An Automated High-level Design Framework for Partially Reconfigurable FPGAs

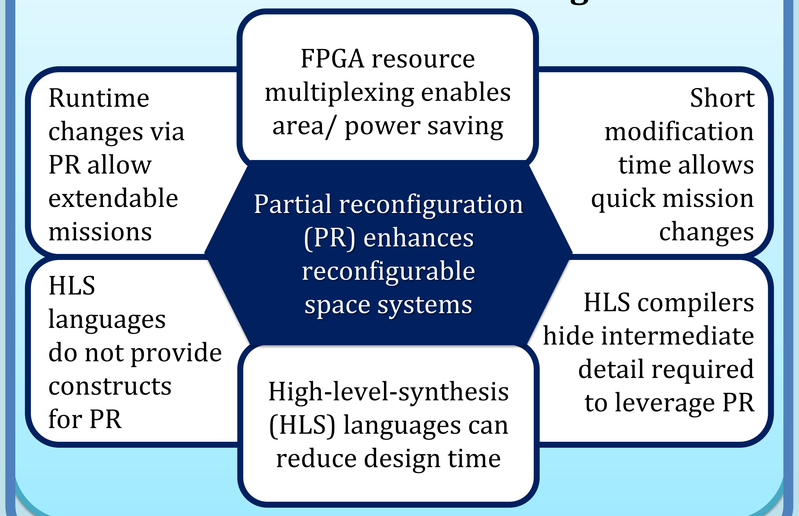
CHREC NSF Center for High-Performance Reconfigurable Computing

Rohit Kumar and Ann Gordon-Ross

UF UNIVERSITY of FLORIDA

Introduction

Motivations and Challenges



PR Partitioning

- PR requires applications to be partitioned in PR architecture(s)
  - PR architecture contains app's static and runtime swappable modules
- Application partitioning is performed on application's PRML model
  - PRML allows applications to identify PR-specific attributes

Fundamental partitioning rules and brief description of the rules' execution results after the rule is applied to an application's PRML model.

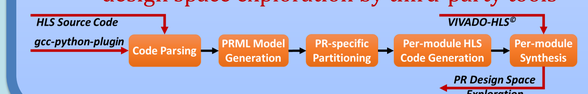
Fundamental Partitioning Rules	Execution results
1. Eliminate hierarchy nodes and memory nodes inside the hierarchy nodes	Eliminates redundant memory nodes by flattening the PRML model.
2. Identify computation and iteration supernode(s)	Reduces the number of nodes by merging interdependent nodes.
3. Identify all execution paths/cycles except symbol paths/cycles and trivial paths (i.e., L1 paths)	Identifies all non-trivial input to output paths.
4. Identify distinct smaller paths (i.e., 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 (i.e., L3 paths) from the L2 paths (break the L2 paths at iteration nodes and iteration supernodes but exclude trivial paths)	Identifies all 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.

PaRAT Methodology and Case Studies

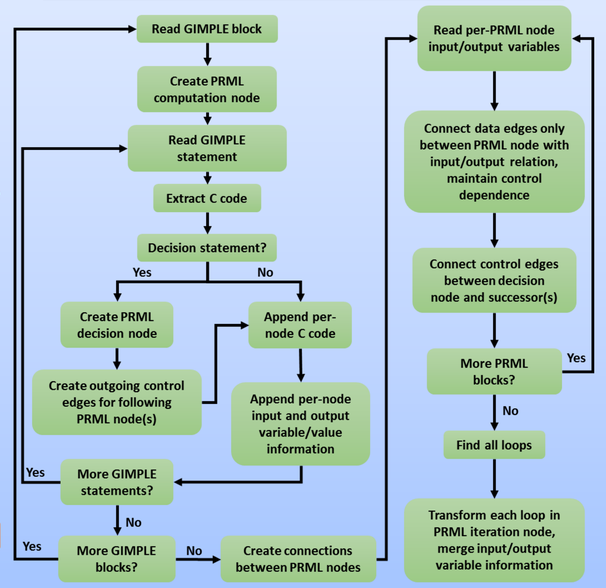
DRAPER LABORATORY

PaRAT Methodology

- Automatically parse and analyze application's HLS source code in C
  - Leverage *gcc-python-plugin* to extract control flow graph (CFG) and data flow analysis
  - Annotate CFG with data flow analysis and CFG's per-block synthesizable HLS source code
- PRML model generation and partitioning
  - PR partitioning requires explicit control/data dependence and PR-specific attributes
  - Convert CFG to PRML model
  - Partition applications based on PR-specific partitioning rules
  - Store partition and synthesis information in portable output data structure to enable PR design space exploration by third-party tools



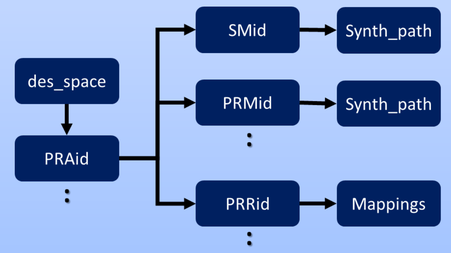
Flowchart for CFG to PRML model conversion



PR Application Development Case Study

- IDEA (International Data Encryption Algorithm)
  - Block cipher, used in pretty good privacy (PGP) v2.0
- Experimental Setup and performance analysis
  - Quantifying traditional PR app dev. time is difficult
  - Fedora 17, 2GB, one Intel i7-3517U@1.9GHz core
  - Execution time was averaged over 100 execution
  - ~4 seconds to generate modules and per-module source code, ~9 seconds for PR partitioning

PaRAT output data structure format



Floating point multiplier application from VIVADO examples

```
#include "fp_mul_pow2.h"
#ifdef ABS
#undef ABS
#endif
#define ABS(n) ((n < 0) ? -n : n)
float float_mul_pow2(float x, int8_t n){
#pragma AP inline
  float_num_t x_num, prod;
  x_num.fp_num = x;
#ifndef AESL_FP_MATH_NO_BOUNDS_TESTS
  if (x_num.bexp == 0xFF || x_num.bexp == 0)
    prod.fp_num = x_num.fp_num;
  else if (n >= 0 && x_num.bexp >= 255 - n) {
    prod.sign = x_num.sign; //
    prod.bexp = 0xFF;      // +/-INF
    prod.mant = 0;         //
  } else if (n < 0 && x_num.bexp <= ABS(n)) {
    prod.sign = x_num.sign; //
    prod.bexp = 0;         // +/-ZERO
    prod.mant = 0;         //
  } else
#endif  {
    prod.sign = x_num.sign;
    prod.bexp = x_num.bexp + n;
    prod.mant = x_num.mant;  }
  return prod.fp_num; }
```

Synthesizable code for multiplier app partitions

```
void fn2_block4(float D_1775, float x_num_fp_num, float prod_fp_num){
D_1775 = x_num_fp_num;
prod_fp_num = D_1775; }
```

```
void fn2_block6(unsigned char D_1779, unsigned char x_num_D_1740_bexp, int8_t n, int D_1780, int D_1781, int D_1782){
D_1779 = x_num_D_1740_bexp;
D_1780 = (int) D_1779;
D_1781 = (int) n;
D_1782 = 255 - D_1781; }
```

```
void fn2_block7(unsigned D_1784, unsigned x_num_D_1740_sign, unsigned prod_D_1740_sign, unsigned char prod_D_1740_bexp, unsigned prod_D_1740_mant){
D_1784 = x_num_D_1740_sign;
prod_D_1740_sign = D_1784;
prod_D_1740_bexp = 255;
prod_D_1740_mant = 0; }
```

```
void fn2_block10(unsigned D_1793, unsigned x_num_D_1740_sign, unsigned prod_D_1740_sign, unsigned char prod_D_1740_bexp, unsigned prod_D_1740_mant){
D_1793 = x_num_D_1740_sign;
prod_D_1740_sign = D_1793;
prod_D_1740_bexp = 0;
prod_D_1740_mant = 0; }
```