# Low-Overhead FPGA Middleware for Application Portability and Productivity

ROBERT KIRCHGESSNER, ALAN D. GEORGE, and GREG STITT, University of Florida

Reconfigurable computing devices such as field-programmable gate arrays (FPGAs) offer advantages over fixed-logic CPU and GPU architectures, including improved performance, superior power efficiency, and reconfigurability. The challenge of FPGA application development, however, has limited their acceptance in high-performance computing and high-performance embedded computing applications. FPGA development carries similar difficulties to hardware design, requiring that developers iterate through register-transfer level designs with cycle-level accuracy. Furthermore, the lack of hardware and software standards between FPGA platforms limits productivity and application portability, and makes porting applications between heterogeneous platforms a time-consuming and often challenging process. Recent efforts to improve FPGA productivity using high-level synthesis tools and languages show promise, but platform support remains limited and typically is left as a challenge for developers. To address these issues, we present RC Middleware (RCMW), a novel middleware that improves productivity and enables application and tool portability by abstracting away platform-specific details. RCMW provides an application-centric development environment, exposing only the resources and standardized interfaces required by an application, independent of the underlying platform. We demonstrate the portability and productivity benefits of RCMW using four heterogeneous platforms from three vendors. Our results indicate that RCMW enables application productivity and improves developer productivity, and that these benefits are achieved with less than 7% performance and 3% area overhead on average.

Categories and Subject Descriptors: B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids

General Terms: Design, Performance, Standardization

Additional Key Words and Phrases: Accelerators, FPGA, heterogeneous computing, middleware, portability, productivity, reconfigurable computing

**ACM Reference Format:**
Robert Kirchgessner, Alan D. George, and Greg Stitt. 2015. Low-overhead FPGA middleware for application portability and productivity. ACM Trans. Reconfigurable Technol. Syst. 8, 4, Article 21 (September 2015), 22 pages.
DOI: http://dx.doi.org/10.1145/2746404

## 1. INTRODUCTION

Field-programmable gate arrays (FPGAs) allow developers to create application-specific hardware architectures, enabling several orders of magnitude performance improvement [El-Ghazawi et al. 2008; Pascoe et al. 2010] while also improving

This work was supported in part by the I/UCRC Program of the National Science Foundation under grants EEC-0642422 and IIP-1161022. The authors gratefully acknowledge equipment provided by Altera, GiDEL, Pico Computing, and Nallatech that helped make this work possible.
Authors' addresses: R. Kirchgessner, 317 Larsen Hall, P.O. Box 116200, Gainesville, FL 32611-6200; email: kirchgessner@hcs.ufl.edu; A. D. George, 327 Larsen Hall, P.O. Box 116200, Gainesville, FL 32611-6200; email: george@hcs.ufl.edu; G. Stitt, 323 Benton Hall, P.O. Box 116200, Gainesville, FL 32611-6200; email: gstitt@ece.ufl.edu.

DOI: http://dx.doi.org/10.1145/2746404

computational efficiency [Williams et al. 2010; Betkaoui et al. 2010] for applications that do not map well to conventional CPU and GPU architectures. This flexibility and efficiency has made FPGAs ideal for various applications, from embedded systems [Garcia et al. 2006] to supercomputers [George et al. 2011].

These benefits, however, come with the added complexity of hardware design, limiting developer productivity relative to fixed-logic devices, and preventing widespread usage of FPGAs. The difficulty of register-transfer level (RTL) design coupled with a lack of standards between FPGA accelerator platforms, herein referred to as platforms, complicates application development and limits code reusability. Due to a lack of standards between platforms, developers must tailor their application to a specific vendor's software and hardware interfaces. This platform-specific development cycle prevents portability, requiring significant developer time and effort to port applications to new platforms. Additionally, vendor-specific procedural APIs further limit portability. Procedural APIs embed platform-specific parameters into application code, including data marshalling and the physical location of application resources.

These portability issues extend to high-level synthesis (HLS) tools and languages, which intend to improve developer productivity. 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 challenge of supporting new platforms to application developers. These problems ultimately reduce HLS tool performance and usability, and end up costing tool vendors and application developers valuable time that could better be spent on developing their tools and applications.

To help overcome the portability and productivity hurdles of FPGA application development, we present RC Middleware (RCMW). RCMW is a layered middleware that enables application and tool portability by creating an application-specific platform abstraction. Developers specify their application's required resources and interfaces at design time, customizing the number, type, size, and data types of interfaces. Using this specification, RCMW provides a portable application-specific hardware and software interface. One major research challenge for enabling application portability is providing standardized interfaces to application-specific resources that are independent of the underlying platform while also minimizing overhead. Platform details such as the number and type of FPGAs, and size and performance of external memories, require careful consideration when mapping an application onto a target platform. To address these challenges, the RCMW toolchain determines the application-to-platform mapping at compile time, selecting an appropriate mapping based on a user-customizable cost function. Using RCMW, developers can focus on their application or tool rather than implementing their designs onto a specific platform.

In this article, we present and evaluate RCMW using four platforms from three vendors: the PROCStar III and PROCStar IV from GiDEL, the M501 from Pico Computing, and the PCIe-385n from Nallatech. We demonstrate the ability to quickly explore different application-to-platform mappings with the RCMW toolchain using a representative convolution case study. We show that the benefits of RCMW can be achieved with minimal overhead—less than 7% performance and 3% area in the common case. We also demonstrate RCMW's productivity benefits by showing that it requires less development time and lines of code for deploying applications compared to the recommended vendor approaches. Finally, we demonstrate application portability using RCMW by executing the same application hardware and software source, for several applications and kernels, across each supported platform.

The remainder of this article is organized as follows. Section 2 presents background and related work. Section 3 presents the RCMW framework and toolchain. Section 4 presents our experiments, results, and analysis. Section 5 presents our conclusions and future work.

## 2. BACKGROUND AND RELATED WORK

The lack of FPGA-accelerator standards has resulted in the development of vendor-specific APIs that limit application portability and developer productivity. To address this issue, OpenFPGA proposed a procedural C-based API standard for managing RC accelerators [OpenFPGA Inc. 2008]. The OpenFPGA standard defines functions for initializing, managing, and communicating with FPGA accelerators but requires developers to embed platform-specific application details such as the physical location of application resources. The Simple Interface for Reconfigurable Computing (SIRC) [Eguro 2010] is an object-oriented communication interface that provides functionality similar to the OpenFPGA standard but enables portability using platform-specific subclasses. Although OpenFPGA and SIRC define comprehensive APIs, both require embedding platform-specific application details, which limits application portability. RCMW also provides a portable API standard but overcomes this limitation by providing an object-oriented representation of application resources that encapsulates platform-specific details and enables application portability.

HLS tools address the productivity hurdles of FPGA application design by providing high-level software-style development environments but typically have limited platform support. HLS tools such as ROCCC [Villarreal et al. 2010] and Impulse-C [Stone et al. 2010] provide a C-style development environment and stream-optimized programming model but take different approaches to platform support. ROCCC generates RTL cores with streaming interfaces but requires that developers handle the platform-specific implementation. Impulse-C generates synthesizable HDL cores and an application driver from a single application source and enables portability using platform-support packages (PSPs). PSPs wrap platform-specific interfaces to enable portability; however, due to their complexity and the large number of available platforms, developing PSPs is typically left as a challenge for the end user. Recent efforts such as Altera OpenCL [Czajkowski et al. 2012] enable developers to create portable FPGA application kernels using OpenCL. Similar to Impulse-C's PSPs, Altera OpenCL uses board-support packages (BSPs) to target a specific platform.

The FUSE framework [Ismail and Shannon 2011] provides an OS-level abstraction of hardware accelerator resources, transparently scheduling software tasks on available hardware accelerators. Similarly, SPREAD [Wang et al. 2013] provides a unified hardware and software threading model but takes advantage of partial reconfiguration to dynamically schedule hardware tasks. Liquid Metal (Lime) [Huang et al. 2008] also defines a unified hardware and software threading model but enables developers to create mixed FPGA and CPU applications using Java. Similar to Lime, hthreads [Andrews et al. 2008] enables developers to create mixed applications but instead uses a C-based POSIX threading model. To target a platform, these tools and frameworks must provide a custom platform-specific hardware and software support package. RCMW is a complementary approach and could be leveraged by these tools and frameworks to generate a customized portable support package, allowing tool developers to focus on improving their tools instead of platform support.

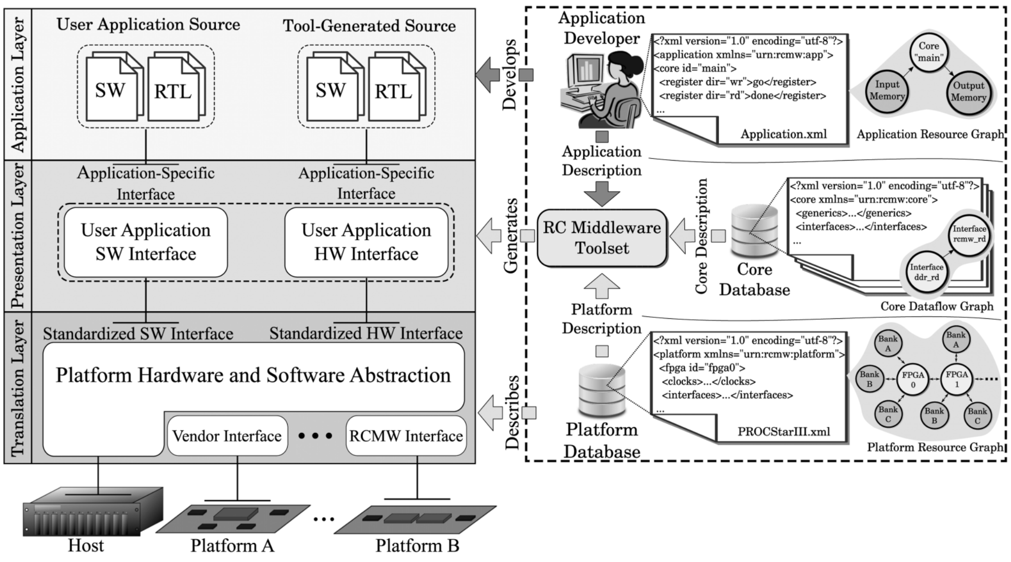
System-design tools such as SpecC [Cai et al. 2001] assist developers with design-space exploration and partitioning applications across multiple devices. SpecC enables developers to create a high-level application specification and refine it to select an architecture model, communication model, and finally create synthesizable RTL. OpenCPI [Kulp 2010] is a component-based application middleware for heterogeneous systems that enables seamless communication between components across devices including FPGAs, GPUs, and CPUs. Similarly, the System Coordination Framework [Aggarwal et al. 2012] simplifies task communication between heterogeneous devices, including CPUs and FPGAs, by creating a partitioned global address space. SIMPPL [Shannon and Chow 2005] and IMORC [Schumacher et al. 2009] provide frameworks for creating applications from networks of components on a single FPGA. SIMPPL wraps IP cores with a core-specific network controller and enables asynchronous communication. IMORC also creates a network of components but uses a multibus interconnect architecture. Although these approaches simplify the development of component-based applications, they still require significant developer time and effort to port application to new platforms. RCMW is a related approach that automatically handles mapping application components and resources onto a target platform using a customizable mapping algorithm. RCMW could be leveraged by these tools to handle FPGA-component mapping and provide portable hardware and software interfaces to components.

An alternative approach to enabling FPGA application portability is to create virtual-FPGA overlays of application-specific resources. Intermediate fabrics [Stitt and Coole 2011] are coarse-grained virtual-FPGA fabrics customized for a particular application domain. Similarly, Reves et al. [2005] presents a device-level middleware with customizable resources for software-defined radio applications. These approaches enable device-level portability by providing the same coarse-grained resources independent of the target device. RCMW is a complementary approach and could provide portable resource interfaces to these virtual-FPGA fabrics.

Platform vendors typically provide tools to assist with application development. Two notable examples are Nallatech's DIMEtalk [Nallatech Ltd. 2007] and GiDEL's PROCWizard [GiDEL Ltd. 2014]. DIMEtalk provides a graphical interface to create networks of components and generate FPGA bitfiles. PROCWizard generates an HDL wrapper and C++ interface based on developer-specified clocks, registers, and customized physical-memory interfaces. Since developers design their applications by customizing platform-specific resources, effort is still required when porting between platforms. To overcome this limitation, RCMW enables developers to configure application-specific resources without assuming any knowledge of the underlying platform.

LEAP scratchpads [Adler et al. 2011] provide cached virtual memory interfaces and simplify FPGA application memory management. Altera's Avalon [Altera Corp. 2007] and ARM's AXI [ARM 2013] protocol were created to enable component interoperability and define streaming and memory-mapped interfaces. RCMW defines interfaces optimized for streaming applications but can be extended to support any interface using the extensible core library. LEAP scratchpads, Avalon, and AXI could be added to RCMW, allowing developers to request the ideal interface for each application resource while maximizing performance and minimizing design area.

RCMW enables application portability by providing an application-specific view of available hardware and software interfaces, independent of the underlying platform. Using RCMW, developers specify the required resources and interfaces needed by their applications at design time, and RCMW handles determination of application-to-platform mapping at compile time. RCMW is extensible, allowing support for new interface and resource types to be added by extending the RCMW core library. An earlier version of this work can be found in Kirchgessner et al. [2013], in which we demonstrate a previous version of RCMW. Since that work, we have developed an RCMW driver that enables us to support platforms without vendor support packages. Leveraging our driver, we added support for the Nallatech PCIe-385n featuring an Altera Stratix-V FPGA and explored this platform in our experiments. Additionally, we have extended the RCMW toolchain to include a best first search algorithm to select the application-to-platform mapping. This algorithm provides a faster alternative to the exhaustive approach presented in our previous work. In addition to a case study demonstrating the RCMW design methodology, we have included results for the platforms from our earlier work in this article for completeness.



```
<?xml version="1.0" encoding="utf-8"?>
<application xmlns="urn:rcmw:app">
<core id="main">
  <register dir="wr">go</register>
  <register dir="rd">done</register>
...
```

```
<?xml version="1.0" encoding="utf-8"?>
<core xmlns="urn:rcmw:core">
  <generics>...</generics>
  <interfaces>...</interfaces>
...
```

```
<?xml version="1.0" encoding="utf-8"?>
<platform xmlns="urn:rcmw:platform">
  <fpga id="fpga0">
   <clocks>...</clocks>
   <interfaces>...</interfaces>
...
```

(a) RC Middleware Layered Model (b) Design Methodology

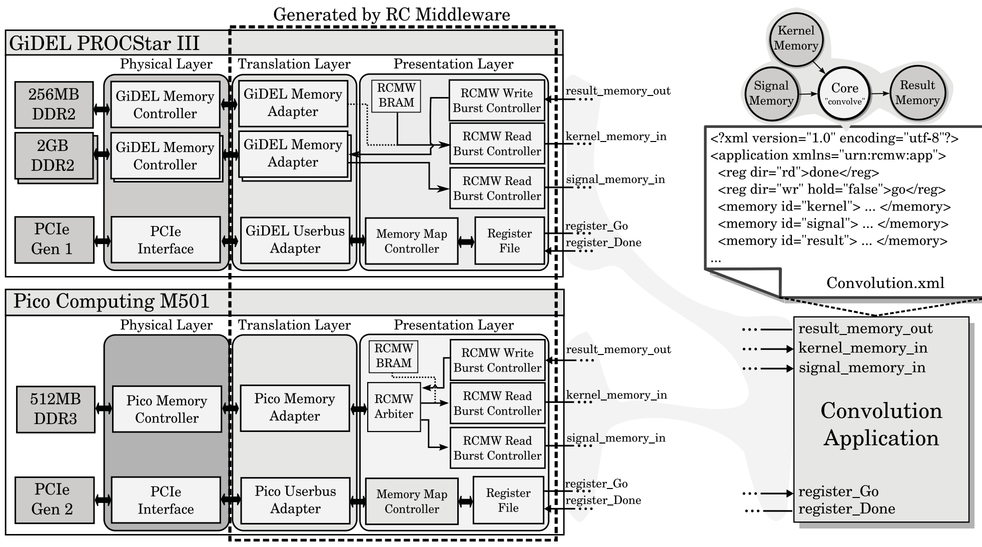
Fig. 1. Overview of RCMW. (a) Layered model of hardware and software abstractions enabling application portability across FPGA platforms. (b) Overview of design methodology for executing applications on specific platforms.

## 3. RC MIDDLEWARE OVERVIEW

To enable FPGA application portability, we must provide a standardized view of application resources independent of the underlying target platform hardware configuration and software API. RCMW enables this standardized view using customizable hardware and software middlewares consisting of three layers of abstraction, as shown in Figure 1(a). From the bottom up, these layers are the translation layer, presentation layer, and application layer. First, the translation layer translates platform-specific hardware and software interfaces to standardized RCMW interfaces. Next, the presentation layer leverages these standardized interfaces, creating the application-specific hardware and software interfaces specified by the developer. Finally, these application-specific resources and interfaces are presented to the developer in the application layer, independent of the underlying platform.

Figure 1(b) overviews RCMW's design methodology. Using RCMW, the application developer only needs to develop the application hardware and create an XML-based description of the application resources and interfaces. The developer then provides the application description to the RCMW toolchain and specifies a supported target platform. RCMW selects an application-to-platform mapping using a customizable cost function optimizing for device area or interface latency. RCMW then uses the selected mapping to generate a ready-to-compile project to create bitfiles and a C++ class that provides interfaces to the application resources and application software stub. Although RCMW is intended to enable application portability regardless of the application class, the RCMW core library currently provides cores optimized for streaming applications, which are the focus of our case studies in this article.

The remainder of this section is organized as follows. Section 3.1 presents the RCMW hardware abstraction layers. Section 3.2 presents the RCMW software abstraction layers. Section 3.3 discusses the RCMW XML metadata formats and extensible core library. Finally, Section 3.4 presents the RCMW toolchain and mapping algorithm.



```
<?xml version="1.0" encoding="utf-8"?>
<application xmlns="urn:rcmw:app">
  <reg dir="rd">done</reg>
  <reg dir="wr" hold="false">go</reg>
  <memory id="kernel"> ... </memory>
  <memory id="signal"> ... </memory>
  <memory id="result"> ... </memory>
...
```

Fig. 2. RCMW hardware abstraction layers enabling application portability between GiDEL PROCStar III and Pico M501.

### 3.1. Hardware Abstraction

Figure 2 illustrates an example of the three layers of hardware abstraction enabling portability for a convolution application. In this example, the developer has specified two input memories, one for the convolution kernel and one for the input signal, and one output memory for the convolution results. Since the GiDEL PROCStar III board has three external memories, each application memory can be assigned to a separate physical memory. The Pico M501, however, only has one external memory, requiring that the three application memories either share a single physical memory or make use of on-chip block RAM (BRAM). The developer also requested a go and done memory-mapped register for triggering the application and waiting for it to finish.

The physical layer consists of the low-level hardware interface controllers for external memories and host communication. We have leveraged vendor-supplied components for these interfaces wherever possible to avoid recreating existing interfaces without any significant benefits. In cases where no vendor components are provided, such as for the Nallatech PCIe-385n, we leveraged Altera/Xilinx IP cores and created custom HDL components. The translation layer is responsible for converting the platform-specific interfaces from the physical layer into a standardized interface that the rest of the RCMW toolchain understands. This layer is generated by the RCMW toolchain leveraging the RCMW core library and depends on the application-to-platform mapping. The presentation layer handles creation of the application-specific interfaces requested by the developer using the standardized interfaces exposed by the translation layer. The presentation layer is customized by the RCMW toolchain based on available platform resources and requested application resources, and it is generated at compile time. The HDL cores leveraged in generating this layer are stored in an extensible core library, which is discussed later in Section 3.3.

To enable a configurable number of developer-requested interfaces to platform resources, the RCMW core library includes a configurable arbitration controller. This arbitration controller can handle multiplexing any number and type of application resources to a physical resource or BRAM. Although having too many application resources mapped to a single platform resource could degrade performance, this

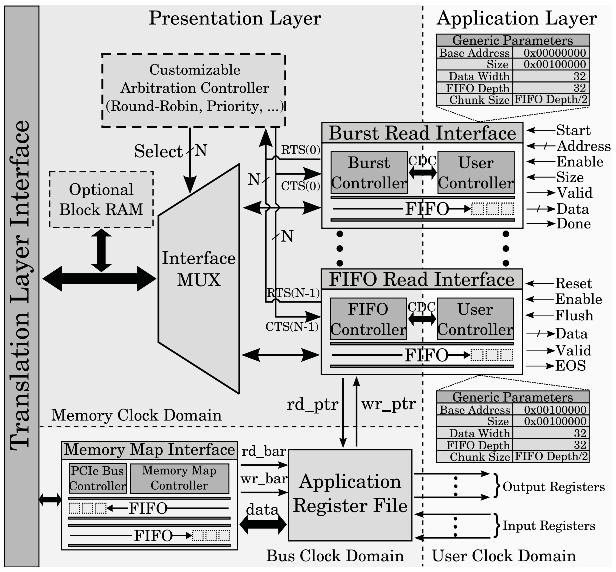
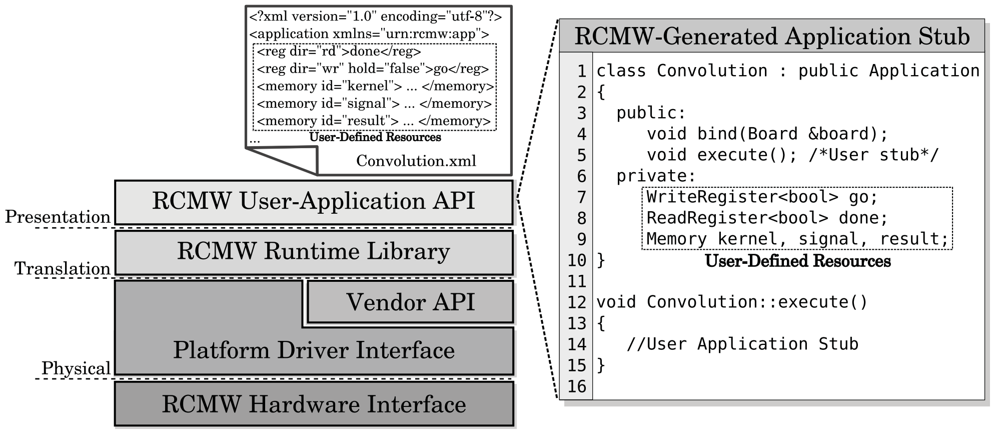


Fig. 3. Overview of RCMW's hardware presentation layer.

controller is required to enable application portability between platforms with different resource configurations. When available physical memory bandwidth is greater than the required application bandwidth, the middleware can saturate multiple application interfaces without significant loss in performance. RCMW's customizable arbitration controller uses a request-to-send (RTS) and clear-to-send (CTS) protocol to arbitrate between application interface controllers. This protocol can be used to implement any arbitration scheme, from simple round-robin to adaptive arbitration schemes similar to Hao and Stitt [2012], allowing RCMW to optimize design area and performance depending on application configuration.

RCMW currently provides two standardized interface protocols: the burst interface and FIFO interface. These protocols were selected because they are commonly used in streaming applications, but additional interface protocols can be supported by extending the RCMW core library. The burst interface enables applications to address an application memory sequentially. The interface word size can be any power-of-two number of bytes. The application specifies the starting byte-aligned *address*, *size* in memory words, and asserts the *start* signal to begin a transfer. The interface will transfer the requested amount of data and assert the *done* signal. The FIFO interface enables application software to read or write data streams to application hardware in a first-in, first-out order. The FIFO word size can be any power-of-two number of bytes. The application first toggles the *reset* signal to reset the FIFO buffer and read/write pointers. Then the application reads/writes data to the interface, asserting the *flush* signal for write interfaces when the stream is empty. When the read or write stream is complete, the *EOS* signal is asserted, indicating the end of the data stream. Both interface types require the *enable* and read *valid* or write *ready* signals for flow control. Flow control is required by all interfaces due to differences in performance between platforms.

Figure 3 provides a detailed illustration of the presentation layer. Each application memory has one or more interface. Using the configurable arbitration module described previously, any number of application memories and interfaces can be mapped by the RCMW toolchain to a physical memory. In the case that multiple application memories



```
<?xml version="1.0" encoding="utf-8"?>
<application xmlns="urn:rcmw:app">
 <reg dir="rd">done</reg>
 <reg dir="wr" hold="false">go</reg>
 <memory id="kernel"> ... </memory>
 <memory id="signal"> ... </memory>
 <memory id="result"> ... </memory>
...
```

```
class Convolution : public Application
{
  public:
     void bind(Board &board);
     void execute(); /*User stub*/
  private:
     WriteRegister<bool> go;
     ReadRegister<bool> done;
     Memory kernel, signal, result;
}

void Convolution::execute()
{
   //User Application Stub
}
```

Fig. 4. Overview of RCMW's software stack and generated C++ application stub.

are mapped by RCMW to the same physical resource, there must be a virtual separation to prevent resources from affecting each other. This virtual separation is created by the RCMW toolchain using the generic parameters, including the base address and memory size, of each HDL interface controller. The base address corresponds to the address in physical memory where the application memory begins. The size of the memory is used to calculate address-wrapping conditions. In addition to memory interfaces, RCMW provides a separate memory-mapped interface to the application. This interface maps application resources, such as memory-mapped registers, to a host-controlled bus. The application layer presents the application-specific HDL interfaces specified in the application description to each application core and generates a vendor-specific project for each FPGA where an application core is mapped.

## 3.2. Software Abstraction

Each hardware abstraction layer described in the previous section has a corresponding layer in software. Figure 4 illustrates the layered software model and RCMW-generated application stub. RCMW uses a portable object-oriented software API that provides standardized interfaces to application resources.

The physical layer corresponds to the software driver interface. Although we try to leverage vendor-supplied drivers wherever possible to minimize development overhead, we developed a RCMW PCI-Express driver for platforms without a vendor-provided driver, such as the Nallatech PCIe-385n. The software translation layer wraps platform-specific APIs and provides a standardized software interface to platform resources. RCMW requires that each supported platform has a subclass of the RCMW *Board* class. This *Board* class defines the required interfaces for the upper API levels, such as blocking and nonblocking DMA read/write, board enumeration and initialization, clock configuration, and bitfile programming. The *Board* class encapsulates *FPGA* and *Memory* objects that represent physical platform components.

The presentation layer handles mapping the application-specific resource interfaces onto the *Board* class interface provided by the translation layer. This layer is generated by the RCMW toolchain as a subclass of the RCMW *Application* class. Figure 4 illustrates the application-specific interface for a convolution example, with two registers, *go* and *done*, and three memories, *kernel*, *signal*, and *result*. The RCMW toolchain-generated *Application* subclass encapsulates an instance of each resource specified by the application description. It provides *Register* objects, which are mapped onto the memory-mapped interface, *Memory* objects, which correspond to hardware sequential interfaces, and *FIFO* objects, which correspond to FIFO hardware interfaces. The *Application* class provides two functions: *bind(...)* and *execute(...)*. The bind function is generated by the RCMW toolchain along with the application stub. The bind function handles the mapping of application resources onto a target platform at runtime, based on the RCMW-selected application-to-platform mapping. The execute function is the stub where developers implement their application software using the resources exposed by the application class. RCMW provides a concurrent API that allows developers to allocate, manage, and communicate with application resources concurrently and portably. At runtime, RCMW handles detection of available platforms, selecting which platform will execute each application, initializing and configuring FPGAs, and managing threads for concurrent transfers. If a bitfile for an application is not available for a particular platform, the developer must first compile the RCMW toolchain-generated project using the vendor toolchain before being able to execute it. If a bitfile is found, the bind function is called on the selected *Board* object instance and assigns the application execute function to an idle software thread. When the application completes, RCMW releases platform resources.

The application layer exposes an application-customized subclass of an *Application* class generated by the RCMW toolchain. This approach provides a portable programming model and allows developers to launch multiple application instances with RCMW handling platform configuration and scheduling. Developers are provided with standardized application interfaces without having to worry about where or how they are mapped onto a target platform.

### 3.3. Metadata and Extensible Core Library

This section provides an overview of the various XML-based metadata formats used in RCMW. There are three different metadata formats: the application description, platform description, and core description. The application description is used by developers to specify their application's required resources and interfaces. The application description contains information about each core in an application, including HDL source files and any register or memory resources required. Application cores can specify any number of register or memory resources with any number and type of resource interfaces. The application description also contains information about the structure of the application, including any core instances, and how those instances are interconnected. An excerpt of the application description from a convolution example can be seen in Figure 5. Although details have been excluded, the overall application description can be understood. In this example, the developer specifies a core called *main*, composed of two source files: *Convolution.vhd* and *Datapath.vhd*. The developer specifies a memory with a burst read interface for storing the kernel data.

The platform description is used to describe a platform's resources, such as FPGAs and memories, as well as their interfaces and physical connections. This description enables the RCMW toolchain to understand the available resources and how they are connected. The platform description contains the hardware details of a platform, with the software details captured by the platform-specific *Board* class as described in Section 3.2. Developers or platform vendors can easily extend RCMW to include a new platform by creating a platform description. In cases where the RCMW core library contains all required HDL components, no additional coding is needed. If the platform requires device-specific IP instances or interfaces not supported by RCMW, the core library must be extended with the necessary components.

The core description describes the interfaces and function of the HDL core components in the RCMW core library. These cores are used by the RCMW toolchain to resolve the connections between application interfaces and platform resources. The

```
<?xml version="1.0" encoding="utf-8"?>
<application xmlns="urn:rcmw:application">
  <core id="main">   <!--Author details and description excluded-->
    <file type="vhd" toplevel="true">Convolution.vhd</file>
    <file type="vhd">Datapath.vhd</file>
    <register direction="write" hold="false" width="1">go</register>
    <register direction="read" width="1">done</register>
    <memory id="kernel">
      <size>256MB</size>
      <interface id="kernel_read">
        <type>rcmw_sequential_rd</type>
        <direction>read</direction>
        <port id="address">
          <width>32</width>
          <type>address</type>
          <direction>out</direction>
        </port>  <!-- Remaining ports excluded-->
      </interface>
    </memory> <!--Remaining resources excluded-->
  </core>
  <instance id="main_instance" count="1" core="main"/>
</application>
```

Fig. 5. Example of an application description XML document.

core metadata describes core generics, clocks, resets, interfaces, and dataflow between interfaces. Additionally, the core metadata contains information about device-specific area and performance costs in terms of lookup tables (LUTs) and average latency. The area-cost data is based on previous postfit results reported by the vendor toolchain. In the case of a core with a generic number of interfaces, the core entity is generated by a core-specific Python script depending on the selected mapping discussed in Section 3.4.

## 3.4. RC Middleware Toolchain

The previous sections presented the hardware and software abstraction layers that enable application portability between heterogeneous platforms using RCMW. To provide an application-specific view of resources, the RCMW toolchain generates customized translation and presentation layers based on the target platform and required application resources. This section presents an overview of the RCMW toolchain and discusses our application-to-platform mapping approach.

Figure 6 illustrates the RCMW toolchain. To use the RCMW toolchain, an application developer only needs to develop the application logic and describe the required resources and interfaces in the application description. The developer then executes the RCMW toolchain, providing the application description and specifying a target platform from the RCMW platform database. The RCMW toolchain determines an application-to-platform mapping based on a configurable cost function. Using the mapping results, the RCMW toolchain calls a C++ stub generator, which creates an application-specific stub similar to Figure 4, an HDL generator, which instantiates the required HDL entities and connects them together as shown in Figure 2, and a vendor-specific project generator for compiling the FPGA bitfile(s).

The mapping process consists of two stages: (1) determine how to map each application resource to platform resources, such as application memories to physical memories, and application cores to FPGAs, and (2) determine how to connect each application interface to the platform resource selected in (1). In the previous version of RCMW

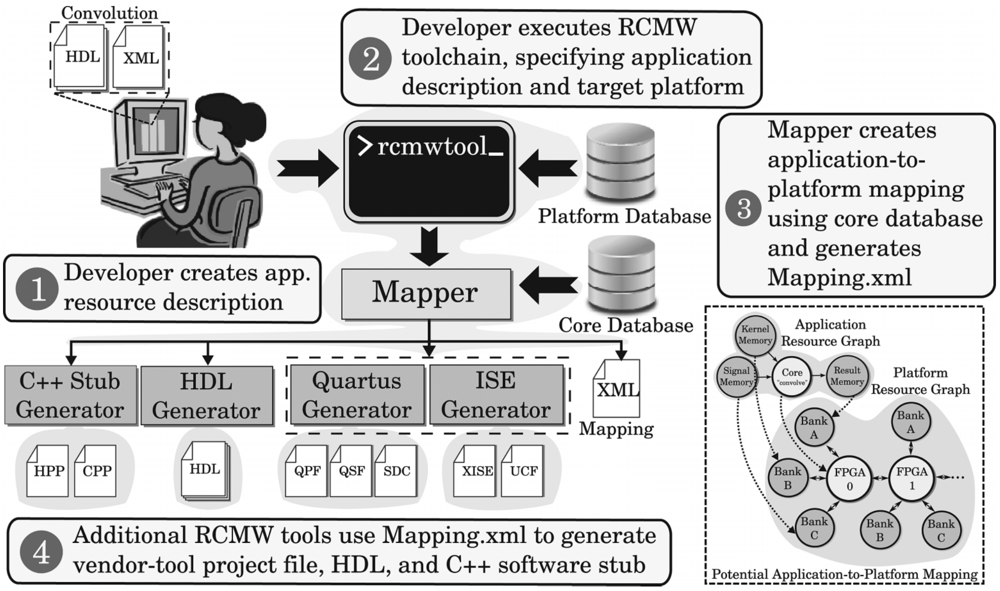

Convolution
HDL
XML
2 Developer executes RCMW toolchain, specifying application description and target platform
>rcmwtool_
Platform Database
3 Mapper creates application-to-platform mapping using core database and generates Mapping.xml
1 Developer creates app. resource description
Mapper
Core Database
C++ Stub Generator
HDL Generator
Quartus Generator
ISE Generator
XML Mapping
HPP
CPP
HDL
QPF
QSF
SDC
XISE
UCF
4 Additional RCMW tools use Mapping.xml to generate vendor-tool project file, HDL, and C++ software stub
Application Resource Graph
Kernel Memory
Signal Memory
Core 'convolve'
Result Memory
Platform Resource Graph
Bank A
Bank B
Bank C
FPGA 0
FPGA 1
Potential Application-to-Platform Mapping

Fig. 6. Overview of RCMW toolchain from application specification to vendor-specific project generation.

described in Kirchgessner et al. [2013], we used an exhaustive search to explore every valid application-to-platform mapping. For applications with few components, this approach is acceptable. However, for large applications and multi-FPGA platforms, the number of possible mappings grows considerably. To overcome this limitation, our updated mapping approach uses heuristics to guide the mapping process.

The first stage generates a list of candidate application-to-platform resource mappings to be considered. Each candidate mapping is generated by selecting an FPGA for each application core and then selecting an appropriate platform resource for each application resource. For example, an application memory could be mapped to a BRAM or an external memory bank. To reduce the number of candidate mappings to be explored in the second stage, we use the number of FPGA boundaries a datapath must pass through as a heuristic to estimate the cost of the path. Candidate mappings that map connected application cores and resources on the same FPGA or local memories are favored over mapping components across multiple FPGAs. Once the list of candidate mappings has been generated, the next stage determines how to implement each candidate mapping and calculates the cost using a customizable cost function.

The second stage explores each candidate mapping and determines how to connect each application interface to the selected platform resource using customizable cores from the RCMW core library. Each core in the RCMW core library is characterized by an XML-based core description that represents the core's interfaces, generics, and dataflow between each interface. An interface in RCMW is characterized by a type, dataflow direction, and collection of ports. Each port in an interface is characterized by a width, direction, and a type such as clock, reset, or data. Interfaces with compatible types, direction, and ports can be mapped together. Mapping interfaces together may require binding core generics to a particular value, such as the data port width. The core description can optionally include a Python-based script that allows for automated generation of an HDL entity declaration in cases where the entity provides a generic number of interfaces, such as the configurable arbitration controller.

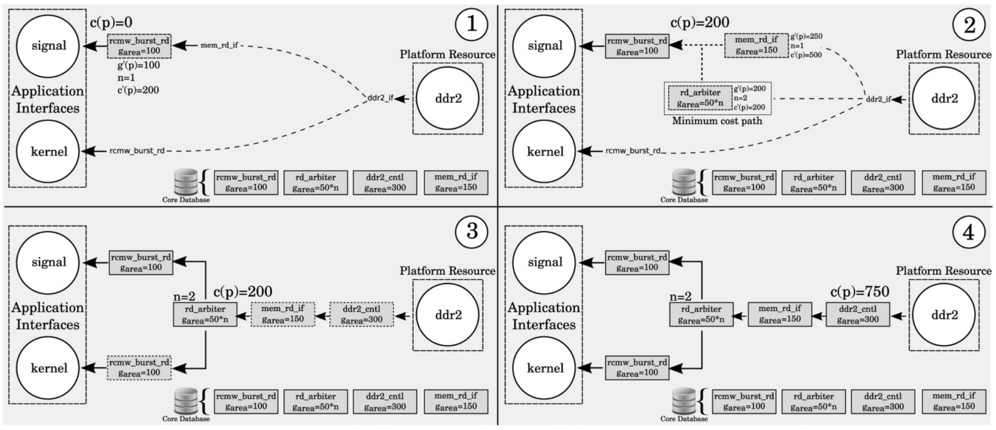


Fig. 7. Overview of mapping process to connect two application read interfaces to a single physical memory.

The process of determining what cores from the RCMW core library are needed to connect an application interface to a target platform resource is similar to path finding, where the starting position is an application interface and the goal is the target platform resource. Each node in the path to the goal corresponds to a core instance from the RCMW core library, including cores that convert interface types, cross clock domains, or merge multiple datapaths using a resource arbiter. At each iteration of this mapping process, there is a set of interfaces that need to be resolved to their target resource and a set of candidate cores from the RCMW core library that match those interface types. The set of candidate cores plus the current path create a new set of paths that need to be explored. The mapper uses a best first search algorithm, selecting the next candidate path to explore using a knowledge-plus-heuristic cost function. The knowledge-based cost is cost of the current core instances in the selected path, such as the estimated FPGA resources or latency. The heuristic cost estimates the cost for any application interfaces that have not yet been resolved in the current path and is estimated by weighting the current knowledge-based cost by the number of unresolved application interfaces. The cost function is defined as $c(p) = g(p) + h(p)$, where $g(p)$ is the knowledge-based cost and $h(p)$ is the heuristic cost. We define $h(p) = g(p)(N_0 - n)u(N_0 - n)$, where $N_0$ is the number of application interfaces that need to connect to the target physical resource, $n$ is the number of application interfaces the current path resolves, and $u$ is the unit step function. This heuristic function provides a lower bound on the path cost by estimating that the unresolved application interfaces will likely require at least the same core instances as those in the current path. The step function acts to remove the heuristic cost once the current path can support all application interfaces. This cost function can be modified to improve mapping results or optimize for different parameters and will be explored further in our future work. After calculating the cost for each candidate mapping, the minimum cost mapping is selected.

Figure 7 illustrates the second stage of the mapping process for resolving two application read interfaces, signal and kernel, to a single DDR2 memory bank. Each step in Figure 7 shows a step of the mapping process. The rectangles refer to cores from the RCMW core library. This example uses a simplified set of cores and area costs to illustrate the mapping process. The cores and their respective costs used in this example can be found at the bottom of each mapping step. In each step of the mapping process, there are multiple open paths that need to be considered, each consisting of a set of core instances from the RCMW core library, and an interface needs to be connected to

Table I. Currently Supported Platforms

| Platform | Vendor | FPGA(s) | Memory per FPGA | Host Interface |
|---|---|---|---|---|
| PROCStar III | GiDEL | 4x Stratix III E260 | 1 x 256MB DDR2<br>2 x 2GB DDR2 | PCI Express Gen1 8x |
| PROCStar IV | GiDEL | 4x Stratix IV E530 | 1 x 512MB DDR2<br>2 x 4GB DDR2 | PCI Express Gen1 8x |
| M501 | Pico Computing | 1x Virtex-6 LX240T | 1 x 512MB DDR3 | PCI Express Gen2 8x |
| PCIe-385n | Nallatech | 1x Stratix V SGSMD5 | 2 x 4GB DDR3 | PCI Express Gen2 8x |

its target platform resource or another core instance. In the first step in Figure 7, the signal application interface is selected first. The RCMW library is searched for candidate cores, which provides the required interface type for the signal interface. One matching core is found and is appended to the current path. In the second step, the previous path is expanded to find two candidate cores: an arbitration core that supports a generic number of interfaces and a low-level controller for interfacing with memory. Since the arbitration controller supports a generic number of interfaces, it reduces the heuristic cost and is selected as part of the minimum cost path. Steps three and four combine several steps using the same method as previous steps to illustrate the remaining cores being selected to complete the mapping. In our experiments, we explore area- and performance-optimizing cost functions. The area-optimizing $g(p)$ is equal to the total estimated LUTs of core instances in the current path. The performance-optimizing $g(p)$ is equal to maximum estimated latency from any application interface to the target platform resource.

## 4. RESULTS AND ANALYSIS

In this section, we present experiments that demonstrate the portability and productivity benefits of RCMW. We evaluate these benefits using four platforms from three vendors, detailed in Table I. First, we begin with a convolution application as a case study. We use RCMW to map the application to each supported platform and look at the differences in application-to-platform mapping results for both area- and performance-optimizing cost functions. Next, we evaluate the performance and area overhead incurred when using RCMW compared to native vendor interfaces. Finally, we evaluate the productivity and portability benefits of using RCMW using several streaming applications. In our experiments, we compiled Altera bitfiles using Quartus II v13.0sp1. We used GiDEL driver version 8.9.3.0. Bitfiles for the Pico M501 were generated using Xilinx ISE 14.7. 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 optimization flag -O3.

### 4.1. Convolution Case Study

This case study explores the complete development cycle for a convolution application using RCMW. Although this example requires a relatively simple resource configuration, it is representative of using the RCMW toolchain for more complex applications. We examine the required developer effort in terms of hardware and software lines of code, as well as lines of XML. We explore the RCMW toolchain results for both area- and performance-optimizing cost functions for each platform. We examine the toolchain execution time, estimated area in LUTs, estimated interface latency in clock cycles, actual postfit area in LUTs, and execution time for each platform. The estimated area and interface latency are used by the RCMW toolchain to evaluate each cost function. The convolution application performs 1D convolution of 32-bit integers. We use a randomly generated 2-million point signal and 96-point kernel.

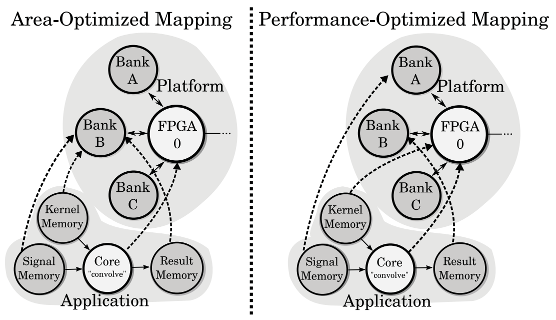


Fig. 8. Generated mappings for PROCStar III/IV.

Table II. Developer Lines of Code

| Source Type | Lines of Code |
|---|---|
| Hardware HDL | 456 |
| Software C++ | 22 |
| App. Description XML | 111 |

Figure 8 illustrates the area- and performance-optimized mappings generated by the RCMW toolchain for the convolution application on the PROCStar III and IV. The circles represent platform and application resources, with the solid lines indicating interfaces between resources. The dotted lines indicate the platform resource to which it is mapped. The area-optimizing cost function selects the mapping with the minimum area in LUTs and does not take into account on-chip BRAM. Using this cost function, the RCMW toolchain maps all application memories to the 2 and 4 GB bank B of the PROCStar III and IV, respectively. This mapping minimizes the number of LUTs by minimizing the number of memory controller instances, which require significantly more area than the RCMW arbitration logic. The performance-optimizing cost function selects the mapping where the greatest latency of all application interfaces is minimized. Using this cost function, the RCMW toolchain maps each application memory to a separate physical memory bank, minimizing the latency for each application interface. Since the kernel is sufficiently small, it is mapped to BRAM.

Table II presents the total hardware and software lines of code, as well as the lines of XML in the application description written by the application developer. We only include lines of code written by the developer, not including spacing or comments. Due to the differences in coding styles and developer experience, this table is meant to compare the relative effort for creating each component of the application.

Table III presents the results of using the RCMW toolchain to map the convolution application onto each supported platform for both performance- and area-optimizing cost functions. The map time is the required execution time for the RCMW toolchain to finish mapping the application to each platform and to generate the associated FPGA project file and C++ software stub. We ran the RCMW toolchain on a quad-core Xeon E5520. The estimated area in LUTs is the area calculated by the RCMW mapper using the postfit area results reported by the FPGA-vendor toolchain for each individual core. The estimated latency in clock cycles is the estimated maximum latency of all application interfaces in the selected mapping. The latency of each interface is calculated by adding the estimated latency of each core instance in the path from the application interface to platform interface. The calculated latency for the area- and performance-optimized mappings are only a few cycles different, as the RCMW MUX component only estimates a single clock cycle for each multiplexed interface. This estimate could be improved by taking into account the type of arbitration used, the number of interfaces, and average transfer length. The area in LUTs is the postfit area reported by the vendor toolchain. This area is similar to the estimated area that was calculated using postfit results for each core individually but is not equal due to optimizations made during the analysis and synthesis, and fitter stages of the vendor toolchain. The execution time is the total time to transfer the input signal and kernel data, perform the convolution,

Table III. Mapping Results for Convolution Application on Each Platform Optimizing for Performance and Area

| Platform | Optimization | Map Time | Estimated Area (LUTs) | Estimated Latency (clock cycles) | Area (LUTs) | Execution Time |
|---|---|---|---|---|---|---|
| PROCStar III | Area | 17.66ms | 17,857 | 26 | 15,348 | 39.3ms |
| | Performance | 18.75ms | 26,619 | 24 | 23,934 | 38.9ms |
| PROCStar IV | Area | 14.27ms | 17,125 | 26 | 16,938 | 39.1ms |
| | Performance | 17.47ms | 25,669 | 24 | 25,194 | 38.8ms |
| M501 | Area | 2.17ms | 12,330 | 38 | 12,396 | 35.1ms |
| | Performance | 2.46ms | 17,403 | 36 | 16,843 | 34.7ms |
| PCIe-385n | Area | 7.16ms | 11,762 | 16 | 10,843 | 29.9ms |
| | Performance | 6.21ms | 15,862 | 14 | 12,997 | 28.9ms |

and transfer the results. We selected an application clock frequency of 150MHz for each platform. The execution times are similar for both performance and area optimization, as the available memory bandwidth is sufficiently higher than the bandwidth required by the convolution core. Depending on the target platform hardware configuration and required application resources, the performance optimization may or may not give significant performance improvements. As illustrated in Figure 8, the area-optimized mapping results in all application memories being mapped to a single external memory bank, requiring only a single memory controller instance. The performance-optimized mapping, however, required two memory controller instances and additional logic for the kernel BRAM. We were able to reduce postfit logic usage of our application by 36% by selecting an area-optimizing cost function, which could enable applications to fit additional processing elements on an FPGA and increase application performance. In our previous work, we used an exhaustive mapping algorithm that explored all possible application-to-platform resource mappings. Given the simplicity of the resource configuration for this case study, both the exhaustive and heuristic mapping algorithms converge to the same mappings for both the area- and performance-optimizing cost functions. The exhaustive algorithm, however, requires more than an order of magnitude longer to find the same mapping even for this simple example.

## 4.2. Analysis of Performance and Area Overhead

In this section, we analyze the interface and area overhead introduced by RCMW. First, we measure the overhead introduced by RCMW's software API by transferring data between host and FPGA for varying transfer sizes. We measure the time required to complete each transfer and calculate the overhead as a percentage reduction in effective bandwidth compared to the vendor-specific API. Next, we measure the overhead introduced by RCMW when transferring data between application and platform memory. We compare the effective bandwidth when transferring data for a single RCMW read/write interface to the vendor-specific interface. 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 the known application clock frequency to calculate the effective bandwidth. We calculate overhead as a percentage reduction in effective bandwidth. Finally, we measure the RCMW area overhead by comparing the relative logic usage of RCMW, vendor, and application components for several simple applications. The area percentages were obtained using the postfit device usage report provided by Altera Quartus II and Xilinx ISE.

Figure 9 presents the effective read and write bandwidth of the RCMW host to FPGA and FPGA to external memory transfers. We find that the M501 and PCIe-385n lead the GiDEL PROCStar III and IV by a factor of two in host/FPGA bandwidth due to the newer generations of PCI Express. The maximum bandwidth of writing from the FPGA to external memory is approximately the same for each platform due to the fixed word

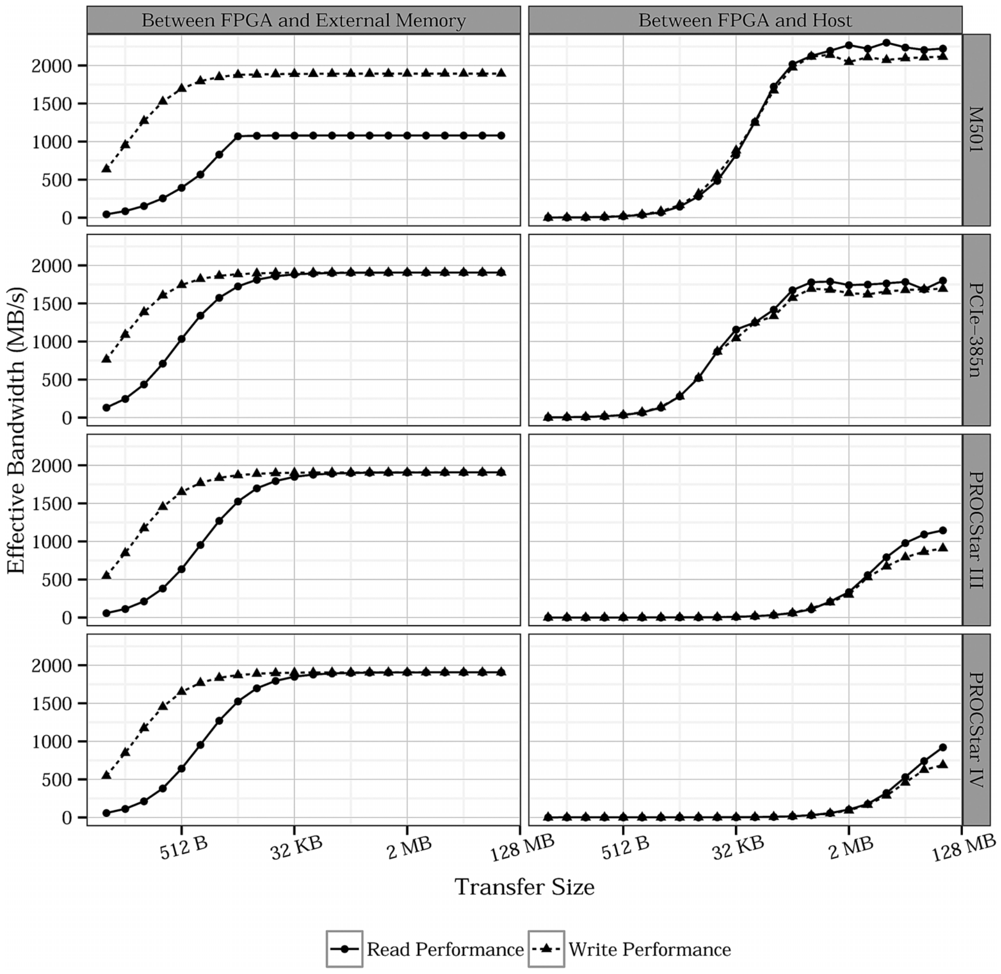


Fig. 9. Host and FPGA read and write performance to external memory for PROCStar III/IV, M501, and PCIe-385n. FPGA-to-memory bandwidth was measured with a word size of 128 bytes at 150MHz.

size of 128 bytes at 150MHz in our benchmarks. The FPGA to external memory read performance is also approximately the same for each platform, with the exception being the M501. The fixed latency for each 4KB read of the M501's AXI memory interface results in the effective read bandwidth plateauing around 1GB/s.

Figure 10 presents the overhead incurred by using RCMW compared to vendor-only baseline interfaces. The Nallatech PCIe-385n is not included in this figure, as the vendor provides no baseline with which to compare.

The left side of Figure 10 presents the RCMW overhead for transfers between the FPGA and external memory. The peak FPGA/memory write overhead was similar for each platform: 50%, 43%, and 43% overhead for the M501, PROCStar III, and PROCStar IV, respectively. For large transfers, this overhead quickly becomes less than 1% for each platform. The peak read overhead is less than 10% for each platform and similarly becomes less than 1% for large transfers. For small transfers, an overhead of 50% equates to only tens of clock cycles, which is relatively insignificant. The peak

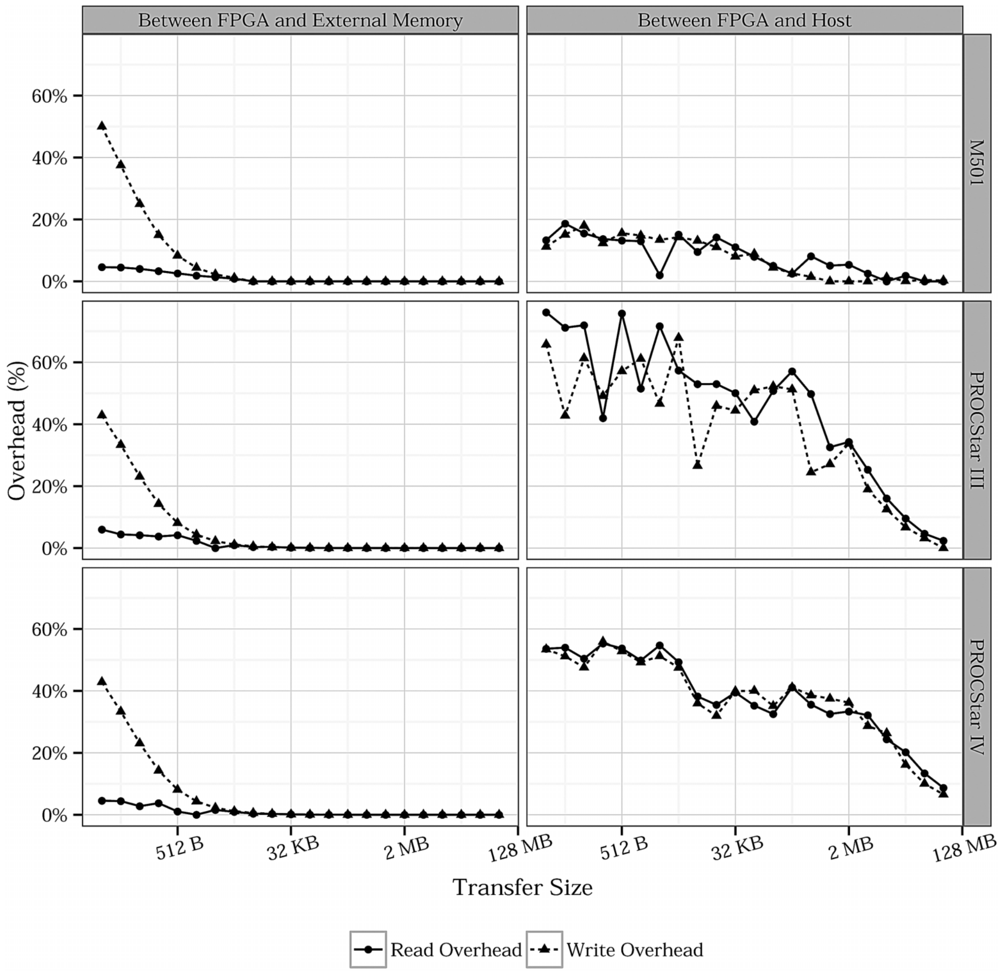


Fig. 10. Host and FPGA read and write overhead to external memory for PROCStar III/IV and M501. FPGA-to-memory overhead was measured with a word size of 128 bytes at 150MHz.

write overhead is greater than the peak read overhead due to the additional cycles required to flush memory buffers and ensure read-after-write consistency.

The right side of Figure 10 presents the RCMW overhead for transfers between the FPGA and host. The high variance found in these graphs for small transfers is due to the variance in the host's OS scheduler. The greatest FPGA/host transfer overhead is incurred by the PROCStar III, which peaks at approximately 80% for reads and 70% for writes. This seemingly high overhead is due to the additional features provided by the RCMW software API, including thread safety and user memory buffer management. Although we could disable these features and significantly reduce this overhead, they are vital to RCMW's concurrent API and therefore are included in our results. Furthermore, this high overhead occurs at small transfer sizes and accounts for approximately 1- to 2ms overhead. Since the M501 provides thread safety for some of their API calls by default, the peak overhead is less, at approximately 20% for both read and writes. Large overheads are restricted to small transfer sizes, resulting in only a few additional microseconds for each transfer. For increasing transfer sizes, this

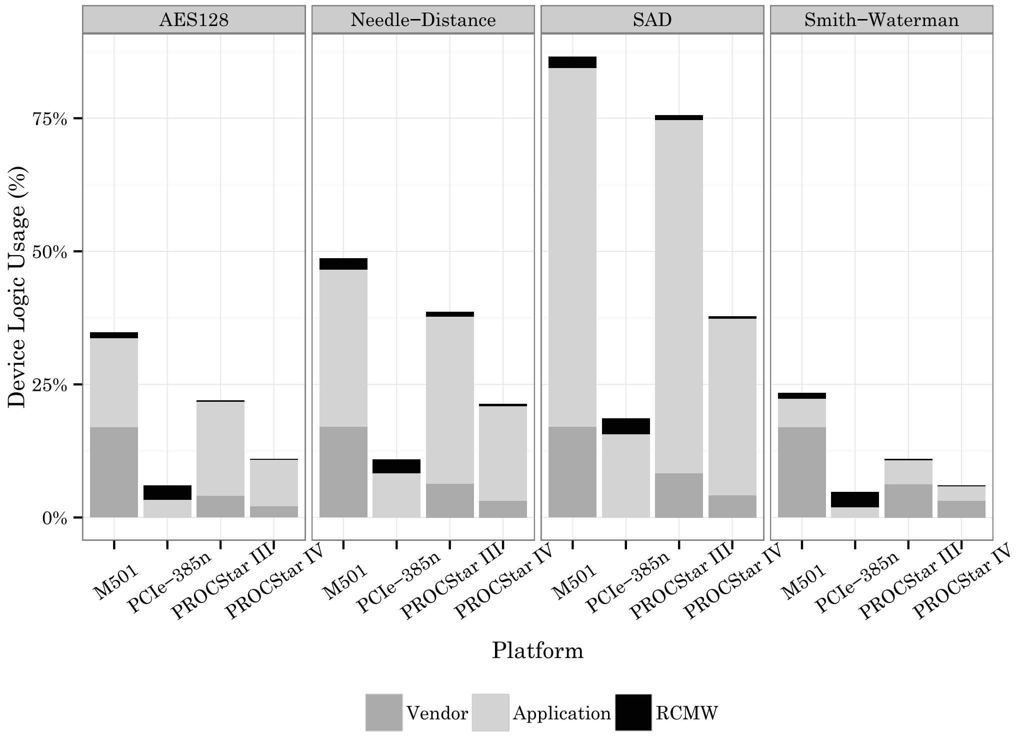


Fig. 11. Area overhead analysis of RCMW compared vendor IP and application components.

overhead is quickly amortized, resulting in less than 1%, 5%, and 7% read and write overhead for the M501, PROCStar III, and PROCStar IV, respectively.

Figure 11 presents the logic usage of RCMW, application, and vendor components. The bottom layer in the stacked bar graph represents the vendor logic usage, the middle layer represents the application logic usage, and the top layer represents RCMW overhead. Each set of bars represents the area breakdown for each platform for a specific application. The PCIe-385n does not have a vendor component, as there is no vendor-provided hardware components. From this figure, we see that RCMW accounts for a very small fraction of the overall design area, typically less than 1% of the total device resources. The largest RCMW area overhead was less than 3% for the sum of absolute differences (SAD) on the Pico M501. For the PCIe-385n, RCMW handles the PCIe and external memory interfaces, resulting in a relatively larger RCMW area usage. Although the total area required by RCMW is platform and application specific, it is important to note that at least a portion of the area resulting from mapping multiple application resources to a single physical memory would be necessary even for non-RCMW implementations. For a performance comparison of these applications and kernels, please see Table V.

## 4.3. Analysis of Productivity

In this section, we analyze the productivity benefits of RCMW by comparing software lines of code (SLoCs), hardware lines of code (HLoCs), and total development time required by the developer. Although lines of code and development time are commonly used for measuring software development productivity, it is worth mentioning

Table IV. Factors Contributing to Improved Developer Productivity Using RCMW

| | |
|---|---|
| Standardized Interfaces | RCMW provides standardized hardware and software interfaces, independent of the underlying platform, enabling code reuse and allowing developers to use a single API for multiple platforms. |
| Application-Specific Interfaces | RCMW provides only the requested resource interfaces independent of the underlying platform, reducing application complexity. |
| Reduced Lines of Code | RCMW reduces hardware and software lines of code by handling typical sources of coding overhead. In hardware, RCMW handles resource arbitration and clock domain crossing (CDC). In software, RCMW manages platform initialization, cleanup, and application multithreading. |
| Application Portability | RCMW enables portability through abstraction. Application developers specify their applications in terms of cores, which can then be re-used in other applications, maximizing productivity. |
| Object-Oriented API | RCMW provides an object-oriented API that provides an objectified representation of application resources, simplifying application development and avoiding procedural API calls. |
| API Validation | RCMW validates API calls to prevent platforms from entering an invalid state, alerting users via C++ exceptions when necessary. |

that these measures are heavily influenced by developer-specific factors such as coding style [Schofield 2005]. To explore the productivity benefits of RCMW, a developer familiar with the Pico Computing M501, GiDEL PROCStar III/IV, and RCMW implemented five cores from OpenCores [2014] using both vendor- and RCMW-specific design flows for each platform. The cores used included an AES128 encryption core, a JPEG encoder, a SHA256 hashing core, an FIR filter, and a 3DES encryption core. Each core was implemented using both the vendor's recommended design flow and the RCMW toolchain. We included all code written by the developer, excluding comments and whitespace. The Nallatech PCIe-385n was excluded from this experiment due to lack of a vendor-specific design flow.

GiDEL and Pico Computing provide different approaches for developers to interface their applications with platform resources. GiDEL provides a graphical tool called *PROCWizard*, which enables developers to customize GiDEL-provided IP cores and resource interfaces. Pico Computing takes a different approach, providing developers with a Xilinx AXI bus interface to platform memory. Pico Computing provides a streaming abstraction in both hardware and software, enabling efficient transfer of data between host and FPGA.

Our experiments indicated that, on average, RCMW required 65% less SLoCs, 41% less HLoCs, and 53% less development time than the GiDEL-specific design flow, and 66% less SLoCs, 59% less HLoCs, and 69% less development time than Pico-specific design flow. Since these numbers are averaged for a single developer, we cannot draw conclusions as to an exact improvement for all developers, but we can support the argument that RCMW improves productivity. These improvements are expected, as RCMW handles many development tasks typically left to the developer. The major factors leading to improved productivity are summarized in Table IV.

In our experiments, the Pico M501 required relatively high HLoCs due to the generic interfaces exposed to developers. Unlike PROCWizard and RCMW, the Pico M501 does not assist developers in customizing platform resources, forcing developers to handle arbitration and clock domain crossing (CDC). GiDEL's approach required less HLoCs due to PROCWizard assisting developers in customizing IP interfaces for their application. GiDEL also handles CDC for memory interfaces, reducing HLoCs. RCMW required the least HLoCs, generating the specific interfaces required by the application and handling all required arbitration and CDC. Similar results were found for total SLoCs, with Pico requiring the most SLoCs, followed by GiDEL and then RCMW. Both

Table V. Execution Time and Area of Several Applications and Kernels on Each Supported Platform Using RCMW

| | M501 | | PCIe 385n | | PROCStar III | | PROCStar IV | |
|---|---|---|---|---|---|---|---|---|
| Kernel/Application | Time | Logic (%) | Time | Logic (%) | Time | Logic (%) | Time | Logic (%) |
| 1D Convolution | 43.3ms | 26 | 32.9ms | 3 | 39.3ms | 14 | 38.9ms | 7 |
| 2D Convolution | 16.0ms | 57 | 12.8ms | 13 | 13.2ms | 44 | 16.3ms | 25 |
| Image Segmentation | 1.73s | 64 | 1.24s | 11 | 1.41s | 53 | 1.39s | 26 |
| Needle-Distance | 161.0ms | 48 | 173.0ms | 10 | 194.0ms | 38 | 203.0ms | 21 |
| OpenCores AES128 | 32.6ms | 32 | 19.3ms | 6 | 25.3ms | 22 | 24.3ms | 11 |
| OpenCores FIR | 21.5ms | 28 | 21.3ms | 3 | 24.5ms | 15 | 24.0ms | 8 |
| OpenCores JPEGEnc. | 23.9ms | 29 | 14.3ms | 3 | 15.3ms | 15 | 19.6ms | 8 |
| OpenCores SHA256 | 73.3ms | 26 | 52.3ms | 4 | 64.1ms | 12 | 63.3ms | 5 |
| Smith-Waterman | 96.0ms | 23 | 104.0ms | 4 | 116.0ms | 11 | 119.0ms | 6 |
| Sum of Abs. Diff. | 18.7ms | 86 | 13.8ms | 18 | 14.7ms | 75 | 19.1ms | 38 |

vendor APIs require developers to manage buffers and platform-specific restrictions such as data alignment and transfer size.

## 4.4. Analysis of Portability

Table V presents the execution time and logic usage of various streaming applications and kernels for all four supported platforms. Each application was executed using the same application source code with a clock frequency of 150MHz and required between two and four streaming interfaces. The OpenCores JPEG encoder required a random-access memory interface to integrate with its on-board peripheral bus (OPB) interface. This table demonstrates the same application hardware and software source executing across heterogeneous platforms. Porting applications across each platform was accomplished with almost no effort, requiring only that the RCMW toolchain be executed once for each application and platform. This table is not meant to be a comparison of platform performance, as the maximum device area and achievable clock frequency was not used for each application. In our tested applications and kernels, however, we found that the PCIe-385n outperforms the PROCStar III and IV for the given implementations and input datasets. The M501 and PCIe-385n use a newer PCIe generation, enabling higher peak from host to FPGA, 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 memory banks of the PROCStar III and PROCStar IV provide an advantage over the M501. It is important to note that with newer platforms with state-of-the-art FPGAs, the increase in FPGA resources enables more application cores to fit on a single FPGA. This trend is illustrated in Table V, as the device-logic usage greatly decreases from the PROCStar III, which uses a Stratix III, to the PCIe-385n, which uses a Stratix V. For high-performance computing applications with data-level parallelism, RCMW could enable significant performance improvements by targeting existing applications to newer FPGA platforms with little to no developer effort.

## 5. CONCLUSIONS

Despite performance and power advantages over conventional many-core CPU and GPU architectures, FPGAs have had limited acceptance in high-performance computing and high-performance embedded computing applications due to their portability and productivity challenges. To help overcome these challenges, we introduced RCMW, which provides an extensible framework that abstracts away platform-specific details to provide an application-centric hardware and software development environment. This environment is customized by the RCMW toolchain using the developer-provided application description and allows developers to focus on the ideal resources and interfaces for their application without worrying about the underlying platform. To create this environment, the RCMW toolchain first selects an application-to-platform mapping using a customizable cost function and then generates the required hardware and software interfaces.

We evaluated RCMW's performance and productivity benefits for four platforms from three vendors. We demonstrated RCMW's ability to quickly explore different application-to-platform mappings using a convolution application case study or both area- and performance-optimizing cost functions. We demonstrated that the benefits of RCMW can be achieved with less than 1% FPGA/memory and 7% host/FPGA transfer overhead in the common case. We also demonstrated that RCMW has relatively low area overhead, requiring less than 3% of logic resources for several applications across all four platforms. We presented evidence that RCMW improves developer productivity by showing that RCMW requires fewer lines of code and total development time for deploying several kernels than vendor-specific approaches. Finally, we demonstrated that RCMW enables portability by showing that the same application source was able to execute without change across each supported platform.

Directions for future work include improving the RCMW mapping algorithm to map application cores to multi-FPGA systems and investigating different mapping cost functions and arbitration schemes to maximize performance for heterogeneous application configurations. Additionally, we will continue to add support for new FPGA accelerator platformsand extend the RCMW core library to include optimized cores for a broad range of application classes.

## REFERENCES

M. Adler, K. E. Fleming, A. Parashar, M. Pellauer, and J. Emer. 2011. LEAP scratchpads: Automatic memory and cache management for reconfigurable logic. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'11)*. ACM, New York, NY, 25–28. http://doi.acm.org/10.1145/1950413.1950421

V. Aggarwal, G. Stitt, A. George, and C. Yoon. 2012. SCF: A framework for task-level coordination in reconfigurable, heterogeneous systems. *ACM Transactions on Reconfigurable Technology and Systems* 5, 2, 7:17:23. http://doi.acm.org/10.1145/2209285.2209286

Altera Corp. 2007. Avalon Memory-Mapped Interface Specification. Available at https://www.altera.com.

D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp. 2008. Achieving programming model abstractions for reconfigurable computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16, 1, 34–44.

ARM. 2013. AMBA AXI and ACE Protocol Specification. Available at http://www.arm.com.

B. Betkaoui, D. B. Thomas, and W. Luk. 2010. Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing. In *Proceedings of the 2010 International Conference on Field-Programmable Technology (FPT'10)*. 94–101.

L. Cai, D. Gajski, and M. Olivarez. 2001. Introduction of system level architecture exploration using the SpecC methodology. In *Proceedings of the 2001 IEEE International Symposium on Circuits and Systems (ISCAS'01)*, Vol. 5. 9–12.

T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. 2012. From OpenCL to high-performance hardware on FPGAS. In *Proceedings of the 2012 22nd International Conference on Field-Programmable Logic and Applications (FPL'12)*. 531–534.

K. Eguro. 2010. SIRC: An extensible reconfigurable computing communication API. In *Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'10)*. 135–138.

T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell. 2008. The promise of high-performance reconfigurable computing. *Computer* 41, 2, 69–76.

P. Garcia, K. Compton, M. Schulte, E. Blem, and W. Fu. 2006. An overview of reconfigurable hardware in embedded systems. *EURASIP Journal on Embedded Systems* 2006, 1, 13.

A. George, H. Lam, and G. Stitt. 2011. Novo-G: At the forefront of scalable reconfigurable supercomputing. *Computing in Science Engineering* 13, 1, 82–86.

GiDEL Ltd. 2014. PROCWizard. Retrieved August 24, 2015, from http://www.gidel.com/procwizard.htm.

L. Hao and G. Stitt. 2012. Bandwidth-sensitivity-aware arbitration for FPGAs. *IEEE Embedded Systems Letters* 4, 3, 73–76. DOI:http://dx.doi.org/10.1109/LES.2012.2209397.

S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah. 2008. Liquid metal: Object-oriented programming across the hardware/software boundary. In *ECOOP 2008—Object-Oriented Programming*, Lecture Notes in Computer Science, Vol. 5142. Springer, 76–103.

A. Ismail and L. Shannon. 2011. FUSE: Front-end user framework for O/S abstraction of hardware accelerators. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM'11)*. 170–177. DOI:http://dx.doi.org/10.1109/FCCM.2011.48.

R. Kirchgessner, A. D. George, and H. Lam. 2013. Reconfigurable computing middleware for application portability and productivity. In *Proceedings of the 2013 IEEE 24th International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'13)*. 211–218.

J. Kulp. 2010. *OpenCPI Technical Summary*. Technical Report. Available at http://opencpi.org.

Nallatech Ltd. 2007. DIMEtalk V3.0. Available at http://www.nallatech.com.

OpenCores. 2014. Open Cores Home Page. Retrieved August 24, 2015, from http://opencores.org.

OpenFPGA Inc. 2008. OpenFPGA Home Page. Retrieved August 24, 2014, from http://openfpga.org.

C. Pascoe, A. Lawande, H. Lam, A. George, W. Farmerie Sun, and M. Herbordt. 2010. Reconfigurable supercomputing with scalable systolic arrays and in-stream control for wavefront genomics processing. In *Proceedings of the Symposium on Application Accelerators in High-Performance Computing*. 13–15.

X. Reves, V. Marojevic, R. Ferrus, and A. Gelonch. 2005. FPGA's middleware for software defined radio applications. In *Proceedings of the 2005 International Conference on Field-Programmable Logic and Applications*. 598–601.

J. Schofield. 2005. The statistically unreliable nature of lines of code. *CrossTalk* 18, 4, 29–33.

T. Schumacher, C. Plessl, and M. Platzner. 2009. IMORC: Application mapping, monitoring and optimization for high-performance reconfigurable computing. In *Proceedings of the 17th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'09)*. 275–278.

L. Shannon and P. Chow. 2005. Simplifying the integration of processing elements in computing systems using a programmable controller. In *Proceedings of the 2005 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*. 63–72.

G. Stitt and J. Coole. 2011. Intermediate fabrics: Virtual architectures for near-instant FPGA compilation. *IEEE Embedded Systems Letters* 3, 3, 81–84.

J. E. Stone, D. Gohara, and G. Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering* 12, 3, 66–73.

J. Villarreal, A. Park, W. Najjar, and R. Halstead. 2010. Designing modular hardware accelerators in C with ROCCC 2.0. In *Proceedings of the 2010 18th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'10)*. 127–134.

Y. Wang, X. Zhou, L. Wang, J. Yan, W. Luk, C. Peng, and J. Tong. 2013. SPREAD: A streaming-based partially reconfigurable architecture and programming model. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21, 12, 2179–2192.

J. Williams, C. Massie, A. D. George, J. Richardson, K. Gosrani, and H. Lam. 2010. Characterization of fixed and reconfigurable multi-core devices for application acceleration. *ACM Transactions on Reconfigurable Technology and Systems* 3, 4, 19:1–19:29. http://doi.acm.org/10.1145/1862648.1862649.

Received April 2014; revised February 2015; accepted March 2015