An Automated High-level Design Framework for Partially Reconfigurable FPGAs

Rohit Kumar and Ann Gordon-Ross
NSF Center for High-Performance Reconfigurable Computing (CHREC)
Department of Electrical and Computer Engineering, University of Florida
Gainesville, Florida, USA
E-mail: {kumar, ann}@chrec.org

Abstract— **Modern field-programmable gate arrays (FPGAs) allow runtime partial reconfiguration (PR) of the FPGA, enabling PR benefits such as runtime adaptability and extensibility, and reduces the application's area requirement. However, PR application development requires non-traditional expertise and lengthy design time effort. Since high-level synthesis (HLS) languages afford fast application development time, these languages are becoming increasingly popular for FPGA application development. However, widely used HLS languages, such as C variants, do not contain PR-specific constructs, thus exploiting PR benefits using an HLS language is a challenging task. To alleviate this challenge, we present an automated high-level design framework—PaRAT (partial reconfiguration amenability test). PaRAT parses, analyzes, and partitions an application's HLS code to generate the application's PR architectures, which contain the application's runtime modifiable modules and thus, allows the application's runtime reconfiguration. Case study analysis demonstrates PaRAT's ability to quickly and automatically generate PR architectures from an application's HLS code.**

Keywords— FPGA, partial reconfiguration, partitioning

I. Introduction

Run-time partial reconfiguration (PR) of field-programmable gate arrays (FPGAs) affords several PR benefits by isolating reconfiguration to designer-specified partially reconfigurable region(s) (PRRs). This isolation enables FPGA applications to modularly and non-intrusively add/change functionality at runtime without halting the entire FPGA's execution, which enables easy adaptation to changes in system goals and improves system power, performance, flexibility, future extensibility, reliability, and maintainability. Since new functionality is added modularly, full application re-design is not required for these additions/modifications, thus PR can reduce the application's time-to-market since functionality can be added/augmented after deployment. Additionally, since PR allows application functionalities to time-multiplex PRRs, area and power requirements and overall system cost can be reduced (i.e., a smaller, cheaper FPGA can mimic the functionality of a larger, more expensive FPGA).

To maximize these PR benefits, application designers must overcome several PR design challenges, such as lengthy source code development time, PR-specific application partitioning, design space exploration, implementing the partitioned application on a target FPGA device, etc. Due to the faster code development time and increased designer productivity afforded by high-level synthesis (HLS) languages (e.g., Vivado-HLS from Xilinx [17], Impulse C from Impulse [9], and Catapult C from Calypto [2]) as compared to hardware description languages (e.g., VHDL and Verilog), FPGA applications are typically designed using HLS code [5][17].

To make HLS languages more accessible to FPGA application designers, most modern HLS languages are C/C++-based, which affords a quick learning curve, facilitates hardware-software co-design and co-verification, and allows leveraging new advancements in software compiler optimization [4] as compared to hardware description languages. However, since C/C++ is a software programming language, generating efficient hardware applications from C/C++ source code is challenging. C/C++ does not provide native, built-in constructs that are crucial for hardware design, such as concurrency, timing, bit accuracy, synchronization, and hierarchy. Additionally, C/C++ has complex language constructs that are difficult to efficiently synthesize into hardware, such as pointers, dynamic memory allocation, polymorphism, etc. [4]. To alleviate these challenges, modern HLS tools provide additional language constructs and use a restricted subset of the C/C++ language, however, these constructs only facilitate hardware design and do not alleviate PR-specific application design challenges.

In addition to challenges posed by high-level languages in PR-specific application design, existing high-level application design techniques, such as software design and hardware-software co-design techniques, do not facilitate PR-specific application design. Software design techniques are aimed for microprocessor-based target platforms but PR-specific applications are targeted for FPGA-based systems. PR-specific application design is significantly different than hardware-software co-design in which the target platform is a microprocessor-FPGA system. In a microprocessor-FPGA system, the FPGA is considered as a single hardware region as compared to the PR-specific application system where the FPGA may or may not be coupled with a processor and the

FPGA is considered as multiple interconnected hardware regions (PRRs).

Generating a PR-specific application from the application's HLS code requires identification of the application's portions/modules that will executed in mutual exclusion and can be modified at runtime. Whereas a designer could manually specify this PR-specific application partitioning to generate the application's PR architecture, this process requires tedious exploration of an unmanageably large design space. The PR architecture defines the application using a static module and one or more partially reconfigurable modules (PRMs). The static module contains the application's fixed functionality and is only configured at FPGA startup (i.e., the static module is never reconfigured during application execution), and the PRMs are reconfigured at runtime to time-multiplex PRR resources.

The static module's, PRMs', and PRRs' sizes and number vary depending on the PR-specific partitioning, and thus provide many PR architectures with varying PR benefits. Design space exploration to identify potential PR architectures that best adhere to designer-specified system goals, such as area, power, and/or performance requirements/constraints, has been addressed by few previous works [10]. Since PR-specific partitioning can range in size granularity from a single instruction per PRM to a single PRM containing all instructions, the PR architecture design space may conceivably contain $O(2^N)$ partitions where N is the total number of instructions in the application, thus designers require automated PR-specific partitioning to facilitate generation, exploration, and evaluation of PR architectures.

In this paper, we present the Partial Reconfiguration Amenability Test (PaRAT) to alleviate the PR design challenges involved in developing a PR-specific application from the application's HLS code. PaRAT evaluates an application's C-based HLS source code to capture the application's control and data dependence information. We represent the control and data dependence information using a partial reconfiguration model language (PRML) model similar to [11], but significantly enhanced to include nested dependencies. We partition this PRML model based on the PR-specific partitioning technique in [11], but our work enumerates all static module and PRM combinations, which enables complete exploration of the PR architecture design space. PaRAT quickly estimates the PR architectures' areas and clock frequencies using the VIVADO-HLS tool [17], which enables comparative evaluation of the PR architectures' tradeoffs with respect to system design goals.

Case study analysis shows that PaRAT can quickly and efficiently generate PR architectures from the application's C code. PaRAT automatically generates PR architectures and uses scripts to identify per-PR architecture area and longest path delay. The per-PR architecture area and longest path delay aid designers in selecting Pareto optimal PR architectures and target FPGA devices using PR design exploration tools such as [10].

II. RELATED WORK

Several previous works focused on PR architecture generation from the application's HLS source code but all previous works either did not perform automated PR-specific partitioning or did not generate all possible PR architectures. Additionally, all previous works either did not consider HLS source code or only considered highly specialized HLS languages.

Banerjee et al. [1] evaluated an application's task graph model and partitioned the graph to generate the PR architecture that minimized application execution time, however, the authors did not perform PR architecture design space exploration and did not consider the application's HLS source code. Shallenberg et al. [15] extended the OSSS HLS language to develop a PR-specific application, but this method required the designer to manually specify the PR-specific partitioning. To alleviate manual PR-specific partitioning, Santombrogio et al. [12] inferred the task graph from the application's HLS code and automatically partitioned the task graph into the static module and PRMs, however, the authors only considered a PR architecture with a fixed number of PRMs and PRRs, which could limit the application's PR benefits and efficiency if the PR architecture was not appropriate with respect to the system goals.

To consider multiple PR architectures, Uchevler et al. [16] evaluated the application's HLS code in CLaSH [3], which is a subset of Haskell. CLaSH provides constructs to allow PR-specific partitioning to create multiple PR architectures for evaluation. However, since CLaSH is a highly specialized language, CLaSH has not been widely adopted. In prior work [11], we alleviated the requirement of a highly specialized language using a simple graph modeling technique to model an application's algorithm. Our work partitioned the graph model using PR-specific partitioning techniques to create all potential PR architectures. Even though that work used a simple graph modeling technique, which is generally applicable and portable, the technique's usability was still limited since an HLS language was not considered.

III. PARTIAL RECONFIGURATION AMENABILITY TEST (PARAT)

PaRAT aims to alleviate PR design challenges, such as parsing and analyzing the application's HLS code, PR-specific partitioning, PR architecture generation, and facilitating PR design exploration. PaRAT's methodology leverages several tools and techniques to alleviate these PR design challenges. In this section, we detail PaRAT's methodology and discuss all relevant tools with examples.

A. Overview

Figure 1 presents an overview of PaRAT's methodology. PaRAT evaluates the application's HLS source code written in VIVADO-HLS-synthesizable C, and parses and analyzes this code using an in-house, heavily modified *gcc-python-plugin* [12] to generate a PRML model [11], which captures the

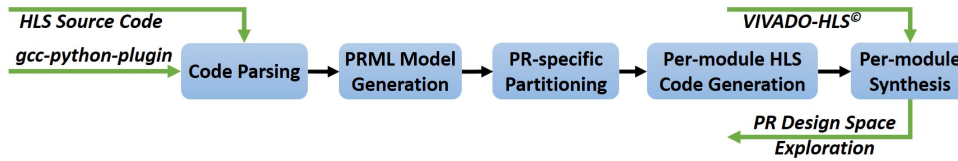

Figure 1. PaRAT's methodology overview.

application's control and data dependencies, and partitions the PRML model to create the static modules and PRMs (PR-specific partitioning). To enable PR design space exploration, PaRAT generates per-module HLS code from the application's HLS code, which are automatically synthesized using VIVADO-HLS to identify per-module area and longest path delay values to facilitate PR design space exploration.

B. *HLS Code Parsing and Analysis*

PaRAT parses and analyzes application's HLS code in VIVADO-HLS-synthesizable C to extract application's control and data dependence information. Since VIVADO-HLS is based on gcc [7], PaRAT leverages a gcc-python-plugin [12] to generate Python data structures that contain the application's control and data dependency information. The gcc-python-plugin is a Python plugin that leverages gcc's built-in plugin mechanisms to access and capture the intermediate gcc compilation information, which includes source code structure, control dependency between source code statements, per-statement input/output variable information, etc.

1) gcc-python-plugin: The gcc-python-plugin sequentially compiles each function in the C code using gcc and stores all intermediate compilation information in a per-function Python data structure that mimics gcc's tree data structure. The plugin updates this per-function data structure on various compilation events as enumerated in the gcc header file *gcc-plugin.h*. Additionally, these compilation events can also trigger a callback function, which is a mechanism that allows external tools to interact with the plugin. Since the plugin triggers a callback function after gcc creates the control and data dependency information in a control flow graph (CFG), PaRAT implements the callback function to analyze the CFG's Python data structure created by the plugin.

The CFG is composed of numbered GIMPLE [7] blocks starting from 0 (e.g., block 0, block 1, etc.,) and edges. GIMPLE is a family of intermediate representations based on gcc's tree data structure. Edges connecting the blocks represent the control flow relation between the GIMPLE blocks. GIMPLE blocks contain GIMPLE statements, which are the C source code pertaining to the GIMPLE block and these GIMPLE statements are stored in the GIMPLE statement's Python data structure.

A GIMPLE statement's data structure contains all information about the associated C statement, such as the C language construct type (e.g., assignment, decision, array, function call, etc.), input/output variable names, variable data type (e.g., int, char, etc.), assigned value, etc. This information is crucial for performing application control and data dependency analysis.

2) Control and Data Dependency Analysis. To analyze an application's control and data dependency, PaRAT sequentially analyzes each GIMPLE block for GIMPLE statement data structures. PaRAT creates a Python list, indexed by the GIMPLE block number, and gathers per-GIMPLE statement original C code to facilitate the generation of per-GIMPLE block HLS code. Additionally, PaRAT aggregates per-statement input/output variable names, variable data types, number of bits required to represent the variable (i.e., sizeof ()), array sizes where needed, assigned values where needed, and value type. PaRAT analyzes every variable's read-write sequence across GIMPLE statements in the GIMPLE block to identify the GIMPLE block's input/output interface. The per-GIMPLE block input/output interface aids PaRAT in generating application's PRML model.

3) PRML Model Generation. PRML [11] provides extensive guidelines, nodes, edges, and per-node and per-edge attributes to model an application. To avoid loopback edges and to facilitate PR-specific partitioning, PRML guidelines ensure that PRML models are always directed acyclic graphs. PRML nodes (i.e., computation, memory, decision, iteration, and hierarchy node) model algorithmic constructs, and control and data edges explicitly model control and data dependency behavior, respectively, between the PRML nodes. Per-node attributes capture the node's input/output variable information (e.g., name, type (register or array), size in number of bits, etc.), area requirements in terms of FPGA resources, longest path delay, external I/O interaction, etc.

The PRML model and CFG differ in several ways due to extensive modeling guidelines provided by PRML, which facilitates PR partitioning and PR architecture identification. The PRML model is always acyclic, but the CFG can be cyclic due to for and while loops. The CFG always has an explicit entry point, assigned as block 0, but the PRML modeling allows multiple entry points. The CFG shows inter-block control dependency, but the PRML requires explicit inter-block control and data dependency.

Generating the PRML model from a CFG is not a straightforward process due to these differences between PRML and the CFG. Additionally, since the number of CFG blocks and the CFG blocks' topology can vary drastically based on the application's control behavior, discussing the entire methodology to generate the PRML model from the CFG is impractical, thus we limit our discussion to an overview of PaRAT's methodology.

Figure 2 shows an overview of PRML model generation from a CFG. PaRAT sequentially analyzes each GIMPLE

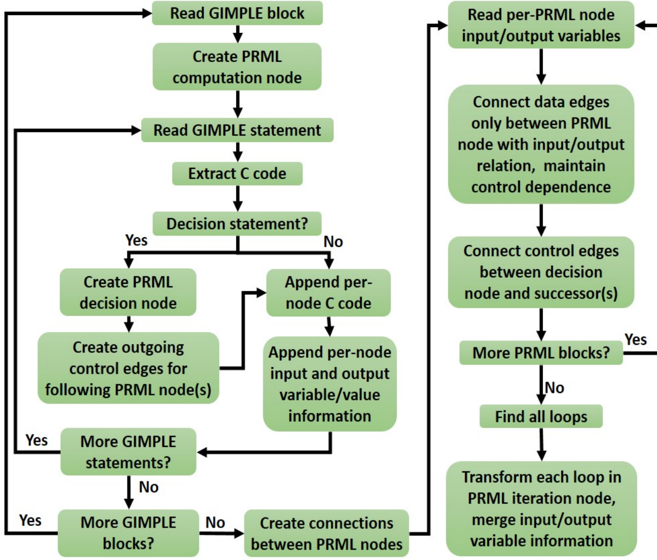

Figure 2. Block diagram representation of PRML model generation from a CFG.

block in the CFG and generates a PRML computation node associated with the GIMPLE block. PaRAT then sequentially reads the per-GIMPLE block GIMPLE statements to identify the source code and the statement type. If the statement type is a decision (i.e., the type for if, for, while, and switch statements), PaRAT creates an additional decision-type PRML node associated with the GIMPLE block. PaRAT stores the source code associated with the GIMPLE statement with the PRML node to facilitate HLS code generation. Additionally, PaRAT processes all input/output variables that are read/written, and the sequence in which these variables are read/written to identify aggregate per-GIMPLE block input/output variables, and stores these variables in the PRML node attributes. PaRAT repeats this process until all GIMPLE blocks are analyzed to create all PRML nodes and per-node attributes.

PaRAT sequentially analyzes the PRML nodes and per-node attributes to create explicit inter-PRML node control and data edges between PRML nodes. For example, PaRAT connects PRML nodes N_x and N_y with an outgoing data edge from N_x to N_y iff N_y falls on a directed path from N_x, and N_y's input variables share one or more variables with N_x's output variables. PaRAT connects nodes N_x and N_y with an outgoing control edge from N_x to N_y iff N_y falls on a directed path from N_x, and N_x and N_y do not share any input/output variables. Connecting all PRML nodes with the edges may result in loops in the PRML model. Since loops are circular paths and carry circular dependencies, loops in a PRML model prevent the partitioning methodology from identifying execution paths. To remove loops, PaRAT replaces each loop's PRML nodes with a PRML iteration node, and merges the loop nodes' attributes and source code in the iteration node's attributes and source code.

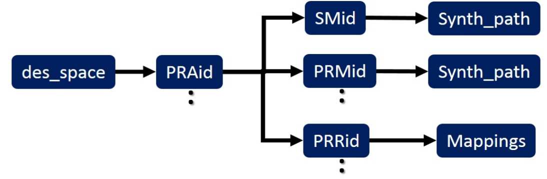

Figure 3. Simplified PaRAT output data structure format

The per-node source code are code snippets from the original C source code and are not synthesizable by VIVADO-HLS. To make the per-node source code synthesizable, the source code should be formatted in VIVADO-HLS-synthesizable C function declarations by utilizing per-node input/output variables as passed-by-reference function parameters. The function definition includes the function declaration and the per-node source code, and is stored in per-node VIVADO-HLS-synthesizable C files, which is used to generate area and longest path delay numbers, and perform PR design space exploration after PaRAT partitions the PRML model.

C. Partitioning

PaRAT leverages PRML's extensive PR-specific partitioning rules [11] to systematically analyze the PRML model to generate the PR architecture(s). The PR-specific partitioning rules leverage graph theory concepts and the per-block attributes to logically segregate common and mutually exclusive sub-graphs, generate the PR architectures as sets of sub-graph combinations, and stores PaRAT's output in a custom data structure that is compatible with the PR design space exploration tool presented in [10]. In this section, we provide a brief overview of PRML's PR-specific partitioning rules as presented in [11].

1) Partitioning rules: PRML describes the partitioning rules using standard graph theory terminology. A PRML model is a directed acyclic graph where PRML nodes and edges are the graph's nodes and edges, respectively. An execution path/cycle is a directed path/cycle between any pair of memory nodes. Trivial paths do not contain any computation or iteration nodes, do not show resource or longest path delay overheads, and thus are ignored during partitioning. A path *X* is a parent/child of a path *Y* if all of the nodes and edges of path *X* are a proper superset/subset of path *Y*. Two paths are siblings if the paths have the same parent node and two paths are clones if the paths have the same terminal node. A path's uncles are the path's parent's siblings or parent's clones. A path's cousins are a path's uncle's children. A weak data/control component is a maximal weakly connected component where all edges are data/control edges. A computation supernode represents a weak data component that contains at least two nodes and all nodes are computation nodes. An iteration supernode represents a weak data component that contains at least two nodes (at least one iteration node) and only computation and iteration nodes.

TABLE I. FUNDAMENTAL PARTITIONING RULES AND BRIEF DESCRIPTIONS OF THE RULES' EXECUTION RESULTS AFTER THE RULE IS APPLIED TO A PRML MODEL

Rule	Fundamental Partitioning Rules	Execution Results
1	Eliminate hierarchy nodes and memory nodes inside the hierarchy nodes	Eliminates redundant memory nodes by flattening the PRML model.
2	Identify computation and iteration supernode(s)	Reduces the number of nodes by merging interdependent nodes.
3	Identify all execution paths except symbol paths and trivial paths (L1 paths)	Identifies all non-trivial input to output paths.
4	Identify distinct smaller paths (L2 paths) from the L1 paths (sequentially break the L1 paths at choice and or-merge nodes but exclude symbol paths and trivial paths)	Identifies smaller data paths from the non-trivial input to output paths based on control choices.
5	Identify distinct smaller paths (L3 paths) from the L2 paths (break the L2 paths at iteration nodes and iteration supernodes but exclude trivial paths)	Identifies small computation kernels.
6	Identify all sets of static module and PRMs based on L2 paths, L3 paths, and node's divergent attribute value	Identifies all possible path combinations considering paths generated by rules 3-5, divides these paths into the PRMs and the static module.
7	Assign PRMs to PRRs: (a) clone PRMs are assigned to the same PRR; (b) sibling PRMs are assigned to different PRRs; (c) cousin PRMs can be assigned to the same or different PRRs	Calculates the number of PRRs required for each combination generated by rule 6 and creates all possible PRM to PRR assignments.
8	Create PR architectures.	Different PR architectures are created for each PRM variant and each PRM to PRR assignment.

TABLE I shows the fundamental partitioning rules and a brief description of the rules' execution results after the rules are applied to a PRML model. TABLE I reveals that PRML nodes can merge or split during partitioning to generate different static module and PRM combinations (i.e., PR architectures). PaRAT takes this node merging and splitting into consideration to generate per-module HLS code and stores these PR architectures and per-module HLS code in a custom output data structure that is compatible with the PR design exploration tool presented in [10]. Designers can use these PR architectures and the per-module HLS code to identify per-PR architecture area and longest path delay numbers and perform PR design space exploration using [10].

2) Output data format: PaRAT's output is an XML document and the document's XML data structure is compatible with the input data structure of the PR design space exploration tool presented in [10]. Figure 3 shows a simplified version of PaRAT's output XML data structure. The root element, *des_space*, contains all PR architectures as child elements, which are each identified by a unique *PRAid*. Each PR architecture contains unique ids for the static module, PRMs, and PRRs associated with the PR architecture and are represented by *SMid*, *PRMid*, and *PRRid*, respectively. The static module's and the PRMs' child elements, *Synth_path*, contains the module's HLS code's storage location in the application's design environment's filesystem. The script to execute the VIVADO-HLS tool for the module's HLS code to find the per-module area and worst case latency results stored in the same storage location as the module's HLS code, and thus, is identified by *Synth_path*. The *PRRid*'s child element, *Mappings*, contains a comma separated list of the *PRMid*s of the PRMs mapped to the PRR represented by *PRRid*.

IV. EXPERIMENTS AND RESULTS

We demonstrate PaRAT's efficacy in facilitating PR application development using a real world application case study. PaRAT parses, analyzes, and partitions the application's C code to generate the PR architectures, the PR architectures' VIVADO-HLS-synthesizable C code, and scripts to invoke VIVADO-HLS to synthesize the PR architectures and to identify per-PR architecture area and longest path delay from the VIVADO-HLS synthesis reports. The per-PR architecture area and longest path delay results can be used with a PR design exploration tool (e.g., [10]) to identify Pareto-optimal PR architectures and target FPGA devices, however, this portion of the design space exploration is out of the scope of this paper.

Our case study leverages the widely available encryption/decryption C application—IDEA (international data encryption algorithm) [8][18]. IDEA is a symmetric-key block cipher that was intended as a replacement for the data encryption standard (DES) [13]. IDEA encryption operates on 64-bit blocks of data using a 128-bit key, and consists of a series of eight identical transformations and an output transformation. The processes for encryption and decryption are similar.

PaRAT analysis on IDEA identified 11 functions and generated 109 potential partitions. Out of these 109 partitions, PaRAT identified 73 partitions to eliminate because these partitions were too small to be assigned as PRMs. Since PRRs have lower bounds on the amount of FPGA resources the PRR can contain, using these 73 partitions would result in significant wasted resources (i.e., internal fragmentation). Figure 4(a) and Figure 4(b) shows the automatically-generated HLS code for example non-trivial and trivial partitions,

```
void fn3_block4(long int a, long int b,
                long int x, long int y,
                long int d, long int *y_0){
 extended_euclidean (a, b, &x, &y, &d);
 *y_0 = y;}
```

(a)

```
void fn10_block11(
  short unsigned int *D_3001){
 *D_3001 = 1;}
```

(b)

Figure 4. Automatically-generated HLS code for partitions: (a) HLS code for a non-trivial partitioning, and (b) HLS code for a trivial partitioning.

respectively. PaRAT leveraged [11] to perform PR partitioning on the application and generated PR architectures with one to five PRRs of varying sizes. PaRAT also generated scripts to perform high-level synthesis of these PR architectures. These scripts aid PaRAT in identification of trivial partitions and aid designers in performing PR design space exploration.

We executed PaRAT on a Fedora 17 [6] virtual machine with 2GB memory and one Intel i7-3517U@1.9GHz processor core on a Ubuntu host machine. We used Python's time module to measure PaRAT's execution time, and averaged the times for 100 executions. PaRAT required less than four seconds to generate the partitions and per-partition HLS code, and less than nine seconds to perform PR partitioning, generate the PR architectures, and generate the automated scripts for high-level synthesis. The design time speedup afforded by PaRAT is substantial, but direct numerical comparison is difficult due to many variable aspects. Without using PaRAT, a designer would have to manually identify the application's partitions and perform the PR partitioning to generate the application's PR architectures for each considered partitioning. This process alone can take days or weeks depending on the application's complexity and requires PR-specific expertise. Thus, PaRAT's capability to rapidly identify and generate the PR architectures from the application's HLS code greatly aids designers in reducing this aspect of PR application development time.

V. CONCLUSIONS

In this paper, we presented a high-level framework—PaRAT (partial reconfiguration amenability test)—to aid designers in automatic and rapid application development for partially reconfigurable (PR) field-progammable gate arrays (FPGAs). PaRAT parses, analyzes, and performs PR partitioning on an application's high-level synthesis (HLS) code written in VIVADO-HLS-synthesizable C code. PaRAT leverages a gcc-python-plugin [12] to extract the application's control and data dependencies, and converts the control and data dependency information to a PR modeling language (PRML) [11] graph, which is a custom modeling language to aid PR partitioning of applications. PaRAT leverages and extends the PR partitioning methodology provided by [11] to generate the application's PR architectures. These PR architectures can be used with a PR design space exploration tool, such as [10], to identify Pareto optimal PR architectures and target FPGA devices, facilitating designers in more efficient and effective PR application design. Case study analysis showed that PaRAT was able to quickly generate PR architectures from the application's HLS code.

Future work includes full integration of PaRAT, a PR design space exploration tool, and an automated vendor-tool implementation flow to create a one-click framework for designing, developing, and implementing a PR application from the application's HLS code.

VI. ACKNOWLEDGMENTS

This work was supported in part by the I/UCRC program of the National Science Foundation (NSF) under Grant Nos. EEC-0642422 and IIP-1161022 and and the NSF CHREC membership support of Draper Laboratory. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. We gratefully acknowledge tools provided by Xilinx.

VII. REFERENCES

[1] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "Physically-aware HW-SW partitioning for reconfigurable architectures with partial dynamic reconfiguration." In Proceedings of the 42nd Annual Design Automation Conference (2005), pp. 335–340.

[2] Catapult high level synthesis and verification, http://calypto.com/en/products/catapult/catapult_overview#productinfo

[3] CLASH: CAES language for synchronous hardware, "http://clash.ewi.utwente.nl/"

[4] J. Cong, and B. Liu, "High-level synthesis for FPGAs: From prototyping to deployment", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2011), pp. 473–491.

[5] A. Cornu, S. Derrien, and D. Lavenier, "HLS tools for FPGA: Faster development with better performance," in Reconfigurable Computing: Architectures, Tools and Applications (2011), pp. 67–78.

[6] Fedora linux, "http://fedoraproject.org/"

[7] GNU compiler internals, "https://gcc.gnu.org/onlinedocs/gccint/"

[8] A. Hämäläinen, M. Tommiska, and J. Skyttä, "6.78 gigabits per second implementation of the IDEA cryptographic algorithm." IEEE Transactions on Field-Programmable Logic and Applications, (2002), pp760–769

[9] Impulse CoDeveloper C-to-FPGA Tools, "http://www.impulseaccelerated.com/"

[10] R. Kumar and A. Gordon-Ross, "Formulation-level Design Space Exploration for Partially Reconfigurable FPGAs." IEEE International Conference on Field-Programmable Technology (2011).

[11] R. Kumar and A. Gordon-Ross, "PRML: A Modeling Language for Rapid Design Exploration of Partially Reconfigurable FPGAs", IEEE Symposium on Field-Programmable Custom Computing Machines (2013).

[12] D. Malcom, "GCC Python Plugin", "https://fedorahosted.org/gcc-python-plugin/"

[13] National Bureau of Standards, Data Encryption Standard, FIPS-Pub.46. National Bureau of Standards, U.S. Department of Commerce, Washington D.C., January 1977.

[14] M. D. Santambrogio, and D. Sciuto, "Design methodology for partial dynamic reconfiguration: a new degree of freedom in the HW/SW codesign", IEEE International Symposium on Parallel and Distributed Processing (2008), 1–8.

[15] A. Schallenberg, W. Nebel, and F. Oppenheimer, "OSSS+R: Modeling and Simulating Self-reconfigurable Systems." In International Conference on Field Programmable Logic and Applications (2006), pp. 1–6.

[16] B. N. Uchevler, K. Svarstad, J. Kuper, & C. Baaij, "System-level modelling of dynamic reconfigurable designs using functional programming abstractions." International Symposium on Quality Electronic Design (2013), pp. 379–385.

[17] Xilinx Inc., Introduction to FPGA Design with Vivado High-Level Synthesis. UG998, July 2013.

[18] L. Xuejia, J. L. Massey. A Proposal for a New Block Encryption Standard. EUROCRYPT (1991), pp. 389-404