Piparazzi: A Test Program Generator for Micro-architecture Flow Verification

Eyal Bin
Outline

- Methodology
- Types of bugs
- Generic Test Plan (GTP)
- Generation
- μArch comparison
- Results and summary
Methodology

Design → Simulation → μArch

Test → Generation

GTP → Interesting μArch cases

Comparison

Bug
Micro-architecture verification

- **Arch level**
  - Microprocessors
  - Core level verification

- **µArch level**

- **RT level**

There are more bugs in the µArch level than in the Arch level.
Different abstraction levels to find a functional bug using expected results

- Arch generator
- Arch expected results
- Arch simulated results
- A functional bug
Classification of bugs found by μArch comparison

- **Bugs found by μArch discrepancy**: timing, mechanism, location

While the test itself does not show a functional problem, it can be inferred from an analysis of the faulty μArch behavior.

Cannot be found by arch comparison
Generic Test Plan – processor independent test plan

- Coverage models for different units in the processor
- Cross-product coverage models
- Coverage models for resources and instructions
- Knowledge reuse

Even though microprocessors have a different μArch, we define a generic μArch test plan covering specific μArch behavior.
An example

For each pipeline in the processor
For each stage in the pipeline
For each stall reason in the stage
Generate a test that creates the stall
Generator

- Model-based (processor independent)
- Based on CSP
- Coverage by generation
- Two main outputs
- Slower than random test generator
μArch model

- Enables the definition of different microprocessors (Out of order, Pipelines, Super scalar, Multithreading)
- A cycle accurate model

Modeling = Configuration of predefined building blocks
Types of building blocks

Hardware
- Fetch
- Dispatcher
- Pipeline
- Cache
- Branch prediction
- Queue

Mechanisms
- Flush
- Forwarding
- Interrupt
- Splitting instruction

Instructions
- add
- sub
- load
- store
- sync

The model:
Each BB may have variables, constraints, parameters, and BBs
Solving the problem

IBM Labs in Haifa

User’s constraints

CSP builder

Engine

Microprocessor Model Constraints

User's constraints

CSP builder

Engine
CSP Builder

Each building block creates its modeled CSP variables
Each building block inserts its constraints:
  Design independent constraints
  Design dependent constraints (modeled)
User’s constraints are inserted
Example for constraints: pipeline stage building block

- **Design independent constraints:**
  - $\text{EXIT} = \text{ENTRY} + \text{TOTAL\_STALL}$
  - $\text{TOTAL\_STALL} = \text{SELF\_STALL} + \cdots$

- **Design dependent constraints:**
  - The conditions and amounts for an instruction to be stalled in the stage ($\text{SELF\_STALL}$)
  - The rules for grouping instructions in the stage
  - The conditions and timing to stall the stage above this one based on the instruction in this stage
The Engine

Solution: Each of the required variables reaches a single value. The constraints are satisfied.

Alg': A dedicated CSP random solver based on MAC (AC3)
Maintain Arc Consistency AC3

1. Bring every arc to a consistent state (if empty set -> backtrack)
2. Choose a variable (if none -> success)
3. Choose a Value (if none -> backtrack (back jumping))
4. Go to 1

A = B + C
A = \{1, 2, 3, 4, 5, 6\}
B = \{3, 7\}
C = \{0, 3\}
Specific issues for big and intricate CSPs

- Insert constraints on-the-fly just when necessary
- Different techniques to improve consistency for better success rate
- Domain specific heuristics for variable ordering
- Massive sharing of sub-constraints
- Domain specific redundant constraints

More backtracks

More time on consistency
μArch comparison

Simulation

A trace

μArch extractor

Actual behavior

Expected behavior

μArch comparator

Generator
Results

- Working with mainstream high-end IBM processors
- We found many bugs; most of them by using \( \mu \text{Arch} \) comparison
- About \( \frac{3}{4} \) of the bugs are performance bugs
- Experiment results: Many interesting \( \mu \text{Arch} \) events have a very low chance of being hit by random test generator (non-\( \mu \text{Arch} \))
- It is usually not easy to find a test that reveals the architectural bug based on a \( \mu \text{Arch} \) discrepancy
Summary

- μArch comparison is useful for finding both performance and Arch bugs
- μArch input language for a generator is important for reaching interesting μArch states
- Bugs have difficulty remaining underground when covering a comprehensive μArch test plan

"Language shapes the way we think, and determines what we can think about" - B.L. Whorf