Introduction

Motivations and Challenges

- FPGA resource multiplexing enables area/power saving
- Partial reconfiguration (PR) enhances reconfigurable space systems
- HLS languages do not provide constructs for PR
- High-level-synthesis (HLS) languages can reduce design time

Fundamental Partitioning Rules

1. Eliminate hierarchy nodes and memory nodes inside the hierarchy nodes
2. Identify computation and iteration superno(s)
3. Identify all execution paths/cycles except symbol paths/cycles and trivial paths (i.e., L1 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)
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)
6. Identify all sets of static module and PRMs based on L2 paths, L3 paths, and node's divergent attribute value
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
8. Create PR architectures.

PR Partitioning

- PR requires applications to be partitioned in PR architecture(s)
- 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.

Execution results

- Eliminates redundant memory nodes by flattening the PRML model.
- Reduces the number of nodes by merging interdependent nodes.
- Identifies all non-trivial input to output paths.
- Identifies smaller data paths from the non-trivial input to output paths based on control choices.
- Identifies all computation kernels.
- Identifies all possible path combinations considering paths generated by rules 3-5, divides these paths into the PRMs and the static module.
- Calculates the number of PRMs required for each combination generated by rule 6 and creates all possible PRM to PRR assignments.
- Different PR architectures are created for each PRM variant and each PRM to PRR assignment.

PaRAT Methodology and Case Studies

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

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
- 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.

Floating point multiplier application from VIVADO examples

```
#include "fp_mixed_plugin.h"

#define ABS(n) ((n < 0) ? -n : n)

#define _fp_mixed_plugin2(float x, int n1) {
  if (n1 < 0 && n1 >= 255)
    return;
  else if (n1 < 0 && n1 == 0xFF)
    return;
  else if (n1 >= 0 && n1 == 0xFF)
    return;
}

void fn2_block6(unsigned char D_1779, unsigned int D_1780, int D_1781,
                unsigned char D_1782)
{
  D_1793 = x_num_D_1740_sign;
  D_1794 = x_num_D_1740_bexp;
  D_1795 = x_num_fp_num;
  prod_D_1740_bexp = prod_D_1740_mant = 0;
  prod_D_1740_sign = D_1784;
  prod_D_1740_bexp = D_1785;
  prod_D_1740_mant = D_1786;
  prod_D_1740_sign = D_1784;
  prod_D_1740_bexp = D_1785;
  prod_D_1740_mant = D_1786;
  prod_D_1740_sign = D_1784;
  prod_D_1740_bexp = D_1785;
  prod_D_1740_mant = D_1786;
}
```

Synthesizable code for multiplier app partitions

```
void fn2_block6(unsigned char D_1779, unsigned char D_1780, int D_1781,
                unsigned char D_1782)
{
  D_1793 = x_num_D_1740_sign;
  D_1794 = x_num_D_1740_bexp;
  D_1795 = x_num_fp_num;
  prod_D_1740_bexp = prod_D_1740_mant = 0;
  prod_D_1740_sign = D_1784;
  prod_D_1740_bexp = D_1785;
  prod_D_1740_mant = D_1786;
  prod_D_1740_sign = D_1784;
  prod_D_1740_bexp = D_1785;
  prod_D_1740_mant = D_1786;
  prod_D_1740_sign = D_1784;
  prod_D_1740_bexp = D_1785;
  prod_D_1740_mant = D_1786;
  prod_D_1740_sign = D_1784;
  prod_D_1740_bexp = D_1785;
  prod_D_1740_mant = D_1786;
```