Compile- and Run-time Services for Distributed Heterogeneous Reconfigurable Computing

Brian M. Holland, James Greco, Ian A. Troxel, Gabe Barfield, Vikas Aggarwal, and Alan D. George

{holland, greco, troxel, barfield, aggarwal, george}@hcs.ufl.edu

High-performance Computing and Simulation (HCS) Research Laboratory
Department of Electrical and Computer Engineering, University of Florida
Gainesville, FL 32611-6200

Abstract—Achieving seamless portability in reconfigurable hardware designs among vendor-specific FPGA platforms continues to be a nontrivial task. The current reliance on custom hardware wrappers and proprietary software APIs is detrimental to the efficiency of FPGA-accelerated application development. The issues with portability are compounded for distributed applications requiring multiple FPGA platform configurations. To address the problem, the USURP framework provides a standardized compile-time hardware/software interface for the design of write once, run anywhere hardware cores. Previous efforts have concentrated on the challenges of seamlessly porting applications between different RC platforms. In this paper we link the USURP standard to an RC cluster management framework, CARMA, to provide distributed heterogeneous reconfigurable computing services. To demonstrate the power of a distributed RC cluster, three case studies in distributed cryptanalysis are presented.

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

I. INTRODUCTION

The current design flow for FPGA development involves a lengthy sequence of tasks for achieving hardware acceleration. Designing an algorithm in a hardware description language is merely the first stage of this process. This code must be merged with a hardware wrapper containing the necessary components for interfacing and communicating with the application core. A software program must also be constructed which interfaces the FPGA driver in order to control the reconfigurable platform. Neither of these additional tasks are trivial in scope and can be further hindered depending upon the developer’s familiarity with the target platform.

The primary reason for the large overhead in FPGA design stems from the field’s lack of portability. Although two RC platforms may use similar peripheral buses and possess common resources (e.g., external SRAM and DMA FIFO), the underlying control structures are often dissimilar enough to require extensive manual translations to achieve cross-platform compatibility. Software designs face similar dilemmas; comparable high-level approaches exist for referencing FPGA resources but are implemented with proprietary APIs, inhibiting portability. Reconfigurable computing lacks the “layers of abstraction” approach found in software design which creates portability through a sequence of interface to which designs must conform. The most tedious aspects of platform-specific translation should be handled by automated compilers. By creating both a hardware and software abstraction layer for FPGA design along with the necessary back-end support (for interfacing the underlying interconnect), developers can create IP cores and software control programs which require no modification (aside from automated recompilation) for use on any supported hardware platform. Not only does this increase the “legacy” value of hardware designs but also it allows algorithms to be distributed over a heterogeneous collection of reconfigurable computers with no additional effort. USURP (USURP’s Standard for Unified Reconfigurable Platforms) provides the necessary compile-time infrastructure to support design portability.

While compile-time FPGA standards are necessary to achieving efficient hardware algorithm development, run-time management is critical for the success of distributed heterogeneous reconfigurable computing. USURP is chiefly responsible for run-time linking of software API calls to the corresponding vendor code. Although the USURP software API and a parallel programming language (e.g., MPI) are sufficient for distributed computing, the Comprehensive Approach to Reconfigurable Management Architecture (CARMA) under development at the University of Florida seeks to address the need for scalable, fault-tolerant, run-time management services for FPGA-accelerated HPC systems. CARMA is a full-featured job management infrastructure and middleware for FPGA-accelerated HPC systems and seeks to ease the transition from traditional-processor to multi-paradigm systems [1]. The CARMA framework is composed of numerous independent software agents that frequently communicate to schedule and execute jobs, configure RC devices, and gather and share system performance information among other management duties. CARMA does not incorporate a specific design-capture tool, programming language, bitstream generator, etc., thereby allowing users to design and build applications in any manner they see fit. The combination of USURP’s portability and CARMA’s run-time JMS features make for a powerful infrastructure for FPGA-accelerated HPC systems.

The remaining sections are organized as follows. Section II outlines previous and related work relevant to the USURP (and CARMA) framework. The hardware/software interface and the run-time services necessary to support USURP are described in Section III. Section IV provides additional information on CARMA’s runtime features and how they relate to the USURP framework. Section V highlights a suite of cryptographic-related applications that utilize and benefit from this standard. A summary of the paper’s contributions and future work is outlined in Section VI.

II. PREVIOUS AND RELATED WORK

Currently, the reconfigurable computing community lacks a unified solution for effectively standardizing the high-level concepts of hardware design and control with low-level implemen-
tation details. High-level language and graphical tools expedite development time, but shift the design challenges to manually porting applications into existing custom interfaces. Some top-down environments (e.g. SRC’s Carte and Nullatec’s DIME) exist, but do not address the greater issue of cross-platform compatibility due to their proprietary nature. Standardization efforts have remained isolated with no specific system achieving widespread deployment.

OpenFPGA [2] is a relatively new initiative to create a standardization body for reconfigurable computing. This organization caters to researchers and developers in government, academia, and industry. By channeling such a diverse collection of FPGA users, this community has made a successful effort at bringing issues such as portability and interoperability to the forefront of reconfigurable computing. What follows is a brief survey of prior endeavors related to USURP.

The Adaptive Computing System (ACS) [3] was one of the first endeavors into FPGA hardware and software standardization. The monolithic design approach of ACS was sufficient for working with the few RC resources in existence at the time, namely the SLAAC-1 and SLAAC-2 RC platforms. However, the emergence of numerous FPGA vendors with stable (though proprietary) communication schemes has forced a decreased reliance on custom third-party channels. Lightweight APIs built upon vendor APIs are used in USURP to reduce the overall complexity. IGOL (Imaging and Graphics Operator Libraries) [4] provides a compile-time and run-time framework for reconfigurable data processing applications. The primary emphasis of IGOL is to abstract the hardware-software interface of reconfigurable systems away from developers. IGOL is built upon the Microsoft Component Object Model (COM) and only targets Celoxica’s RC-1000 boards for Handel-C hardware design. USURP instead provides common control commands for hardware/software design in VHDL and C, respectively. Often, the subject of portability extends beyond support for cross-platform development into issues concerning modular algorithm development. A version of the Basic Local Alignment Search Tool (BLAST) [5] seeks to improve current methodologies for making legacy FPGAs designs more accessible to new RC platforms. Unfortunately, the RC-BLAST authors define “A portable implementation [as] one where the nonrecurring engineering costs associated with moving the implementation from one FPGA board to another is limited to interfacing issues and do not involve reworking the design.” While this is an important first step, USURP’s hardware/software interface standardization seeks to further reduce non-recurring engineering costs.

In addition to the code portability challenge USURP directly addresses, other efforts are underway to provide the feature-rich, run-time environment that high-performance computing (HPC) users have come to expect. Job Management Services (JMS) such as multitasking, job scheduling and data staging, among others have been developed to coordinate jobs on large-scale HPC systems in order to improve their utilization and availability. As traditional HPC systems are augmented with FPGA accelerators, these and other run-time services must adapt to account for the unique requirements of these new resources. For example, research at the University of Wisconsin [6] seeks to improve run-time scheduling techniques for managing FPGA usage based on metrics such as speed, area, power, etc. Although this approach begins to address the scheduling limitations found in previous work such as the Load Sharing Facility at George Washington University [7], the performance of any scheduler will be limited by its flexibility to assign a particular algorithm to an arbitrary reconfigurable resource. Ensuring the run-time portability of a hardware application across all platforms, a challenge not addressed by the University of Wisconsin system, becomes increasingly necessary. In addition, the scalability of this approach has yet to be tested beyond single-node implementations and the need for additional JMS features beyond job scheduling and deployment must be addressed and CARMA is an attempt to fill this void.

The MATLAB Compiler for distributed heterogeneous computing systems (MATCH) [8] provides a powerful mechanism for creating parallel hardware descriptions. The overall framework is capable of supporting not only reconfigurable computers but also embedded processing boards and specialized equipment such as DSP cards. MATCH is based upon the functional decomposition of Matlab code with the FPGA-bound components translated into RTL VHDL. In contrast, USURP relies upon existing VHDL hardware design and MPI programming models thereby promoting data decomposition over functional division in distributed algorithms. USURP standardization improves both hardware and software design, allowing them to reside separately instead of providing one, all-encompassing wrapper to abstract the custom implementations. A similar project [9] uses functional decomposition of signal processing simulation components across a distributed cluster of resources which can include FPGAs. A middleware helps to standardize the system architecture and ease development of reconfigurable components. However, this system and MATCH rely on a controller/worker topology for functional decomposition which tend to limit scalability depending on the architecture. USURP designs are intended to be fully scalable and require no central authority in the context of application execution.

The CHAMPION software design environment [10] facilitates the use of predefined functions and modules (referred to as “glyphs”) to accelerate development of multi-FPGA applications across multiple vendor platforms. The tool is primarily comprised of a library of glyphs (extendable by the user) for creating frameworks to underlying reconfigurable systems. USURP extends this general concept by narrowly defining the interfaces of prebuilt wrapper components. The resulting communication channels have standardized protocols which are independent of the underlying platform. Many software vendors provide powerful graphical tool suites like CHAMPION for constructing and abstracting RC infrastructures. However, the customizable aspects of these automated wrapper generators must be balanced with the need for ease of portability. Extra features result in a more adaptable tool but will lead to increased need for manual translation when migrating code across platforms.

Research into ubiquitous reconfigurable computing [11] has yielded a system similar to both USURP and CARMA. The concept of ubiquitous computing is defined as a system where minimal attention or understanding is required by the user to create or operate a program. This approach has long been a chief
tenant of traditional software architecture and has grown in importance in reconfigurable computing over the past decade. The basic components of the ubiquitous system are distributed resource management and remote function call handling. By using the freely available Jini middleware and generic APIs (supported by vendor-specific backend modules), the system is able to provide hardware abstraction. The ubiquitous framework is only defined for the XESS XCV-800 Virtex Board and is primarily based around a Java-to-hardware design flow. USURP is a logical successor to this project since it contains increased platform support, clearly defined software APIs, and richly featured run-time services. Both USURP and CARMA seek to expand on the prior research in compile- and run-time services. Both USURP and CARMA seek to create a standardized and more richly featured reconfigurable computing environment.

III. USURP Compile-time Services

Support for the USURP unified framework requires the development of a software API and a unified hardware wrapper. The software API and hardware wrapper are compile-time structures with standardized interfaces. The backends are customized to support the transition between the unified interface and the vendor-provided communication channel. The construction of these USURP components is a fairly complex process involving intimate knowledge of the underlying platform. However, this is a one-time development cost and the custom portions of the API and wrapper will be invisible to user applications. This section provides a top-down description of the compile-time USURP framework components.

A. Software API

The Universal FPGA Software API is a standardized programming model for FPGA-based application design. The API provides a generic medium for addressing and controlling FPGA resources from a software coprocessor. Reconfigurable devices often reside as expansion devices within a conventional von Neumann architecture. Thus, a distributed cluster of FPGA resources is heavily dependent on the software infrastructure to support parallel communication and operation. The abstraction API removes the software programmer’s need to understand the physical interconnection protocol between the FPGA and software resource. The underlying vendor API/driver is called by the USURP functions, with all proprietary features handled by the framework. Supporting a new FPGA resource under the standard requires linking the USURP functions to the appropriate vendor constructs. Fig. 1 illustrates the available API for use in a USURP software design. Although the underlying hardware specifics may vary greatly, the goal of any user is to read and write to the FPGA with as little effort as possible using whatever means necessary. USURP allows such communication to be done in a portable and efficient manner.

The USURP software API is meant to be a general suite of operations that encapsulates the maximum amount of functionality common in reconfigurable platforms. This approach is formulated out of the authors’ experiences with legacy FPGA code. However, this approach will not always be valid for a particular application nor will forward compatibility be guaranteed for new platforms created without thought for the USURP standard. RC cards may contain multiple FPGAs which allow new avenues for direct high-speed communication. Application-specific resources such as networking ports or analog-to-digital converters must be addressable to user software and/or FPGA applications. The base USURP system does not contain suitable functions or definitions to address these additional possibilities. However, the framework does allow for user-definable extensions to the function library. These avenues for expansion of the framework encourage portability and consideration for future projects involving similar inclusions of more exotic hardware.

B. Hardware Wrapper

The Universal Hardware Wrapper (Fig. 2) provides the software application with a unified interface to FPGA resources. A user-modifiable memory map interfaces common memory elements (e.g. register blocks, SRAM) to the host PC communication bus. The wrapper does not provide the host PC with direct access to the hardware application core; a wide variety of PC/FPGA communication interfaces (e.g. PCI, Rapid I/O, Serial) and RC vendor implementations of the interfaces prevent the design of a unified bus. By only allowing the host PC indirect access through memory resources, we can greatly enhance core portability between different RC platforms.

The hardware application core has concurrent access to the memory elements interfaced through the Universal Hardware Wrapper. As implemented on three platforms to date, these memory resources include a 1024-bit register block, 4 kB of internal block RAM, and several MB of external SRAM. These sizes were chosen somewhat arbitrarily based upon the researcher’s current needs, but can be augmented as required and RC platforms allow. In addition, non-standard features can be integrated into the USURP framework through the Extended Hardware Wrapper. The extended wrapper allows the USURP standard to include vendor-specific features without breaking

Setup Procedures
int USURP_Discovery(int fpga_id)
int USURP_Init(int fpga_id)
int USURP_Finalize(int fpga_id)

Configuration Procedures
int USURP_Set_clk(int fpga_id, int clk_id,
double freq_req, double *freq_act)
int USURP_Load(int fpga_id, char *bitfile)

Register Transfer Procedures
int USURP_Reg_read(int fpga_id, int addr,
int *data)
int USURP_Reg_write(int fpga_id, int addr,
int data)

DMA Procedures
int USURP_DMA_read(int fpga_id, int addr,
int *data, int len)
int USURP_DMA_write(int fpga_id, int addr,
int *data, int len)

Fig. 1. Universal FPGA Software API
platform independence. For example, the Nallatech BenNUEY-PCI-4e platform features four Ethernet ports that can be integrated into the wrapper’s memory map.

IV. CARMA AND USURP RUN-TIME SERVICES

CARMA provides run-time JMS middleware to which users submit jobs on any system node. Since CARMA is typically deployed as a fully distributed service, a Job Manager (JM) exists on each node in the system and can pass jobs to an appropriate node based on resource needs or loading levels. Users describe their application’s runtime requirements in the form of a Directed Acyclic Graph (DAG). DAG files are one of several common graph-based job description methods by which HPC users submit job information in order to optimize job scheduling and deployment. DAGs for CARMA have been extended to relay RC-related information such as number of and type(s) of FPGA resources required as well as other information beyond the scope of this paper (e.g. application fault-tolerance policy). The local JM to which the job is submitted is responsible for ensuring the job’s proper execution and result delivery.

Upon receiving a job, the local JM requests the required type and number of nodes upon which to perform job execution from the local Task Scheduler (TS). The job of task scheduling (as with all other CARMA services) is fully distributed, as central schedulers inherently have poor scalability and fault-tolerance. The TS on each node responds to scheduling requests from that node’s local JM. Based on the user’s criteria as defined in the DAG, the scheduler performs a query on the local copy of the system-information database which provides an accurate representation of the system to find an appropriate machine on which the task should execute. The local information database is updated very quickly (i.e. simulation projects demonstrate system-wide updates in the millisecond range for a thousand-node system) with global run-time information (e.g. CPU utilization, configuration loaded on a given FPGA, etc.) by the Gossip-Enabled Monitoring Services (GEMS) [12]. For parallel jobs, an appropriate collection of machines is selected. To make this determination, the CARMA scheduler performs true dual-paradigm scheduling by taking advanced RC-specific resource needs into account (i.e. with greater sophistication than a simple semaphore). For example, the time to configure FPGAs can be a rather large overhead, so the scheduler looks for FPGAs that have the required configuration already loaded. Also, if a task cannot be immediately executed on a particular FPGA, the scheduler can be made to schedule the task based on speculative information as to which machines and FPGAs will be free in the near future.

Fig. 3 provides a conceptual diagram of the interaction between USURP and CARMA run-time services. User applications can be executed strictly within the USURP system. Explicit management services are often unnecessary for small cluster designs. Users can write and compile their software control applications against shared object libraries for the Universal Software API (described in previous section) and whatever communication paradigm they choose (e.g. message passing or shared memory). CARMA run-time services become important for large-scale cluster management. At runtime, board-interfacing commands (i.e. the backend code that implements the Universal FPGA Software API) are handled by a low-overhead agent within CARMA dedicated to serving such requests for each board, know as the Board-Interface Module (BIM). This concept provides an additional layer of hardware
abstraction by keeping users from actually acquiring direct control over the RC resource (as is the case with all vendor FPGA board APIs). In today’s systems, a user’s process performs an “open card” on the FPGA which keeps other users from gaining access to the board (via the driver) while it is in use. This process inherently serializes access to the board and makes multi-tasking difficult to manage. In addition, if a user’s process that has control over the board crashes, the driver is often sent into an unrecoverable state, keeping all other users from accessing the board. In the authors’ experience, this problem can sometimes be fixed by manually restarting the driver but more often requires a system reboot. This approach is unacceptable when trying to build a multi-tasking system and greatly reduces system fault tolerance and availability. From USURP, a user’s process (through the API calls) requests operations to be performed via CARMA’s BIM. This scheme increases system security (by stopping unauthorized access), utilization (by allowing advanced configuration management schemes, multitasking and performance monitoring) and fault tolerance (by providing an agent to check the health of RC boards).

V. USURP OVERHEAD

Fig. 4 illustrates the overhead between a USURP and a “custom” wrapper for the ADM-XRC-II platform. Both of these wrappers are based upon example interfaces that were provided with the Alpha Data board (referred to as SDK). Consequently, the underlying control and DMA mechanisms are very similar with minor differences primarily at the connections to FPGA resources (block RAMs, register banks, etc.) and the application core. It is not surprising that the sustained throughputs for these two configurations vary only by a few percent at worst case. It should be noted that both subtle changes in the wrapper architectures and a rather large variance in the DMA benchmarking run-times are the primary reasons for the discrepancies in the throughput results. These graphs represent average transfer rates but single-point experiments will vary based upon some minor, uncontrollable events (e.g. CPU interrupts, synchronization stalls, etc.)

Fig. 5 and Fig. 6 highlight the differences between the USURP wrapper and the associated custom interfaces for the Nallatech platforms. The BenNUEY-PCI USURP wrapper is based upon a low-level wrapper that predates Nallatech’s DIMEtalk, a customizable network of FPGA components for interfacing Nallatech hardware. The baseline “custom” wrapper uses DIMEtalk since this approach would be typical of a current designs targeting a Nallatech platform. The differences between these two interfaces are more pronounced than with the Alpha Data platform but much of the same underlying transfer structure remains in place. The USURP wrapper was originally written using the older, low-level infrastructure to increase performance. However, the overall benefits remain minimal and work is currently underway to migrate the wrapper to use the current Nallatech DIMEtalk interface as the underlying baseline. The Nallatech BenNUEY-PCI-4e platform is the newest to be included under USURP and is based off the DIMEtalk interface. Consequently, the performance difference is virtually negligible.
The minimal changes of the USURP wrappers versus the customized interfaces had negligible effect on the resource utilizations and communication delay. The USURP interface was not intended to radically modify existing communication methodologies. It should be noted that only FPGA components and not the underlying hardware drivers are modified. Consequently both the USURP and “custom” versions share many of the same resources and protocols. All of the wrappers under discussion consumed less than 6% of total FPGA resources. The overall latency between the two wrappers is nearly identical. From the application core’s perspective, no extra penalties are incurred for accessing FPGA resources in a USURP wrapper. (e.g. Block RAMs still require 2 cycle delay for reads, and 1 cycle for writes.) Communication over the PCI bus requires at most one extra cycle to initiate DMA transfers for some platforms. This single cycle penalty also exists for accessing external SRAM.

VI. CASE STUDIES

The USURP and CARMA run-time services provide an efficient environment for application development on distributed heterogeneous reconfigurable computers. While FPGAs are extremely powerful computational engines, they must be configured and managed effectively before an application can benefit from them. The proceeding case studies were selected for their emphasis on distributed, cryptographic-type problems. In many instances, cryptographic algorithms have inherently large computation to communication ratios making them ideal for distributed computing. Each of these algorithms began as a VHDL design targeted for the USURP hardware framework. The resulting code was synthesized for the Alpha Data ADM-XRC-II, Nallatech BenNUEY-PCI, and Nallatech BenNUEY-PCI-4e RC platforms. No manual translation was required to target these platforms since they are supported under USURP. A USURP software API program was created to interface with the FPGA resources with MPI providing the necessary interprocess communication and synchronization. CARMA was used to manage the FPGA resource availability and application execution. A lightweight USURP protocol was used to identify the FPGA resource (and appropriate configuration file) corresponding to the each MPI process. The USURP run-time functionality will eventually be incorporated with the CARMA services to create a more robust identification mechanism.

A. N-Queens

N-Queens is a computationally complex depth-first search algorithm and is consequently useful as a cryptographic benchmark. The algorithm involves computing the number of possible combinations of \( N \) queens placed upon a chessboard of size \( N \times N \) such that no queen can attack another queen (under the rules of chess). No direct mathematical solution exists to accurately find all solutions for this sequence of numbers. An exhaustive search must be used to guarantee a correct result and the case study’s implementation has a computation complexity of \( O(N^3) \). The basic approach begins by placing a queen in the first row of the first column. A queen is placed in the second row such that it cannot attack the first queen and likewise a third is placed so as to not conflict with the prior two. The algorithm continues forward until a queen cannot be legally placed in the next column and subsequently backtracks by shifting the previous queen down one row and proceeding forward again. All valid combinations of \( N \) queens are recorded as they occur and the application terminates when the first queen is in the bottom row and no additional solutions can be found.

This algorithm contains a significant amount of both coarse and fine-grain parallelism and represents an excellent candidate for hardware acceleration. The problem is typically decomposed into multiple application kernels, each one computing all possible combinations for the case when the first queen is placed into a particular row. The parallel nature of these kernels allows their computation on distributed RC platforms to occur with virtually zero overhead. For small chessboard sizes, one FPGA platform may be sufficient to solve an entire problem. However, as in the case with the ADM XRC-II, filling an FPGA to capacity with N-Queens kernels can cause strained routing logic, reduced clock speed, and overall performance penalties. This problem is exacerbated as larger chessboard sizes require larger registers to store answers which requires additional FPGAs to store the expanded cores.

Table 1 illustrates the execution times for various configurations of the Florida distributed RC cluster. Each FPGA contains five N-Queens kernels capable of simultaneously solving up to five rows of an \( N \)- sized chessboard. Any quantity of FPGAs in the cluster can solve every N-Queens chessboard size. However, up to three sequential computations by the kernels may be required for resource-limited configurations. For a chessboard size of \( 5 \times 5 \), any number of FPGAs will have sufficient kernels to execute completely in parallel. However, a single FPGA must use its kernels twice to solve a size of \( 10 \times 10 \) and consequently has an execution time twice that of two or three FPGA configurations which possess sufficient kernels. Similar run-time patterns can be found for sizes of \( 12 \times 12 \) and \( 15 \times 15 \). A single FPGA must perform three successive computations, two FPGAs require two rounds, while three FPGAs can execute the entire operation in parallel. In the best case for board size \( 15 \times 15 \), three FPGAs achieve slightly greater then three times speedup over one FPGA due to the reduced control logic necessary for parallel execution. The USURP framework and the heterogeneous clusters allow greater concurrent execution by providing and managing more reconfigurable resources with minimal overhead.

B. DES Encryption

The Data Encryption Standard (DES) algorithm has been an important cryptographic system since its approval by the American National Standards Institute (ANSI) in 1981. Although the

Table 1: N-Queens execution time (secs) for various chessboard sizes

<table>
<thead>
<tr>
<th>Chessboard Size</th>
<th>1 FPGA (5 cores)</th>
<th>2 FPGAs (10 cores)</th>
<th>3 FPGAs (15 cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 \times 5</td>
<td>2.36E-5</td>
<td>2.36E-5</td>
<td>2.45E-5</td>
</tr>
<tr>
<td>10 \times 10</td>
<td>9.90E-3</td>
<td>4.98E-3</td>
<td>4.98E-3</td>
</tr>
<tr>
<td>12 \times 12</td>
<td>3.46E-1</td>
<td>2.27E-1</td>
<td>1.17E-1</td>
</tr>
<tr>
<td>15 \times 15</td>
<td>7.11E+1</td>
<td>4.44E+1</td>
<td>2.33E+1</td>
</tr>
</tbody>
</table>
Advanced Encryption Standard (AES) became official in 2001. DES remains a relatively secure algorithm, especially in the Triple DES (3DES) variant. DES itself is a symmetric key block cipher. Both encryption and decryption are performed using the same private 64-bit key. This key is then used to ‘seed’ a series of 16 regular permutations on a 64-bit block of text. The fundamental problem with breaking codes is not having very much information with which to work. FPGAs are an excellent platform for doing large volumes of computation based upon a small, static data set. The theoretical DES crack presented here requires only one plain- and cipher-text pair along with a few parameters dictating the range of keys to search.

DES cryptanalysis case studies are not without precedence in the reconfigurable computing environment. One endeavor [13] employs eight custom-made FPGA boards to implement “Mat-sui’s linear cryptanalysis.” Although the attack is successful in 2.3 hours, the 13-bit key size of the experiment reduces the applicability of the system to typical DES implementations. Another project [14] uses a time-memory trade-off cryptanalysis attack but requires 14.9 days to complete an exhaustive search. The distributed RC system used for this attack contains 58 Reconfigurable Architecture based on Scalable Hardware (RASH) units. Each unit is comprised of a general microprocessor board and seven “EXE-boards”, each containing eight Altera FLEX10K100 FPGAs. The 3248 reconfigurable devices used in this system confirm the size and complexity of the projected USURP-based DES attack illustrated in the results section.

Table 2 illustrates the necessary run-times for DES exhaustive searches on four theoretical setups for reconfigurable clusters. These statistics are based on initial experiments with the encryption core. The DES algorithm was implemented using a pipelined architecture with an application kernel testing one key per clock cycle. Successful experimentation has been conducted with DES kernels executing at 100 MHz frequency, however place and route tools have calculated the critical path under 4ns thus validating the possibility of clock speeds in excess of 200 MHz. The first experimental setup, Single FPGA from Table 2 estimates a runtime of over 500,000 days for a single kernel operating at this clock frequency. Area usage statistics cite one kernel as occupying approximately one-sixth of a Xilinx 2V6000 chip. Using this area estimate and projecting for a cluster of 3,248 FPGAs (i.e. the reconfigurable resources in the aforementioned “RASH” system), the Basic Cluster experiment reduces the execution time by four orders of magnitude. The Improved CLK experimental setup performs the DES exhaustive search approximately two days faster then the actual RASH system. The fourth experiment, Optimal incorporates the previous parameters plus more efficient resource utilization which should allow eight kernels and not reduce the design’s optimal clock frequency. The scalability of USURP is illustrated by the DES example, where a singular design is capable of utilizing thousands of FPGAs to solve large problems. USURP and CARMA run-time services eliminate the management difficulties of RC clusters; only the complexities of physically linking over three thousand FPGAs platforms restrict implementation of a DES exhaustive search.

C. Knapsack Problem

The knapsack problem was proposed by Ralph Merkle and Martin Hellman [15] as the backbone for a public key cryptography system. Although the system itself was proven insecure, it was a pioneer in the field of asymmetric ciphers. More interesting than the cryptographic extension is the actual knapsack problem itself, which is NP-complete. Trapdoor mathematics form a fundamental basis for encryption protocols. Problems such as the Knapsack (and others such as the Traveling Salesman) require an exponential amount of computation to determine an optimal solution based upon a relatively small amount of data.

The knapsack problem is a trial and error problem. A set of ‘objects’ is predefined, each one associated with a unique weight. A knapsack is user-defined to hold a certain weight. The problem is determining the specific set of objects, that when placed in the knapsack, create the desired weight. Although there are many simplifications that can be performed to improve a knapsack implementation beyond worst-case exhaustive approaches, the overall crux of the algorithm revolves around a scalable number of kernels, each trying certain ranges of possible combinations. Again, the inherent parallelism of this approach makes it ideal for distributed computing. Although the knapsack problem itself may not currently be relevant, it does illustrate the applicability of that class of NP-Complete problems to reconfigurable computing.

Table 3 summarizes the results from experimentation with software and hardware knapsack applications. The algorithm decomposes the problem set (i.e. the possible combinations of objects) among the number of available kernels for parallel execution. The software version performs all computations on a 2.4GHz Xeon processor. The FPGA implementation executes on the distributed reconfigurable cluster using the assigned number of hardware resources. The speedups gained through the use of multiple FPGAs over a single RC platform approach linear (e.g. 30 elements had 2.00 \times 2.89 \times speedup for two and three FPGAs, respectively) for sufficiently large

<table>
<thead>
<tr>
<th>20 elements</th>
<th>25 elements</th>
<th>30 elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software</td>
<td>0.1287</td>
<td>5.055</td>
</tr>
<tr>
<td>1 FPGA</td>
<td>0.1809</td>
<td>7.046</td>
</tr>
<tr>
<td>(2 kernels)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 FPGAs</td>
<td>0.1153</td>
<td>3.559</td>
</tr>
<tr>
<td>(4 kernels)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 FPGAs</td>
<td>0.097</td>
<td>2.376</td>
</tr>
<tr>
<td>(6 kernels)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
problems. Both the four and six kernel arrangements consistently outperform the software implementation despite the serial nature of kernel execution. The knapsack problem illustrates that algorithms with sufficient coarse-grain parallelism can be efficiently decomposed to distributed reconfigurable clusters. Even for relatively simplistic operations found in the knapsack algorithm (e.g. addition, comparison), the microprocessor could not outperform a relatively small number of FPGA kernels. However, such improvements would be significantly less feasible without USURP compile- and run-time services.

VII. CONCLUSIONS AND FUTURE WORK

The USURP framework is proposed as the unifying standard sought by the OpenFPGA community. It remains a user-centric standard, based upon the needs and experiences of current researchers. It capitalizes on the current trend of extremely diverse hardware and software implementations all performing relatively similar high-level operations. Conversely, the overall functionality of a design may remain consistent, but it is unlikely that communication channels of two arbitrary designs will ever perfectly conform. By imposing standards and rewarding conformity with increased compile-time and run-time support and reduced development time will the efficiency and productivity of reconfigurable computing advance.

The applications presented in this paper are representative of types of problems that can benefit from hardware acceleration, distributed operation, and RC standardization. N-Queens shows that a coarse-grain, computationally complex algorithm can achieve speedup with the availability of multiple RC platforms. While multiple application kernels can be placed on a single FPGA, additional resources provide more reconfigurable fabric for increased parallelism. For sufficiently large problems, bigger RC clusters removed the serial portions of the application thereby providing improvements over single FPGA implementations. DES illustrates not only the possibility of effective, large-scale algorithm kernel deployment but also the necessity for run-time services to manage such endeavors. It would be inefficient to design a custom lightweight scheduling protocol for every large scale project and ill-advised to rely on a new, untested system for programs with lengthy execution times (e.g. DES attacks). The knapsack problem shows that for even mathematically simplistic applications, RC clusters can achieve speedup assuming sufficient parallelism is exploitable. Like N-Queens and DES, the massive number of independent test cases in the knapsack problem provides ample opportunity for problem decomposition. While the speedups afforded by a multi-FPGA system are not surprising, the possibility for significant performance improvements despite the distributed, heterogeneous nature of the target RC cluster is noteworthy. The benefits of USURP and CARMA greatly overshadow the minimal overheads imposed through compile-time standardization and run-time management.

While the performance improvements of the case studies are a strong indicator of the success of USURP and distributed heterogeneous clusters, algorithm design efficiency is also an important consideration. While USURP has shown some (albeit small) run-time overhead, its has greatly reduced the development time for FPGA designs. Absolutely no changes or translation were needed for the case studies to target a particular FPGA platform. The applications were written with external interfaces compliant to the USURP standard and the resulting communication channel never proved a source of error during experimentation. This level of assurance proved extremely valuable when debugging errors because the hardware wrapper could be safely eliminated as a likely source of error. USURP allowed a hardware design to effortlessly move from simulation to hardware experimentation; only the extra time required for the automated synthesis and place and route tools to execute was required. The potential longevity of USURP designs was highlighted by the authors’ ability to compile and run other researchers’ code (e.g. N-Queens) without the need to comprehend the VHDL and with only cursory descriptions of the necessary hardware/software communication requirements.

The FPGA platform configurations currently available to USURP and CARMA can limit the overall performance versus traditional CPU clusters for communication-intensive applications. However, access to RC platforms with faster host processor interconnections will remedy this problem. Also, the emergence of multi-core CPUs has also begun to both question and illustrate the need for multi-FPGA designs. While software will retain many of the same challenges of migrating to parallel domains, the inherent data flow programming model of hardware remains very amenable to multiple FPGAs. Since FPGA applications are already decomposed into multiple functional blocks connected by communication channels, these computational units can essentially be separated across any distance provided a channel is available. The creation and mapping of channels both in physical board hardware and HDL infrastructures is nontrivial but still can be abstracted without inherently modifying the underlying programming model. Nothing prevents a USURP type framework from managing more direct communication channels to enable multi-FPGA designs. Such projects will become more critical as platforms with more tightly woven connection schemes are incorporated into USURP and CARMA.

Several immediate goals exist for the improvement of USURP. The expansion of the standard to support additional platforms is always a top priority. The Cray XD1 is a prime choice for inclusion since it will open up large-scale HPC machines to the USURP framework. Such platforms will also be the first real need for a USURP extended software API to manage the additional resources and protocols of the XD1 system. The inclusion of application mapper support in USURP will also be necessary for increased viability of the standard. High-level languages are revolutionizing the speed and efficiency of hardware design and their inclusion will greatly help USURP application development. More integration with CARMA is also under development to increase the capabilities of the USURP run-time system. By increasing the compile-time standards and run-time services, distributed heterogeneous reconfigurable computing will become more efficient and accessible to FPGA researchers.

VIII. ACKNOWLEDGMENTS

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


