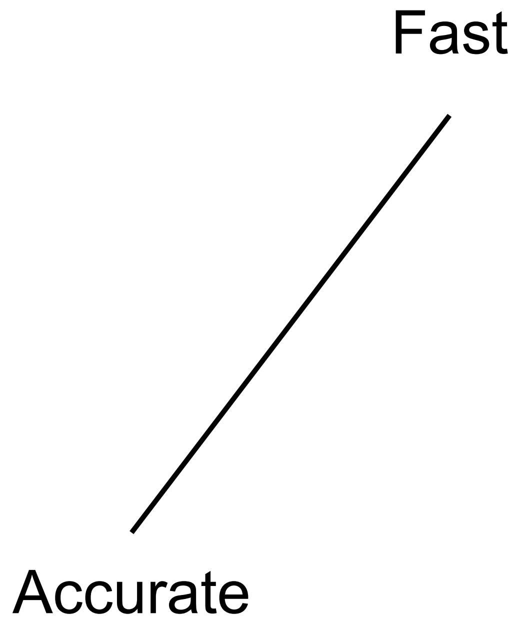
JIT-Assisted Fast-Forward Embedding and Instrumentation to Enable Fast, Accurate, and Agile Simulation

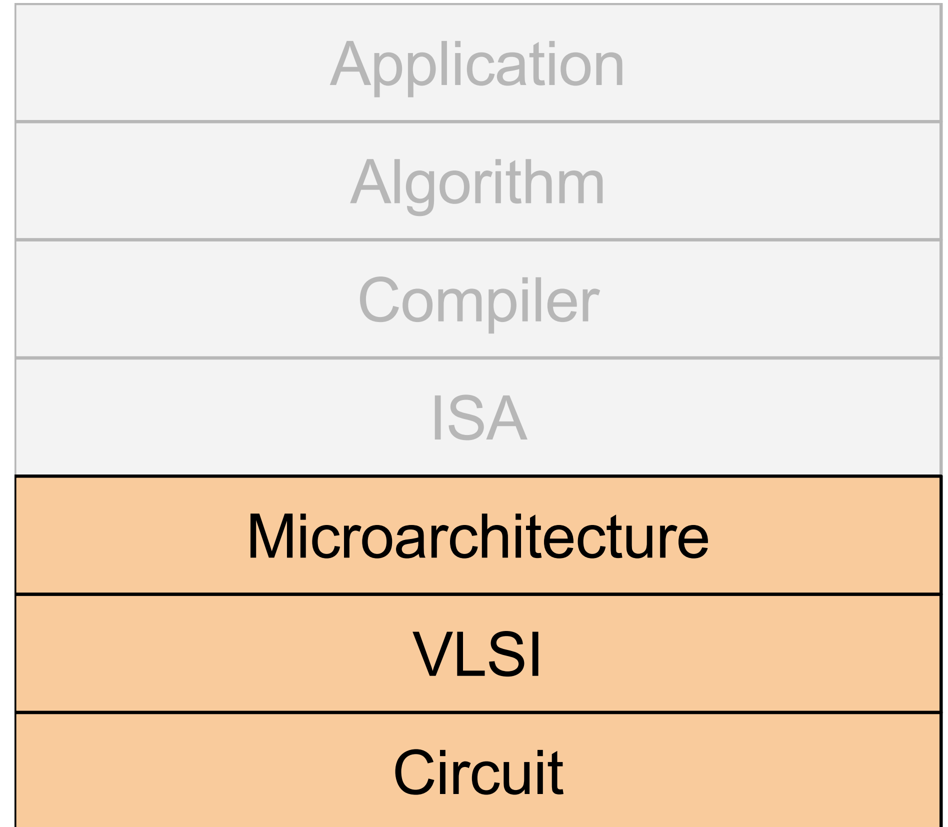
Berkin Ilbeyi and Christopher Batten

Computer Systems Laboratory
School of Electrical and Computer Engineering
Cornell University

ISPASS-2016, April 2016

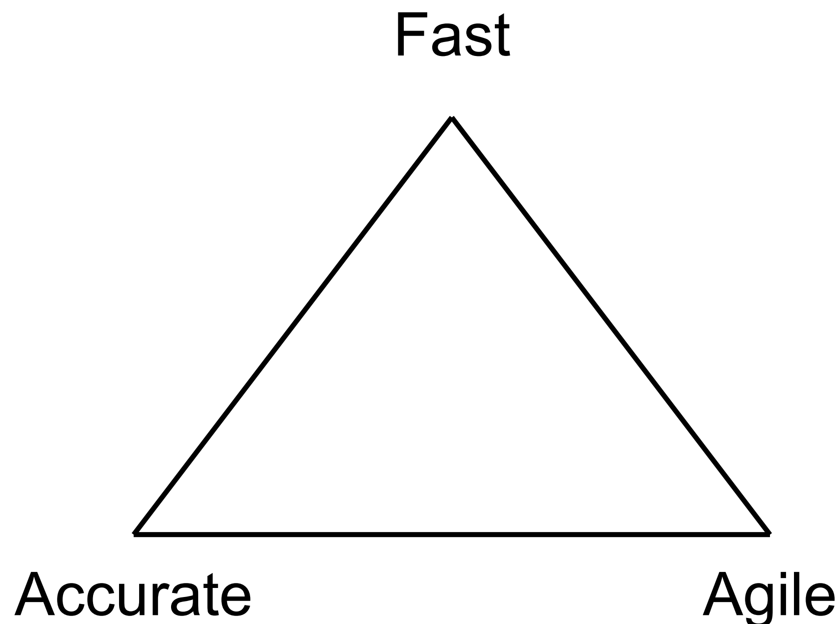
Motivation

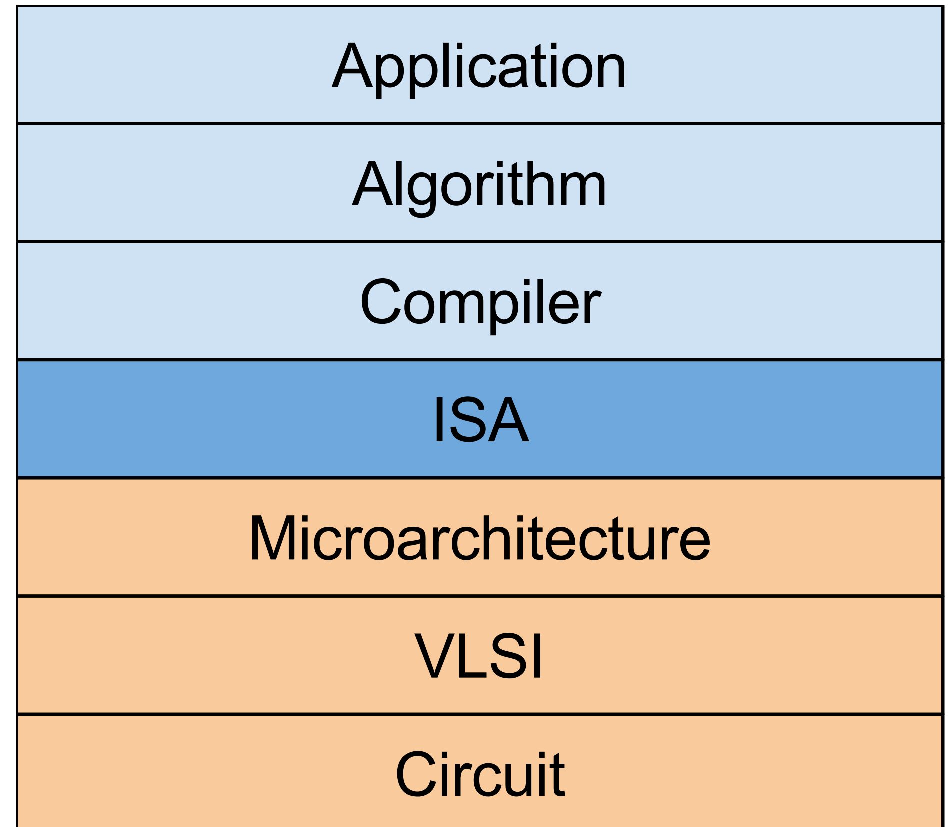




“Traditional” Computer Architects

Motivation





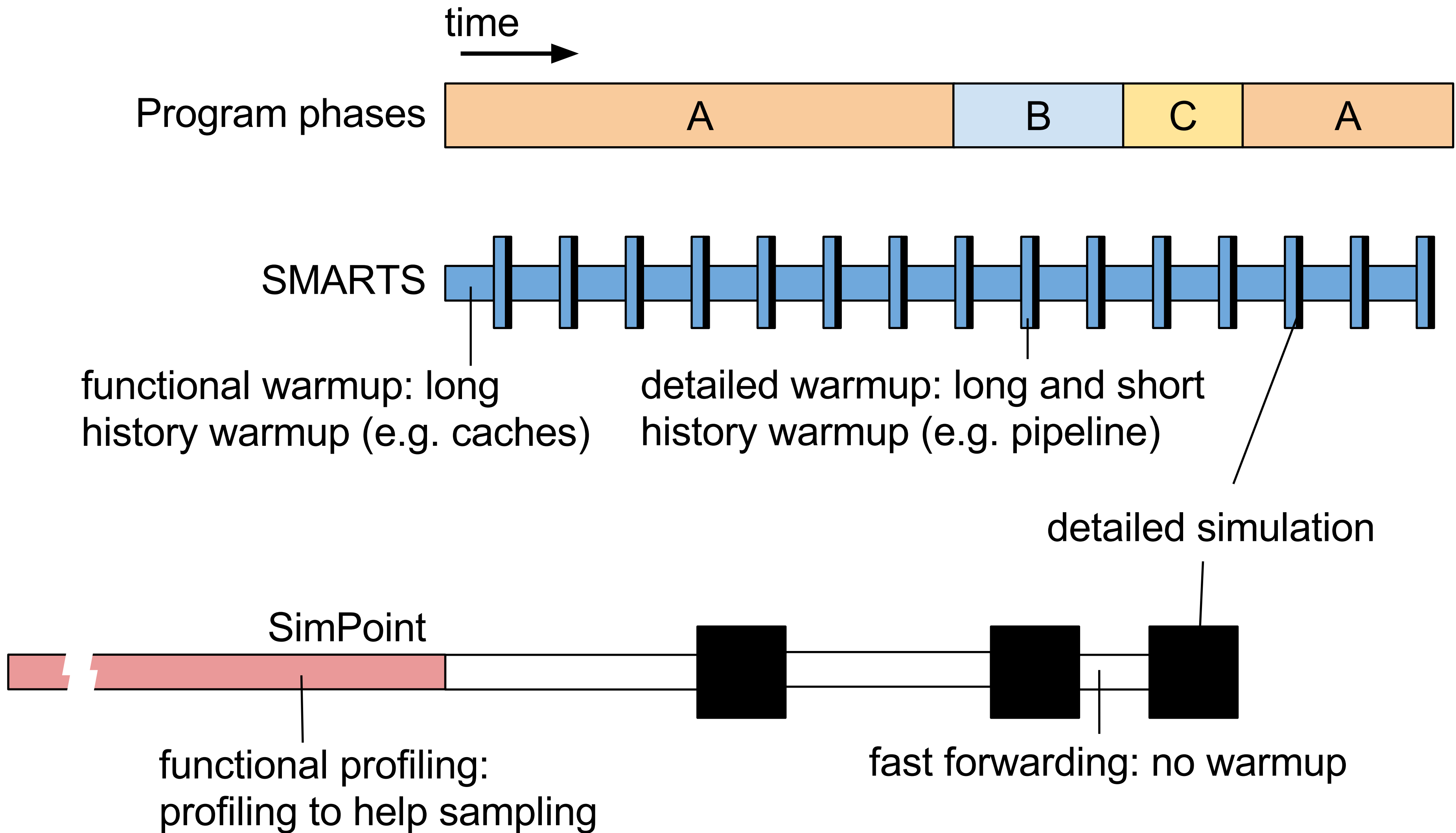
Emerging workloads require re-thinking the entire compute stack.
Existing Fast and Accurate simulation methodologies exploit the fact that portions of the stack are “locked”.

Agile: quickly make changes in any layer of the stack without penalties

Simulation and Evaluation Methodologies

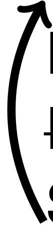
	Fast	Accurate	Agile
Native execution	yes	no	yes*
Interpreter based ISS	no	no	yes
Dynamic Binary Translation based ISS	yes	no	yes
~~Detailed simulation~~	~~no~~	~~yes~~	~~yes~~

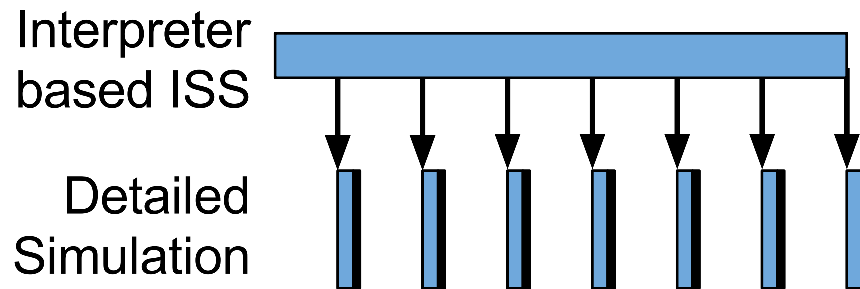
Running detailed simulation to completion for realistic workloads is unfeasible (**over a year of simulation time!**)



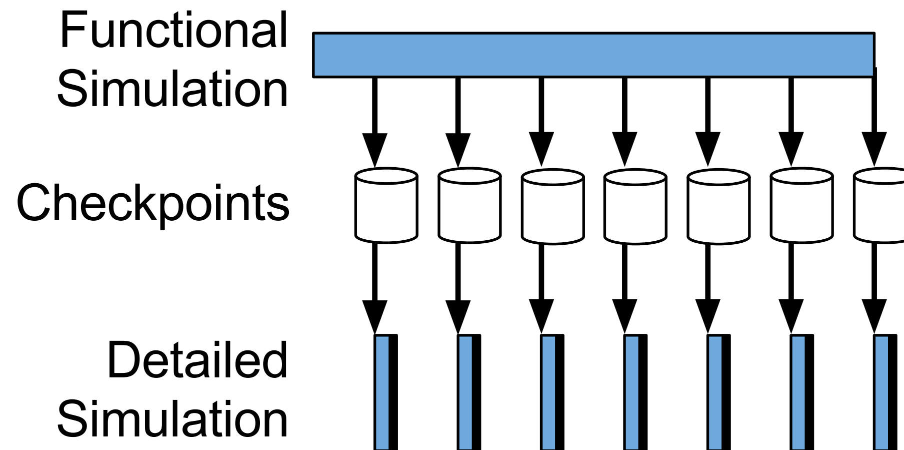
Over 99.9% of detailed simulation can be eliminated!

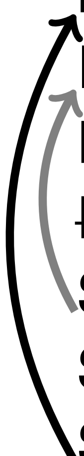
	Fast	Accurate	Agile
Native execution	yes	no	yes*
Interpreter based ISS	no	no	yes
Dynamic Binary Translation based ISS	yes	no	yes
~~Detailed simulation~~	~~no~~	~~yes~~	~~yes~~
Sampling: Fast-forwarding	no	yes	yes



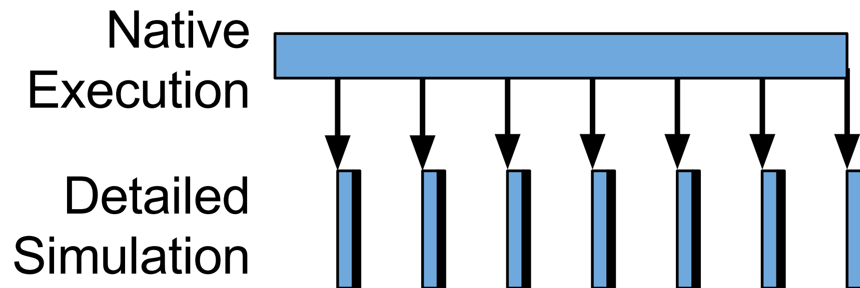


	Fast	Accurate	Agile
Native execution	yes	no	yes*
Interpreter based ISS	no	no	yes
Dynamic Binary Translation based ISS	yes	no	yes
~~Detailed simulation~~	~~no~~	~~yes~~	~~yes~~
Sampling: Fast-forwarding	no	yes	yes
Sampling: Checkpointing	yes	yes	no



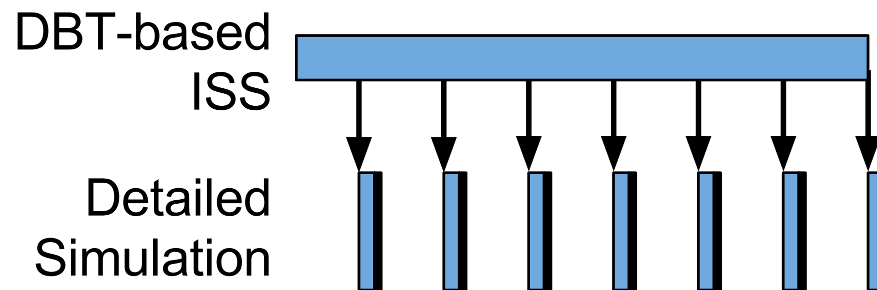


	Fast	Accurate	Agile
Native execution	yes	no	yes*
Interpreter based ISS	no	no	yes
Dynamic Binary Translation based ISS	yes	no	yes
~~Detailed simulation~~	~~no~~	~~yes~~	~~yes~~
Sampling: Fast-forwarding	no	yes	yes
Sampling: Checkpointing	yes	yes	no
Sampling: Native-on-native execution	yes	yes	yes*

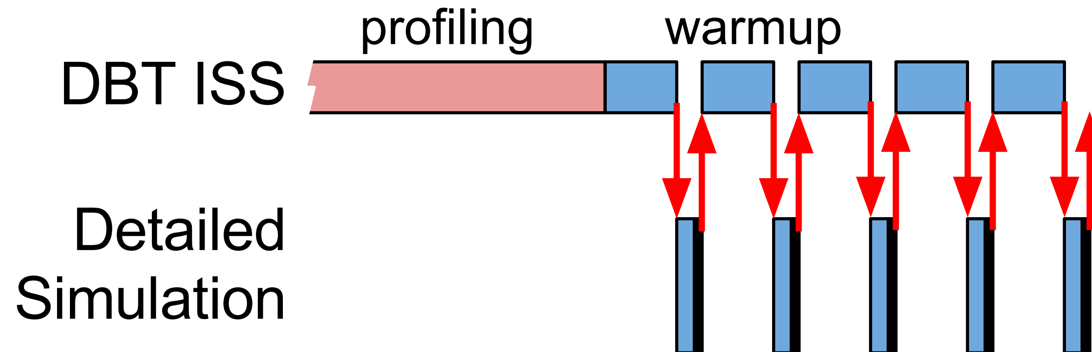


*Not fully agile because the target ISA needs to match the host ISA

	Fast	Accurate	Agile
Native execution	yes	no	yes*
Interpreter based ISS	no	no	yes
Dynamic Binary Translation based ISS	yes	no	yes
~~Detailed simulation~~	~~no~~	~~yes~~	~~yes~~
Sampling: Fast-forwarding	no	yes	yes
Sampling: Checkpointing	yes	yes	no
Sampling: Native-on-native execution	yes	yes	yes*
Goal	**yes**	**yes**	**yes**

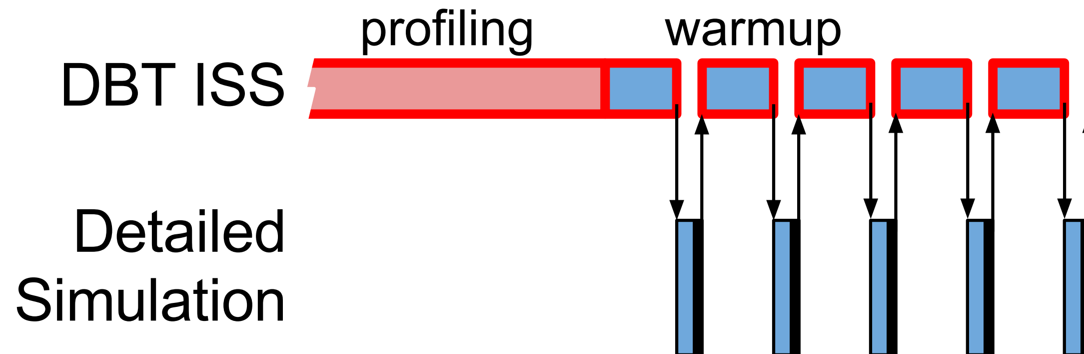


Challenges of using DBT ISS for Fast Forwarding



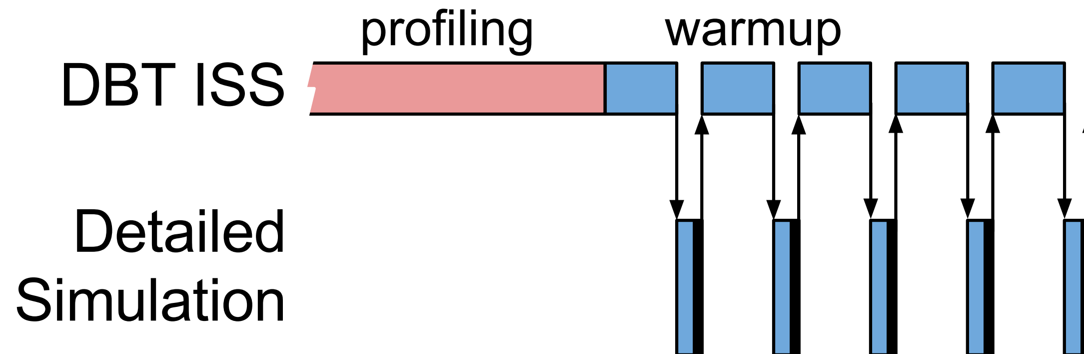
- Problem: Large architectural state transfers may hurt performance
  - Solution: JIT-Assisted Fast Forward Embedding (JIT-FFE). Embed DBT ISS in the detailed simulator which allows fast- and zero-copy architectural state transfer

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
  - Solution: JIT-Assisted Fast Forward Instrumentation (JIT-FFI). Use Pydgin and RPython's meta-tracing JIT to easily add JIT-compiled instrumentation.

Challenges of using DBT ISS for Fast Forwarding



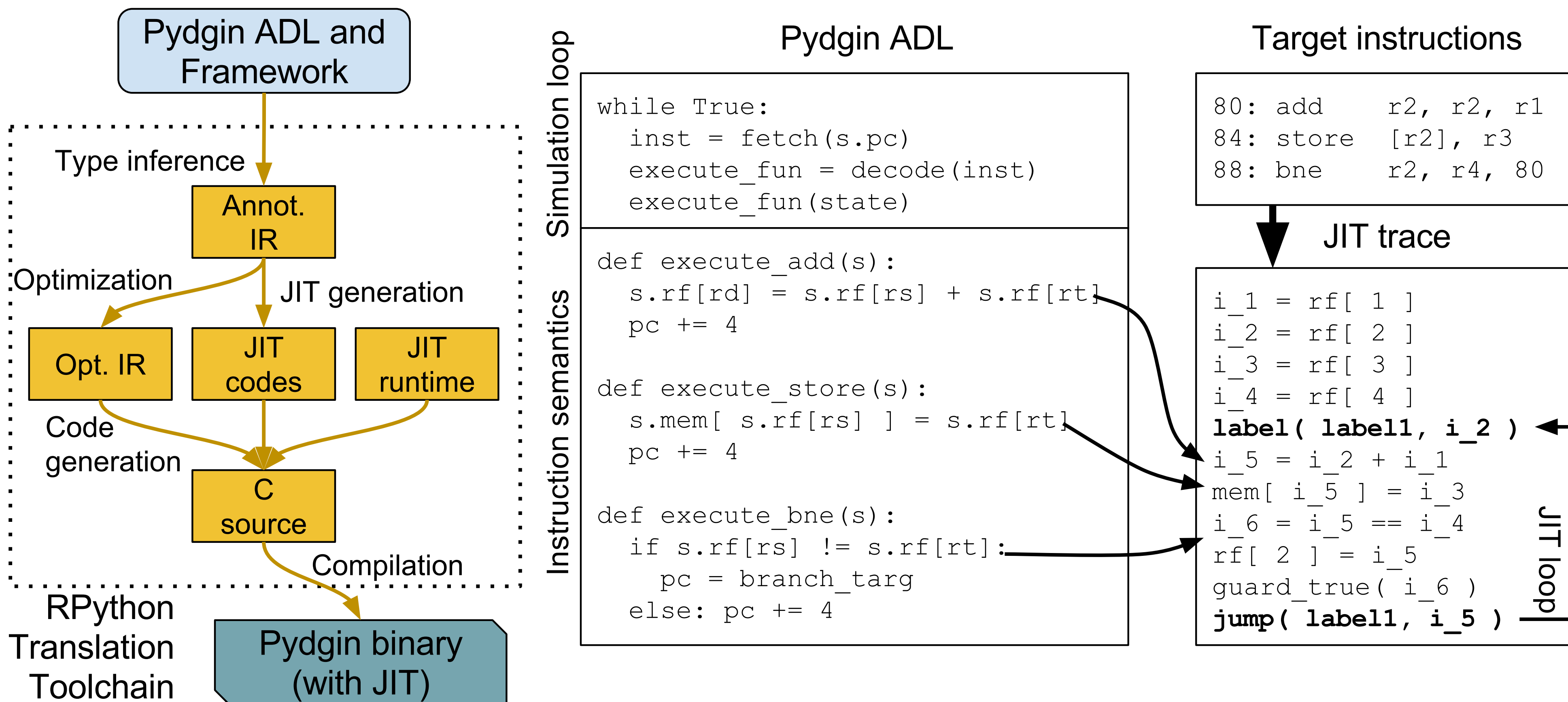
- Problem: Large architectural state transfers may hurt performance
  - Solution: JIT-Assisted Fast Forward Embedding (JIT-FFE). Embed DBT ISS in the detailed simulator which allows fast- and zero-copy architectural state transfer
- Problem: Functional profiling and warmup may hurt performance
  - Solution: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

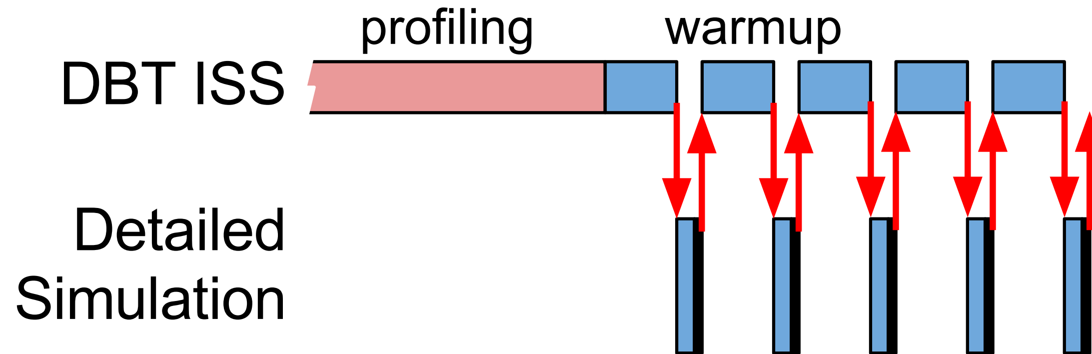
Pydgin Overview

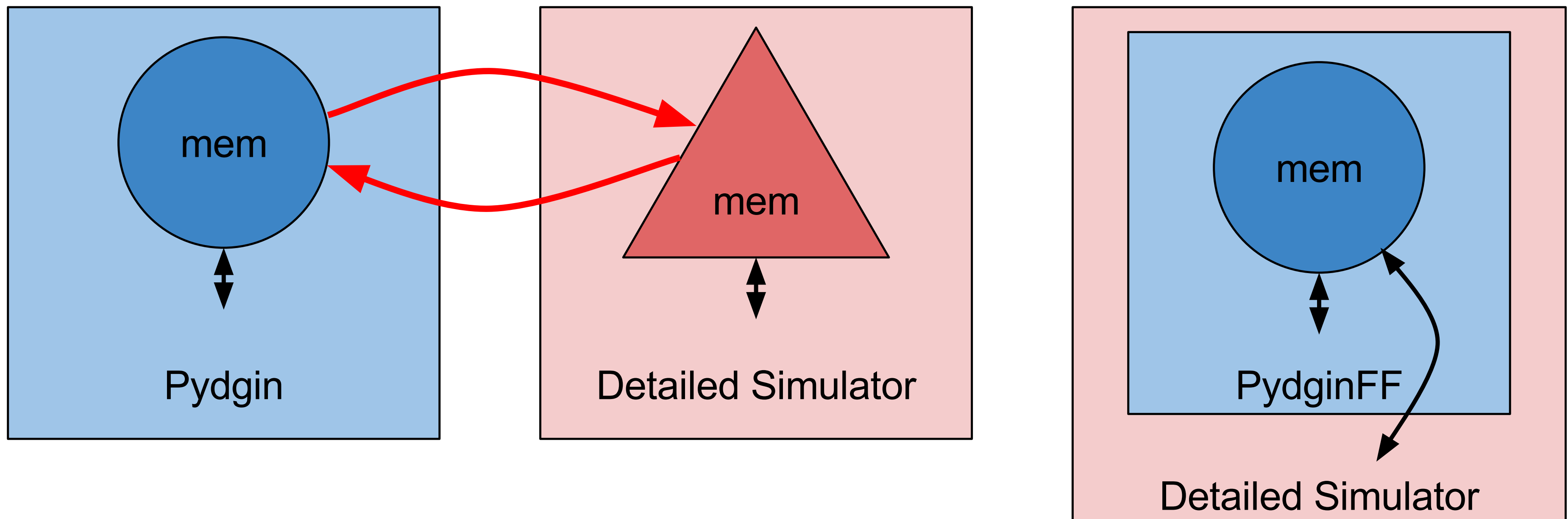


Meta-Tracing JIT: the trace of interpreter interpreting the instructions is JIT compiled

Outline

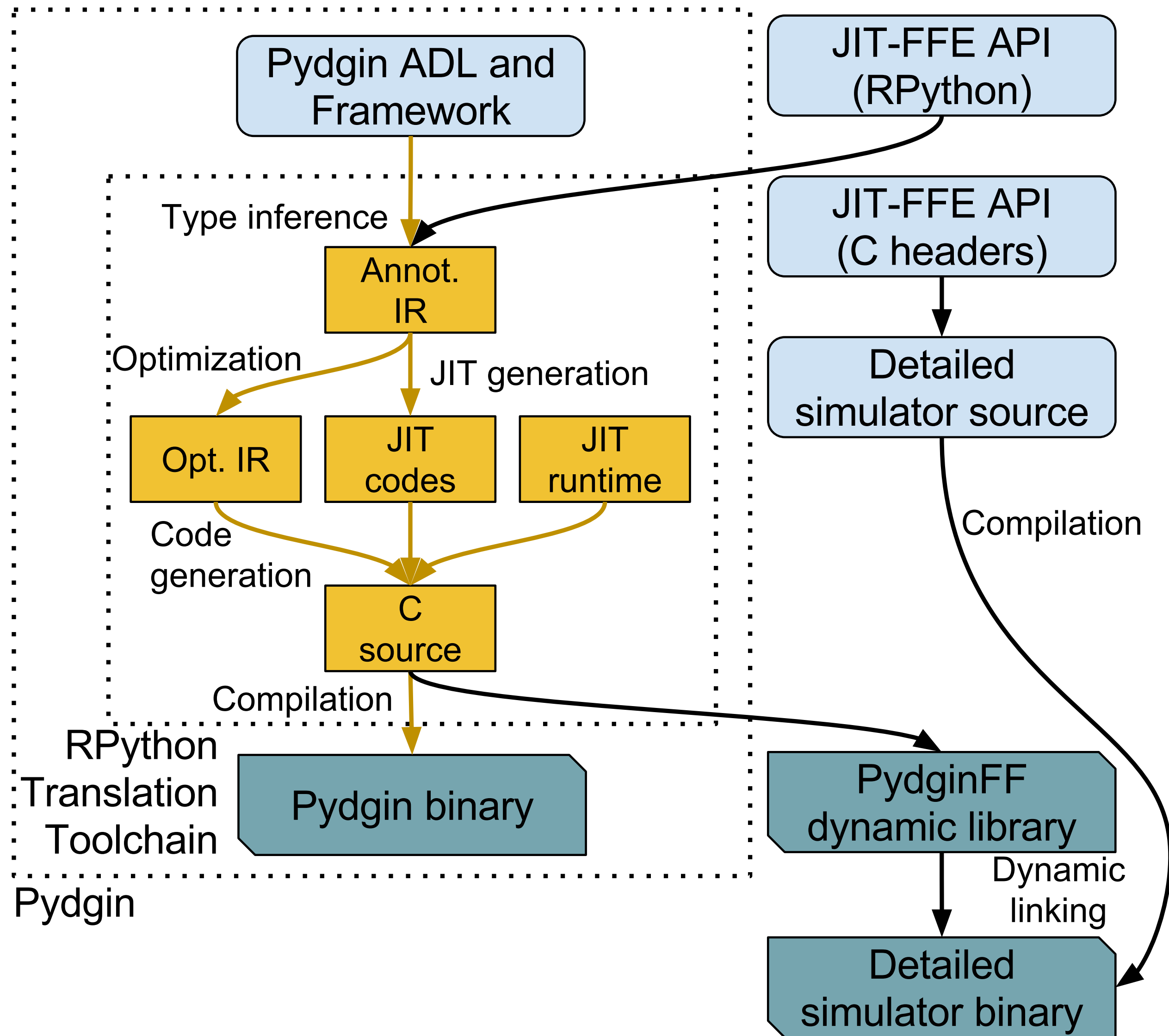
- Motivation
- Pydgin Overview
- **JIT-Assisted Fast-Forward Embedding**
- JIT-Assisted Fast-Forward Instrumentation
- Results





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

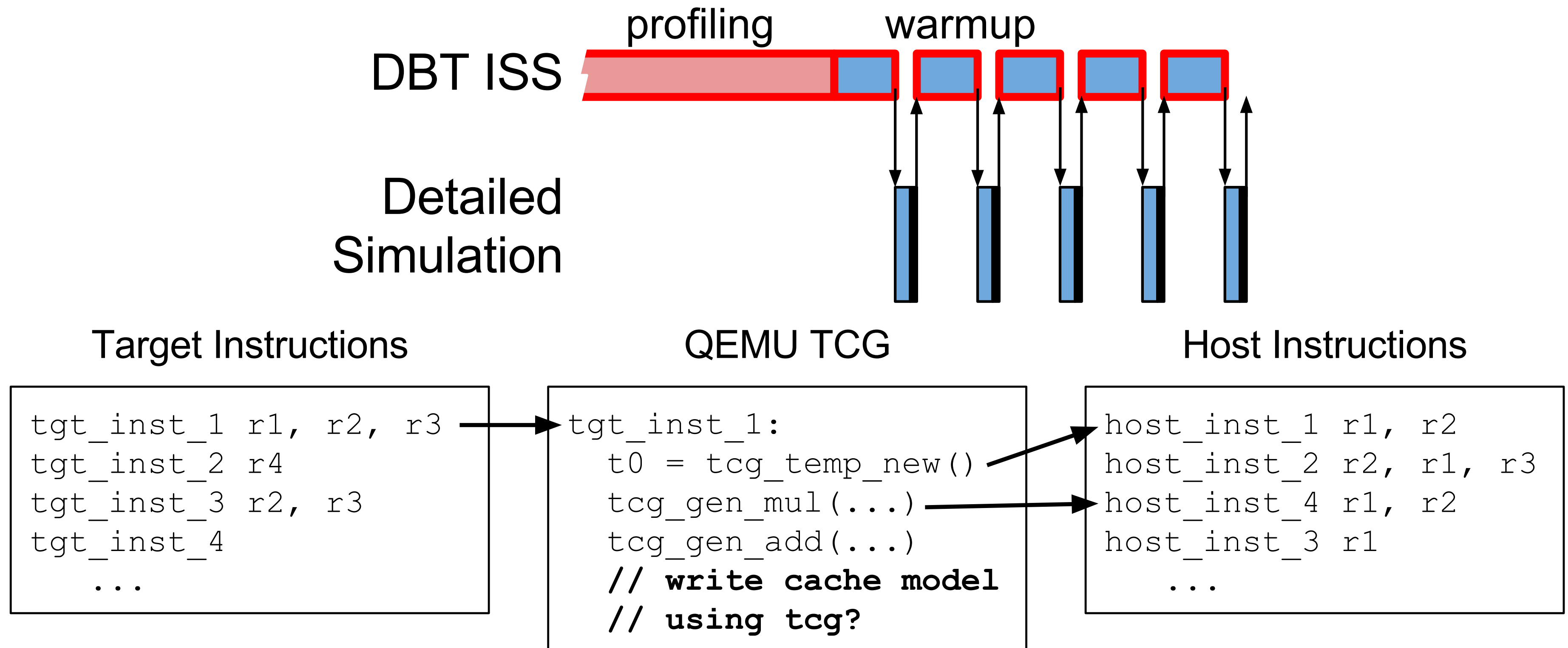
JIT-Assisted Fast-Forward Embedding



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)

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


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

Instrumentation

```
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( label1, i_5 )
```

JIT loop

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

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
```

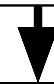
Instrumentation

```
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 loop

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
```

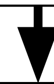
Instrumentation

```
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

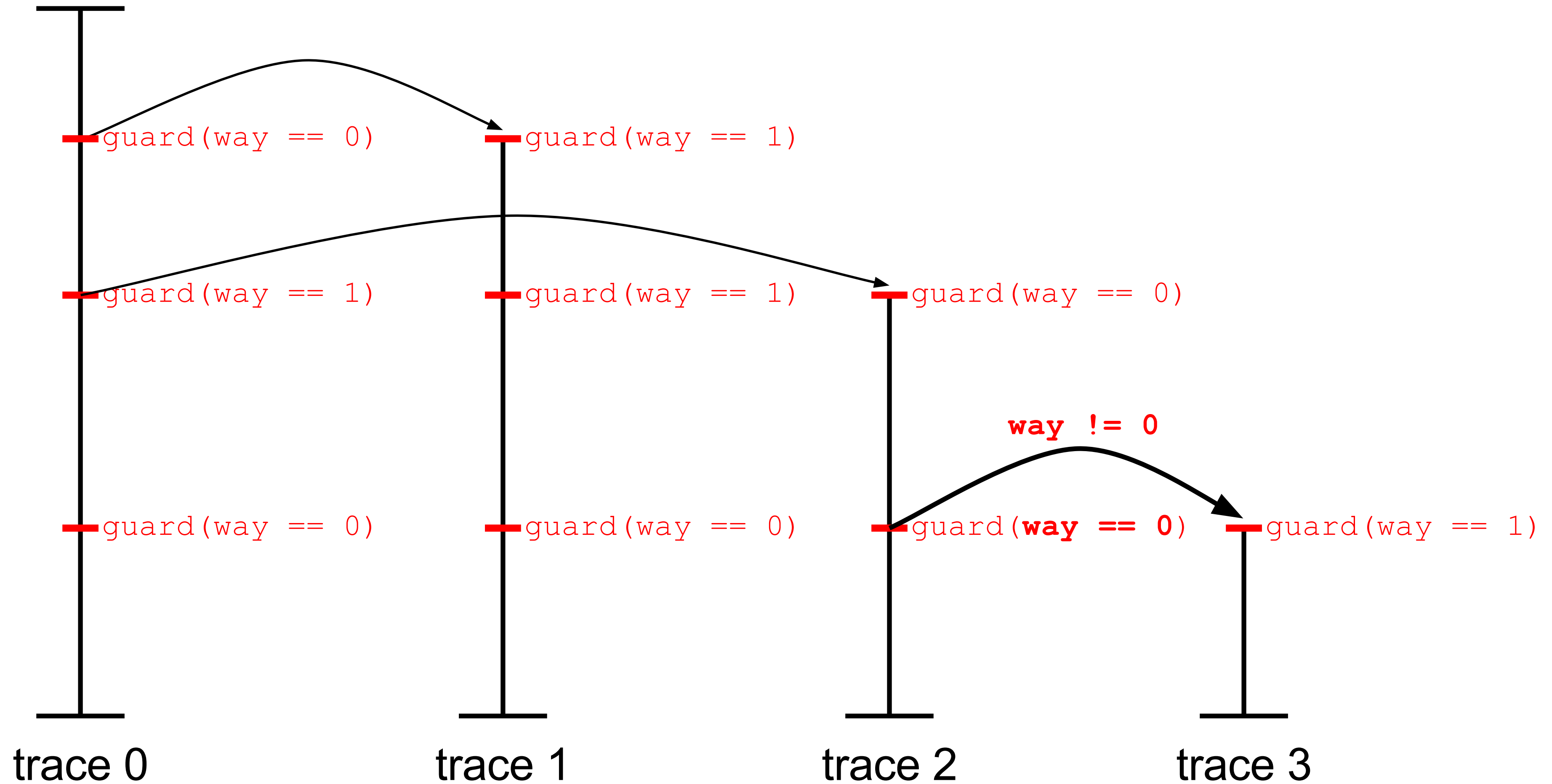
```
i_1 = rf[ 1 ]
i_2 = rf[ 2 ]
i_3 = rf[ 3 ]
i_4 = rf[ 4 ]
label( label1, i_2 )
i_5 = i_2 + i_1
mem[ i_5 ] = i_3
call(instrument_memop)
i_6 = i_5 == i_4
rf[ 2 ] = i_5
num_insts += 3
guard_true( i_6 )
jump( label1, i_5 )
```

JIT loop

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

JIT-FFI-outlined instrumentation.

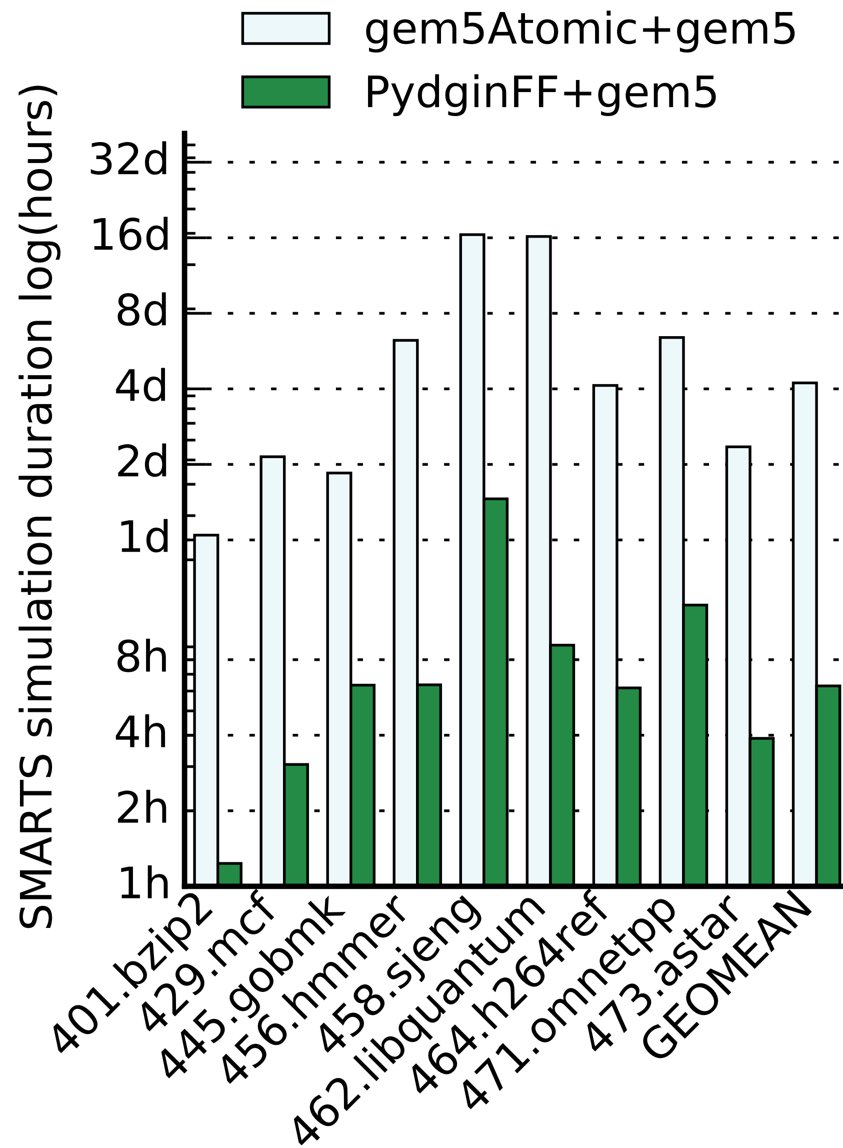
Inlined Instrumentation with Data-Dependent Control-Flow

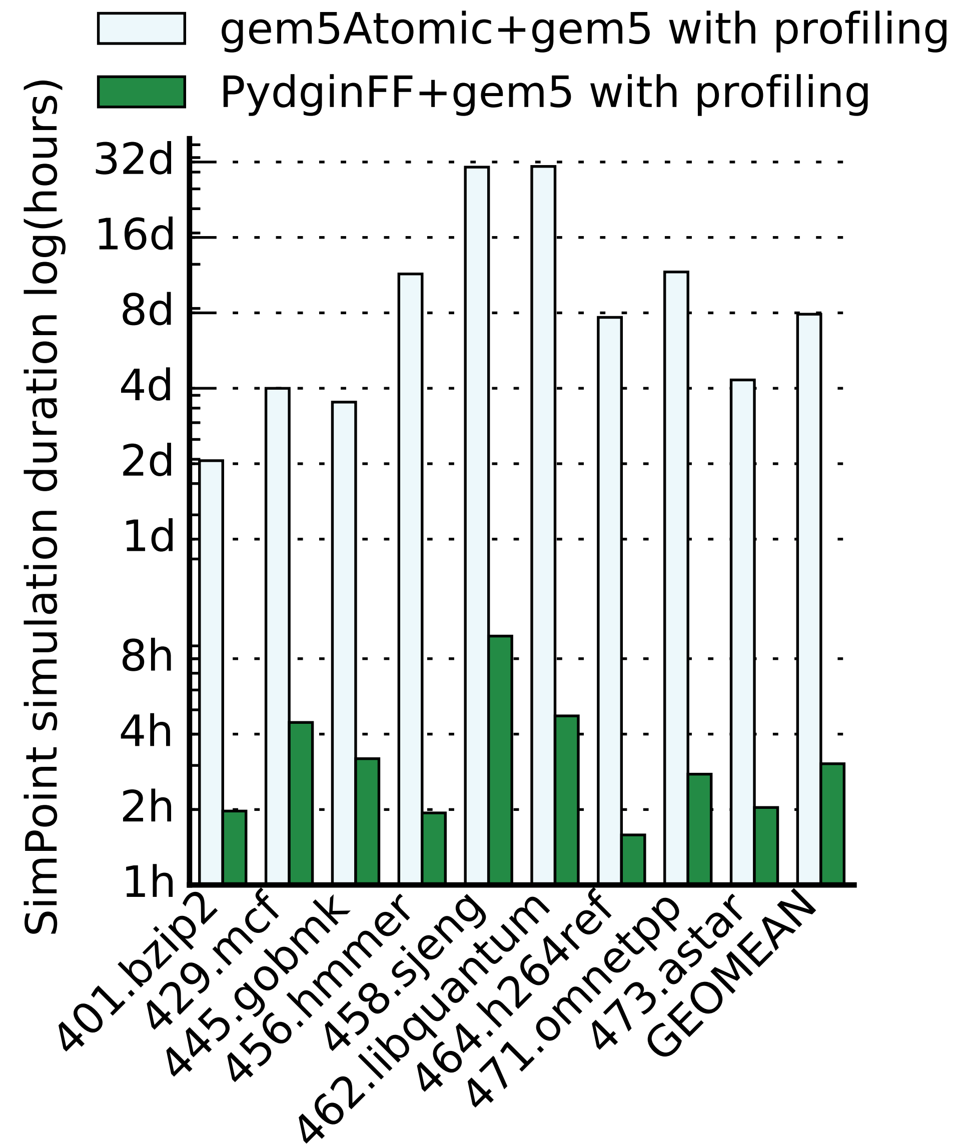


Outline

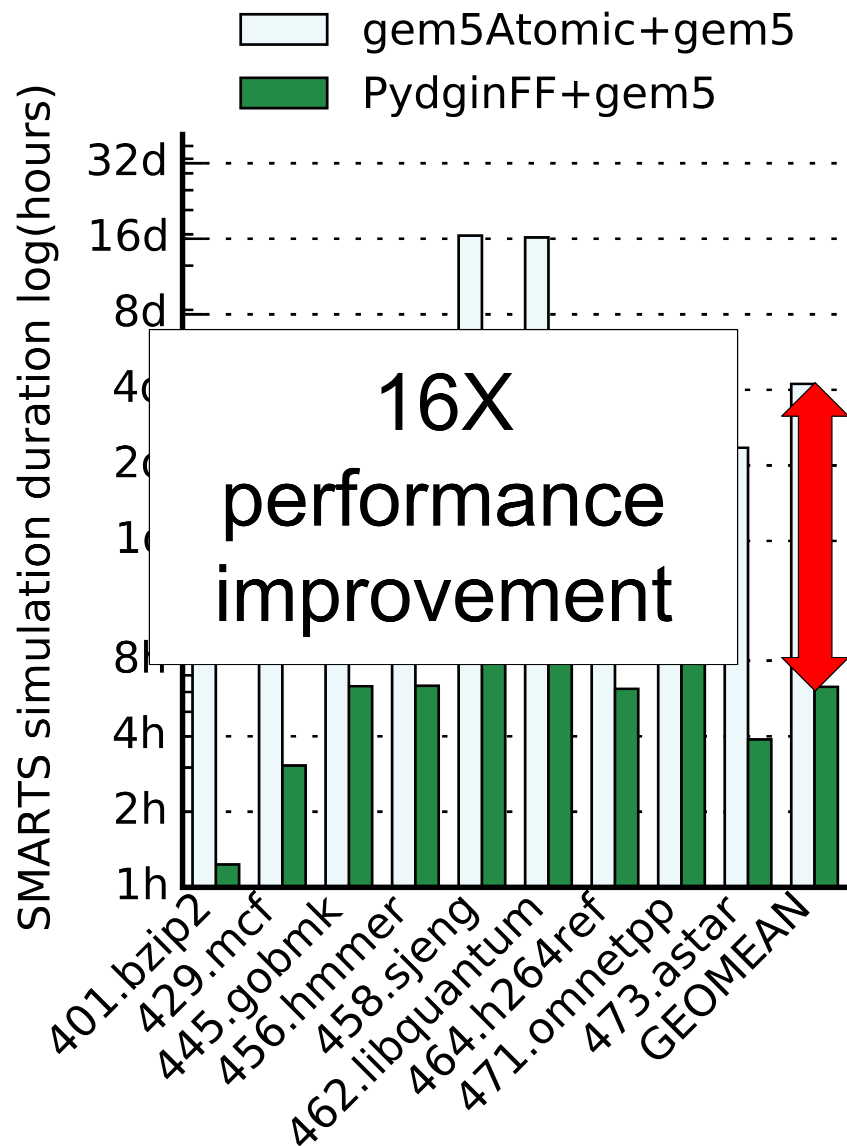
- Motivation
- Pydgin Overview
- JIT-Assisted Fast-Forward Embedding
- JIT-Assisted Fast-Forward Instrumentation
- **Results**

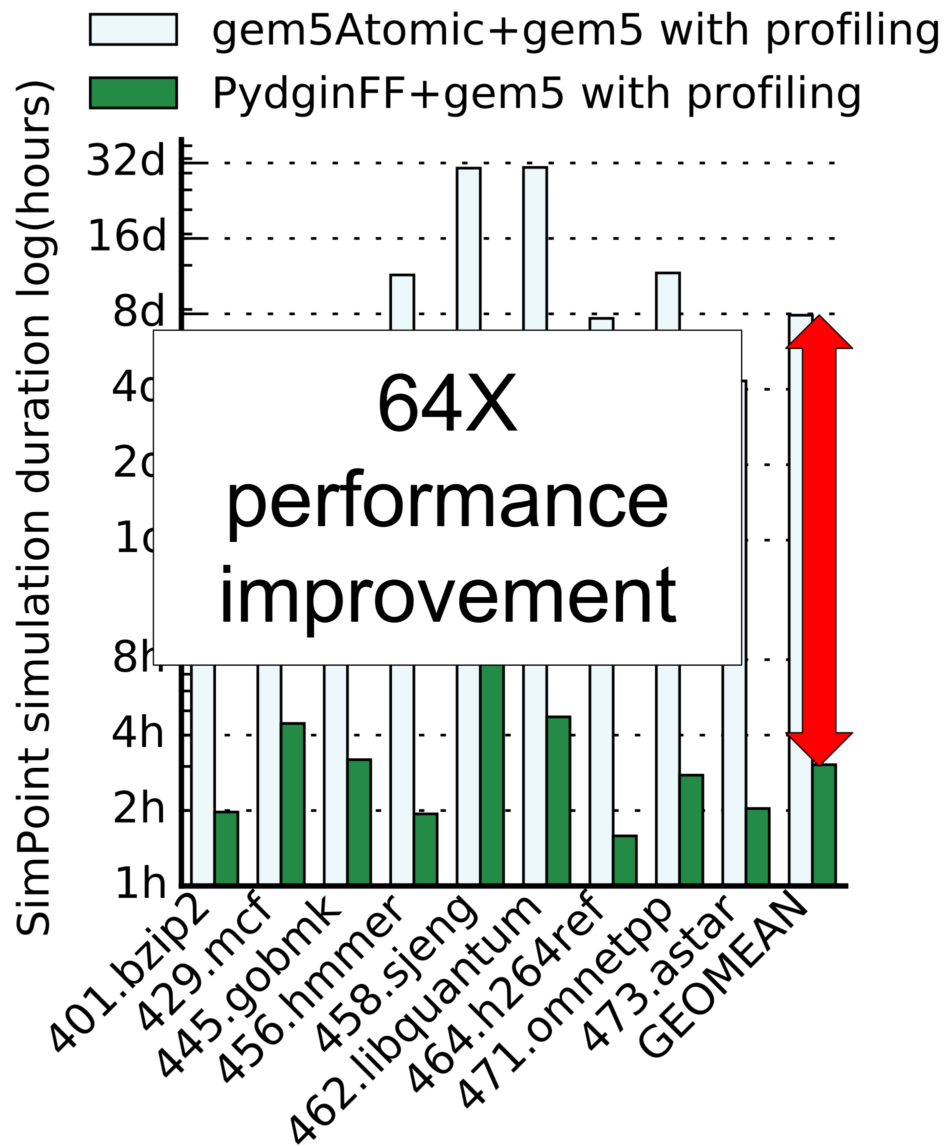
SMARTS and SimPoint Results



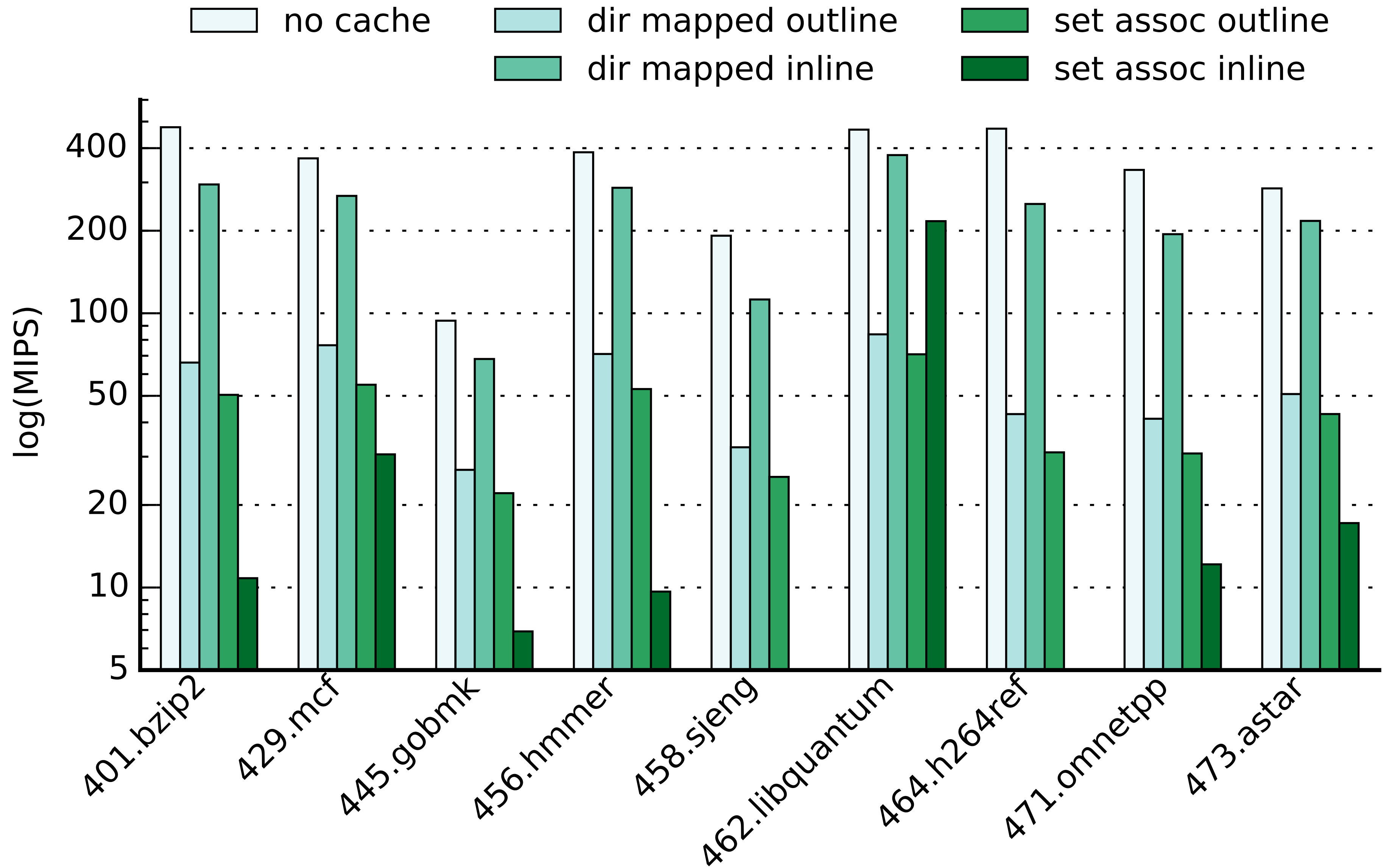


SMARTS and SimPoint Results





JIT-FFI Functional Warmup Case Study



Conclusion

- 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.