JIT-Assisted Fast-Forward Embedding and Instrumentation to Enable Fast, Accurate, and Agile Simulation

Berkin Ilbeyi and Christopher Batten

Computer Systems Laboratory
School of Electrical and Computer Engineering
Cornell University

ISPASS-2016, April 2016
Motivation

"Traditional" Computer Architects

Fast

Accurate

Application
Algorithm
Compiler
ISA
Microarchitecture
VLSI
Circuit
Emerging workloads require re-thinking the entire compute stack. Existing Fast and Accurate simulation methodologies exploit the fact that portions of the stack are “locked”.

**Agile:** quickly make changes in any layer of the stack without penalties
## Simulation and Evaluation Methodologies

<table>
<thead>
<tr>
<th>Method</th>
<th>Fast</th>
<th>Accurate</th>
<th>Agile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native execution</td>
<td>yes</td>
<td>no</td>
<td>yes*</td>
</tr>
<tr>
<td>Interpreter based ISS</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Dynamic Binary Translation based ISS</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Detailed simulation</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Running detailed simulation to completion for realistic workloads is unfeasible (over a year of simulation time!)
### Motivation

Over 99.9% of detailed simulation can be eliminated!

<table>
<thead>
<tr>
<th>Program phases</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### SMARTS

- **functional warmup**: long
- **history warmup** (e.g. caches)
- **detailed warmup**: long and short
- **history warmup** (e.g. pipeline)
- **detailed simulation**

#### SimPoint

- **functional profiling**: profiling to help sampling
- **fast forwarding**: no warmup
<table>
<thead>
<tr>
<th>Motivation</th>
<th>Pydgin Overview</th>
<th>JIT-FFE</th>
<th>JIT-FFI</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native execution</td>
<td><strong>yes</strong></td>
<td>no</td>
<td>yes*</td>
<td></td>
</tr>
<tr>
<td>Interpreter based ISS</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Dynamic Binary Translation based ISS</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Detailed simulation</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Sampling: Fast-forwarding</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td></td>
</tr>
</tbody>
</table>

**Diagram:**

```
    Interpreter based ISS
     |                  |
     |                  |
     |                  |
     |                  |
     |                  |
```

```
    Detailed Simulation
     |
     |
     |
     |
     |
```
<table>
<thead>
<tr>
<th></th>
<th>Fast</th>
<th>Accurate</th>
<th>Agile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native execution</td>
<td>yes</td>
<td>no</td>
<td>yes*</td>
</tr>
<tr>
<td>Interpreter based ISS</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Dynamic Binary Translation based ISS</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Detailed simulation</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Sampling: Fast-forwarding</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Sampling: Checkpointing</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Motivation</td>
<td>Pydgin Overview</td>
<td>JIT-FFE</td>
<td>JIT-FFI</td>
</tr>
<tr>
<td>------------</td>
<td>----------------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>Native execution</td>
<td>yes</td>
<td>no</td>
<td>yes*</td>
</tr>
<tr>
<td>Interpreter based ISS</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Dynamic Binary Translation based ISS</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Detailed simulation</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Sampling: Fast-forwarding</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Sampling: Checkpointing</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Sampling: Native-on-native execution</td>
<td>yes</td>
<td>yes</td>
<td>yes*</td>
</tr>
</tbody>
</table>

*Not fully agile because the target ISA needs to match the host ISA
<table>
<thead>
<tr>
<th></th>
<th>Fast</th>
<th>Accurate</th>
<th>Agile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native execution</td>
<td>yes</td>
<td>no</td>
<td>yes*</td>
</tr>
<tr>
<td>Interpreter based ISS</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Dynamic Binary Translation based ISS</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Detailed simulation</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Sampling: Fast-forwarding</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Sampling: Checkpointing</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Sampling: Native-on-native execution</td>
<td>yes</td>
<td>yes</td>
<td>yes*</td>
</tr>
<tr>
<td>Goal</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

**DBT-based ISS**

**Detailed Simulation**
Challenges of using DBT ISS for Fast Forwarding

Problem: Large architectural state transfers may hurt performance

Solution: JIT-Assisted Fast Forward Embedding (JIT-FFE). Embed DBT ISS in the detailed simulator which allows fast- and zero-copy architectural state transfer
Challenges of using DBT ISS for Fast Forwarding

- Problem: Large architectural state transfers may hurt performance
  - Solution: JIT-Assisted Fast Forward Embedding (JIT-FFE). Embed DBT ISS in the detailed simulator which allows fast- and zero-copy architectural state transfer.

- Problem: Functional profiling and warmup may hurt performance
Challenges of using DBT ISS for Fast Forwarding

Problem: Large architectural state transfers may hurt performance
  Solution: JIT-Assisted Fast Forward Embedding (JIT-FFE). Embed DBT
  ISS in the detailed simulator which allows fast- and zero-copy architectural
  state transfer

Problem: Functional profiling and warmup may hurt performance
  Solution: JIT-Assisted Fast Forward Instrumentation (JIT-FFI). Use Pydgin
  and RPython’s meta-tracing JIT to easily add JIT-compiled instrumentation.

PydginFF = Pydgin [ISPASS-2015] + JIT-FFE + JIT-FFI
Outline

- Motivation
- Pydgin Overview
- JIT-Assisted Fast-Forward Embedding
- JIT-Assisted Fast-Forward Instrumentation
- Results
Meta-Tracing JIT: the trace of interpreter interpreting the instructions is JIT compiled
Outline

- Motivation
- Pydgin Overview
- JIT-Assisted Fast-Forward Embedding
- JIT-Assisted Fast-Forward Instrumentation
- Results
Problem: Large architectural state transfers may hurt performance.

JIT-Assisted Fast-Forward Embedding

- Pydgin ADL and Framework
  - Type inference
  - Optimization
  - Code generation
  - Compilation

- Annot. IR
- Opt. IR
- JIT codes
- JIT runtime
- C source
- Compilation

- Pydgin binary
- PydginFF dynamic library
- Detailed simulator binary

- JIT-FFE API (RPython)
- JIT-FFE API (C headers)

- Detailed simulator source
- Compilation
- Dynamic linking
Outline

- Motivation
- Pydgin Overview
- JIT-Assisted Fast-Forward Embedding
- JIT-Assisted Fast-Forward Instrumentation
- Results
**Challenge**

- **Problem**: Functional profiling and warmup may hurt performance.
- **Solution**: JIT-Assisted Fast Forward Instrumentation (JIT-FFI). Use Pydgin and RPython’s meta-tracing JIT to easily add JIT-compiled instrumentation.
PydginFF (Pydgin + JIT-FFI)

```python
while True:
    inst = fetch(s.pc)
    execute_fun = decode(inst)
    execute_fun(state)
    instrument_inst(state)

    def execute_add(s):
        s.rf[rd] = s.rf[rs] + s.rf[rt]
        pc += 4

    def execute_store(s):
        s.mem[ s.rf[rs] ] = s.rf[rt]
        pc += 4

    def execute_bne(s):
        if s.rf[rs] != s.rf[rt]:
            pc = branch_targ
        else:
            pc += 4

    def instrument_inst(s):
        s.num_insts += 1
```

Target instructions

- 80: add r2, r2, r1
- 84: store [r2], r3
- 88: bne r2, r4, 80

**JIT trace**

- i_1 = rf[1]
- i_2 = rf[2]
- i_3 = rf[3]
- i_4 = rf[4]
- label( label1, i_2)
- i_5 = i_2 + i_1
- mem[ i_5 ] = i_3
- i_6 = i_5 == i_4
- rf[2] = i_5

**num_insts += 3**
- guard_true( i_6 )
- jump( labell, i_5 )

JIT-FFI-inlined instrumentation. Note the +3 optimization.
Motivation

Pydgin Overview

JIT-FFE

• JIT-FFI •

Results

PydginFF (Pydgin + JIT-FFI)

```python
while True:
    inst = fetch(s.pc)
    execute_fun = decode(inst)
    execute_fun(state)
    instrument_inst(state)

def execute_add(s):
    s.rf[rd] = s.rf[rs] + s.rf[rt]
    pc += 4

def execute_store(s):
    s.mem[ s.rf[rs] ] = s.rf[rt]
    pc += 4
    instrument_memop(s, s.rf[rt])

def execute_bne(s):
    if s.rf[rs] != s.rf[rt]:
        pc = branch_targ
    else: pc += 4

def instrument_inst(s):
    s.num_insts += 1

def instrument_memop(s, addr):
    idx = addr & idx_mask
    tag = addr & tag_mask
    for way in range( num_ways ):
        if tag == tags[idx][way]:
            return tag
    return refill( idx, tag )
```

Target instructions

80: add r2, r2, r1
84: store [r2], r3
88: bne r2, r4, 80

JIT trace

i_1 = rf[ 1 ]
i_2 = rf[ 2 ]
i_3 = rf[ 3 ]
i_4 = rf[ 4 ]
label( label1, i_2 )
i_5 = i_2 + i_1
mem[ i_5 ] = i_3
idx = i_3 & idx_mask
    tag1 = i_3 & tag_mask
tag2 = tags[idx][0]
    guard(tag1 == tag2)
i_6 = i_5 == i_4
rf[ 2 ] = i_5
    num_insts += 3
    guard_true( i_6 )
    jump( label1, i_5 )

JIT-FFI-inlined instrumentation. Note the +3 optimization.

JIT-FFI-inlined instrumentation.
## PydginFF (Pydgin + JIT-FFI)

### Simulation loop

While True:
- `inst = fetch(s.pc)`
- `execute_fun = decode(inst)`
- `execute_fun(state)`
- `instrument_inst(state)`

### Instruction semantics

**def execute_add(s):**
- `s.rf[rd] = s.rf[rs] + s.rf[rt]`
- `pc += 4`

**def execute_store(s):**
- `s.mem[ s.rf[rs] ] = s.rf[rt]`
- `pc += 4`
- `instrument_memop(s, s.rf[rt])`

**def execute_bne(s):**
- If `s.rf[rs] != s.rf[rt]:`
  - `pc = branch_targ`
- Else: `pc += 4`

**def instrument_inst(s):**
- `s.num_insts += 1`

**def instrument_memop(s, addr):**
- `idx = addr & idx_mask`
- `tag = addr & tag_mask`
- For `way in range(num_ways):`
  - If `tag == tags[idx][way]:`
    - Return `tag`
- Return `refill(idx, tag)`

### Target instructions

- `80: add r2, r2, r1`
- `84: store [r2], r3`
- `88: bne r2, r4, 80`

### JIT trace

- `i_1 = rf[1]`
- `i_2 = rf[2]`
- `i_3 = rf[3]`
- `i_4 = rf[4]`
- `label(label1, i_2)`
- `i_5 = i_2 + i_1`
- `mem[i_5] = i_3`
- `call(instrument_memop)`
- `i_6 = i_5 == i_4`
- `rf[2] = i_5`
- `num_insts += 3`
- `guard_true(i_6)`
- `jump(label1, i_5)`

### JIT-FFI-inlined instrumentation

Note the +3 optimization.

### JIT-FFI-outlined instrumentation

---

*Cornell University*  
Berkin Ilbayi  
14 / 19
Inlined Instrumentation with Data-Dependent Control-Flow
Outline

- Motivation
- Pydgin Overview
- JIT-Assisted Fast-Forward Embedding
- JIT-Assisted Fast-Forward Instrumentation
- Results
SMARTS and SimPoint Results

SMARTS simulation duration (hours)

- gem5Atomic + gem5
- PydginFF + gem5

SimPoint simulation duration (hours)

- gem5Atomic + gem5 with profiling
- PydginFF + gem5 with profiling
SMARTS and SimPoint Results

- SMARTS Results
  - gem5Atomic+gem5
  - PydginFF+gem5
  - 16X performance improvement

- SimPoint Results
  - gem5Atomic+gem5 with profiling
  - PydginFF+gem5 with profiling
  - 64X performance improvement
JIT-FFI Functional Warmup Case Study

- no cache
- dir mapped outline
- set assoc outline
- dir mapped inline
- set assoc inline

---

401.bzip2  429.mcf  445.gobmk  456.hmmer  458.sjeng  462.libquantum  464.h264ref  471.omnetpp  473.astar

log(MIPS)
Sampling, JIT-Assisted Fast-Forward Embedding, and JIT-Assisted Fast-Forward Instrumentation added on top of Pydgin allows fast, accurate, and agile simulation.

Compared to using an interpreter-based instruction set simulator, PydginFF+gem5 is $16 \times$ faster using SMARTS, and $64 \times$ faster using SimPoint.

In the absence of data-dependent control flow, inlined instrumentation gives very good performance, otherwise outlining can be used.

We have also developed set-associative cache model without data-dependent control flow and L2 cache modeling.