Runtime Programmable Pipelines for Model Checkers on FPGAs

Mrunal Patel, Shenghsun Cho, Michael Ferdman, Peter Milder
FPGA Accelerates Model Checking

• Model checking ensures system correctness
  – By exploring all states of a system
  – Hours or days of run time with general purpose cores

• FPGAs show orders-of-magnitude speedup in run time
  – 3 hours → 12 seconds reported by prior work [Cho’18]
FPGA Preparation Time Voids The Benefit

- Models change while system development in progress
  - Fixing bugs, adding features, etc.
- Every model change results in new hardware logic
  - Requires hours to prepare (syn, P&R) new model checkers
- Prevents rapid system development iteration

<table>
<thead>
<tr>
<th>Total Time</th>
<th>Software Based Model Checker</th>
<th>FPGA Based Prior Work</th>
</tr>
</thead>
</table>

Synthesis
Place & Route
Our Approach: Programmable Pipeline

- Instruction controlled model checker on FPGAs
- Eliminate preparation time
  - No FPGA synthesis and P&R for new/modified models
- Limited overhead when comparing to prior work
  - Maintain 80% - 90% performance in run time
Outline

• Overview

• FPGA based model checking
  • Programmable Pipeline for FPGA Model Checkers
  • Evaluation
  • Conclusions
Explicit State Model Checking

• Bit vector explicitly represents system state
  – Contains PC, registers, variable values, etc.

• State space is logically represented as a graph
  – Edges represent all possible transitions of system states
  – Some states represent spec violations (code assertions)

• Model checker explores the graph
  – Visits all successors of each node
    (e.g., breadth-first traversal)
  – Discovers violating states
Model Checker Overview

- State space exploration:
  1. Take a state from queue, generate all successors
  2. Check and log violating states
  3. If successors not visited before, enqueue them
- Explore until queue is empty
Model Checking Challenges

• Costly computation for general purpose cores
  – Bit manipulation, memory compare, hashing, etc.

• Limited parallelism
  – Shared state queue and visited-state storage
Model Checking on FPGAs

- FPGA dedicated logic accelerates the computation
- Independent resources enables parallelism
  - Performance grows linearly with num of model checker cores
  - Limited by FPGA BRAM capacity and BRAM usage per core
- Prior work shows significant speedup in run time
Model Checking on FPGAs

- FPGA dedicated logic accelerates the computation
- Independent resources enables parallelism
  - Performance grows linearly with num of model checker cores
  - Limited by FPGA BRAM capacity and BRAM usage per core
- Prior work shows significant speedup in run time

Speedup in run time does not mean overall speedup
FPGA “Preparation Time” Problem

- HLS directly translates models into hardware
  - Every model change generates new hardware circuit
  - Synthesis and P&R for every new/modified model
- High resource utilization causes long P&R
  - Hours of waiting to generate the bitstream
  - Multiple iterations for timing closure

Preparation time kills the run time speedup
Outline

• Overview
• FPGA based model checking
  • Programmable Pipeline for FPGA Model Checkers
  • Evaluation
  • Conclusions
Replacing Model-Specific Logic

- Programmable pipelines replace model-specific logic
  - Successor State Generator
  - State Validator
- Maintain the same throughput as model-specific logic
Multi-Core for Parallelism

- Many independent model checker cores
- Control and violating state logging via AXI ports

Number of cores determines performance
Programmable Pipeline

- VLIW style pipeline
- 4 main stages for successor state generation
  - Instruction Fetch, Variable Select, Execution, Store
Instruction Fetch

- Instruction contains control signals for following stages
  - Including constants for value calculation
- Instructions stored in BRAM
  - Guaranteed latency and one instruction per cycle
  - Independent access for model checker cores

Instructions increase per-core BRAM usage
Variable Select

• Load variables and constants required for calculation
  – Variables from the parent state vector
  – Constants from instruction

• Each variable select unit loads one variable
  – Number of select units dependents on models
Execution

• A grid of ALUs to calculate:
  – Condition value
  – New variable values to be updated

• Limit ALU connection to reduce instruction length
  – ... hence reduce BRAM usage
Execution

- Two types of ALUs:
  - Normal ALU for doing calculation
  - Load ALU for loading values from the state vector
    - Indexed array access

- Limit num of load ALU to reduce connection

Normal ALU control bits

<table>
<thead>
<tr>
<th>SRC2</th>
<th>OP</th>
<th>OUT</th>
</tr>
</thead>
</table>

Load ALU control bits

<table>
<thead>
<tr>
<th>BASE_SEL</th>
<th>PID</th>
<th>OP</th>
<th>OUT</th>
</tr>
</thead>
</table>

PID=0: use SRC1 as index, PID=1: use PID as index

ALU designed to minimize connection and BRAM usage
Store

- Update variables inside the parent state vector
  - Based on condition calculated in the execute stage
- Each variable store unit updates variable
  - Number of store units depends on models
- One PC store unit dedicated for updating PC
Pipeline Parameters

• Stage requirements vary for different models
  – Variable select: number variables and constants
  – Execute: width and depth for the ALU grid
  – Store: number of variable that needs updating

• Affects the length of the instructions
  – ... which affects BRAM usage per model checker core
  – ... which affects number of cores can fit into an FPGA
  – ... which affects performance

  Overhead of programmability

Longer instruction $\rightarrow$ fewer cores $\rightarrow$ lower performance
Outline

• Overview
• Background
• Programmable Pipeline for FPGA Model Checkers
  • Evaluation
  • Conclusions
Evaluation

- Programmable model checker on FPGAs
  - Programmable pipeline for successor generator
  - With overhead of programmability (fewer cores)
- Baseline model checker on FPGAs: FPGASwarm\textsuperscript{[Cho’18]}
  - HLS based successor generator
  - No overhead of programmability (max num of cores)
- Common configurations for both model checker cores
  - Same frequency
  - Same per-core throughput (one-state-per-cycle)
  - Same queue size and visited state checker

Performance only depends on num of cores
Benchmarks

- 6 models from the BEEM database
  - Publicly available benchmark model set for model checkers

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>State Vec. (bytes)</th>
<th>Var. Sel. Units</th>
<th>ALUs Grid</th>
<th>Store Units</th>
<th>Inst. Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anderson.8</td>
<td>24</td>
<td>2</td>
<td>2x3</td>
<td>3</td>
<td>131</td>
</tr>
<tr>
<td>Bakery.8</td>
<td>28</td>
<td>2</td>
<td>2x5</td>
<td>3</td>
<td>167</td>
</tr>
<tr>
<td>Lamport.8</td>
<td>20</td>
<td>2</td>
<td>2x3</td>
<td>2</td>
<td>114</td>
</tr>
<tr>
<td>Leader_Filters.7</td>
<td>32</td>
<td>1</td>
<td>2x4</td>
<td>1</td>
<td>107</td>
</tr>
<tr>
<td>Mcs.6</td>
<td>24</td>
<td>2</td>
<td>1x1</td>
<td>2</td>
<td>64</td>
</tr>
<tr>
<td>Peterson.7</td>
<td>28</td>
<td>3</td>
<td>2x4</td>
<td>2</td>
<td>129</td>
</tr>
</tbody>
</table>
Results: Superset Checker for All

- One programmable checker for all benchmarks
  - Use the maximum parameter values
  - Load the model checker once for all benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>State Vec. (bytes)</th>
<th>Var. Sel. Units</th>
<th>ALUs Grid</th>
<th>Store Units</th>
<th>Inst. Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anderson.8</td>
<td>24</td>
<td>2</td>
<td>2x3</td>
<td>3</td>
<td>131</td>
</tr>
<tr>
<td>Bakery.8</td>
<td>28</td>
<td>2</td>
<td>2x5</td>
<td>3</td>
<td>167</td>
</tr>
<tr>
<td>Lamport.8</td>
<td>20</td>
<td>2</td>
<td>2x3</td>
<td>2</td>
<td>114</td>
</tr>
<tr>
<td>Leader_Filters.7</td>
<td>32</td>
<td>1</td>
<td>2x4</td>
<td>1</td>
<td>107</td>
</tr>
<tr>
<td>Mcs.6</td>
<td>24</td>
<td>2</td>
<td>1x1</td>
<td>2</td>
<td>64</td>
</tr>
<tr>
<td>Peterson.7</td>
<td>28</td>
<td>3</td>
<td>2x4</td>
<td>2</td>
<td>129</td>
</tr>
<tr>
<td>Superset</td>
<td>32</td>
<td>3</td>
<td>2x5</td>
<td>3</td>
<td>172</td>
</tr>
</tbody>
</table>
Results: Superset Checker for All

• Maintain at least 60% run time performance
  – Still significant faster than software model checkers

• Waste BRAM for models with short state vectors

Unnecessary BRAM usage hurts performance
Optimization: Best-Fit Checkers

• Superset checker wastes BRAM from some models
• Solution: Pre-generate a model checker library

① Sweep parameters to pre-generate model checkers
  – State vector size, number of sub blocks in each stage
  – Does not affect preparation time or run time

② When given a model, analyze its parameters

③ Load the best-fit model checker to FPGA
  – With the closest parameters that can check that model
Optimization: Best-Fit Checkers

- Regain performance using best-fit checkers
  - Performance only affected by overhead of programmability

Recover 80% - 90% performance of prior work
Conclusions

- Model checker on FPGAs shows significant speedup
  - Orders-of-magnitude speedup in run time from prior work
- But the FPGA preparation time voids the speedup
  - Synthesis and P&R required for every new/modified models
- Programmable pipeline eliminates preparation time
  - Avoid synthesis and P&R
  - Pre-compiled best-fit bitstreams to minimize overhead
  - Maintain 80% - 90% of the run time performance

shencho@cs.stonybrook.edu
Backups
Backups
Backups
Model Checking Time

- FPGA preparation time is significantly longer than model checking runtime by software.
Background

- **Promela**
  - ND: Non-determinism factor
  - PC: Current state
  - PID: Process ID

```c
byte balance=1;
active [2] proctype customer() {
    byte cash=0;
    S: if :: goto W;
        :: goto end;
    fi;
    W: if :: d_step { balance=balance-1;
        cash=cash+1; }
        goto end;
    fi
    end:
}
```
Instruction Fetch

- address <= \{PC, ND\};

- Instruction format

<table>
<thead>
<tr>
<th>Selection</th>
<th>Execute 0</th>
<th>Execute 1</th>
<th>Execute 2</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit 0</td>
<td>ALU 0</td>
<td>ALU 0</td>
<td>ALU 0</td>
<td>Unit 0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Unit m</td>
<td>ALU n</td>
<td>ALU n</td>
<td>ALU n</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>ALU n</td>
<td>ALU n</td>
<td>ALU n</td>
<td>Unit 0</td>
</tr>
<tr>
<td></td>
<td>ALU n</td>
<td>ALU n</td>
<td>ALU n</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>ALU n</td>
<td>ALU n</td>
<td>ALU n</td>
<td>Unit 0</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
(Software) Swarm Verification

• Expose parallelism in model checkers
  – Replace one large model checker with many small ones
  – Each “verification task” (VT) explores part of state space
    • VTs will overlap in exploration
    • ... but combination will statistically cover the space

• Advantages:
  – Massive, completely independent parallelism
  – Memory usage per model checker: GBs → MBs

\[\text{State Space} / \text{Model Checker} \rightarrow \text{State Space} / \text{Model Checker VTs}\]
Pipeline Stage Registers

- PID
- M selected values
- N immediate values (Constants)
- Several temporaries
- Instruction