Portable Library Development for Reconfigurable Computing Systems

Proshanta Saha, Esam El-Araby, Miaoqing Huang, Mohamed Taher, Tarek El-Ghazawi,
The George Washington University
{sahap, esam, mqhuang, mtaher, tarek}@gwu.edu

Chang Shu, Kris Gaj
George Mason University
{cshu, kgaj}@gmu.edu

Alan Michalski, Duncan Buell,
University of South Carolina
{michalsk, buell}@engr.sc.edu

Abstract

As Reconfigurable Computing (RC) systems become more common place among application scientists and developers, a mechanism for porting existing work to other platforms is increasingly desirable. The constantly changing technologies and architectures in today's RC platforms present a challenge to any developer wishing to move from an early development system to a newer system. Many new RC systems do not offer a complete development environment, and often requires the end user to choose design tools, languages, and hardware library packages that are compatible with their system. Rewriting basic cores for each platform can be a daunting task. Unlike the High Performance Parallel Computing community, which provides highly optimized and portable open source libraries to its community, to the best of our knowledge such an effort does not exist for the RC community. In this work we propose a methodology for developing RC libraries and present the challenges involved in each step. This paper also presents work done in developing an extensive portable library using the SRC6, Cray XD1, and SGI RC100 platforms as a case study.

Keywords: Library Design, Reconfigurable Computing, Code Reuse, Component Based Design, FPGA, SRC6, XD1, SGI RASC.

1. Introduction

With an increasing number of RC system vendors in the market, there appears to be very little work done to assist users in migrating existing work on to new systems. While the RC systems are fairly unique and customized, the tool chain provided to the end user is often just the drivers and the necessary HDL (Verilog/VHDL) top level modules required to access the devices on the platform. The task of building highly optimized hardware macros it seems is left entirely to the end user. This unfortunately results in a lot of code re-write that is mostly unnecessary and sometimes perceived as 're-inventing the wheel'. Our experience on the Reconfigurable Computing Library (RCLib) project at the George Washington University (GWU) along with our academic partners George Mason University (GMU) and University of South Carolina (SC) has shown that there is a significant amount of overlap among different applications in a field. The RCLib project decomposes common applications in domains such as bioinformatics, cryptography, image processing, matrix arithmetic, sorting, and others into re-usable hardware cores.

Throughout the development cycle of RCLib we have adopted a methodology for creating such library cores that addresses concerns in key areas such as performance vs. portability, code reuse vs. application specific needs, collaboration between multiple sites, generic vs. architecture specific library distribution, licensing issues, and several other concerns. This paper is broken in 5 sections. Section 2 discusses related work done in the field by hardware vendors, independent software vendors (ISV), and open source efforts. Section 3 discusses the methodology of building a library. Section 4 describes a case study using the SRC6, Cray XD1, and SGI RC100 platforms. And finally section 5 concludes this paper with future work and lessons learned.

2. Related Work

The RC library effort has been taken on at various stages and forms by multiple groups each with a particular niche. The main contributors are the field programmable gate array (FGPA) chip manufacturers, RC system vendors, independent software vendors (ISVs), and users and enthusiasts. FGPA chip manufacturers such as Xilinx and Altera provide basic cores to handle tasks such as memory operations, Block RAM (BRAM) initialization, hard core instantiation, and few other basic operations in their library. Soft cores are also provided for commonly used functions from both in-house and user contributed macros. Chip vendors also provide

development tools that aid users in rapid prototyping with applications such as Xilinx Sysgen [1] and Altera DSP Builder [2] that can be useful when designing components which require chip specific implementations. Licensing restrictions also require that the library cores be used only on particular set of processors.

System and board vendors provide basic library modules for their users, mostly in the form of tightly coupled software. Vendors such as SRC Computers, Annapolis Microsystems, Starbridge, Nallatech and several others provide complete solutions catered to their architecture. SRC Computers for example provides a C based language compiler, Carte-C [3], for their SRC platforms (SRC6e, SRC6, CompactMap, etc). Carte-C allows users to define a hardware application using basic library cores without utilizing hardware description languages (HDL). Annapolis Microsystems provides a Java based GUI schematic tool, CoreFire [4], for their entire line of hardware acceleration daughter cards (Firebird, Wildstar, etc) which allow for users to drag and drop available library components on to their design sheets. Starbridge Systems provides a win32 based GUI schematic tool, Viva [5], for their line of HyperComputers (HC-36, HC-62, etc) which also allows users to drag and drop library components on to their active design sheets. Nallatech provides DimeTalk [6] for their H1XX and BenXX series of systems and boards, and is a hybrid of both C based language and schematic tool that allows users to design their applications utilizing a library of existing hardware cores. Unfortunately the software tools provided are restricted to the platforms manufactured by the vendors, and tend to incur pricy support and upgrades.

ISVs generally provide a set of commonly used functions with their development tools. Development tools such as DSPLogic RC Toolbox [7], Impulse CoDeveloper [8], Mitrionics Mitrion SDK [9], Celoxica DK Design Suite [10], and others provide a basic set of portable cores for RC systems, such as the Cray XD1 and SGI RASC, allowing the user with the flexibility to retarget their application. Although the efforts by ISVs have enabled them to be somewhat independent from a given architecture, the tools tend to focus primarily on portability and as a consequence aren't able to take full advantage of the unique features of each target architecture.

Finally, RC users and enthusiasts provide domain specific library macros, predominantly in specific niche fields such as communication and DSP. The contributions are mostly provided 'as-is' and are usually architecture specific or often generic HDL proof of concepts. One such open source library is the Open Cores project [11] which provides access to various DSP, crypto, system architecture, controllers, and other miscellaneous cores.

3. Developing Libraries

In this section we propose a set of guidelines for designing and developing portable hardware libraries. There are several key concerns in developing a hardware library, namely: library scope; packaging and interface definition to allow for quick porting of cores to a new architecture; performance versus portability issues with regards to relying on architecture or chip specific features; collaboration and intellectual property (IP) management; quality control to prevent unnecessary delays in the release of a library; and distribution and licensing. The remainder of this section focuses on the unique challenges for developing hardware libraries and the proposed a path to address with the issues.

3.1 Domain Analysis

As with any effort, it is desirable to generate a comprehensive library that covers multiple domains in great depth. Unfortunately man power and resources will dictate the size and direction of the library effort. However, our experience on the RCLib project has shown that the amount of overlap found in functions of a given domain allows for a more targeted library to be pursued. Focusing on the core functionality of a domain can provide ways to extend to various other applications.

Unlike software libraries, it is difficult to generalize hardware library cores. The interface, data types, size of operands, and loop iterations for example need to be known and fixed prior to creating a hardware instantiation. Several approaches have been introduced to offset this limitation. Tools such as Mitrion-C, Viva, CoreFire, DSPLogic and others provide a type inference mechanism, determining the correct data types and sizes at compile time. Tools such as Carte-C provide pre-processor parsers that can generate the required interface and changes at compilation time.

The performance tradeoff between generality and application specific functionality will have to be carefully considered. This tradeoff will determine the level in which the functionality is broken down to. Unlike software libraries, which are generally intended to run in a sequential fashion, hardware libraries are designed to exploit concurrency. Breaking down the functionality into finer units allows for more re-use, however, it may require significant effort by the hardware library user to ensure that pipeline lengths remain manageable and meet stringent timing and performance requirements. To offset this restriction, performance analysis and

characterization of the hardware cores can assist users in determining the types of cores that are suitable for their designs. These include information such as the core's data types, data sizes, pipeline lengths, latency, and delays.

3.2 Standard Interface Definitions

The purpose of creating a hardware library is not only to generate a common set of useful application cores, but also to modularize the applications themselves to promote code reuse. Reducing the amount of context switching or reconfiguration has been shown to decrease execution time and increase performance of a hardware macro [12][13][14]. This can be achieved by creating reusable modules and building new library cores based on existing modules. Modularization also helps in partial reconfiguration, where configuration time can be reduce by replacing only the parts that need to be changed [15][16].

To ensure that the macros can be used together, a common port list scheme and naming convention should be adopted. Creating a set of query ports such as busy, wait, and done for example will facilitate handshaking between cores. Providing a similar interface for all cores will not only enhance readability and improve the debugging efforts, it will also allow for greater portability and integration.

3.3 Portability

A key issue usually decided early on is the number and types of RC platforms that the hardware library would support. Although universal portability is the ideal, it is neither feasible nor practical. Even platforms from the same vendor can be drastically different from one generation to the next, making the economics of creating backward compatible hardware cores costly. In the relatively new field of RC, new advances in technology happen at such a fast pace, that it is not desirable to keep programming using techniques that were designed as a work around for a previous generation's shortcomings.

There are several steps that can be taken to make porting to a new platform less taxing. The first method is to diligently document optimization clues for the regions of the code that cause the most delay and thus affect the clock speed. This will aid future developers in deciding how best to meeting timing. The second step is to agree between the vendors and users on a minimum clock speed which the hardware cores should meet. This requires that vendors supply top level modules that can meet the target clock speed. The third step is to ensure that no architecture specific code is introduced, as this will affect the possible platforms that the library can be ported to. Assumptions such as BRAM sizes, specific instantiation of hard cores (e.g. MULTS), and etc should be avoided if possible. A fourth step is to ensure that system specific workarounds are avoided. This could however severely affect the performance of the hardware core, as the customized optimizations may be critical to the proper functionality of a design. Following these steps will help future developers in porting the design to newer hardware.

Once the decisions are made about the particular platforms of interest, it is important not to duplicate efforts in porting designs to the various platforms. Keeping hardware macros at an HDL level will significantly reduce the time and effort necessary to port hardware cores across platforms. A majority of vendors support the inclusion of netlists into a design. The only part the end user would be responsible is the necessary vendor specific wrappers to communicate with the library macros, as vendor architecture and interfaces may change from platform to platform. A standard interface is desirable, and is currently a topic being addressed groups such as OpenFPGA [17] and the Spirit Consortium [18].

3.4 Collaboration

Working with a large group of contributors require thorough planning to ensure that there are no conflicts, ensuring that the proper access privileges are given to the moderators of the library and end users is essential. Security often times takes a back seat to ease of access, and this can be a costly error.

Deciding where and how to store user contributions and management privileges is a policy decision and should be considered carefully, as tools and security hardware alone cannot be a replacement for sound policy. In the event that anonymous access is allowed to a particular core repository, it is important to respect the intellectual property of contributors by ensuring licensing requirements are properly displayed. If proprietary material is provided, the conditions for their use should be properly acknowledged by the end user. This is particularly essential in early access and non-disclosure agreements made with the contributor.

The topic of source control can easily fill volumes [19][20][21][22][23], and it has, but that is not the intent of this paper. The use of source control software that meets the needs and requirements of the library developers and contributors is highly recommended.

3.5 Benchmarking and Testing

This section discusses the importance of maintaining an up-to-date benchmark and testing suite for verification and release purposes. There are several ways to ensure that the updates checked into

the library repository do not interfere with other library components and function as expected. Unit testing, regression testing, and release builds are necessary to ensure the quality of a release. All too frequently, release builds are performed only a short time prior to their expected release dates. This may not be sufficient enough for contributors to discover faults in their design, particularly when multiple hardware libraries are affected by their changes. More frequent builds and regression tests help developers quickly determine the stability of their contributions and verify that previously working modules still function as expected. However hardware library tests can be very resource and time consuming and at times it can take several hours to properly run a single core through the tool chain. Regression tests should be performed along with the builds to allow for an automated capture of errors and failures.

Benchmarks serve as a quality assurance check to ensure that the current release exceeds and/or meets the performance of previously released libraries. Here the term benchmark refers to the performance benchmark of the hardware library components, and does not necessarily reflect that of industry specific benchmark tests. Benchmarks allow end users and library developers to determine their impact in switching to the new release. Providing benchmark results for all releases and their requirements can help an end user determine which platform and release is suitable for their needs.

3.6 Distribution and Licensing

There are several decisions that have to be made prior to a distribution release. One important decision is the distribution format. Because of the lack of standardization among vendors as to the mechanism of importing third party library cores, wrappers will have to be generated to enable the use of the library components. This wrapper is specific to each vendor thus requires a separate library distribution for each platform. This can be time consuming as well as challenging to keep up with the various different platforms and their subsequent releases. Since the wrappers are tightly integrated with the vendor's IDE releases and APIs, the library distribution would have to reflect those changes in order to guarantee compatibility.

A second option, and perhaps the more straight forward alternative, would be to release the source code package containing HDL source, black box files, interface documentation, and handshaking protocols to the end user and assume that the user would be able to create the necessary wrappers to utilize the library. This will alleviate the need to develop separate releases for each platform and for each release of the vendor's IDE and APIs. Although this has its advantages, the user would need to have intimate knowledge of the library core and be able to make modifications necessary to meet vendor specific requirements such as timing, port lists, registers, and etc.

Licensing is perhaps one of the most confusing aspects of a software distribution or library distribution in the open source community. Unfortunately violating or ignoring the original consent by the authors not only diminishes the value and efforts of the open source contributors, it can also inadvertently create legal worries for further distribution of work spawned from it. It is important to understand the level of freedom the project wishes to bestow upon the community, and still respect the original contributors' intent and intellectual property

There are several other licenses available but the aim of this paper is not to examine them but rather to give an overview of the importance of choosing a license agreement. To find out information about more licensing models can be found at the Open Source Initiative [24].

The incorporation of various different types of licenses brings about the confusing part in building a library. While chip manufacturers (Xilinx, Altera, etc) and/or system vendors (SRC, AMS, Cray, SGI, etc.) provide basic cores freely to its users, license restrictions allow use only on particular platforms and architectures. This may require that certain soft cores, regardless of how trivial they may be, be re-written to prevent litigation and to allow for free distribution. Soft cores from open source communities may also impose a different license agreement, requiring careful understanding of parts that may or may not be bundled with other license agreements. The end user may also suffer from similar confusion as they may not be able to determine which license agreement holds precedence.

To reduce the litigation burden and offer freely distributable libraries, the recommended method is to avoid utilizing third party cores all together. When doing so is not an option, providing pre-requisites for utilizing the hardware library will alleviate the need to bundle third party libraries and packages. This includes requiring the end user to pre-install the necessary hardware and software libraries, and in effect will satisfy the third party licensing requirements.

4. Case Study

This section discusses the efforts in creating a portable library and the choices taken to minimize

porting efforts between the SRC6, CrayXD1, and SGI RASC platforms.

4.1 Domain Analysis

The application scope was chosen based on input from the RC community as well as domain experts in the respective fields. Various different domains were targeted including Secret Key Ciphers, Binary Galois Field Arithmetic, Elliptic Curve Cryptography, Long Integer Arithmetic, Image Processing, Bioinformatics, Sorting, and Matrix Arithmetic. The applications developed in each domain were analyzed for an acceptable balance between code reuse and performance and library cores were developed accordingly. The core utility, resource utilization, the effort required by the developer to pipeline the cores, and efficiency of the design produced from the library, collectively determine application core breakdown levels. Often times, efficient implementations of an application kept the core levels at too high a level, thereby diminishing the reuse of the core. In such cases the domain applications were re-examined to find overlap between applications and were redesigned to utilize the common cores. In the RCLib project novel approaches such as variable size operands and common skeletons were used to efficiently instantiate hardware implementations of various image processing and elliptical curve cryptography applications.

4.2 Portability

To ensure portability simple guidelines were adopted. Library cores were all developed in either Verilog or VHDL. A common packaging format was chosen, which includes HDL code, debug/simulation code, black box file with interface and constraint definitions, core documentation, and a Makefile. This helps in the automation of library builds and distribution. By maintaining a minimal set of files, future library maintainers and developers have enough information required to modify the library package should constraints change in the future. A good example of this is when the timing constraints are changed to adapt to faster clock speeds. Libraries included in the RCLib follow a minimum clock speed of 100MHz (10ns) as it was an accepted clock speed in the SRC, Cray XD1, and SGI RASC RC systems. The RCLib also provides detailed documentation as to the pipeline latency and a timing diagram to help the end user and library developers get a better understanding as to the usage of the core. A standard set of extensions for the file names and a consistent naming convention, was chosen to facilitate library integration. The minimum is as follows:

.v/.vhd	for Verilog/VHDL source code
.c/.cpp	for Host C/C++ source code
.blk	for black box interface file
.tex/.PDF	for LaTeX/PDF documentation
test.c/.cpp	for test code
Makefile	for the Makefile

Sticking to a guideline ensures that a library build can discern between HDL source code (Verilog or VHDL), host source code (C or C++), interface file (black box) and correctly identify which pre-processors and compilers that are needed for each instance. Following a naming convention also prevents any name space clashes, ensures that library cores are not over written when they are built/archived, and avoids runtime and compile time errors during the integration step

Auxiliary functions with the identical names posed a significant problem during our library integration step. Several libraries utilized similar performance measurement routines and test functions and thus clashed with the inclusion of multiple libraries due to the lack of support for overloading in C. To prevent this from occurring, a common library was created for the auxiliary functions.

Choosing a common documentation format for the RCLib was quite challenging. A correct documentation format can help the library developer maintain an up-to-date user guide for the library core. During a library build, which can change multiple times before a release, keeping documentation current is important to reflect the most recent changes. Creating and updating man pages or manuals from non-standard or static formats such as Microsoft Word or PDF format may be difficult. Sticking to an open source format such as TeX can ensure that the documentation can be converted into the necessary viewing format and also ensures the ability to update the document as necessary. Information such as resources used (FFs, MULT, CLB, LUTs, and etc), timing, and other information can updated during a library build to keep the document up-to-date. The documentation format selected for the RCLib project is PDF for the end user and LaTeX for the library developer.

Although all hardware cores in the RCLib project currently target Xilinx FPGAs, they were designed with very little to no assumption of available peripherals, memory, hard cores. This ensures that future porting to different processing elements such as Altera or future architectures can be done with minimal effort.

4.3 Collaboration

The collaboration and development framework was chosen based on vendor compatibility, to ensure that the libraries created were readily usable. There are several collaboration solutions available, and by far the most widely adopted is the Concurrent Versions System (CVS) source control tool [19]. Although there are several other solutions that provide more features, the sheer number of CVS users usually convince new adopters of its sustainability. CVS provides a basic set of tools necessary for collaboration, such as creation of branches, tags, change history, and much more.

Subversion (SVN) [20], provides most of the functionality of CVS with a key difference, SVN was built with internet collaboration in mind. SVN provides a secure web interface in which people would be able to check-in and check-out code, and also provides a robust security mechanism different from the user access list required by CVS. SVN is also able to version control directories, renames, and properties of files.

The source control software CVS was largely chosen due to its support by SRC Computers. SRC's scripts allowed the pull, parsing, and building of the library for unit testing, regression testing, and finally release build. Rigorous regression testing was used to ensure that additions to the library core would not break existing core functionality. This was a particularly difficult experience, as previously mentioned, due to the namespace clashes that can occur when utilizing library cores from different domains which use similar auxiliary function names. This was primarily due to the limitation in the C language as well as a lack of a naming convention for auxiliary functions across all the libraries developed. This also provided valuable lessons that encouraged developers to provide complete and up-to date unit test cases to ensure that the benchmarking and testing performed is able to discover the finer errors that could be otherwise difficult to detect.

4.4 Benchmarking and Testing

Tests can aid library developers in discovering faults and conflicts with their contributions. Short tests were created as sanity tests, which quickly verifies that the RCLib hardware cores compiles, builds, and passes simple functionality tests. This step was sufficient to catch simple failures. Longer tests, i.e. regression tests, were used to go through an entire suite of tests to ensure that there are no conflicts. Regression tests ran a comprehensive check against an entire RCLib to test for compatibility with each individual library core, the runtime environment, compiler options, tool chain, and other requirements. Regression tests were essential in discovering more subtle faults that would otherwise go undetected in simple tests. Frequent builds were done to help developers quickly determine the stability of their contributions and verify that previously working modules still function as expected. However hardware library tests can be very resource and time consuming, and at times it can take several hours to properly run a single core through the tool chain. Regression tests were performed along with each build and were essential in identifying cross library errors.

4.5 Distribution and Licensing

Licensing was perhaps one of the most confusing aspects of the RCLib project. There are various open source licenses available, and picking one was not a trivial task. Choosing a popular licensing model can help in modifying, utilizing, and integrating other existing work with similar licensing agreements, such Gnu is not UNIX (GNU) General Public License (GPL) [25]. GNU GPL, written and maintained by the Free Software Foundation, is widely used by the open source community, particularly for software developed for the Linux environment. The terms of the license is quite simple and aimed at complete freedom of use and only requires that the parts modified be shared with the community to benefit all and to also prevent derivative work from becoming proprietary.

Another popular license in the open source community is the Berkeley Software Distribution (BSD) license [26]. This license is particularly friendly to commercial users who do not wish to release their modified source back into the community and allows the packages to be re-licensed into a proprietary license if needed. There are several other license models considered, but appear to be more restrictive, for more information about other licensing models please refer to [24].

Unfortunately, mixing proprietary licenses with open source libraries is tricky. During our library development process, we were posed with several licensing decisions such as whether to use freely available cores from Xilinx and/or system vendors (SRC, Cray, SGI, etc.) and live with the license restrictions that allow use only on particular platforms and architectures. The alternative is to re-write the cores, regardless of how trivial they may be to allow for open distribution. We were also faced with similar licensing concerns with soft cores, such as those provided free for use from tools such as Xilinx Sysgen and others, which at times provided vague licensing restrictions. Licensing concerns also crept up during the development of the library

regression testing and benchmarking suites. Open source licenses such as GNU Multi-precession (GMP) library with GPL licensing, LiDIA's non-commercial use license, as well as OpenSSL BSD style license provided similar concerns. It was often difficult to determine which license agreement held precedence.

Trying to discover other third party tools that are compatible with the preferred license may not be trivial either. Not all third party tools are alike, and require the scrutiny of thorough research as to their ease of use, features, documentation, performance, development language, and other tradeoffs. Although it is possible to recreate software that can enable common functionality offered by the third party tools, it may not meet the specifications or performance requirements necessary for the library.

To alleviate the need to include the necessary tools with the library, the RCLib assumes that the end user is responsible for the availability of the necessary third party libraries and tools on their systems. All effort is made to ensure that the tools are available to the public and are well maintained.

The RCLib project provides two package formats. The first format is a binary release designed for the SRC6 platform. The library build and distribution process was done via the vendor's guidelines to ensure usability across available SRC platforms. The library is built in both binary and source package formats. Due to the complex nature of the library development which includes the use of basic Xilinx cores and other open source tools for benchmarking and testing, each with varying degrees of restrictions such as non-profit vs. for profit use, the license format chosen is fashioned after the GNU LGPL [27] license and assumes that the end user is in compliance with the license restrictions imposed by the third party components. Tables 1 to 8 show the available highly optimized library cores in each domain after performing a domain analysis.

The second format provided is a source code package which includes the HDL source code, black box interface file, documentation file, test code, test vectors. This package format was used in porting the library on to the CrayXD1 and SGI RASC platforms. Performance numbers and cost savings, compared to a typical workstation, utilizing the library cores in the respective target platforms are shown in tables 9 to 12. Power consumption, chassis size, and cost assumptions are shown in table 9.

Bioinformatics	
Application	**Cores**
Smith-Waterman	Scoring
	Maximum Score Search

Table 1: Bioinformatics Library Cores

Secret Key Ciphers	
Application	**Cores**
IDEA	encryption
	decryption
	breaking
DES	encryption
	decryption
	breaking
RC5	key scheduling
	encryption
	decryption
	breaking

Table 2: Secret Key Cipher Library Cores

Elliptic Curve Cryptography	
Application	**Cores**
Scalar Multiplication	Normal Basis
	Polynomial Basis
Project to Affine	Normal Basis
	Polynomial Basis
Point Addition	Normal Basis
	Polynomial Basis
Point Doubling	Normal Basis
	Polynomial Basis

Table 3: Elliptic Curve Cryptography Library Cores

Binary Galois Field Arithmetic	
Application	**Cores**
Polynomial Basis	
Trinomial	Squaring
	Multiplication
	Inversion
Pentanomial	Squaring
	Multiplication
	Inversion
Special Field	Squaring
	Multiplication
	Inversion
NIST	Squaring
	Multiplication
	Inversion
Normal Basis	
NIST	Squaring
	Multiplication
	Inversion

Table 4: Binary Galois Field Arithmetic Library Cores

Image Processing	
Application	**Cores**
Wavelet	Discrete Wavelet Transform
	Inverse Discrete Wavelet Transform
	Correlation and Histograming
Filtering	Gaussian Filtering
	Smoothing Filtering
	Sharpening Filtering
	Blurring Filtering
	Prewitt Filtering
	Sobel Edge Filtering
	Median Filtering
Buffering	Line Buffer

Table 5: Image Processing Library Cores

Long Integer Arithmetic	
Application	**Cores**
1024 bit	Montgomery Multiplier w/ Carry save adder
	Montgomery Multiplier w/ Carry propagate adder
	Modular Exponentiation w/ Carry save adder
	Modular Exponentiation w/ Carry propagate adder
1536 bit	Montgomery Multiplier w/ Carry save adder
	Montgomery Multiplier w/ Carry propagate adder
	Modular Exponentiation w/ Carry save adder
	Modular Exponentiation w/ Carry propagate adder
2048 bit	Montgomery Multiplier w/ Carry save adder
	Montgomery Multiplier w/ Carry propagate adder
	Modular Exponentiation w/ Carry save adder
	Modular Exponentiation w/ Carry propagate adder
3027 bit	Montgomery Multiplier w/ Carry save adder
	Montgomery Multiplier w/ Carry propagate adder
	Modular Exponentiation w/ Carry save adder
	Modular Exponentiation w/ Carry propagate adder

Table 6: Long Integer Arithmetic Library Cores

Matrix Arithmetic	
Application	**Cores**
Bit Matrix	Bit Matrix Multiplication
	Bit Matrix Transpose

Table 7: Matrix Arithmetic Library Cores

Sorting	
Application	**Cores**
Sorting	Bitonic Sorting
	Stream Sorting
	Odd-Even Sorting
	Heap Sorting
	Quick Sorting
Scheduling	Sorting Scheduler

Table 8: Sorting Library Cores

Platform	Number of FPGAs	FPGA Type	Maximum Frequency	RC System VS. Workstation		
				Cost	Power	Size
SRC6	4	XC2V 6000	100Mhz	200x	3.64x	33.3x
Cray XD1	6	XC2V P50	100MHz	100x	20x	95.8x
SGI RC100	6	XC4L X200	200Mhz	400x	11.2x	34.5x

Table 9: Target RC system specifications

Application	Speedup	Savings		
		Cost	Power	Size
Smith Waterman	1138x	6x	313x	34x
DES Breaker	6757x	34x	1856x	95.8x
IDEA Breaker	641	3x	176x	19x
RC5 Breaker	1140	6x	313x	34x

Table 10: Performance of sample library cores on an SRC6 system

Application	Speedup	Savings		
		Cost	Power	Size
Smith Waterman	2794	28x	140x	29x
DES Breaker	12162	122x	608x	127x
IDEA Breaker	2402	24x	120x	25x
RC5 Breaker	2321	23x	116x	24x

Table 11: Performance of sample library cores on a Cray XD1 system

Application	Speedup	Savings		
		Cost	Power	Size
Smith Waterman	8723	22x	779x	253x
DES Breaker	38514	96x	3439x	1116x
IDEA Breaker	961	2x	86x	28x
RC5 Breaker	6838	17x	610x	198x

Table 12: Performance of sample library cores on an SGI RC100 system

To request a copy of the RCLib hardware library package or for more information please visit http://hpc.gwu.edu/library.

5. Conclusion and Future Work

In this paper we detailed the proposed methodology for developing hardware library cores. The paper outlines the core issues related to creating a portable hardware library including domain analysis, code reuse, packaging, interface definition, portability, collaboration, benchmarking and testing, distribution and licensing. As shown in our case study, the process of building a portable set of hardware library components requires careful consideration and planning to alleviate the pitfalls that frequently occur when collaborating with a large group of developers, system vendors, chip vendors, and users.

For future work, we are looking to port our library from our case study platforms the SRC6, Cray XD1, and SGI RASC to newer architectures such as the DRC and processing elements such as Altera. Work is also underway to define a thin layer between the user application core and the vendor provides core interface services that will allow truly portable hardware library cores.

6. Acknowledgements

We would like to thank the various contributors to the library. We would also like to thank Dan Poznanovic and Paul Gage of SRC Computers for their tireless efforts in integrating the necessary components to allow for end user library development and deployment in the Carte-C environment.

7. References

[1] Xilinx Inc. System Generator, http://www.xilinx.com/products/software/sysgen/features.htm

[2] Altera Corp. DSP Builder, http://www.altera.com/products/software/products/dsp/dsp-builder.html

[3] SRC Computers Inc. Carte Programming Environment, http://www.srccomputers.com/CarteProgEnv.htm

[4] Annapolis Microsystems Inc. CoreFire Design Suite, http://www.annapmicro.com/corefire.html

[5] Starbridge Systems Inc. Viva development software, http://www.starbridgesystems.com/viva-software/what-is-viva/

[6] Nallatech Inc. DIMEtalk, http://www.nallatech.com/?node_id=1.2.2&id=19

[7] DSPLogic Reconfigurable Computing Toolbox, http://www.dsplogic.com/home/products/rctb

[8] Impuse CoDeveloper FPGA Compiler, http://www.impulsec.com/fpga_c_products.htm

[9] Mitrionics Mitrion SDK, http://www.mitrionics.com/default.asp?pId=23

[10] Celoxica DK Design Suite, http://www.celoxica.com/products/dk/default.asp

[11] OpenCores, http://www.opencores.org/

[12] Bharat P. Dave; "CRUSADE: Hardware/Software Co-Synthesis of Dynamically Reconfigurable Heterogeneous Real-Time Distributed Embedded Systems,"; DATE, p.97, Design, Automation and Test in Europe (DATE '99), 1999.

[13] Banerjee, S.; Bozorgzadeh, E.; Dutt, N.; Physically-aware HW-SW partitioning for reconfigurable architectures with partial dynamic reconfiguration; Design Automation Conference, 2005. Proceedings. 42nd, 13-17 June 2005 Page(s):335 – 340

[14] Rafael Maestre, Fadi J. Kurdahi, Milagros Fernandez, Roman Hermida, Nader Bagherzadeh, A Framework for Reconfigurable Computing: Task Scheduling and Context Management, Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, Volume 9, Issue 6, Dec. 2001 Page(s):858 – 873

[15] I. Gonzalez, S. Lopez-Buedo and F.J. Gomez-Arribas, “Implementation of Secure Applications in Self-Reconfigurable Systems”, Elsevier Microprocessors and Microsystems, 2007

[16] I. Gonzalez, S. Lopez-Buedo, F. J. Gomez and J. Martinez, “Using Partial Reconfiguration in Cryptographic Applications: An Implementation of the IDEA Algorithm”, Lecture Notes in Computer Science 2778, pp. 194-203, 2003

[17] OpenFPGA, http://www.openfpga.org/

[18] The Spirit Consortium, http://www.spiritconsortium.org/home

[19] Concurrent Versions System (CVS), http://www.nongnu.org/cvs/

[20] Subversion(SVN) Source Control system, http://subversion.tigris.org/

[21] Revision Control System(RCS), http://www.gnu.org/software/rcs/rcs.html

[22] IBM Rational Clearcase, http://www-306.ibm.com/software/awdtools/clearcase/index.html

[23] MS Visual Source Safe, http://msdn.microsoft.com/vstudio/products/vssafe/default.aspx

[24] Open source Initiative, http://www.opensource.org/

[25] GNU's not Unix General Public License,
http://www.gnu.org/copyleft/gpl.html
[26] BSD License,
http://www.opensource.org/licenses/bsd-license.php
[27] GNU Lesser General Public License,
http://www.gnu.org/copyleft/lgpl.html