Threadmill: A Post-Silicon Exerciser for Multi-Threaded Processors

Authors: Allon Adir, Maxim Golubev, Shimon Landa, Amir Nahir, Gil Shurek, Vitali Sokhin, Avi Ziv

Presenters: Abraham Addisie and Dong-hyeon Park
Outline

Motivation
Overview of Threadmill
Key Techniques of Threadmill
Conclusion
Questions
Debate!
Motivation – Why Post Silicon?

• Post-Silicon validation is becoming the next-level vehicle for functional verification
Problems of Post-Silicon Validation

- Loading tests externally have high communication and memory overhead
- Existing on-platform test generators are too complex
- Developing full OS system takes time

Solution: Simple on-platform test generator
Outline

Motivation

Overview of Threadmill

Key Techniques of Threadmill

Conclusion

Questions

Debate!
Threadmill Architecture

- Test Template
- System Topology & Configuration
- Architectural Model Testing Knowledge
- Generator & Kernel
- Builder

Threadmill Image
- Test Template
- Topology
- Architectural Model
- Generation
- Execution
- Checking
- OS services

- Accelerator
- Silicon
Test-Template Language

**Test Program Template**

Variable: \( \text{addr} = 0x100 \)
Values: Variable:
Bias: register-dependency

Instructions:
- Store \( R5 \) \( \rightarrow \) ?
- Repeat (\( \text{addr} < 0x200 \))
  - Instruction: Load \( \text{reg} \) \( \leftarrow \) \( \text{addr} \)
- Select
  - Instruction: Add ? \( \leftarrow \) reg + ?
    - Bias: sum-zero
  - Instruction: Sub ? \( \leftarrow ? - ? \)
- \( \text{addr} = \text{addr} + 0x10 \)

**Test Program**

Resource Initial Values:
- \( R6=8, R3=-25, \ldots, R17=-16 \)
- \( 100=7, 110=25, \ldots, 1F0=16 \)

Instructions:
- 500: Store \( R5 \) \( \rightarrow \) FF0
- 504: Load \( R4 \) \( \rightarrow \) 100
- 508: Sub \( R5 \) \( \rightarrow \) \( R6-R4 \)
- 50C: Load \( R4 \) \( \rightarrow \) 110
- 510: Add \( R6 \) \( \rightarrow \) \( R4+R3 \)
- ...
- 57C: Load \( R4 \) \( \rightarrow \) 1F0
- 580: Add \( R9 \) \( \rightarrow \) \( R4+R17 \)
Execution Process

Generate Test Case → Execute Test Case → Check Result

1. Generate Test Case
2. Execute Test Case
3. Check Result
Outline

Motivation
Overview of Threadmill

Key Techniques of Threadmill

Conclusion
Questions
Debate!
Threadmill Design

Goals:
- Simple
- Fast
- Good Coverage
- Multi-Threaded

Key Techniques:
- Static Test Generation
- Floating-Point Generation
- Concurrent Test Generation
- Multi-pass Consistency Checking
- Debugging Tests
Dynamic vs Static Generation

But what do you do when you really need to know the state of the CPU?
Observing Machine State

I want to send:
`beq $s0, $s1, exit`  # if (x==y)
but will it branch or not?
What’s $s0? $s1?

Threadmill

Instruction Stream

```
addi $t1, $t1, 1  # i = i+1
add $s0, $s0, $t1  # x = x+i
sub $s1, $s1, $t1  # y = y-i
```

uArch State

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$t1</td>
<td>7</td>
</tr>
<tr>
<td>$s0</td>
<td>3</td>
</tr>
<tr>
<td>$s1</td>
<td>3</td>
</tr>
</tbody>
</table>
Observing Machine State

Threadmill

$s0 = 3$
$s1 = 3$

Interrupt Handler

Instructions Executed

- addi $t1, $t1, 1 # i = i+1
- add $s0, $s0, $t1 # x = x+i
- sub $s1, $s1, $t1 # y = y-i
- 0xdeadbeef # Bogus

PC

Static Test Generation

Static Test Generation

<table>
<thead>
<tr>
<th>uArch State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t1$</td>
</tr>
<tr>
<td>$s0$</td>
</tr>
<tr>
<td>$s1$</td>
</tr>
</tbody>
</table>
Observing Machine State

Now I know $s0=$s1, so
beq $s0, $s1, exit  # if (x==y)
will branch to exit.

Instructions Executed

addi $t1, $t1, 1  # i = i+1
add $s0, $s0, $t1  # x = x+i
sub $s1, $s1, $t1  # y = y-1
beq $s0, $s1, exit  # if (x==y)

uArch State

<table>
<thead>
<tr>
<th></th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t1</td>
<td></td>
</tr>
<tr>
<td>$s0</td>
<td>3</td>
</tr>
<tr>
<td>$s1</td>
<td>3</td>
</tr>
</tbody>
</table>
Floating-Point Instructions

How to choose argument values?

random $\rightarrow$ inefficient

FPgen: Generate table of interesting test cases “off-line”

Threadmill Floating Point Tests

Random Generation FPgen Table

Floating Point Search Space

Corner Cases

Overflow, Underflow, etc
Concurrent Test Generation

shared_addr = 0xBASE + random()

Possible Collision Types
- Write-Write: True Collision
- Write-Read: False Collision
- ...

Need to be random, but consistent across all the test threads

Synchronization Techniques

Shared Random Seed
Multi-Pass Consistency Checking

Main Focus: Detecting Bugs in Multi-Threaded Consistency
Multi-Pass Consistency Checking

Main Focus: Detecting Bugs in Multi-Threaded Consistency

Execution 0

Threadmill

Reference Model

Registers, Memory values, etc

Execution i+1

Threadmill

Test Result

Equal

Reference Model

Each execution can have different timing or order of operation, but should end with the same result.
Debugging Tests

• Restart the failed exerciser image a few test-cases before the failure

• Take the test-template that causes the bug and run it on a pre-silicon test generation
Conclusion

**Strengths**

- Fast and Light-weight.
- Automatic testing and checking of multi-threaded execution.

**Weakness**

- Doesn’t check datapath or permanent bugs.
- Little evaluation of performance or coverage of the platform (paper).
Questions?
Debate

Threadmill’s failure detection mechanism only checks for bugs in multi-threaded interactions, and do not check bugs in the datapath. Is this adequate?

Instead of generating tests on-chip, isn’t it better to simply load pre-generated tests?