# VirtualRC: A Virtual FPGA Platform for Applications and Tools Portability

Robert Kirchgessner, Greg Stitt, Alan George, Herman Lam
NSF Center for High-Performance Reconfigurable Computing (CHREC)
Department of Electrical and Computer Engineering
University of Florida
{kirchgessner, gstitt, george, hlam}@chrec.org

## ABSTRACT

Numerous studies have shown significant performance and power benefits of field-programmable gate arrays (FPGAs). Despite these benefits, FPGA usage has been limited by application design complexity caused largely by the lack of code and tool portability across different FPGA platforms, which prevents design reuse. This paper addresses the portability challenge by introducing a framework of architecture and middleware for virtualization of FPGA platforms, collectively named VirtualRC. Experiments show modest overhead of 5-6% in performance and 1% in area, while enabling portability of 11 applications and two high-level synthesis tools across three physical platforms.

## Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: Real-time and embedded systems.

## General Terms

Performance, Design.

## Keywords

FPGA, portability, virtual architectures.

## 1. INTRODUCTION

Field-programmable gate arrays (FPGAs) have been widely shown to often achieve significant performance improvements compared to microprocessors [6], graphics-processing units (GPUs) [10], et al., while also reducing power consumption [10]. Despite such advantages, many application designers have avoided FPGAs due to significantly lower design productivity as compared to other devices [3].

Although numerous factors lead to low productivity [4], a major contributor is the lack of application *portability* [4] across FPGA boards and systems, herein referred to as *platforms*. Differences in platform architectures prevent developers from exploiting common design reuse techniques, forcing them to redesign significant portions of an application and write platform-specific register-transfer-level (RTL) code. This problem also extends to debugging, performance analysis, and high-level synthesis (HLS) tools [4] which could ideally support any platform architecture. Existing tools, however, require a platform-support package for each individual platform, making it infeasible to support the numerous available platforms.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*FPGA'12, February 22–24, 2011, Monterey, California, USA.*
*Copyright 2012 ACM 978-1-4503-1155-7/12/02...$10.00.*

To address these problems, we introduce a framework for FPGA platform virtualization called VirtualRC (Virtual Reconfigurable Computing). VirtualRC enables application portability by providing a configurable virtual platform architecture and corresponding software middleware that the framework can potentially map onto any physical platform. With VirtualRC, application designers target a user-customizable virtual platform, which simplifies development and enables the same RTL code to execute on any supported physical platform. In this paper, we evaluate VirtualRC on three PCIe and PCI-X FPGA platforms from GiDEL, Pico Computing, and Nallatech, demonstrating a modest performance overhead of 5-6% and an area overhead of less than 1% using application case studies and benchmarks. We showcase application portability across three platforms with 11 different RTL applications that required no coding changes. We similarly demonstrate the portability of RTL synthesized from two HLS tools, ROCCC [11] and AutoESL [6].

## 2. RELATED WORK

Previous works have addressed portability via application-specialized platform interfaces. Saldaña et al. [8] proposed a method of enabling the MPI programming model across FPGA platforms via HW/SW middleware. Reves et al. [7] presented a portable virtual architecture specific to software-defined radio applications. VirtualRC is conceptually similar, but also enables virtual FPGA platforms where designers can configure any application-specialized platform architecture. Coole et al. [1]

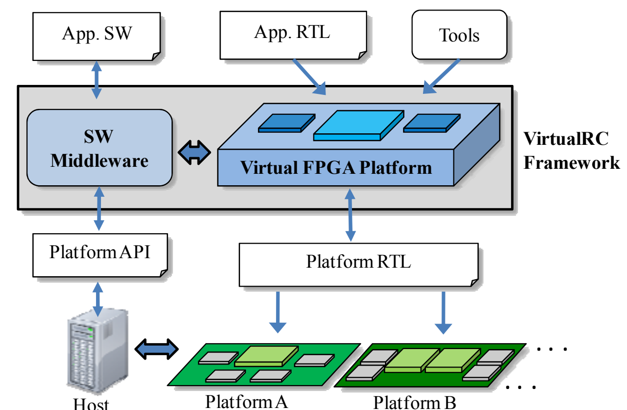


**Figure 1:** Overview of the VirtualRC framework for FPGA platform virtualization, which enables application and tool portability across multiple physical FPGA platforms.

introduced virtual FPGA devices for fast placement and routing, which is complementary to VirtualRC platform virtualization.

Standardized APIs such as OpenFPGA's GenAPI [5] and Intel's Acceleration Abstraction Layer (AAL) [2] address portability by providing a standardized software API for communicating with platform resources. Similarly, OpenCL [9] provides communication between heterogeneous devices. VirtualRC provides a unique API, but could potentially use any interface.

# 3. VirtualRC

As shown in Figure 1, VirtualRC provides a configurable *virtual FPGA platform* and a *software middleware* API for communication with the virtual platform.

To use VirtualRC, an application designer or tool first analyzes application characteristics and then requests a corresponding virtual platform architecture based upon provided configuration options. For example, a designer or tool could request one external memory with a 32-bit read port for the streaming of floating-point inputs, and another external memory with a 16-bit write port for writing fixed-point results. Given this request, VirtualRC generates a virtual platform, represented by an empty RTL entity, whose interface matches the requested configuration of resources. For the previous example, the virtual platform interface would have a 32-bit input corresponding to the read port of one virtual memory, and a 16-bit output corresponding to the write port of the second memory, in addition to control signals. The application designer then writes their application RTL code using the virtual platform as a top-level interface. Alternatively, an HLS tool could generate an application circuit that connects to the virtual platform interface. Finally, a set of platform RTL, ideally provided by the physical platform vendor, implements the virtual platform architecture on the physical platform by converting the interfaces and protocols into those used by the physical platform. Although the exact structure of platform RTL depends upon the virtual and physical platforms, for the platforms we evaluated most of this RTL consisted of simple control logic and specialized buffers for changing streaming data widths.

In creating the virtual platform architecture, we analyzed numerous FPGA platforms from GiDEL, Nallatech, DRC, Pico, and XtremeData, and identified several architectural features common to all platforms: 1) one or more FPGAs; 2) a platform bus for communicating between the host and the FPGA platform; 3) an FPGA communication controller that allows software to access on-chip resources such as block RAM and registers; and 4) one or more external memories. The virtual platform architecture provides these same four resources, as shown in Figure 2, with a unified interface and communication protocol that allows designers and tools to use the same RTL code on any supported platform. Details of the interfaces and communication protocols are omitted here for brevity.

The virtual platform supports the following configuration options. For the external memories, the virtual platform allows designers to specify the number of virtual memories, the size, the number of ports (read or write), and the data width of each port. To enable an arbitrary number of ports per virtual memory, the virtual memory interface contains an arbiter using a round-robin policy for concurrent accesses. Virtual memories may either be mapped externally to physical memory or internally to on-chip memory, depending upon application requirements and available resources. Currently, designers are required to manually perform this mapping, which we will automate in future work. For the FPGA

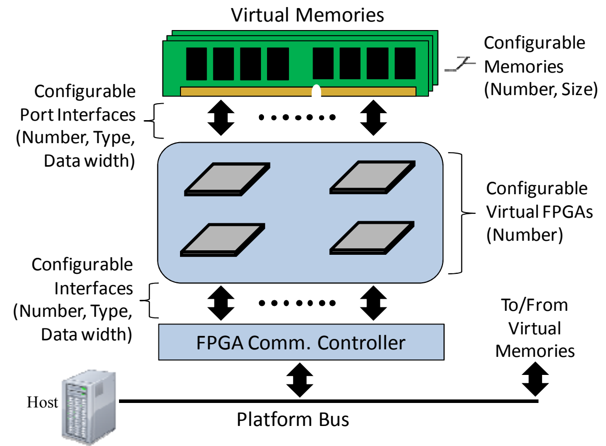


**Figure 2:** Virtual platform architecture overview.

communication controller, the platform allows designers to configure different data widths and numbers of controllers. The platform also supports a configurable number of virtual FPGA devices, although at present each virtual device simply acts as a top-level entity for a corresponding physical FPGA.

To enable software code portability, VirtualRC provides a simple C++ middleware API with overloaded read and write primitives that enable transparent communication of a variable or array to and from virtual resources. The API directly translates VirtualRC communication routines to virtual components into native API calls to the physical platform. The goal of the current API is to show proof of concept for communication with the virtual platform; the middleware could potentially use any API.

# 4. EXPERIMENTS

In this section, we describe the experimental setup (4.1) and then analyze performance and area overhead (4.2). We then evaluate the portability of applications and high-level synthesis tools (4.3) using VirtualRC across multiple physical platforms.

## 4.1 Experimental Setup

In our experiments, we evaluate VirtualRC on three different FPGA platforms: the GiDEL PROCStar III; Nallatech H101; and Pico Computing M501. Each platform has a significantly different platform architecture and API, and all use a variety of FPGAs from different vendors. To support each platform, we manually created the platform RTL shown in Figure 1, which required several days to several weeks. However, vendors could add this support in much less time due to familiarity with their platforms.

Bitstreams for the GiDEL PROCStar III were generated using Altera Quartus 9.1 SP2. Bitstreams for the Pico M501 and Nallatech H101 were generated using Xilinx ISE 12.3. Driver versions used were 8.8 (GiDEL), FUSE 1.5 (Nallatech) and a pre-release version of the M501 drivers (Pico). All software was compiled used g++ version 4.4.3 with –O3 optimizations.

## 4.2 Overhead Analysis

This section analyzes VirtualRC overhead. Section 4.2.1 presents case studies that evaluate application performance overhead. We then analyze FPGA memory bandwidth overhead (4.2.2), software middleware overhead (4.2.3), and resource overhead (4.2.4).

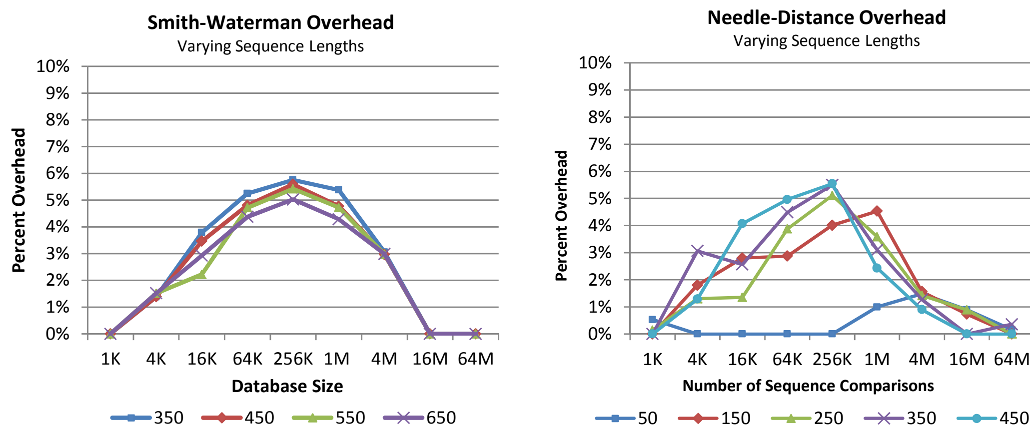


**Figure 3:** Performance overhead of VirtualRC compared to native PROCStar III implementations of Smith-Waterman and Needle-Distance applications.

### *4.2.1 Application Case Studies*

This section evaluates VirtualRC performance overhead by comparing application performance of native implementations for a physical platform with implementations targeting VirtualRC on the same platform. We obtained source code for previously published implementations of Smith-Waterman and Needle-Distance [6]. We chose these two bioinformatics applications due their existing implementation on the GiDEL PROCStar III and substantial speedup over an optimized software baseline.

To evaluate overhead, we used VirtualRC to create the virtual platform interface required by the applications. We then manually mapped the application interfaces to the virtual resource interfaces. Porting the application required no modifications to the application code, and took approximately half an hour per application (not including compilation). We also created software based upon the original application software using the VirtualRC API. We then executed these applications on the PROCStar III, both with and without VirtualRC, for varying input sizes.

Figure 3 illustrates the performance overhead of the Smith-Waterman and Needle-Distance applications on VirtualRC for the PROCStar III. Across all input sizes, VirtualRC had a peak performance overhead of 6% for Smith-Waterman and 5% for Needle-Distance, with the overhead approaching 0% for large inputs sizes in both cases. This overhead was due to differences in the software required to interface VirtualRC with the original FIFO interface. In future work, we will add a configurable FIFO interface option to VirtualRC that will minimize this overhead. VirtualRC had an area overhead of approximately 1% for both applications due to the configurable virtual resources.

### *4.2.2 External Memory Bandwidth Overhead*

For each platform (PROCStar III, M501, and H101), we measured the effective memory bandwidth with and without VirtualRC for varying transfer sizes. For VirtualRC, we configured a single virtual memory with a width equal to the native memory width. For the case without VirtualRC, we used a state machine and FIFO buffer to characterize a streaming application without blocking. We then measured the number of clock cycles required to transfer a specified amount of data.

Table 1 evaluates the FPGA to external memory overhead. Overhead was smallest on the PROCStar III due to similarities between the GiDEL IP and VirtualRC interfaces, with a maximum overhead of 5% for small transfers and negligible overhead for large transfers. For the H101, maximum overhead was 8% for small transfers and again negligible for large transfers. The VirtualRC overhead was largest on the M501 due to significant differences in interface and features from VirtualRC. The M501 IP limits transfer from 32 to 4096 bytes. Although VirtualRC adds additional overhead, it also enables transfers of any size. We measured a peak overhead of 25% for small transfers, which decreased substantially for transfers over 1KB. Since FPGA applications commonly use large data streams, the overhead of VirtualRC in these cases would be negligible.

### *4.2.3 Software Middleware Overhead*

In order to analyze the software overhead for memory accesses, we measured the effective bandwidth for each platform using the VirtualRC API versus physical API for varying transfer sizes.

Table 2 evaluates the host to external memory overhead. The PROCStar III had the smallest overhead of 6% due to similarities between GiDEL and VirtualRC APIs. Maximum overhead on the H101 was 27% for small transfers, and became negligible for transfers above 32 KB. The largest overhead of 46% was measured for the M501, but only for transfers of 16 to 64 bytes. Since data transfers between host and FPGAs are costly, applications tend to avoid small transfers, making the overhead of VirtualRC insignificant for common situations.

**Table 1:** Read (write) overhead from FPGA to external memory due to VirtualRC memory virtualization.

| | 16B | 1KB | 16KB | 256KB | 1MB |
|---|---|---|---|---|---|
| PROCStar III | 5%(4%) | 3%(2%) | 1%(1%) | 0%(0%) | 0%(0%) |
| H101 | 4%(8%) | 0%(0%) | 0%(0%) | 0%(0%) | 0%(0%) |
| M501 | 25%(19%) | 5%(4%) | 2%(2%) | NA | NA |

**Table 2:** Read (write) overhead from host to FPGA memory due to VirtualRC middleware API.

| | 16B | 1KB | 16KB | 256KB | 1MB |
|---|---|---|---|---|---|
| PROCStar III | 5%(0%) | 6%(0%) | 0%(1%) | 0%(0%) | 0%(0%) |
| H101 | 16%(27%) | 13%(27%) | 12%(17%) | 0%(0%) | 0%(0%) |
| M501 | 44%(46%) | 1%(0%) | 1%(0%) | 0%(1%) | 0%(0%) |

### *4.2.4 FPGA Resource Overhead*

Resource overhead from VirtualRC stems from three major components: the FPGA communication controller; the virtual

memory read interface; and the virtual memory write interface. We determined overhead for each component using the appropriate vendor tool (Quartus, ISE). Each component was found to require less than 1% of total device resources in terms of LUTs, registers, and block RAMs. The exact resource usage on each platform varied based on how similar the native IP interface was to the VirtualRC interface specification.

**Table 3:** Demonstration of VirtualRC application portability.

| | PROCStar III | | M501 | | H101 | |
|---|---|---|---|---|---|---|
| | Freq. (MHz) | Time (ms) | Freq. (MHz) | Time (ms) | Freq. (MHz) | Time (ms) |
| *1D Convolution FP* | 125 | 39.29 | 125 | 247.90 | 100 | 91.06 |
| *2D Convolution FP* | 106 | 13.18 | 106 | 15.18 | 100 | 43.25 |
| *Option Pricing* | 125 | 12.15 s | 125 | 14.40 s | - | - |
| *Sum Abs. Differences* | 98 | 14.72 | 98 | 15.62 | 98 | 86.71 |
| *Needle Distance* | 125 | 194.00 | 125 | 116.20 | 100 | 199.51 |
| *Smith Waterman* | 125 | 116.00 | 125 | 133.00 | 100 | 225.00 |
| *Image Segmentation* | 125 | 12.40 | 125 | 16.39 | 100 | 4.81 |
| *OpenCores SHA256* | 125 | 64.05 | 125 | 120.49 | 100 | 25.97 |
| *OpenCores FIR* | 125 | 24.51 | 125 | 413.80 | 100 | 106.16 |
| *OpenCores AES128* | 125 | 25.33 | 125 | 503.78 | 100 | 126.18 |
| *OpenCores JPEG Enc.* | 125 | 15.29 | 125 | 23.93 | 100 | 21.24 |

**Table 4:** Demonstration of VirtualRC tool portability.

| | PROCStar III | | M501 | | H101 | |
|---|---|---|---|---|---|---|
| | Freq. (MHz) | Time (ms) | Freq. (MHz) | Time (ms) | Freq. (MHz) | Time (ms) |
| *ROCCC 8pt FFT* | 125 | 15.66 | 125 | 16.61 | 100 | 39.91 |
| *ROCCC 5-tap FIR* | 125 | 17.78 | 125 | 18.73 | 100 | 40.57 |
| *AutoESL Convolution* | 125 | 4.29 | 125 | 7.31 | 100 | 2.49 |

## 4.3 Portability Analysis

In this section we demonstrate application and tool portability provided by the VirtualRC framework. To evaluate portability, we created a variety of applications for VirtualRC and also obtained examples from OpenCores (www.opencores.org) that we modified to use VirtualRC. We then executed each application and tool on the three aforementioned vendor platforms, using the exact same application code. Each application consists of RTL code in addition to C++ code using the VirtualRC middleware that transfers data to and from the virtual platform. Since performance was not the goal of these experiments, a frequency of 125 MHz was used except in cases where the estimated design operating frequency was lower.

Table 3 summarizes a list of applications used to demonstrate portability on all three platforms. VirtualRC successfully executed all applications on each platform, with the exception of Option Pricing, which would not fit on the H101. This limitation was not a consequence of VirtualRC, but was caused by non-parameterized RTL code that we could not reduce in size to fit on the older Virtex-4 FPGA on the H101. It is important to note that this table is *not* intended as a performance comparison due to significant platform differences. Instead, the purpose here is to show characteristics of each application on each system.

Table 4 demonstrates tool portability. As with Table 3, these results are *not* intended to be compared across devices or between tools. Instead, these results verify that VirtualRC enables identical HLS code synthesized into RTL to work seamlessly across multiple platforms that are not directly supported by the tools. Therefore, VirtualRC potentially enables HLS tools to support any platform without effort from the tool vendors.

# 5. LIMITATIONS AND FUTURE WORK

There are several limitations that we plan to address as future work which include expanding VirtualRC to enable embedded processor architectures, further reducing overhead for different use cases, estimating virtual platform performance for design exploration, and integrating alternative software APIs.

# 6. CONCLUSIONS

To address challenges in FPGA design productivity owing to lack of code portability, we introduced VirtualRC. VirtualRC provides developers with a user configurable virtual platform interface, enabling portability across any supported platform. We evaluated VirtualRC using applications and benchmarks, and show a performance overhead of 5-6% with an area overhead of less than 1%. We also demonstrated that VirtualRC enables application and tool portability by executing the same code for 11 different applications and two tools across three physical platforms.

# 7. ACKNOWLEDGEMENTS

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. EEC-0642422. The authors gratefully acknowledge vendor equipment and tools provided by Altera, GiDEL, Nallatech, Pico Computing, and Xilinx that helped make this work possible.

# 8. REFERENCES

[1] Coole, J., and Stitt, G. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP Int. Conf. on* (2010).

[2] Intel. Intel QuickAssist Technology AAL (White Paper).

[3] Jones, D., Powell, A., Bouganis, C.-S., and Cheung, P. GPU versus FPGA for high productivity computing. In *Field Programmable Logic and Applications (FPL), 2010 Int. Conf. on* (2010).

[4] Merchant, S., Holland, B., Reardon, C., George, A., Lam, H., Stitt, G., Smith, M., Alam, N., Gonzalez, I., El-Araby, E., Saha, P., El-Ghazawi, T., and Simmler, H. Strategic challenges for application development productivity in reconfigurable computing. In *Aerospace and Electronics Conference, 2008. IEEE National* (2008).

[5] OpenFPGA. OpenFPGA GenAPI version 0.4 Draft for Comment.

[6] Pascoe, C., Lawande, A., Lam, H., George, A., Sun, Y., Farmerie, W., and M., H. Reconfigurable supercomputing with scalable systolic arrays and in-stream control for wavefront genomics processing. In *Proc. of Symposium on Application Accelerators in High-Performance Computing* (2010).

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

[8] Saldaña, M., Patel, A., Madill, C., Nunes, D., Wang, D., Chow, P., Wittig, R., Styles, H., and Putnam, A. MPI as a programming model for high-performance reconfigurable computers. *ACM Trans. Reconfigurable Technol. Syst. 3* (November 2010), 22:1–22:29.

[9] Stone, J., Gohara, D., and Shi, G. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering 12*, 3 (may-june 2010), 66 –73.

[10] Tian, X., and Benkrid, K. High-performance quasi-monte carlo financial simulation: FPGA vs. GPP vs. GPU. *ACM Trans. Reconfigurable Technol. Syst. 3* (November 2010), 26:1–26:22.

[11] Villarreal, J., Park, A., Najjar, W., and Halstead, R. 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.