Reconfigurable Computing Middleware for Application Portability and Productivity

Robert Kirchgessner, *Member, IEEE,* Alan D. George, *Fellow, IEEE,* and Herman Lam, *Member, IEEE*
NSF Center for High-Performance Reconfigurable Computing (CHREC)
Department of Electrical and Computer Engineering, University of Florida
Gainesville, Florida, 32611-6200
Email: {kirchgessner, george, hlam}@chrec.org

***Abstract*—Reconfigurable computing (RC) devices such as field-programmable gate arrays (FPGAs) offer significant advantages over fixed-logic, many-core CPU and GPU architectures, including increased performance for many computationally challenging applications, superior power efficiency, and reconfigurability. Difficulties of using FPGAs, however, has limited their acceptance in high-performance computing (HPC) and high-performance embedded computing (HPEC) applications. These difficulties stem from a lack of standards between FPGA platforms and the complexities of hardware design, and lead to higher costs and time to market over competing technologies. Differences in FPGA platform resources such as the type and number of FPGAs, memories and interconnects, as well as vendor-specific procedural APIs and hardware interfaces, inhibits application portability and code reusability. Despite efforts to reduce FPGA application design complexity through technologies such as high-level synthesis (HLS) tools, platform support and portability remains limited, and is typically left as a challenge for application developers. In this paper, we present a novel RC Middleware (RCMW), an extensible framework which enables FPGA application portability and enhances developer productivity by providing an application-centric development environment. Developers focus specifically on the optimal resources and interfaces required by their application, and RCMW handles the mapping and translation of those resources onto a target platform. We demonstrate that RCMW enables application portability over three heterogeneous platforms from two vendors, using both Xilinx and Altera FPGAs, with less than 10% performance and area overhead for several application kernels, and microbenchmarks for the common case. We present the productivity benefits of RCMW, showing that RCMW reduces required number of hardware and software driver lines of code and total development time with respect to native platform deployment methods for several application kernels.**

***Index Terms*—FPGA, portability, productivity, reconfigurable computing**

I. INTRODUCTION

Reconfigurable computing (RC) offers significant advantages over conventional fixed-logic devices such as CPUs and GPUs. The flexibility of reconfigurable devices such as FPGAs enable developers to create application-specific hardware architectures, maximizing performance [1], [2] and power efficiency [3] for applications which may not map well to conventional architectures. This efficiency has made FPGAs promising in a variety of computing areas, from embedded systems [4] to supercomputers [5].

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant Nos. EEC-0642422 and IIP-1161022.

These benefits, however, come with the added complexity of architecture design and thus an added challenge in developer productivity relative to fixed-logic devices such as CPUs and GPUs. The difficulty of hardware design coupled with a lack of standards between FPGA platforms, herein referred to as platforms, complicate application development and limit code portability and reuse. Due to a lack of standards between platforms, FPGA application developers are required to tailor their application to a specific vendor's software and hardware interfaces. Developers must decompose and map their application to a specific platform based on available FPGAs, memories and interconnects. This platform-specific development cycle restricts application portability, and requires significant developer time and effort to port applications to new platforms. Furthermore, the rampant use of procedural software APIs by platform vendors has further limited application portability. The procedural programming paradigm embeds platform specific parameters such as the physical location of application resources into software code. This embedding leads application developers to develop platform-specific code, increasing cost and development time when porting applications to heterogeneous platform configurations.

Technologies such as high-level synthesis (HLS) tools which are designed to reduce the difficulties of FPGA application development also suffer from these issues. Due to differences between platforms, HLS tools must provide a method to support various platforms. Tools such as Impulse-C [6] accomplish this via per-platform support packages which convert platform-specific to tool-specific interfaces. Tools such as ROCCC [7] generate application cores with platform-agnostic interfaces, and require developers to handle platform-specific mapping and implementation. Although HLS tools typically provide support for at least one platform out of the box, the growing number of HLS tools and FPGA platforms outpaces the ability of tool vendors to provide platform support, leaving the mapping of HLS tools to emerging platforms as a challenge for application developers. Furthermore, the diversity of available platforms forces HLS tools to make optimization-restricting assumptions about available platform resources. The target platform may not be able to provide the required interfaces and bandwidth required for certain optimization techniques [8]. These problems ultimately reduce HLS tool performance and usability, and end up costing tool vendors and developers valuable time which could better be

spent on developing their tools or applications.

In order to overcome the portability and productivity hurdles of RC application development, we present a novel RC Middleware (RCMW). RCMW is a layered middleware which enables application and tool portability by providing an application-centric view of available resources. Using RCMW, application developers can focus on the ideal resource partitioning for their application. Users specify at design time the required resources and interfaces via an XML framework, customizing the number, type, size and data types of various resources. Using this specification, the middleware provides a portable hardware and software interface. The developer can then focus on developing their application, rather than translating their application to a particular platform. This application-specific interface is generated by the middleware toolset, which handles mapping application resources to a target platform as well as generating the hardware and software stacks to realize that mapping. In this paper, we evaluate RCMW using three platforms from two vendors: the PROCStar III and PROCStar IV from GiDEL and the M501 from Pico Computing. We demonstrate application portability by executing the same application hardware and software source code across each platform. We support our argument for RCMW's productivity benefits by demonstrating that RCMW requires less development time and lines of code for deploying applications on each platform compared to the recommended vendor approaches. We also show that these benefits can be achieved with less than 10% performance and area overhead. The remainder of this paper is organized as follows: Section II presents related works. Section III presents the RC Middleware framework and toolset. Section IV presents our experiments and results. Section V discusses our proposed future work. Section VI presents our conclusions.

II. Related Work

In our previous work on VirtualRC [9], we presented a method enabling portability using parameterized, customizable IP components with support for several platforms. We provided a procedural software API which abstracted away vendor-specific software interfaces, but required developers to handle resource translation manually. RCMW further develops this concept by providing an application-centric development environment, where RCMW handles the translation of user application to platform resources, and provides a high-level object-oriented software abstraction.

In [10], Coole et. al. presents intermediate fabrics (IFs), coarse-grained virtual FPGA fabrics designed to significantly improve application compilation time. IFs also enable application portability across platforms which implement the same fabric. Similarly in [11], Reves et. al. presents an application-specific coarse-grained FPGA architecture for software-defined radio (SDR). Applications targeting this coarse-grained architecture are portable across any platform which implements that architecture. RCMW is a complementary approach which provides platform resource abstractions, and could be leveraged by methods such as intermediate fabrics to enable portability across heterogeneous platforms.

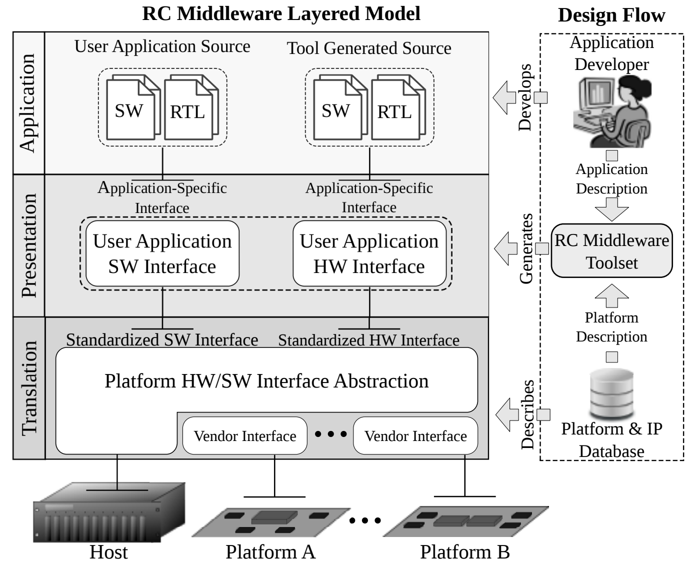

Fig. 1. RC Middleware overview.

Another common approach to enabling FPGA portability is FPGA-on-FPGA overlays, such as the ZUMA architecture presented by Brant et. al. in [12]. This approach provides application bitfile compatibility between devices, enabling application portability between FPGAs without requiring bitfile recompilation. RCMW does not aim to provide FPGA bitfile compatibility, but rather focuses on the broader goal of platform portability. RCMW could provide a portable interface to platform resources, enabling overlay portability across heterogeneous platform configurations.

Similar to high-level synthesis tools, OpenCL [13] and Liquid Metal (Lime) [14] provide high-level programming models for application development, but also aim to provide portability across different types of compute devices. While Lime focuses on efficient code portable between host and FPGA coprocessor, OpenCL takes a broader approach to enable code portability between a wide variety of computing devices. Both OpenCL and Lime focus on providing a high-level, portable models for application development. In order to target a particular FPGA-based platform, the components of each model must be mapped to available platform resources. Lime accomplishes this in part using the Lime Runtime (LMRT), but additional effort is required to support different platforms. By targeting RCMW, both OpenCL and Lime would gain portability across RCMW-supported platforms.

III. Approach

Fig. 1 illustrates the layered structure of RCMW's hardware and software abstractions. From the bottom up, the middleware is composed of three layers in both hardware and software: the translation layer, the presentation layer and the application layer. The translation layer converts platform-specific resource interfaces to a standardized RCMW interface. Next, the presentation layer leverages this standardized interface to provide application-specific resource interfaces requested by the developer's application description. Finally, these interfaces are

presented to the developer in the application layer, providing an application-centric development environment.

The remainder of this section discusses the layered architecture enabling our hardware abstraction (Section III-A), layered software abstraction (Section III-B) and the RC Middleware toolset (Section III-C).

A. Hardware Abstraction

Fig. 2 illustrates RCMW's layered hardware model. This layered model abstracts away vendor-specific platform details and creates the application-specific hardware interfaces requested by the application developer.

1) Translation Layer: The translation layer handles converting vendor-specific physical interfaces for memories, inter-FPGA communication, host communication and other physical interfaces into an RCMW-standardized interface format. This layer provides standardized interfaces for any physical interfaces which may be used to provide application resources. This layer may also expose standardized interfaces to FPGA block RAM, allowing user applications to take advantage of the low latency of on-chip memory for sufficiently small application memory resources.

2) Presentation Layer: The presentation layer handles converting the standardized physical resource interfaces provided by the translation layer into the interfaces expected by the developer's application. The presentation layer interfaces are generated by leveraging an RCMW IP database, which contains IP components for various interface conversions, resource multiplexing, arbitration, and interface controllers.

In order to enable an arbitrary number of user interfaces to map to a physical resource, the presentation layer can optionally include a configurable arbitration controller, shown in Fig. 3 using read interfaces. This controller handles multiplexing any number and type of user interfaces to an interface exposed by the translation layer. Since available memory interface bandwidth is typically greater than the required bandwidth for a single interface, the middleware can typically saturate multiple user interfaces with data without loss of performance. RCMW's arbitration controller uses a ready-to-send (RTS), clear-to-send (CTS) protocol to arbitrate memory accesses. Each interface controller consists of a configurable transfer buffer which buffers incoming read or outgoing writes. When a configurable number of bytes remain in the buffer, the interface controller will assert the RTS signal, indicating that it is ready to access memory. When the arbitration logic asserts the CTS signal, the interface takes control of the physical memory bus. When CTS is de-asserted, the interface must give up control of the bus. The RTS/CTS protocol enables easy addition of new arbitration schemes. If only one memory is mapped to a physical resource, no arbitration logic is needed; the interface controller is mapped directly to the translation layer. RCMW currently provides two interface types for applications: the burst and FIFO interface. The burst interface enables user applications to address different regions of memory sequentially. The user specifies a starting address, size, and asserts the start signal to begin a transfer. The FIFO interface enables application software to read/write data from hardware in first-in, first-out order from the host, without requiring explicit addressing. The enable/valid signals on both interfaces provide a flow control handshake which is necessary due to variations in the underlying platform and application resources.

As shown in Fig. 3, the presentation layer can provide any number of interfaces of either FIFO or burst type. Each interface can be another interface to the same application resource, or a different application resource. To enable this functionality, each interface controller is configured by a set of generic parameters, including: data width, base address, size and buffer length. The data width is the application-specified word size, which can be any power of two number of bits. The base address corresponds to the address in physical memory of the interface. The size of the memory specifies the top of the interface memory space, and is used for calculating address wrapping conditions in the FIFO and burst interfaces. The buffer length specifies the maximum number of words the interface can buffer before requiring physical memory access. In addition to memory interfaces, RCMW provides a memory-mapped interface to the user application. The memory-mapped interface provides register-style resources to the application.

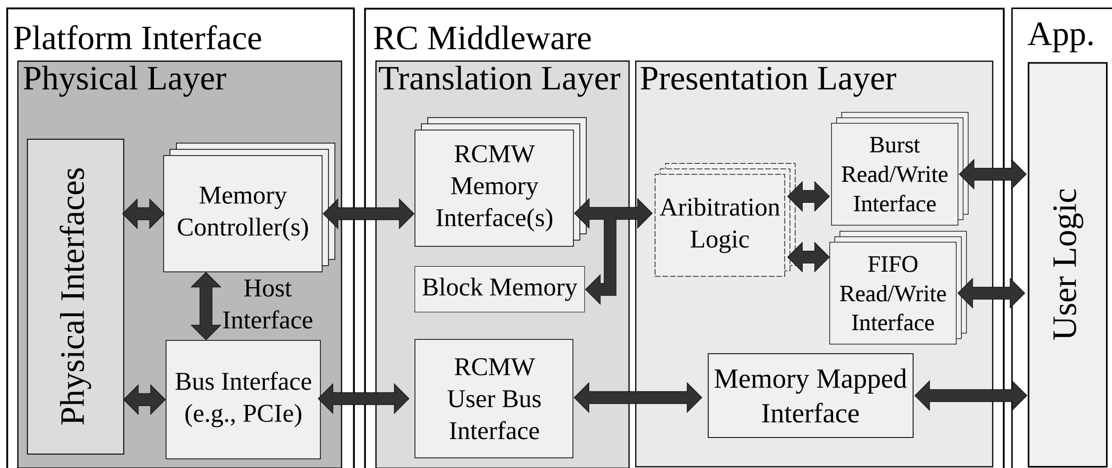

Fig. 2. RC Middleware layered hardware abstraction.

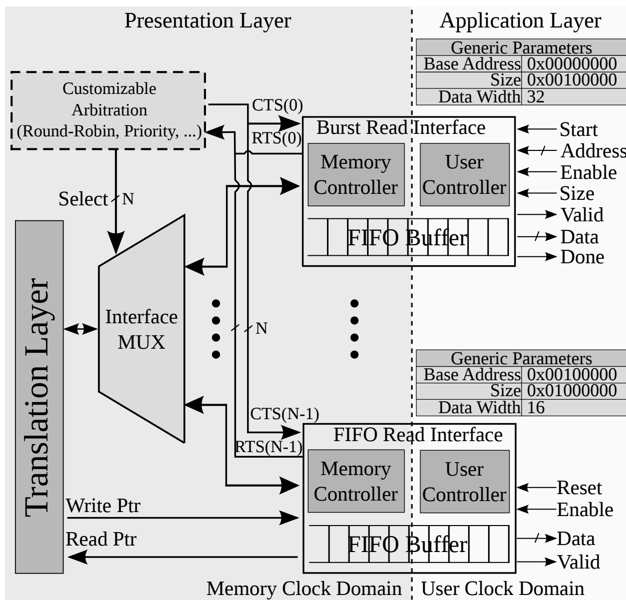

Fig. 3. High-level presentation layer hardware diagram.

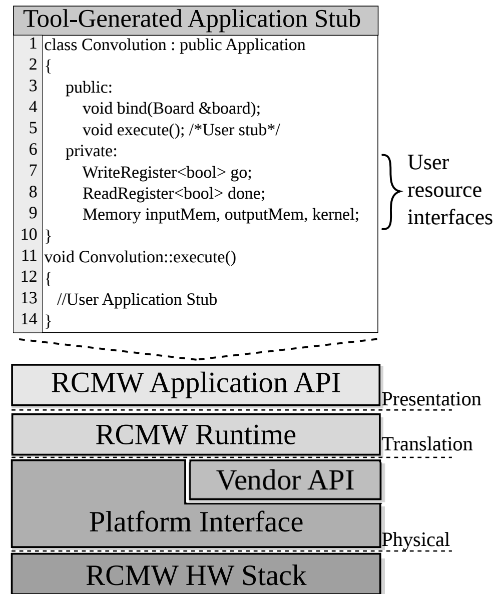

Fig. 4. RC Middleware layered software abstraction.

In order to describe the physical interfaces available on a target platform, RCMW employs an extensible XML framework which describes platform interfaces and components. This framework allows description of FPGAs and memory components on a platform, as well as how they are connected together. For each FPGA, the framework describes available interfaces as well as pin assignments, and timing constraints. In our context, an interface is a collection of pins which aggregate to perform a more complexed function, such as a DDR memory interface. Using this framework, users and platform vendors can easily add support for new platforms.

3) Application Layer: The application layer presents the application-specific hardware interfaces requested by the developer. RCMW provides the application-layer interface in the form of a top-level entity declaring only the requested application interfaces. The user implements their application in the architecture body of the top-level entity using the interfaces required by the application, without worrying about how or where resources are mapped onto the target platform.

B. Software Abstraction

Fig 4 illustrates RCMW's layered software model. RCMW provides a portable object-oriented software API based on C++11 which provides a uniform application-centric development environment regardless of the underlying platform. RCMW takes advantage of C++11's improved threading model, memory management, and features to maximize application performance and simplify development.

1) Translation Layer: The translation layer presents standardized software interfaces to platform resources. These standardized interfaces are exposed through an abstract *Board* class in RCMW's software API. Each platform supported by RCMW is required to have a subclass of the *Board* class, implementing the required interfaces including: blocking read-/write, asynchronous read/write, board enumeration and bitfile initialization for each FPGA. The *Board* class encapsulates a set of *Memory* and *FPGA* objects which represent the physical components available on a specific platform. Each *FPGA* object stores a reference list to *Memory* objects for which the corresponding *FPGA* is physically connected. This objectified representation provides a common interface to platform resources, and captures the layout of the physical platform. These resource interfaces are built upon either the vendor-specific API, or an RCMW-specific driver interface.

2) Presentation Layer: The presentation layer handles mapping the application specification onto the *Board* class interface provided by the translation layer. This layer is generated by the RCMW toolset as a subclass of the RCMW *Application* class. Fig. 4 illustrates the application-specific interface for a simple convolution example, with two registers *go* and *done* and three memory interfaces *input*, *output* and *kernel*. The *Application* class encapsulates an instance of a resource interface object corresponding with each interface requested in the application description XML. It provides *Register* objects, which correspond to registers mapped onto the memory-mapped interface, *Memory* objects, which correspond to hardware burst interfaces, and *FIFO* objects, which correspond to FIFO hardware interfaces. The *Application* class interface consists of two main functions: *bind* and *execute*. The *bind* function is generated by the RCMW toolset, and handles mapping application-specific resources onto platform-specific resources. The *execute* function is generated as a stub for the application source. The developer implements his application in this function, using the resource objects exposed by the *Application* class instance. At runtime, RCMW handles detecting available platforms, determining what platform to run the application on, and handles initializing and loading the required bitfiles. RCMW then calls *bind* on the selected *Board* instance, and assigns the application *execute* routine to an idle software thread. When the application thread returns, RCMW automatically handles releasing hardware resources.

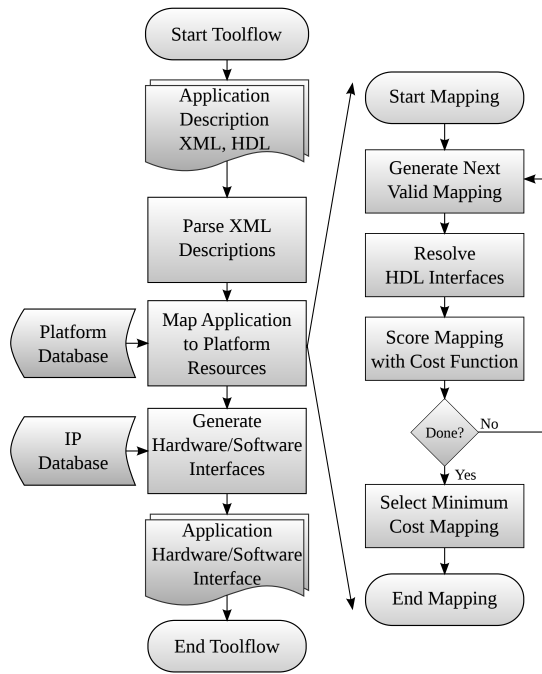

Fig. 5. RC Middleware toolflow.

3) Application Layer: The application layer exposes an application-customized subclass of an *Application* class generated by the RCMW toolset. Developers need only implement their application driver in the *execute* stub function using the resource objects exposed by their customized application class. The RCMW API also provides application-layer utilities, such as a thread-safe timing library, to further improve productivity. This approach provides a portable programming paradigm. Developers are provided with application resource interfaces without having to worry about where or how they are mapped onto a target platform.

C. RC Middleware Toolset

The RCMW toolset handles generating the hardware and software presentation layers to map application resources onto platform resources. The toolset handles mapping application to platform resources, and exposes only the requested interfaces as a top-level in hardware and software, greatly simplifying development. Fig. 5 illustrates the toolflow used to generate the hardware/software interfaces using a developer's application description. The toolflow is broken down into three major steps: parsing the XML description, mapping application to platform resources, and generating the HDL/C++ for the hardware/software interfaces. The most complex step of the toolflow is the mapping step, which determines where application resources such as application memory and FIFO resources should be mapped onto the physical platform. Depending on the target platform configuration, available IP components, and developer-requested application resources, the resulting mapping may be significantly different between heterogeneous platforms. RCMW provides an extensible XML framework which can be used to add new components to the RCMW IP database for use during the mapping step. This enables developers to define custom application interfaces and components.

The mapping step determines the lowest-cost mapping of application to physical resources given a configurable cost function. During the mapping step, RCMW iterates over all valid application to physical mappings. For example, if there are two physical memories, A and B, of size 512MB each, and the developer requests two 256MB application memories, there are four valid mappings to consider: both application memories mapped to physical memory A, both application memories mapped to physical memory B, application memories mapped to physical memory A and B respectively, and application memories mapped to physical memory B and A, respectively.

For each valid mapping, RCMW resolves required HDL components, determining what RCMW IP instances are required to enable that mapping. For example, if two application memories are mapped to the same physical memory, RCMW will need to instantiate an arbitration and multiple interface controllers. In order to guide the mapping process, each IP in RCMW is accompanied by metadata which estimates maximum operating frequency and device resource utilization. Using this information, each mapping is scored using a customizable cost function. RCMW's default cost function is based on the number of IP instances that are used for a particular mapping, favoring the simplest mapping. After all valid mappings have been scored, the mapping stage returns the map with the lowest cost. The toolset then generates the user hardware and software presentation layer interfaces based on the selected mapping. For hardware, a structural HDL component is created connecting the appropriate IP instances from the RCMW IP library. For software, the *bind* function of the user *Application* subclass is generated, mapping user-application resources to physical *Board* resources. Finally, the RCMW toolset generates Makefiles for hardware and software compilation, as well as a configured project file for vendor-specific FPGA toolchains. In order to compile their application, a developer only needs to implement their application in the hardware and software stubs, and run the appropriate Makefile.

Using the RCMW toolset, developers can quickly explore the application design space, modifying application resources by adding registers, tweaking memory interface properties and arbitration schemes, and try different mapping cost-functions which will optimize their designs for different parameters such as performance or area. If a developer receives an application source designed using RCMW, all they need to do is run the RCMW toolset and specify their target platform to obtain a translated and compile-ready version of the application.

IV. Experimental Results

In this section, we present our experimental design (Section IV-A), and then provide an analysis of performance and area overhead incurred using RCMW (Section IV-B). Next, we evaluate the productivity benefits of RCMW (Section IV-C). Finally, we demonstrate the portability of applications and kernels (Section IV-D).

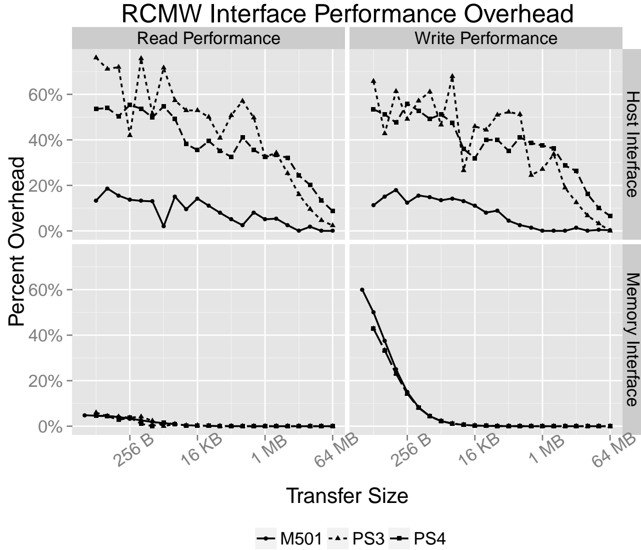

Fig. 6. Host and FPGA read and write performance overhead to external memory for Pico M501, PROCStar III (PS3) and PROCStar IV (PS4).

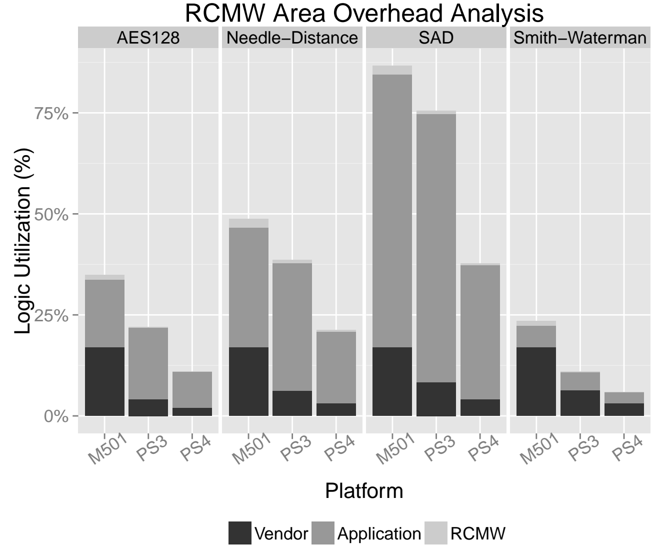

Fig. 7. Area overhead analysis of the RC Middleware compared to logic utilization of vendor IP and application components.

A. Experimental Setup

In our experiments, we evaluate RCMW using three platforms from two vendors: the GiDEL PROCStar III and PROCStar IV, and the Pico Computing M501. Each vendor provides significantly different hardware configurations, software APIs and use different FPGA vendors. The PROCStar III uses a PCIe 1.0 8-lane interface to four Stratix III E260 FPGAs, with one 256 MB external SDRAM and two 2GB DDR-II SODIMMS per FPGA. The PROCStar IV provides a similar configuration, with four Stratix IV E530 FPGAs, with one 512MB external SDRAM and two 4GB DDR-II SODIMMS per FPGA . The Pico M501 uses a PCIe 2.0 8-lane interface to a Virtex-6 LX240T with one bank of 512MB DDR3 memory.

In our experiments, we compiled all Altera bitfiles using Quartus II v11.1 service pack 1. We used GiDEL driver version 8.9.3.0 for the PROCStar board tests. Bitfiles for the Pico M501 were generated using Xilinx ISE 14.1. We used Pico driver version 5.2.0.0.

RCMW's software API was compiled using GCC v4.7.2 with C++11 support. All software was compiled using compiler optimizations -O3.

B. Performance and Overhead Analysis

In this section, we analyze the overhead introduced RCMW hardware and software stacks. We analyze the performance overhead incurred while performing read and write transfers to the external memory from the host and from the FPGA as a percentage overhead of the bandwidth achieved using the native platform interfaces. We analyze the area overhead incurred by comparing the relative logic utilization of RCMW to the logic utilization of vendor IP and application components for several kernels. In order to measure the overhead introduced by RCMW's software stack, we perform host to FPGA memory transfers for varying payload sizes using a single RCMW read and write interface from the host. We measure the time required to complete each transfer and calculate the overhead as a percentage increase over the baseline platform DMA transfer time. Next, in order to measure the overhead introduced by RCMW when transferring data between application hardware and external memory resources, we compared the average bandwidth achieved when transferring various data sizes for a single read and write interface with the average bandwidth achieved without RCMW. To measure the FPGA to external memory bandwidth, we count the total number of clock cycles required to perform a transfer of a given size, and use that to determine bandwidth. We calculate overhead as a percentage increase over the baseline. Finally, we measure RCMW's area overhead for applications and kernels, and compare the logic utilization with the logic utilization of vendor IP and application components. The area percentages were obtained using the post-fit component device utilization summary provided in Altera Quartus II and Xilinx ISE.

Figure 6 illustrates the read and write performance overhead introduced by the RCMW software and hardware stacks for various transfer sizes. The top and bottom rows show host and memory interface overhead, respectively, and the left and right columns show read and write overhead for each interface, respectively. The greatest overhead is found in the host to external memory transfer overhead, peaking at approximately 80% and 70% overhead for reads and writes with the PROCStar III, respectively. This high overhead is due to additional features provided by the RCMW API, including thread-safety. Since the M501 provided thread-safety for some API calls by default, the peak overhead is less, approximately 20% for both reads and writes. These high overheads are restricted to small transfer sizes, resulting in an increase of a few microseconds to complete these transfers. For increasing

transfer sizes, the overhead quickly approaches zero, as the transfer time of data to the board becomes more significant. This behavior is also demonstrated in the FPGA to memory read and write performance. The peak overhead for FPGA to memory transfers was approximately 60% overhead for write operations on the M501. This overhead is introduced due to additional levels of buffering in RCMW's hardware stack, which also requires the data to be flushed to memory to ensure consistency before marking a transfer as completed. For these small transfers, a latency of 60% equates to tens of clock cycles, which is relatively insignificant. For large transfers, this latency becomes less significant as data transfer time increases.

Figure 7 illustrates the logic utilization of RCMW, Application and Vendor IP components. The bottom layer in the stacked bar graph represents the vendor IP utilization, the middle layer represents the application logic utilization and the top layer represents RCMW overhead. Each triplet of bars represent the area utilization breakdown for each platform, for different applications. The height of each bar represents the total logic utilization required by each application, with the shaded regions indicating what percentage of that utilization comes from each component. From this graph, we can see that RCMW accounts for a very small fraction of the total design area, typically less than 1% of the total device resources. The total area percentage required by RCMW depends on the users requested interface configuration. It is important to note that in instances where multiple application resources are mapped to a physical memory, a portion of the overhead introduced by RCMW would be necessary even for a non-RCMW based implementation, such as the memory arbitration controller.

C. Productivity Analysis

In this section, we analyze the productivity benefits provided by RCMW. For lack of a better method, we measure productivity in terms of software lines of code (SLoC), hardware lines of code (HLoC) and total development time. These factors have been commonly used as measures for software development productivity. It is worth mentioning that lines of code is a rather obscure measure, and developer coding style has a significant impact. In these experiments, a developer familiar with all three platforms and RCMW translated several cores from OpenCores to each platform, including: SHA256, JPEG Encoder, FIR filter, AES128, and 3DES kernels. Each core was developed using the vendor recommended design flow and RCMW's design flow. In measuring lines of code, all code written by the developer was counted, excluding comments.

GiDEL and Pico Computing provide different approaches for developers to deploy their application. GiDEL provides a powerful graphical interface called PROCWizard, which enables developers to customize IP cores and select between different interfaces to their platforms resources, specify registers and select between different clock domains. Once a developer finishes specifying different IP parameters, PROCWizard generates a configured hardware template and PROC API software interface to their platform. Pico Computing provides a different, lower-level approach, giving developers configurable access to a Xilinx AXI interconnect to interface with platform memory. Pico provides a streaming abstraction in both hardware and software, which enables efficient streaming of data from host to FPGA, as well as a PicoBus for simple IO operations such as application register transfers.

Our experiments indicated that on average RCMW required 65% less SLoC, 41% less HLoC, and 53% less development time than GiDEL-based implementations, and 66% less SLoC, 59% less HLoC, and 69% less development time than Pico-based implementations. Although this improvement is averaged for five applications and a single developer, it supports the argument that RCMW provides a productivity improvement. This outcome is expected, since RCMW handles many tasks typically left to the developer. The major factors leading to these differences include: level of software/hardware interface complexity, clock-domain crossing (hardware), buffer memory management (software) and API restrictions.

In our experiments, Pico required a relatively high number of HLoC due to the low-level interfaces exposed to developers; requiring users to handle the AXI interconnect protocol and any clock-domain crossing (CDC) for their application. GiDEL required less lines of hardware code, having a large chunk of coding automatically generated by their PROCWizard tool. GiDEL also handles CDC for memory interfaces, further reducing HLoC, but they do not handle CDC for registers. RCMW required the least HLoC, generating the interfaces specified by the user, and handling all required CDC. Similar results were found for total SLoC, with Pico requiring the most lines of code followed again by GiDEL, and then RCMW. Both vendor APIs require developers to manage buffers and imposed restrictions on data transfers such as data alignment and size restrictions. These factors increase the amount of coding developers are responsible for in their software. In order to reduce developer coding overhead, RCMW provides a variety of different features, such as templated read/write functions which can handle variable data types. Additionally, RCMW handles buffer management and garbage collection internally.

D. Portability Analysis

Table I presents the execution time and logic utilization for various applications and kernels executing across all three platforms. Each application was executed using the same user application source with RCMW. This table demonstrates the same application source, both hardware and software, executing across heterogeneous platforms. Porting applications across each platform was accomplished with almost no effort, requiring only that the RCMW toolset be executed once for each platform, with each application. We note that no single platform performs the best for every tested application due to having different strengths and weaknesses. For instance, the M501 uses a newer PCIe generation, enabling higher peak bandwidths for host to FPGA transfers, making transfer heavy applications like Smith-Waterman, which streams a large database from host to FPGA, perform better. For applications that require more memory interfaces, such as Image Segmentation, the two additional banks the PROCStar III and PROCStar IV provides an advantage over the M501.

TABLE I
DEMONSTRATION OF RC MIDDLEWARE ENABLING APPLICATION PORTABILITY ACROSS THREE PLATFORMS.

	M501		**PROCStar III**		**PROCStar IV**	
Kernel/Application	Time	Utilization	Time	Utilization	Time	Utilization
1D Fixed Convolution FXD	43.3 ms	26%	39.3 ms	14%	38.9 ms	7%
2D Convolution FXD	16.0 ms	57%	13.2 ms	44%	16.3 ms	25%
Image Segmentation	1.73 s	64%	1.41 s	53%	1.39 s	26%
Needle-Distance	161 ms	48%	194 ms	38%	203 ms	21%
OpenCores AES128	32.6 ms	32%	25.3 ms	22%	24.3 ms	11%
OpenCores FIR	21.5 ms	28%	24.5 ms	15%	24.0 ms	8%
OpenCores JPEGEncoder	23.9 ms	29%	15.3 ms	15%	19.6 ms	8%
OpenCores SHA256	73.3 ms	26%	64.1 ms	12%	63.3 ms	5%
Smith-Waterman	96.0 ms	23%	116 ms	11%	119 ms	6%
Sum of Absolute Differences	18.7 ms	86%	14.7 ms	75%	19.1 ms	38%

V. FUTURE WORK

RCMW provides an extensible framework for enabling application portability and improves productivity. However, there are still additional features and application-specific optimizations which could improve performance and functionality. Additionally, to better understand the productivity benefits of RCMW, a larger case-study using a variety of application developers is needed.

In the future we will investigate application-specific memory optimizations. Additionally, we will investigate new API features such as common collectives operations for distributing data, including: broadcast, scatter and gather. Although RCMW enables application portability across FPGAs, one remaining portability-limiting factor remains with FPGA-specific IP cores. FPGA vendors such as Xilinx and Altera provide vast libraries of IP cores which users can leverage in developing their applications. These IP cores, however, are not portable between FPGAs. In the future we will investigate methods for enabling IP portability.

VI. CONCLUSIONS

Despite performance and power efficiency advantages over conventional many-core CPU and GPU architectures, FPGAs have continued to suffer from a productivity hurdle. To overcome this hurdle, we introduced the RC Middleware (RCMW). RCMW provides an extensible framework and toolset, which provides an application-centric hardware and software development environment. This application-centric environment can be customized through an XML framework, enabling developers to focus on their ideal application partitioning rather than targeting a specific platform configuration.

We evaluated RCMW on three platforms from two vendors, showing that RCMW enables application portability with less than 1% overhead for large transfers sizes. We also showed that RCMW accomplished this with relatively low area overhead, requiring less than 1% of total device logic resources for several applications across all three platforms. We demonstrated that RCMW enables portability by showing that the same application source was able to execute without change across heterogeneous platforms. Finally, we presented evidence that RCMW improves developer productivity, by showing that RCMW requires less lines of code and total development time for deploying several kernels than vendor-specific approaches.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge vendor equipment and tools provided by Altera, Xilinx, GiDEL and Pico Computing that helped make this work possible.

REFERENCES

[1] C. Pascoe, A. Lawande, H. Lam, A. George, W. F. Sun, and M. Herbordt, "Reconfigurable Supercomputing with Scalable Systolic Arrays and In-Stream Control for Wavefront Genomics Processing," in *Proceedings of International Conference on Engineering of Reconfigurable Systems and Algorithms*, Jul. 2010.

[2] S. Che, J. Li, J. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with gpus and fpgas," in *Application Specific Processors, 2008. SASP 2008. Symposium on*, june 2008, pp. 101–07.

[3] X. Tian and K. Benkrid, "High-Performance Quasi-Monte Carlo Financial Simulation: FPGA vs. GPP vs. GPU," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 4, p. 26:126:22, Nov. 2010. [Online]. Available: http://doi.acm.org/10.1145/1862648.1862656

[4] P. Garcia, K. Compton, M. Schulte, E. Blem, and W. Fu, "An overview of reconfigurable hardware in embedded systems," *EURASIP Journal on Embedded Systems*, vol. 2006, 2006.

[5] A. George, H. Lam, and G. Stitt, "Novo-g: At the forefront of scalable reconfigurable supercomputing," *Computing in Science Engineering*, vol. 13, no. 1, pp. 82–86, 2011.

[6] "Impulse Accelerated Technologies - Software Tools for an Accelerated World." [Online]. Available: http://www.impulseaccelerated.com/

[7] "ROCCC 2.0." [Online]. Available: http://www.jacquardcomputing.com/

[8] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing Modular Hardware Accelerators in C with ROCCC 2.0," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual Int. Sym. on*, may 2010, pp. 127–134.

[9] R. Kirchgessner, G. Stitt, A. George, and H. Lam, "VirtualRC: A Virtual FPGA Platform for Applications and Tools Portability," in *Proc. of International ACM/SIGDA Symposium on Field Programmable Gate Arrays (FPGA)*, Feb 2012.

[10] G. Stitt and J. Coole, "Intermediate Fabrics: Virtual Architectures for Near-Instant FPGA Compilation," *Embedded Systems Letters, IEEE*, vol. PP, no. 99, p. 1, 2011.

[11] X. Reves, V. Marojevic, R. Ferrus, and A. Gelonch, "FPGA's middleware for software defined radio applications," in *Field Programmable Logic and Applications, 2005. Int. Conf. on*, 2005.

[12] A. Brant and G. Lemieux, "ZUMA: An Open FPGA Overlay Architecture," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, 29 2012-may 1 2012, p. 9396.

[13] J. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science Engineering*, vol. 12, no. 3, p. 6673, 2010.

[14] S. Huang, A. Hormati, D. Bacon, and R. Rabbah, "Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary," in *ECOOP 2008 - Object-Oriented Programming*, ser. Lecture Notes in Computer Science, J. Vitek, Ed. Springer Berlin Heidelberg, 2008, vol. 5142, p. 76103.