Implementing a Bytecode Optimizer for PretVM

Shaokai Jerry Lin

How to have guarantees about the maximum latency between the camera capturing a frame and a display showing the frame to the surgeons?

Designing CPS is an ever more complex problem.

Timing
(Execution Time, Precision)

Networking

Memory Footprint

Multi-core

Performance,
Dependability,
Heterogeneity

Energy Consumption

Debuggability

Predictability

Deadline Violation Check

A New Coordination Language for CPS

> “Lingua Franca is a polyglot, declarative, coordination language for real-time, concurrent (and distributed) systems.”

A Classic CPS Example

```
reactor Sensor {
    output out:int
    timer t(0,100 msec)
    state cnt:int(0)

    reaction(t) -> out {= /* Imperative C code here */ =}
}
```

Application Logic

```
reactor Processing {
    input in:int
    output out:int

    reaction(in) -> out {= /* Process measurement */ =}
}
```

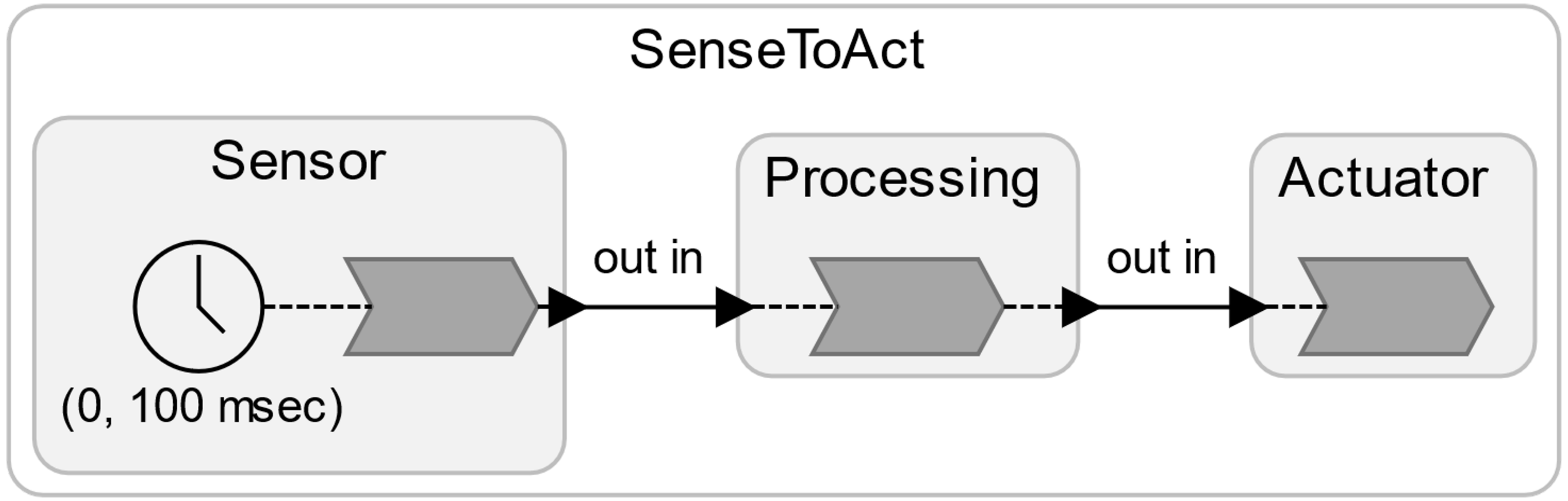
Application Logic

```
reactor Actuator {
    input in:int

    reaction(in) {= /* Drive actuator */ =}
}

```

Application Logic



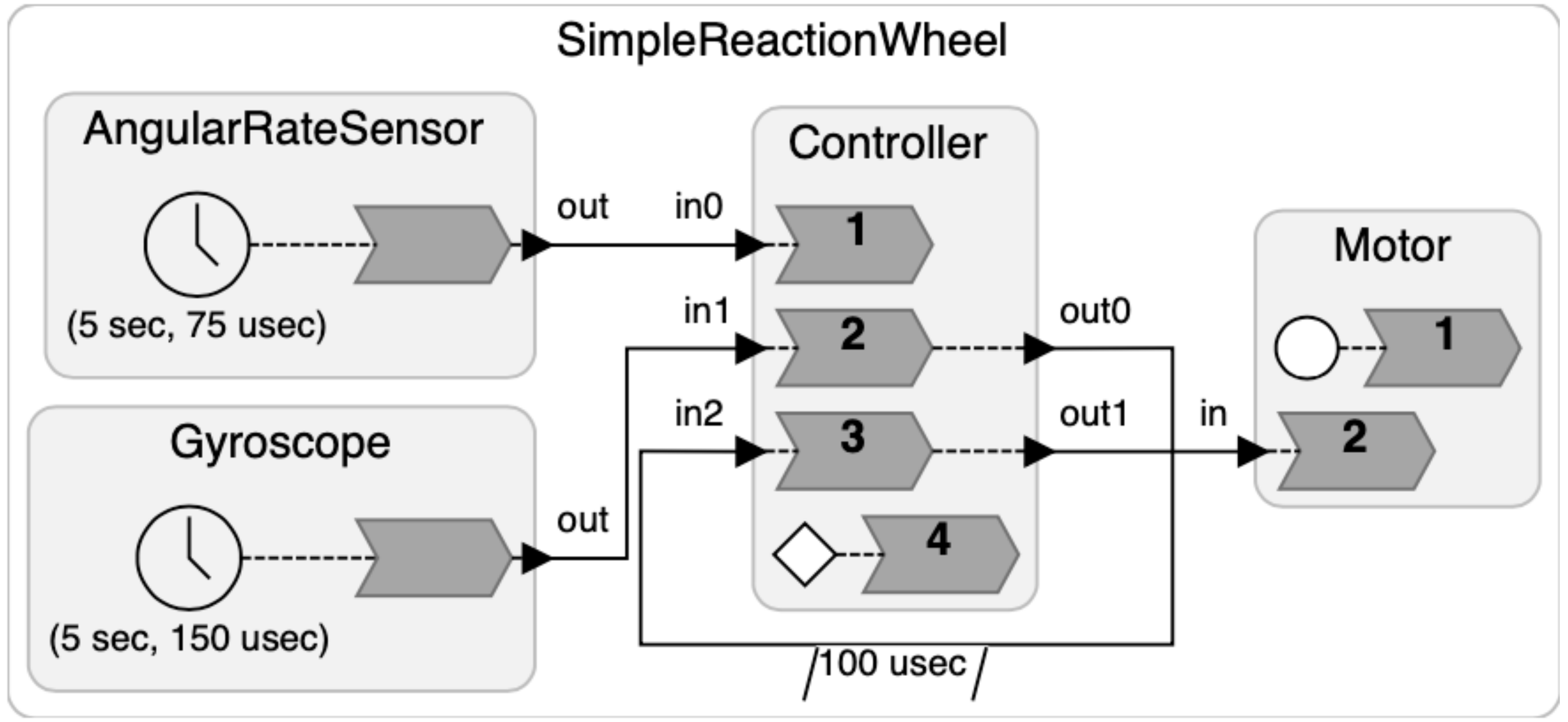
* Thanks to Erling Jellum for creating this slide.

PretVM

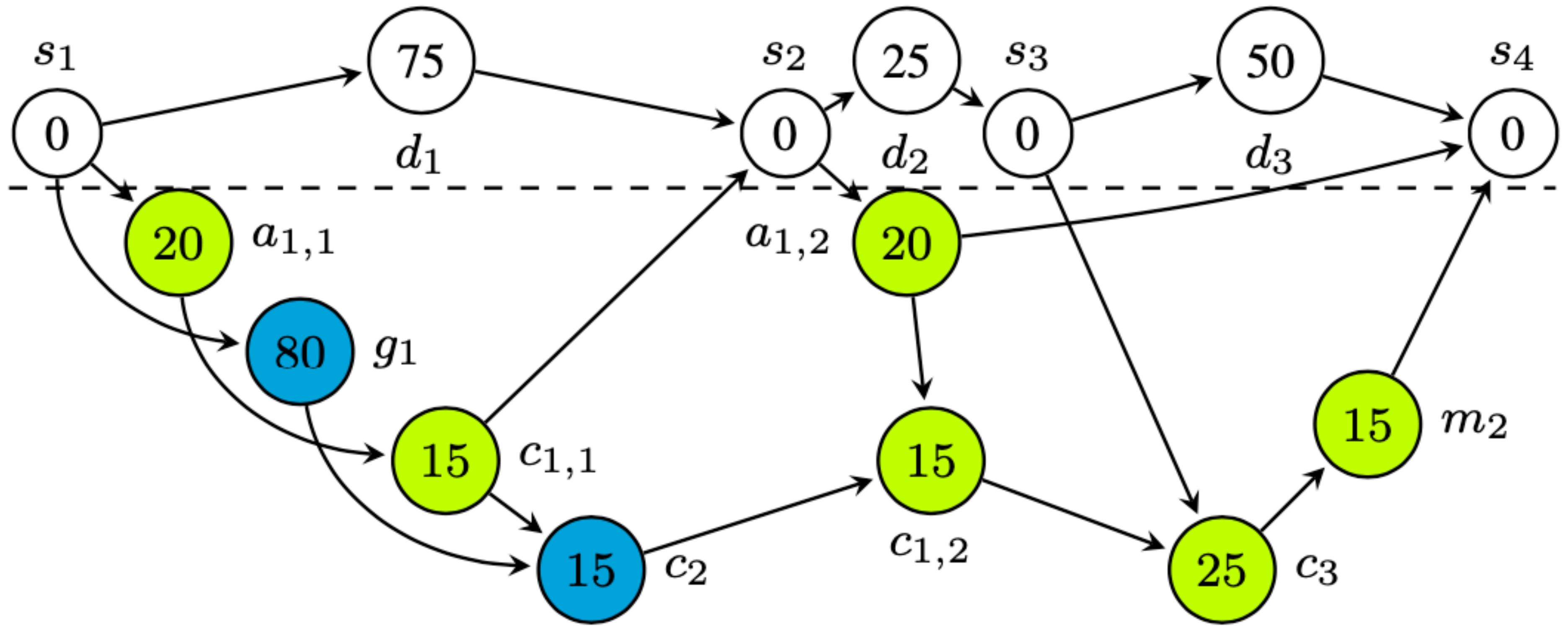
Instruction Set

Instruction	**Semantics**
ADD rs1, rs2, rs3	Add to an integer variable (rs2) by an integer variable (rs3) and store the result in a destination variable (rs1).
ADDI rs1, rs2, rs3	Add to an integer variable (rs2) by an immediate (rs3) and store the result in a destination variable (rs1).
ADV rs1, rs2, rs3	ADVance the logical time of a reactor (rs1) to a base time register (rs2) + an increment register (rs3).
ADVI rs1, rs2, rs3	Advance the logical time of a reactor (rs1) to a base time register (rs2) + an immediate value (rs3).
BEQ rs1, rs2, rs3	Take the branch (rs3) if rs1 is equal to rs2.
BGE rs1, rs2, rs3	Take the branch (rs3) if rs1 is greater than or equal to rs2.
BLT rs1, rs2, rs3	Take the branch (rs3) if rs1 is less than rs2.
BNE rs1, rs2, rs3	Take the branch (rs3) if rs1 is not equal to rs2.
DU rs1, rs2	Delay Until a physical timepoint (rs1) plus an offset (rs2) is reached.
EXE rs1	EXEcute a reaction (rs1)
JAL rs1	Store the return address to rs1 and jump to a label (rs2).
JALR rs1, rs2, rs3	Store the return address in destination (rs1) and jump to baseAddr (rs2) + immediate (rs3)
STP	SToP the execution.
WLT rs1, rs2, rs3	Wait until a variable (rs1) owned by a worker (rs2) to be less than a desired value (rs3).
WU rs1, rs2, rs3	Wait Until a variable (rs1) owned by a worker (rs2) to be greater than or equal to a desired value (rs3).

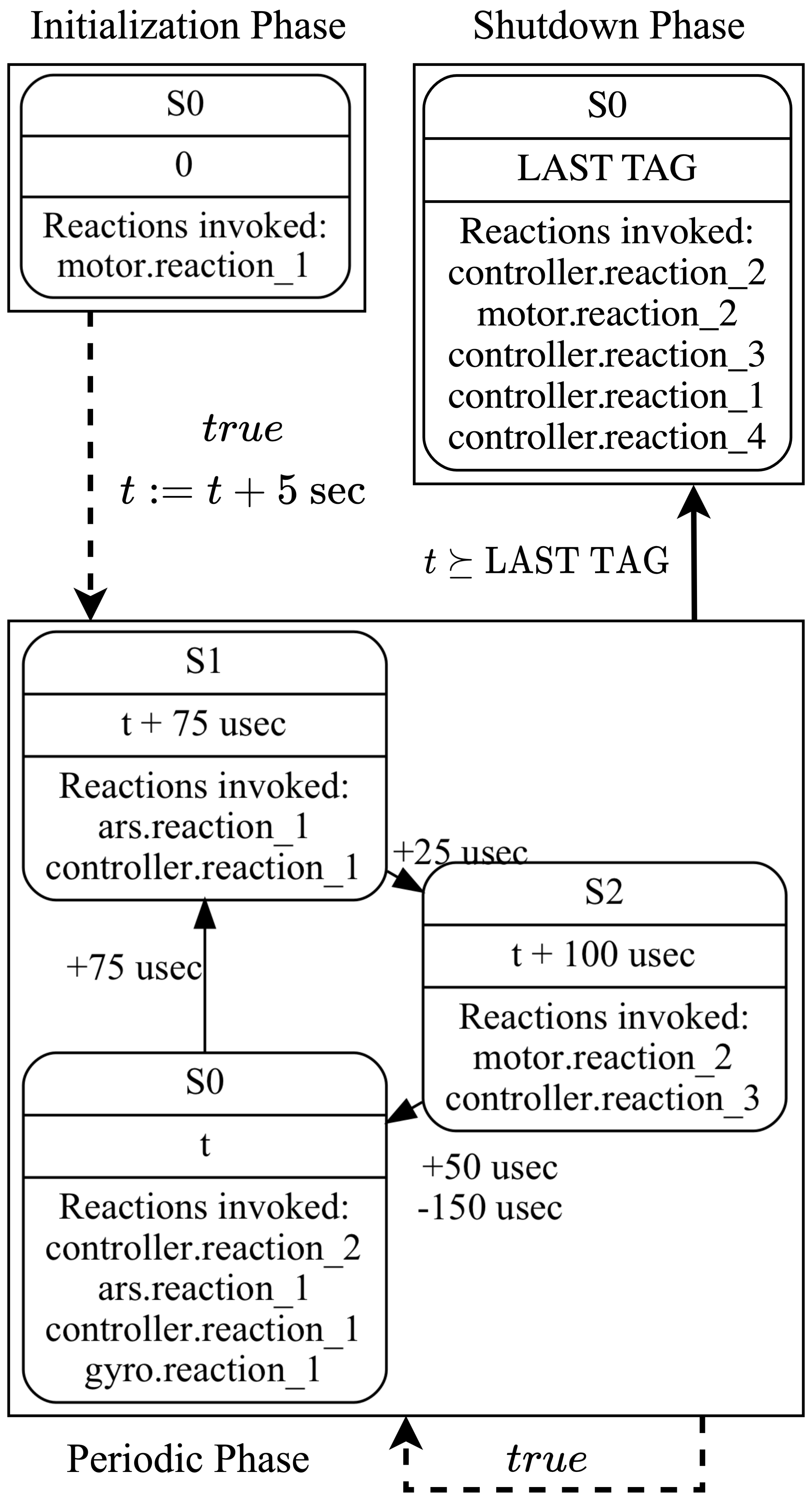
Compiling LF to PretVM Bytecode



Lingua Franca Program

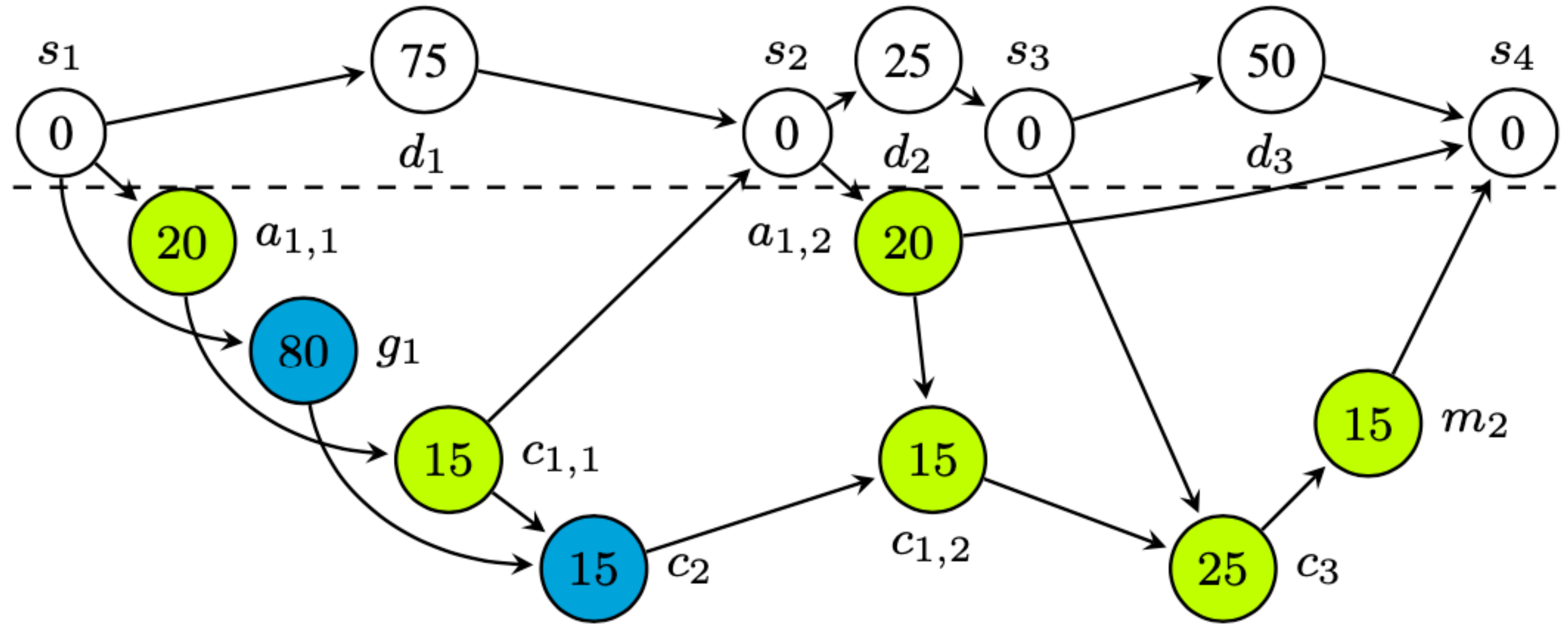


Partitioned DAG



State Space Diagram

Compiling LF to PretVM Bytecode



Partitioned DAG

1) `EXE` g_1
2) `ADDI counter`$_{\text{blue}}$`, counter`$_{\text{blue}}$`,` 1
3) `WU counter`$_{\text{green}}$`,` 2
4) `BEQ in1_is_present,` ***true***
5) `JAL` line_9
6) `EXE` c_2
7) `EXE out0_pre_connection_helper`
8) `EXE in1_post_connection_helper`
9) `ADDI counter`$_{\text{blue}}$`, counter`$_{\text{blue}}$`,` 1
10) `DU time_offset,` $150\mu s$
11) `ADDI offset_inc,` $150\mu s$
12) `JAL return_addr`$_{\text{blue}}$`,` **SYNC_BLOCK**
13) `BGE time_offset, timeout,` **SHUTDOWN**$_{\text{blue}}$
14) `JAL return_addr`$_{\text{blue}}$`,` **PERIODIC**$_{\text{blue}}$

PretVM Bytecode

Opt. 1: Collective Time Advancement

1. Advance reactor 1’s time
2. Advance reactor 2’s time
3. Advance reactor 3’s time

…

1000. Advance reactor 1000’s time

→

1. Advance a shared time

On Raspberry Pi 4B, each line takes ~2 us. 1000 lines could take ~2 ms, which is a lot of time. The optimized code have ~2 us of *constant* overhead.

Peephole Optimization

1. Advance reactor 1’s time
2. Advance reactor 2’s time
3. Advance reactor 3’s time

…

1000. Advance reactor 1000’s time

A set of reactors using a shared time register:

Peephole Optimization

1.
2. Advance a shared time
3. Advance reactor 3’s time

…

1000. Advance reactor 1000’s time

A set of reactors using a shared time register:

- Reactor 1
- Reactor 2

Peephole Optimization

1.
2. Advance a shared time
3. Advance reactor 3’s time

…

1000. Advance reactor 1000’s time

A set of reactors using a shared time register:

- Reactor 1
- Reactor 2

Peephole Optimization

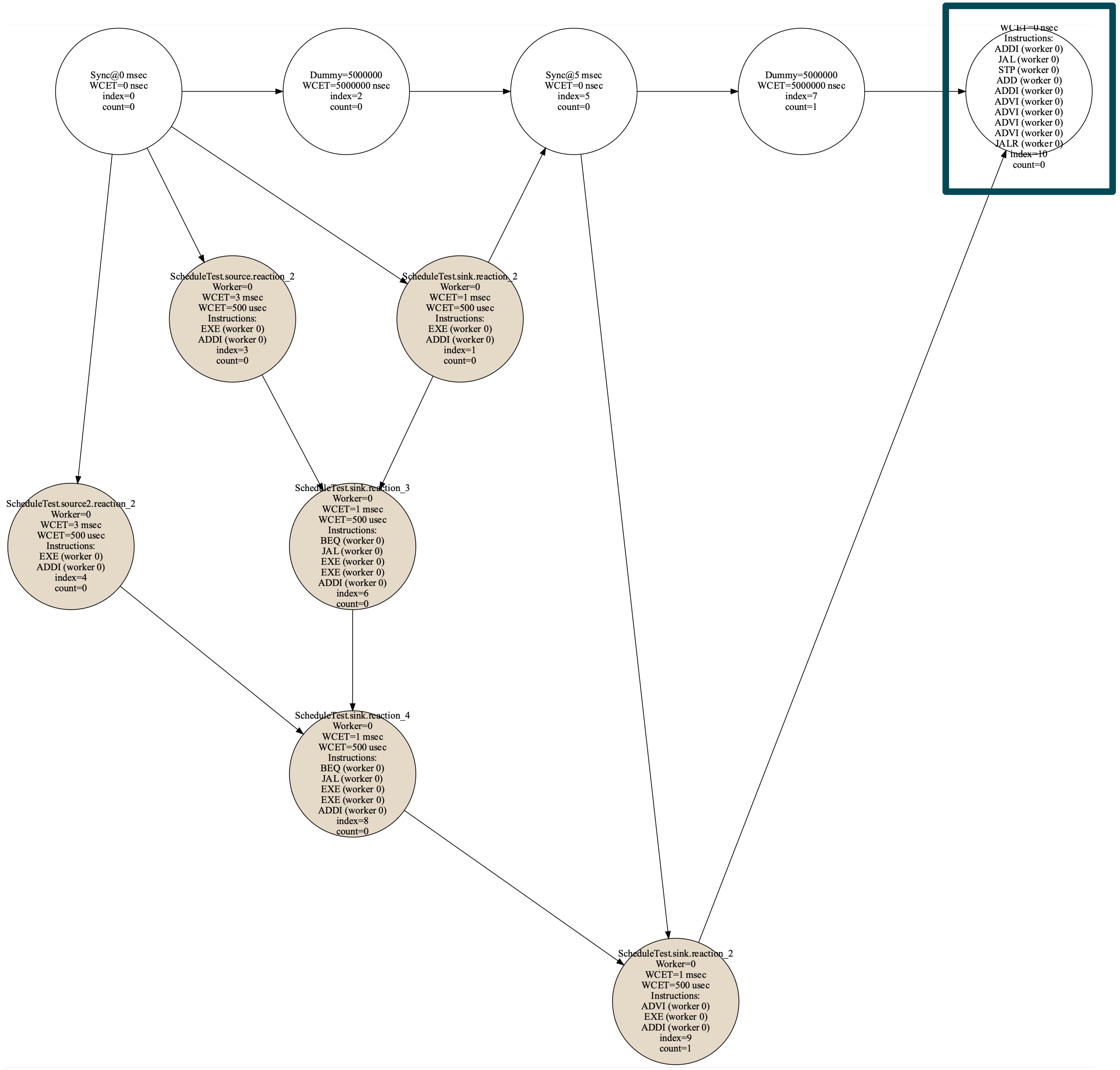
1.
2.
3. Advance a shared time

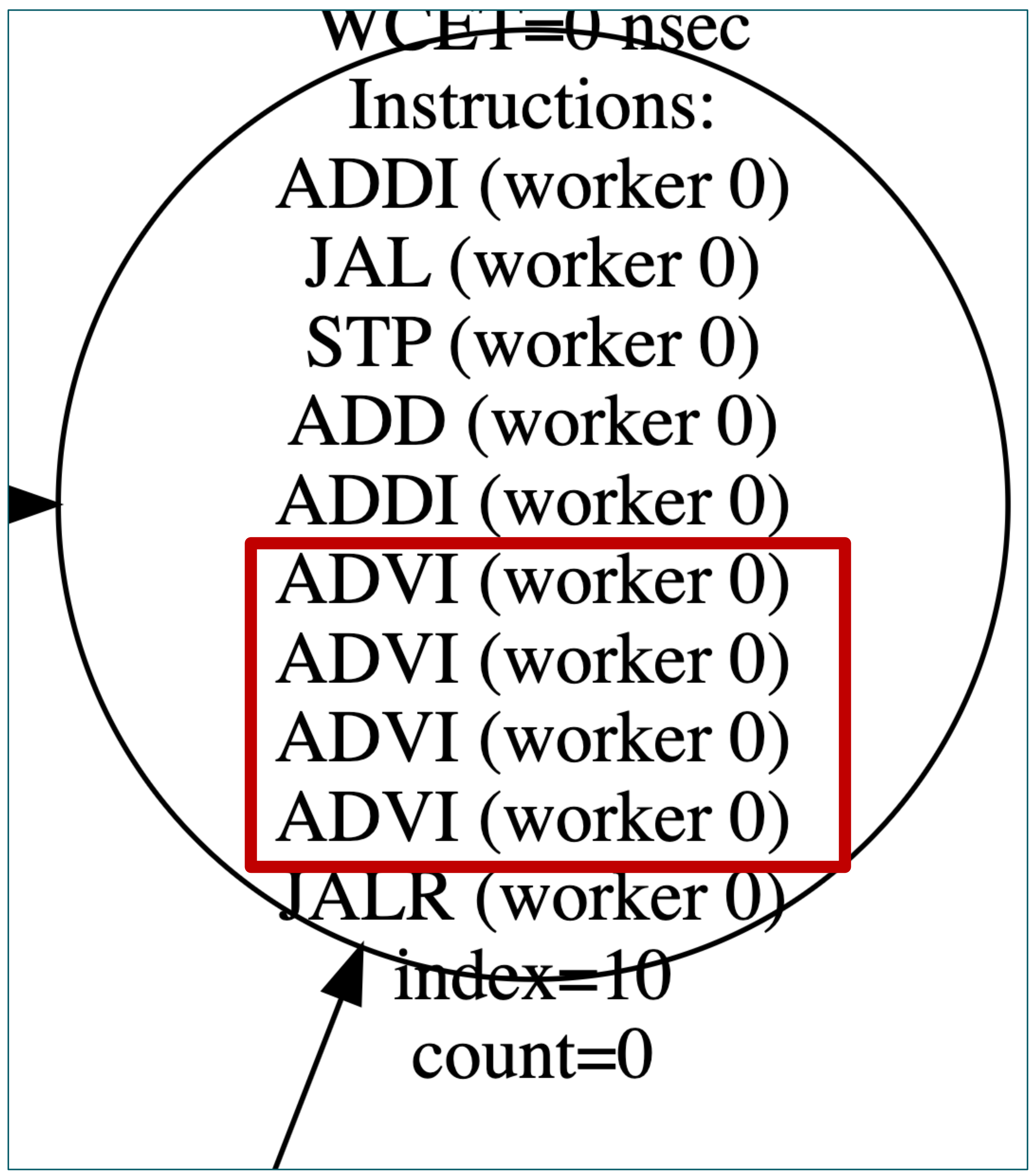
…

1000. Advance reactor 1000’s time

A set of reactors using a shared time register:

- Reactor 1
- Reactor 2
- Reactor 3


Sync@0 msec
WCET=0 nsec
index=0
count=0
Dummy=5000000
WCET=5000000 nsec
index=2
count=0
Sync@5 msec
WCET=0 nsec
index=5
count=0
Dummy=5000000
WCET=5000000 nsec
index=7
count=1
WCET=0 nsec
Instructions:
ADDI (worker 0)
JAL (worker 0)
STP (worker 0)
ADD (worker 0)
ADDI (worker 0)
ADVI (worker 0)
ADVI (worker 0)
ADVI (worker 0)
ADVI (worker 0)
JALR (worker 0)
index=10
count=0
ScheduleTest.source.reaction_2
Worker=0
WCET=3 msec
WCET=500 usec
Instructions:
EXE (worker 0)
ADDI (worker 0)
index=3
count=0
ScheduleTest.sink.reaction_2
Worker=0
WCET=1 msec
WCET=500 usec
Instructions:
EXE (worker 0)
ADDI (worker 0)
index=1
count=0
ScheduleTest.source2.reaction_2
Worker=0
WCET=3 msec
WCET=500 usec
Instructions:
EXE (worker 0)
ADDI (worker 0)
index=4
count=0
ScheduleTest.sink.reaction_3
Worker=0
WCET=1 msec
WCET=500 usec
Instructions:
BEQ (worker 0)
JAL (worker 0)
EXE (worker 0)
EXE (worker 0)
ADDI (worker 0)
index=6
count=0
ScheduleTest.sink.reaction_4
Worker=0
WCET=1 msec
WCET=500 usec
Instructions:
BEQ (worker 0)
JAL (worker 0)
EXE (worker 0)
EXE (worker 0)
ADDI (worker 0)
index=8
count=0
ScheduleTest.sink.reaction_2
Worker=0
WCET=1 msec
WCET=500 usec
Instructions:
ADVI (worker 0)
EXE (worker 0)
ADDI (worker 0)
index=9
count=1


WCET=0 nsec
Instructions:
ADDI (worker 0)
JAL (worker 0)
STP (worker 0)
ADD (worker 0)
ADDI (worker 0)
ADVI (worker 0)
ADVI (worker 0)
ADVI (worker 0)
ADVI (worker 0)
JALR (worker 0)
index=10
count=0

Opt. 2: Procedure Extraction

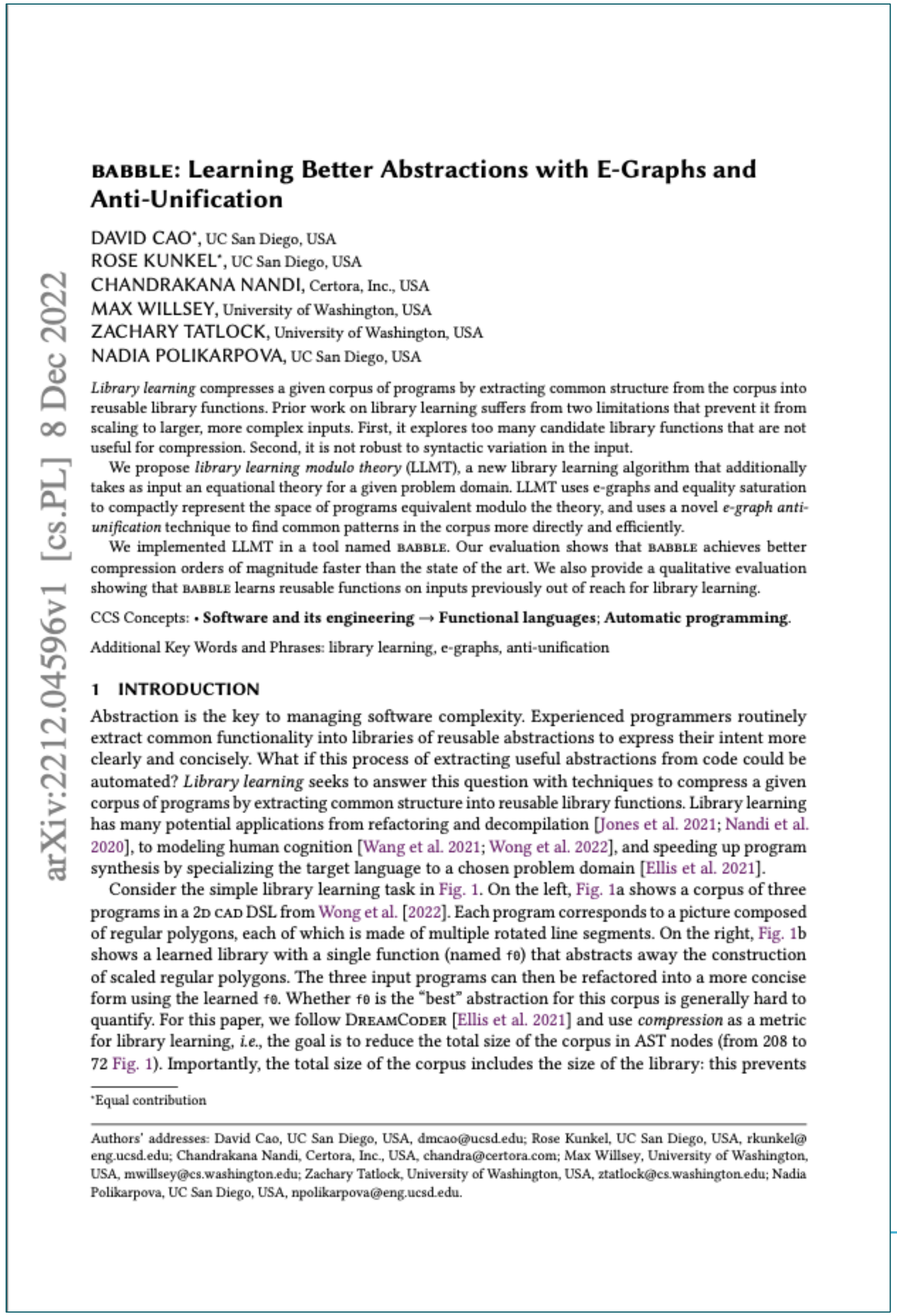
MAIN:
1. <Line A>
2. <Line B>
3. <Line C>
Repeat A-C 100 times...
298. <Line A>
299. <Line B>
300. <Line C>

PROCEDURE:
1. <Line1>
2. <Line2>
3. <Line3>
4. JALR return_addr

Main:
5. JAL PROCEDURE
6. JAL PROCEDURE
...
104. JAL PROCEDURE

Promoting code reuse!

Opt. 2: Procedure Extraction



arXiv:2212.04596v1 [cs.PL] 8 Dec 2022

BABBLE: Learning Better Abstractions with E-Graphs and Anti-Unification

DAVID CAO*, UC San Diego, USA
ROSE KUNKEL*, UC San Diego, USA
CHANDRAKANA NANDI, Certora, Inc., USA
MAX WILLSEY, University of Washington, USA
ZACHARY TATLOCK, University of Washington, USA
NADIA POLIKARPOVA, UC San Diego, USA

Library learning compresses a given corpus of programs by extracting common structure from the corpus into reusable library functions. Prior work on library learning suffers from two limitations that prevent it from scaling to larger, more complex inputs. First, it explores too many candidate library functions that are not useful for compression. Second, it is not robust to syntactic variation in the input.

We propose *library learning modulo theory* (LLMT), a new library learning algorithm that additionally takes as input an equational theory for a given problem domain. LLMT uses e-graphs and equality saturation to compactly represent the space of programs equivalent modulo the theory, and uses a novel *e-graph anti-unification* technique to find common patterns in the corpus more directly and efficiently.

We implemented LLMT in a tool named BABBLE. Our evaluation shows that BABBLE achieves better compression orders of magnitude faster than the state of the art. We also provide a qualitative evaluation showing that BABBLE learns reusable functions on inputs previously out of reach for library learning.

CCS Concepts: • **Software and its engineering → Functional languages**; **Automatic programming**.

Additional Key Words and Phrases: library learning, e-graphs, anti-unification

1 INTRODUCTION

Abstraction is the key to managing software complexity. Experienced programmers routinely extract common functionality into libraries of reusable abstractions to express their intent more clearly and concisely. What if this process of extracting useful abstractions from code could be automated? *Library learning* seeks to answer this question with techniques to compress a given corpus of programs by extracting common structure into reusable library functions. Library learning has many potential applications from refactoring and decompilation [Jones et al. 2021; Nandi et al. 2020], to modeling human cognition [Wang et al. 2021; Wong et al. 2022], and speeding up program synthesis by specializing the target language to a chosen problem domain [Ellis et al. 2021].

Consider the simple library learning task in Fig. 1. On the left, Fig. 1a shows a corpus of three programs in a 2D CAD DSL from Wong et al. [2022]. Each program corresponds to a picture composed of regular polygons, each of which is made of multiple rotated line segments. On the right, Fig. 1b shows a learned library with a single function (named `f0`) that abstracts away the construction of scaled regular polygons. The three input programs can then be refactored into a more concise form using the learned `f0`. Whether `f0` is the "best" abstraction for this corpus is generally hard to quantify. For this paper, we follow DREAMCODER [Ellis et al. 2021] and use *compression* as a metric for library learning, *i.e.*, the goal is to reduce the total size of the corpus in AST nodes (from 208 to 72 Fig. 1). Importantly, the total size of the corpus includes the size of the library: this prevents

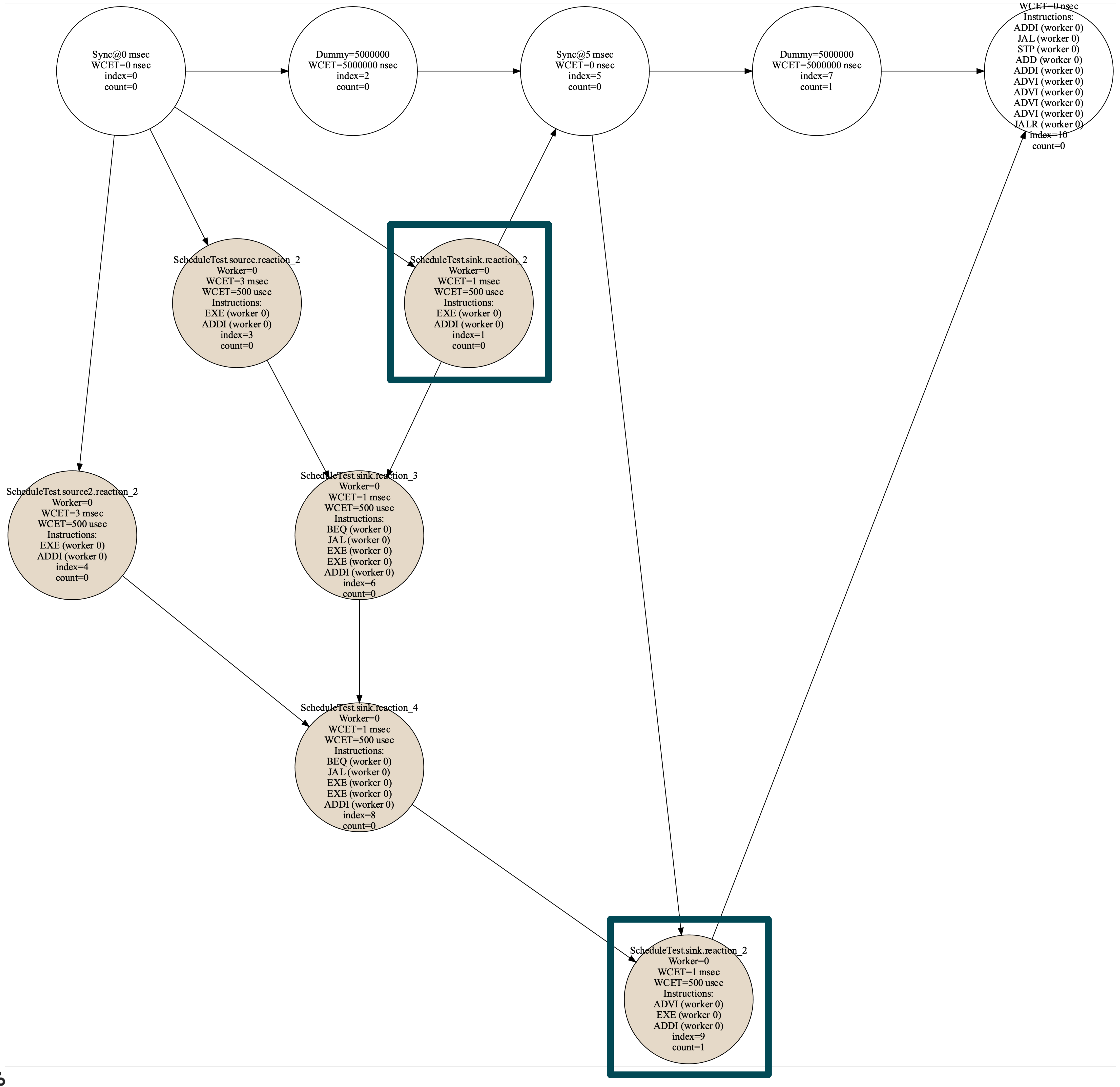
---

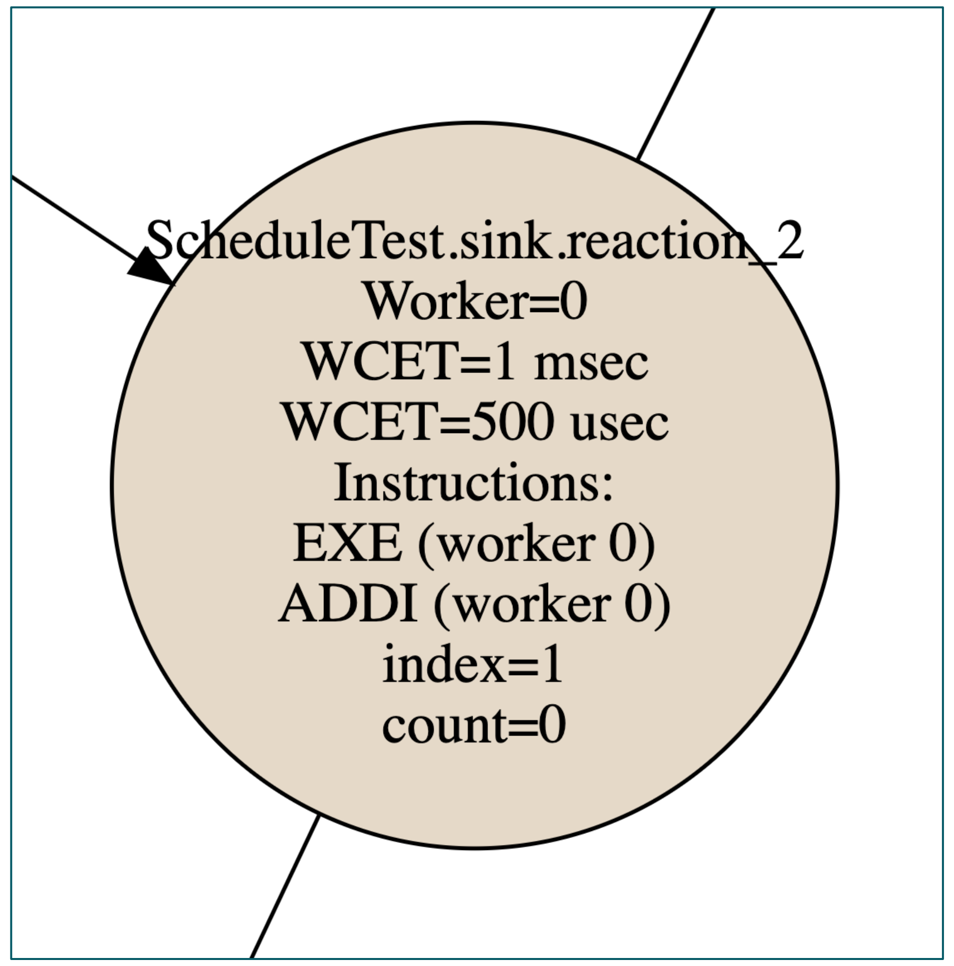
[*] Equal contribution

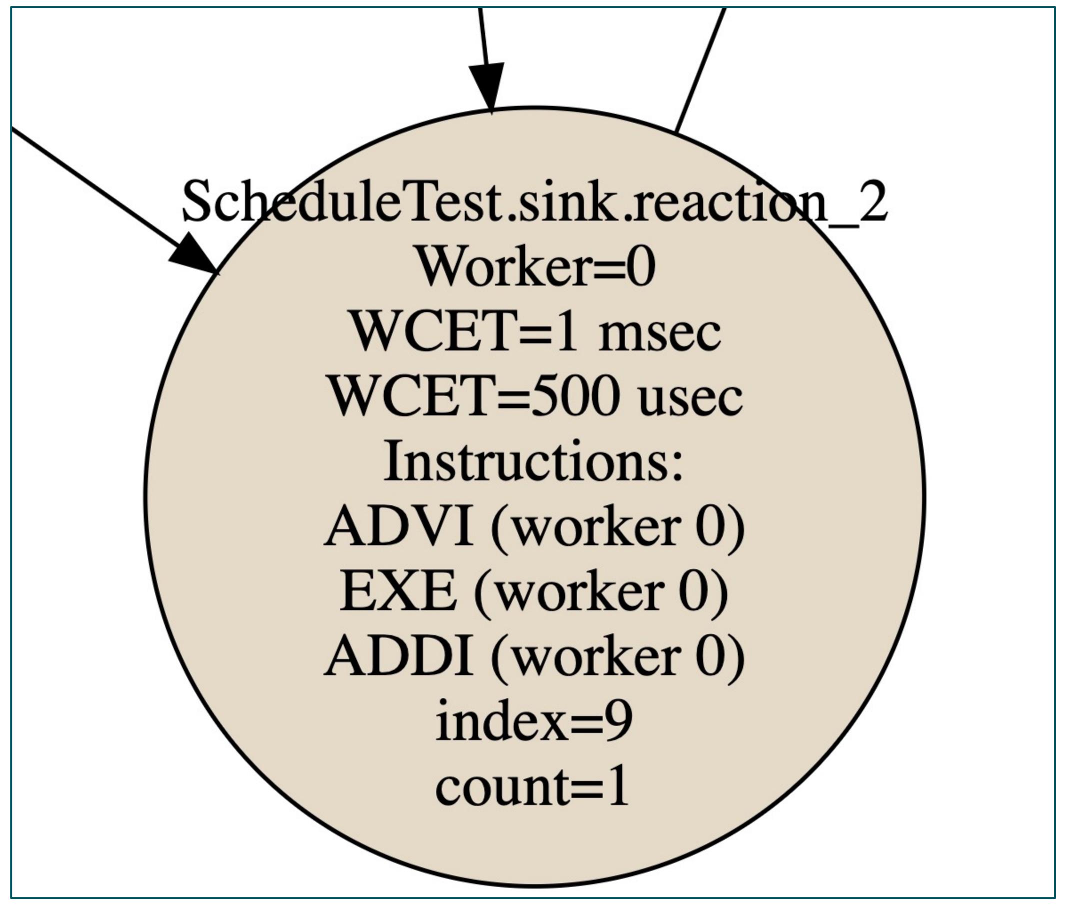
Authors' addresses: David Cao, UC San Diego, USA, dmcao@ucsd.edu; Rose Kunkel, UC San Diego, USA, rkunkel@eng.ucsd.edu; Chandrakana Nandi, Certora, Inc., USA, chandra@certora.com; Max Willsey, University of Washington, USA, mwillsey@cs.washington.edu; Zachary Tatlock, University of Washington, USA, ztatlock@cs.washington.edu; Nadia Polikarpova, UC San Diego, USA, npolikarpova@eng.ucsd.edu.

PROCEDURE:
1. <Line1>
2. <Line2>
3. <Line3>
4. JALR return_addr

Main:
5. JAL PROCEDURE
6. JAL PROCEDURE
...
104. JAL PROCEDURE


Sync@0 msec
WCET=0 nsec
index=0
count=0
Dummy=5000000
WCET=5000000 nsec
index=2
count=0
Sync@5 msec
WCET=0 nsec
index=5
count=0
Dummy=5000000
WCET=5000000 nsec
index=7
count=1
WCET=0 nsec
Instructions:
ADDI (worker 0)
JAL (worker 0)
STP (worker 0)
ADD (worker 0)
ADDI (worker 0)
ADVI (worker 0)
ADVI (worker 0)
ADVI (worker 0)
ADVI (worker 0)
JALR (worker 0)
index=10
count=0
ScheduleTest.source.reaction_2
Worker=0
WCET=3 msec
WCET=500 usec
Instructions:
EXE (worker 0)
ADDI (worker 0)
index=3
count=0
ScheduleTest.sink.reaction_2
Worker=0
WCET=1 msec
WCET=500 usec
Instructions:
EXE (worker 0)
ADDI (worker 0)
index=1
count=0
ScheduleTest.source2.reaction_2
Worker=0
WCET=3 msec
WCET=500 usec
Instructions:
EXE (worker 0)
ADDI (worker 0)
index=4
count=0
ScheduleTest.sink.reaction_3
Worker=0
WCET=1 msec
WCET=500 usec
Instructions:
BEQ (worker 0)
JAL (worker 0)
EXE (worker 0)
EXE (worker 0)
ADDI (worker 0)
index=6
count=0
ScheduleTest.sink.reaction_4
Worker=0
WCET=1 msec
WCET=500 usec
Instructions:
BEQ (worker 0)
JAL (worker 0)
EXE (worker 0)
EXE (worker 0)
ADDI (worker 0)
index=8
count=0
ScheduleTest.sink.reaction_2
Worker=0
WCET=1 msec
WCET=500 usec
Instructions:
ADVI (worker 0)
EXE (worker 0)
ADDI (worker 0)
index=9
count=1


ScheduleTest.sink.reaction_2
Worker=0
WCET=1 msec
WCET=500 usec
Instructions:
EXE (worker 0)
ADDI (worker 0)
index=1
count=0


ScheduleTest.sink.reaction_2
Worker=0
WCET=1 msec
WCET=500 usec
Instructions:
ADVI (worker 0)
EXE (worker 0)
ADDI (worker 0)
index=9
count=1

Current Progress

Toward Opt. 1:

- Set up the code base for optimization passes ✅
- Refactoring: a stronger notion of registers ✅
- (Wrestling with a concurrency bug) ⏳ (80%)
- ADV => ADD ⏳ (30%)
- Peephole optimizer ⏳ (50%)

Toward Opt. 2:

- Finding a procedure extraction strategy: finding identical nodes in DAGs ✅
- Generate procedures and jumps 💤

I am trying to get both done by the end of the week.

Future work: optimizing w.r.t. multiple objectives

Performance, Dependability, Heterogeneity

Timing
(Execution Time, Precision)

Memory Footprint

Energy Consumption

Predictability

Networking

Multi-core

Debuggability

Deadline Violation Check

Thank You!

It’s been a great semester with you all.

☺