of such small transfer sizes.

B. Case Study: N-Queens

N-Queens is a computation-intensive algorithm in which \( N \) number of queens are placed onto an \( N \times N \) size chess board such that no queen can attack another. The algorithm design consists of a series of nested loops, which magnifies the algorithm’s complexity to \( O(N^3) \), in order to perform the proper search sequence for the solutions. As expected from an algorithm complexity of \( O(N^3) \), as the board size increases linearly, the number of solutions increases exponentially. The depth first exhaustive search with backtracking design begins with a queen in the first row of the first column. Another queen is positioned in the second column such that it cannot attack the first queen. Upon finding the second columns position, the user must place another queen in the third column such that it cannot attack the previous two. If the queen can attack the others, then it must be moved to the next best position or the queen in the previous column must be moved to its next best position. The algorithm terminates when the first queen is in the last row and can no longer find a solution.

Given the history of the N-Queens algorithm at the authors’ laboratory at the University of Florida, it is a perfect test case for standardization within the USURP framework. Various custom versions of this algorithm have been ported to the Nallatech BenNUEY-PCI and BenNUEY-4e, Celoxica RC-1000, Alpha Data ADM-XRC and ADM-XRC-II, Cray XD1, and SGI Altix 350 platforms. The core has been written using the VHDL, DIME-C, Handel-C, and Impulse-C languages. For such an important legacy benchmark, a significant amount of time and effort was spent solely adjusting hardware interfaces and software control commands when porting between platforms. Despite the difficulties of manual porting, the success of such endeavors illustrates that such standardization is possible. Information can pass between hardware and software unimpeded despite the medium of communication.

Implementing N-Queens with the USURP framework incurs a small amount of area overhead on the Nallatech BenNUEY-PCI and Alpha Data ADM-XRC-II platforms (see tables 4 and 5, respectively). There are two table entries for the USURP framework, with and without partially disabling the register file. N-Queens only requires two registers to operate (totaling 40 bits of information); partially disabling the 1024 bit register file saves 960 flip-flops. The RAM interface can also be disabled to remove approximately 95 and 560 LUTs on the Nallatech and Alpha Data platforms, respectively.
Table 4: Hardware resources to implement N-Queens on the BenNUEY-PCI using the USURP framework and a custom solution

<table>
<thead>
<tr>
<th></th>
<th>USURP-1</th>
<th>USURP-2</th>
<th>Custom</th>
<th>Diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF</td>
<td>6491</td>
<td>5531</td>
<td>5265</td>
<td>5.1%</td>
</tr>
<tr>
<td>LUT</td>
<td>19750</td>
<td>19750</td>
<td>18728</td>
<td>4.9%</td>
</tr>
<tr>
<td>BRAM</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 5: Hardware resources to implement N-Queens on the ADM-XRC-II using the USURP framework and a custom solution

<table>
<thead>
<tr>
<th></th>
<th>USURP-1</th>
<th>USURP-2</th>
<th>Custom</th>
<th>Diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF</td>
<td>5570</td>
<td>4610</td>
<td>4403</td>
<td>4.7%</td>
</tr>
<tr>
<td>LUT</td>
<td>18238</td>
<td>17713</td>
<td>16588</td>
<td>6.6%</td>
</tr>
<tr>
<td>BRAM</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>

The execution time for the USURP and custom implementations are virtually identical (Fig. 5). The limited bandwidth requirement of N-Queens allows the USURP framework to incur no overhead across all tested board sizes. Even problem sets that have very small computation to communication ratios (the 4x4 board requires only 64 operations) are identical to microsecond resolutions. Between platforms, the execution time results for the ADM-XRC-II and BenNUEY-PCI should not be expected to be identical as each have different hardware/software interfaces. However, for boards larger than 6 x 6, the differences are less than 1%.

As an anecdote, it should be noted that the USURP N-Queens core was developed from a custom solution designed for the BenNUEY-PCI platform. After being verified on the Nallatech platform, it was possible to move the core to the ADM-XRC-II without modifying the VHDL. The platform switch took less then an half hour (which was entirely consumed by the place and route process). We envision using the USURP framework to rapidly benchmark application cores across multiple platforms.

Fig. 6. Data throughput for 2-D FFT on the ADM-XRC-II and BenNUEY-PCI

C. Case Study: 2-D FFT

The 2-Dimensional Fast Fourier Transform (2-D FFT) can be used to convert a signal from the spatial domain to the frequency domain. The conversion is usually done to allow for frequency-based signal analysis or digital filtering of the original signal. Implementations of the 2-D FFT algorithm on an FPGA coprocessor are limited primarily by the available PC/FPGA bandwidth. For an $N \times N$ matrix, the 2-D FFT core requires $O(2N^2)$ data transfers and has a has a computational complexity of $O(10N^2\log(N))$ for a radix-2 implementation.

This particular version of the 2-D FFT algorithm is a unique benchmark in that it was the first application explicitly written for the USURP framework. Unlike N-Queens, it was not based off existing legacy code. As such, there is no existing custom implementation for baseline comparison. Any attempt at creating such a baseline would result in a fairly synthetic comparison and the authors have not attempted such a comparison. Deviating from the USURP framework would primarily include removing custom components not in use (which are subsequently eliminated by the synthesizer anyways) resulting in little noticeable difference.

Table 6 describes the hardware resources used by the USURP framework when implementing the accelerated 2-D FFT core. The overhead in FFs, LUTs, and BRAM is minimal considering a custom-tailored solution would not be able to significantly reduce the resource requirements and still provide the same functionality.

Table 6: Hardware resources to implement a 2-D FFT on the ADM-XRC-II and BenNUEY-PCI using the USURP framework

<table>
<thead>
<tr>
<th></th>
<th>ADM-XRC-II</th>
<th>BenNUEY-PCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFs</td>
<td>760</td>
<td>556</td>
</tr>
<tr>
<td>LUTs</td>
<td>1112</td>
<td>1221</td>
</tr>
<tr>
<td>BRAM</td>
<td>4 kB</td>
<td>4 kB</td>
</tr>
</tbody>
</table>

Fig. 5. Execution time for N-Queens on the ADM-XRC-II and BenNUEY-PCI
The data in Fig. 6 is reported in megasamples (equivalent to 4 MB/s) and represents the number of 16-bit fixed-point complex matrix elements processed by the 2-D FFT core in a second. The 2-D FFT’s throughput trends correspond to the DMA throughput measurements. Small matrices cannot overcome the DMA transfer overhead; large matrices saturate the available communication bandwidth. For matrices larger than 64x64, the USURP framework has no effect on the performance of the core.

V. CONCLUSIONS AND FUTURE WORK

The USURP framework has the potential to provide enormous benefit to the RC community through a standardized compile-time hardware/software interface and run-time communication standard. The USURP software architecture empowers application developers with a common interface to the FPGA through a universal API. Applications developed with the Universal FPGA Software API can be ported to USURP-compatible architectures without wasted redesign or recompilation efforts. The Universal Hardware Wrapper exposes common hardware resources to the application developer in a unified fashion. Cores developed to interface with the wrapper can be ported to USURP-compatible RC platforms with only a new place and route.

Two hardware-accelerated applications have been developed under the USURP framework. The N-Queens and 2-D FFT cores are write once, run anywhere implementations and have been successfully implemented on the Alpha Data ADM-XRC-II and Nallatech BenNUEY-PCI platforms. The USURP framework dramatically improved efficiency and predictability of application execution performance.

The next immediate goal of the USURP project is to apply the framework to additional reconfigurable platforms, particularly the Cray XD1. Implementing the USURP framework on the XD1 architecture will allow us to unify a much larger and more diverse set of hardware under the standard framework. The XD1 also gives us a platform to explore the incorporation of vendor-specific features into the framework using the Extended FPGA Software API and Extended Hardware Wrapper.

Another powerful suite of tools for FPGA development that merit integration into a unified framework are C-based HDL mappers. These languages allow for rapid algorithm development, especially for those with limited hardware experience. Currently, C-based mapper compilers produce hardware descriptions for a vendor-specific standard, which at best supports only a limited number of RC platforms. These vendor solutions can be thinly wrapped for integration into the USURP compilation standard. However, ideally, future compilers could directly target the USURP framework.

Deploying parallel FPGA jobs across multiple RC platforms under a standard management framework is one of the last major hurdles for true reconfigurable supercomputing. Multiple FPGA designs were briefly discussed with regards to CARMA and the possible extensions to the USURP. Also, inter-FPGA communication through the USURP front-end should be able to integrate all FPGA resources, independent of their structure and will allow USURP to fully transform from a compile-time tool to a run-time heterogeneous multi-FPGA design and management infrastructure.

VI. ACKNOWLEDGMENTS

We gratefully thank Cray and Nallatech for their continued equipment support.

REFERENCES