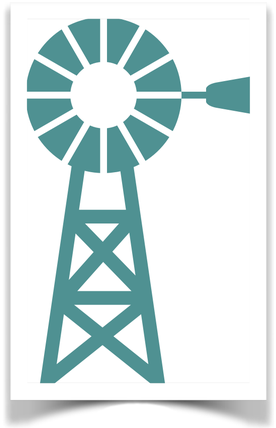
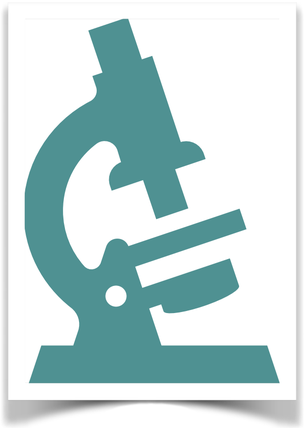
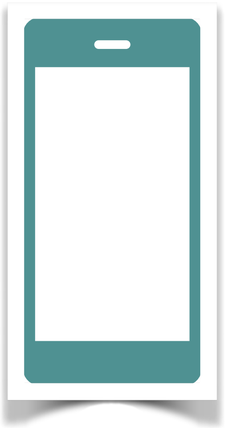
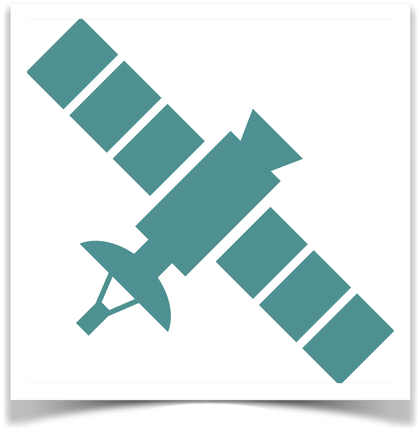
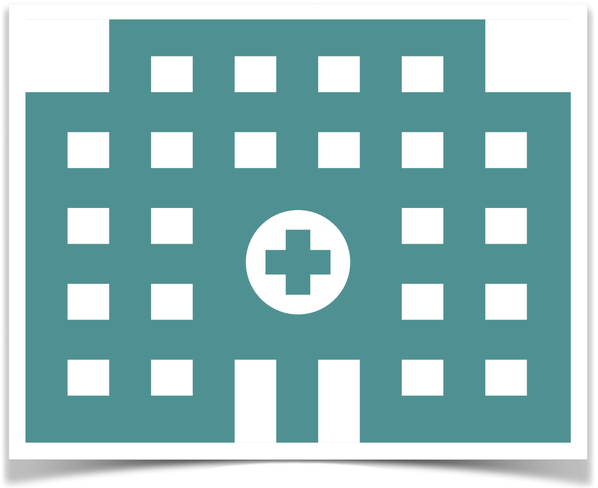
# Simple and Efficient Concurrent Programming via Synchronization Synthesis

Kostas Ferles

UToPiA

The University of Texas at Austin
Computer Science

# Software-Powered Society

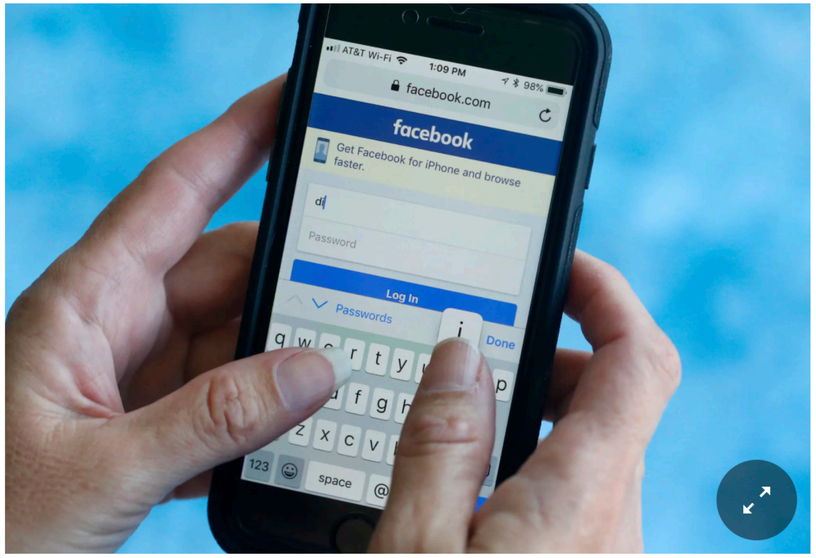
# Bugs Everywhere!

# Bugs Everywhere!

## *Irish Hospitals Are Latest to Be Hit by Ransomware Attacks*

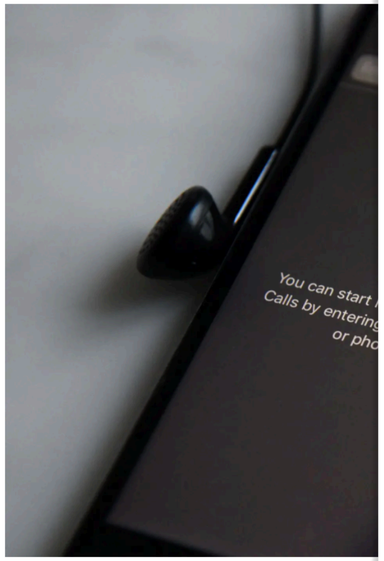
Hospitals in Ireland, Ne
Diego are reeling from c

## *Facebook Hack Included Search History and Location Data of Millions*




Facebook said Friday that a security breach had affected 30 million users, 20 million fewer than originally thought. Wilfredo Lee/Associated Press

## *In a Stumble for Apple, a FaceTime Bug Lets iPhone Users Eavesdrop*



The FaceTime bug could also give a caller
camera. Roslan Rahman/Agence France-Pres

## *Cisco to Pay $8.6 Million to Settle Government Claims of Flawed Tech*


A vulnerability in Cisco surveillance software was identified in 2008 by a whistle-blower, but the company continued to sell the software until July 2013. Albert Gea/Reuters

# Bugs Everywhere!

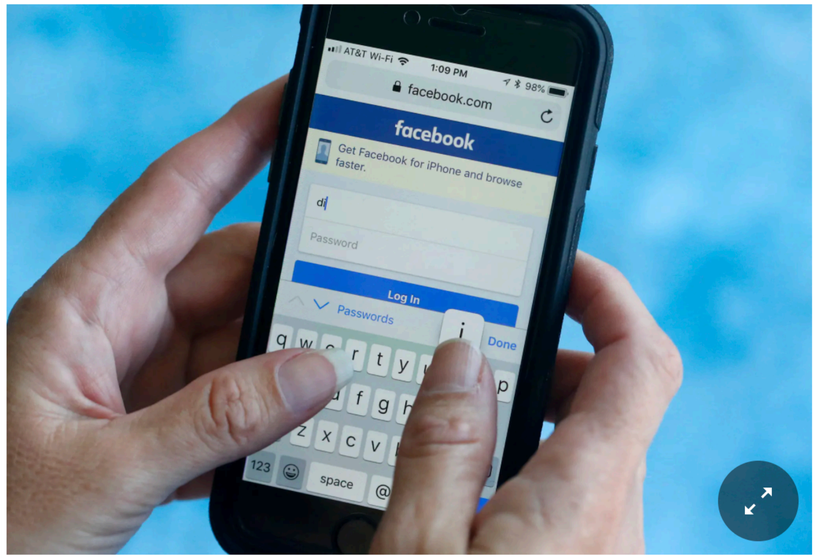
## Formal Methods to The Rescue

### *Irish Hospitals Are Latest to Be Hit by Ransomware Attacks*

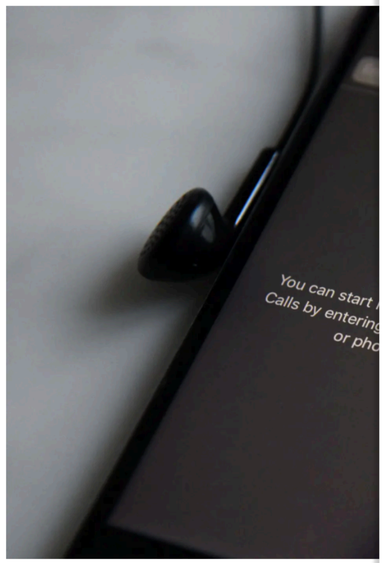
Hospitals in Ireland, Ne
Diego are reeling from

### *Facebook Hack Included Search History and Location Data of Millions*

8




### *In a Stumble for Apple, a FaceTime Bug Lets iPhone Users Eavesdrop*




The FaceTime bug could also give a caller
camera. Roslan Rahman/Agence France-Pres

### *Cisco to Pay $8.6 Million to Settle Government Claims of Flawed Tech*


A vulnerability in Cisco surveillance software was identified in 2008 by a whistle-

# Bugs ~~Everywhere~~! Nowhere

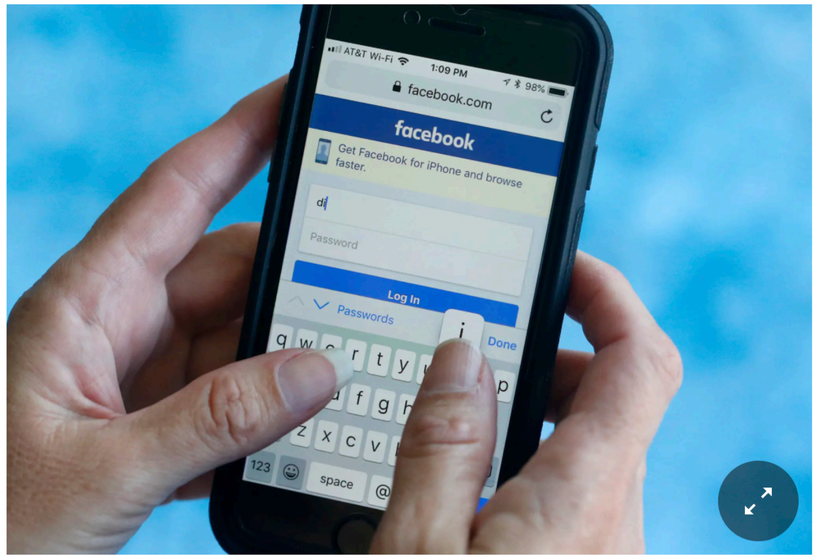
## Formal Methods to The Rescue

### *Irish Hospitals Are Latest to Be Hit by Ransomware Attacks*

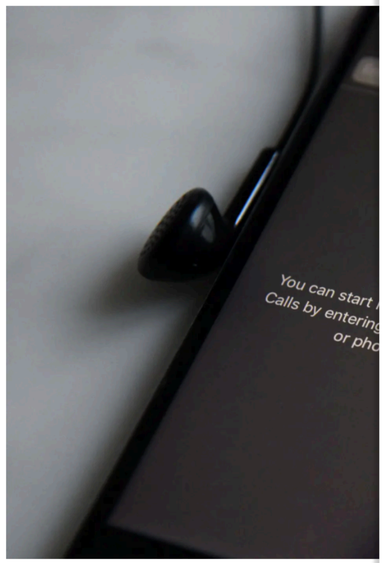
Hospitals in Ireland, Ne
Diego are reeling from

### *Facebook Hack Included Search History and Location Data of Millions*

8




### *In a Stumble for Apple, a FaceTime Bug Lets iPhone Users Eavesdrop*



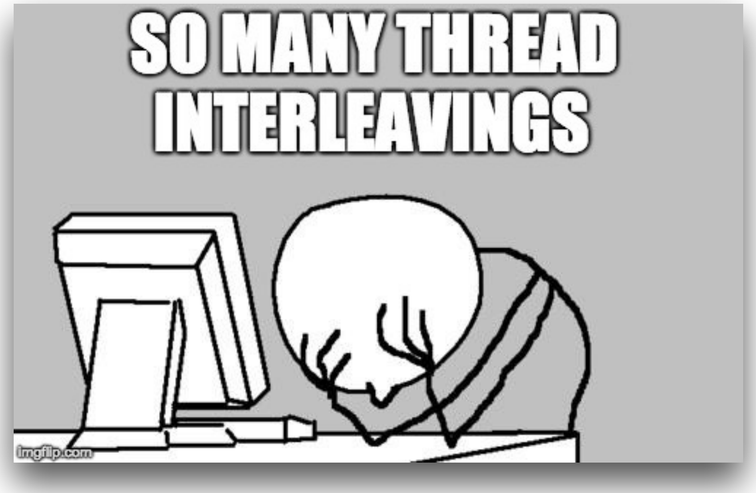
The FaceTime bug could also give a caller
camera. Roslan Rahman/Agence France-Pres

### *Cisco to Pay $8.6 Million to Settle Government Claims of Flawed Tech*


A vulnerability in Cisco surveillance software was identified in 2008 by a whistle-

# Motivation

Concurrent programming is hard!



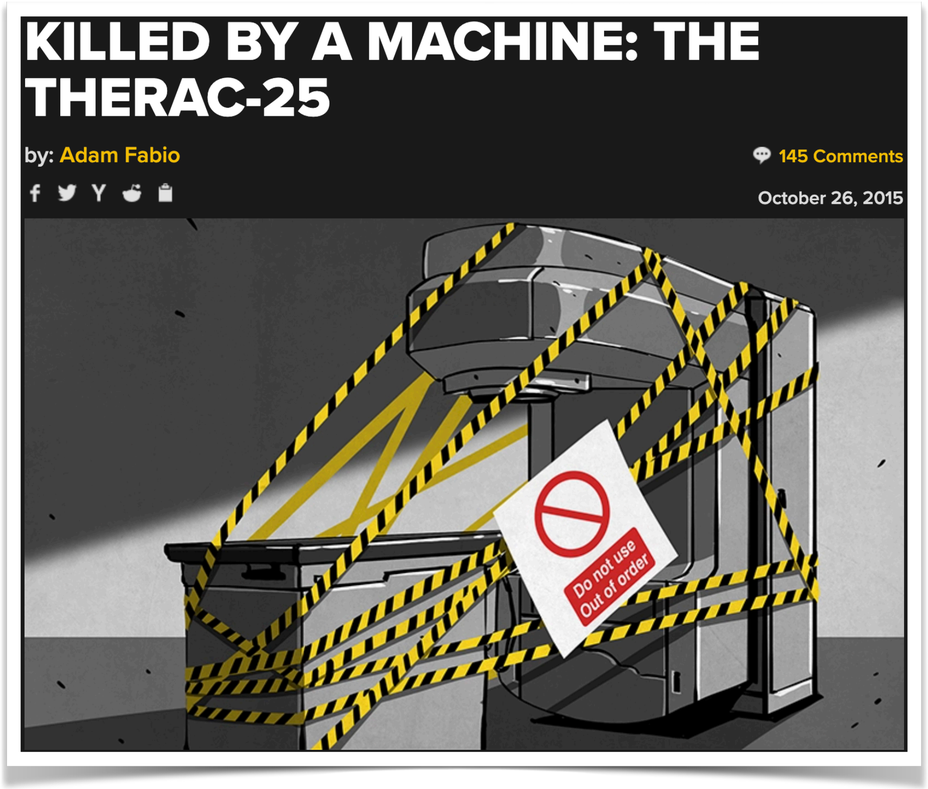

Concurrent code is a perpetual source of painful bugs:

*Deadlocks, atomicity violations, starvation, data races, …*

# Motivation

If buggy concurrent code is deployed, things can go terribly wrong!

## Infamous concurrency bugs

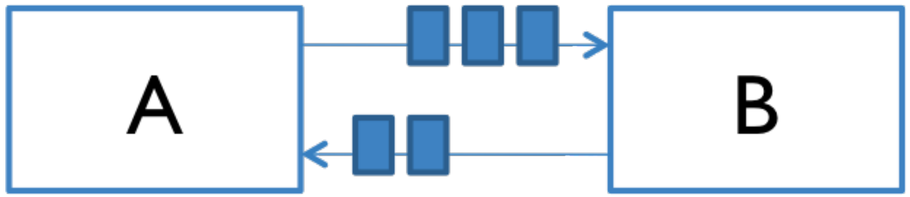


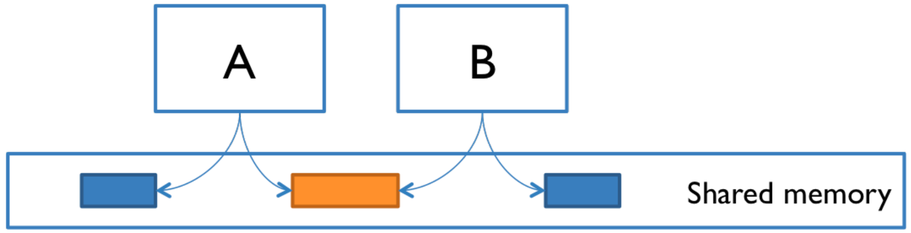

# Research Question

Can we employ formal method techniques to simplify concurrent programming without sacrificing performance?

# Shared Memory vs Message Passing

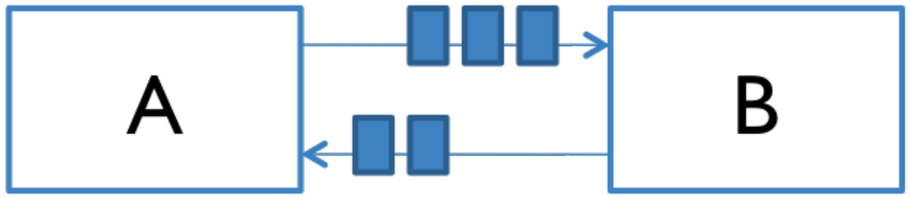
Two main models of concurrency:
*message-passing* and *shared memory*

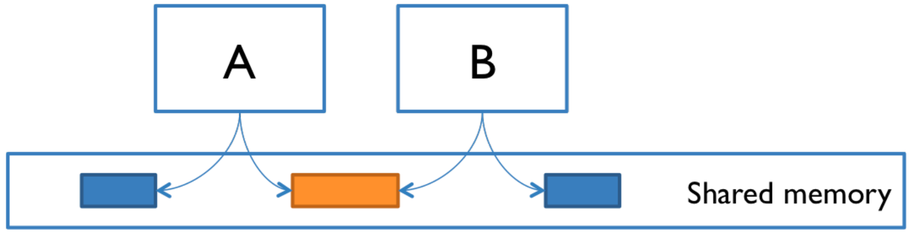


# Shared Memory vs Message Passing

Two main models of concurrency: *message-passing* and *shared memory*

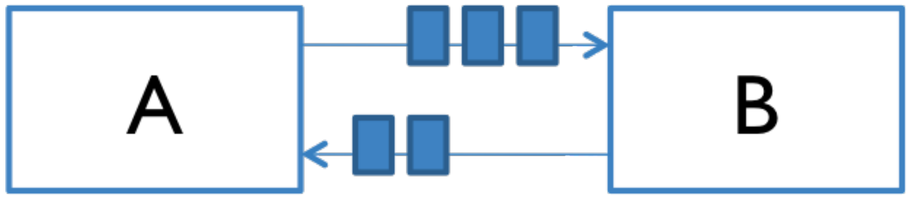


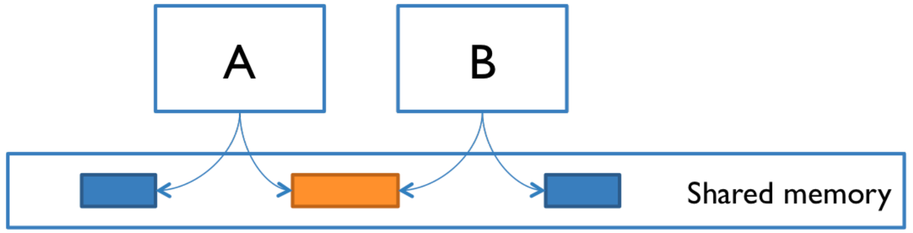

Threads pass messages; no shared state

# Shared Memory vs Message Passing

Two main models of concurrency:
*message-passing* and *shared memory*



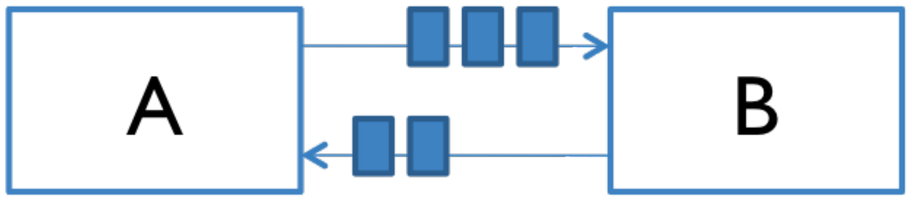

Threads pass messages; no shared state



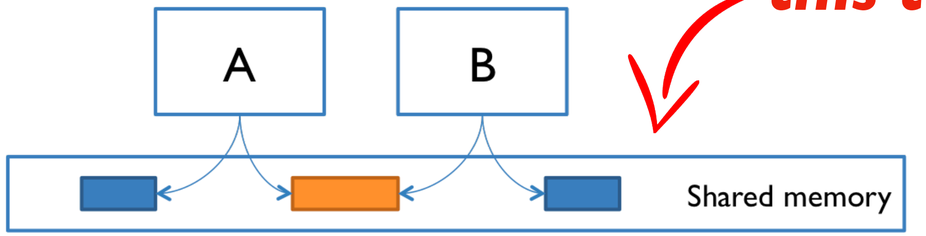

Threads exchange data through shared memory

# Shared Memory vs Message Passing

Two main models of concurrency:
*message-passing* and *shared memory*




Threads pass messages; no shared state




***Focus of this talk***

Threads exchange data through shared memory

# Shared Memory Primitives

Locks

Used to provide mutual exclusion

Condition variables

Signal threads when shared resource becomes available

# Shared Memory Primitives

Locks

Condition variables

Used to provide mutual exclusion

Signal threads when shared resource becomes available

Typically used in conjunction with each other to implement the **monitor** programming abstraction

# The Monitor Abstraction

```
class Mutex {
  private boolean h;

  private Lock l = new ...;
  private Condition heldC
      = l.newCond();

  public void acq() {
    l.lock();
    while (h)
      heldc.await();

    h = true;
    l.unlock();
  }

  public void rel() {
    l.lock();
    h = false;
    heldC.signal();
    l.unlock();
  }
}
```

- Class that mediates access to shared resources

# The Monitor Abstraction

```
class Mutex {
  private boolean h;

  private Lock l = new ...;
  private Condition heldC
      = l.newCond();

  public void acq() {
    l.lock();
    while (h)
      heldc.await();

    h = true;
    l.unlock();
  }

  public void rel() {
    l.lock();
    h = false;
    heldC.signal();
    l.unlock();
  }
}
```

- Class that mediates access to shared resources
- Threads access shared resources only by calling monitor methods
- Monitor implementation should guarantee **deadlock freedom** and **atomicity**

# Implementing Monitors

```
class Mutex {
  private boolean h;

  private Lock l = new ...;
  private Condition heldC
      = l.newCond();

  public void acq() {
    l.lock();
    while (h)
      heldc.await();

    h = true;
    l.unlock();
  }

  public void rel() {
    l.lock();
    h = false;
    heldC.signal();
    l.unlock();
  }
}
```

- Monitors typically implemented using **locks** and **condition variables**

# Implementing Monitors

```
class Mutex {
  private boolean h;

  private Lock l = new ...;
  private Condition heldC
      = l.newCond();

  public void acq() {
    l.lock();
    while (h)
      heldc.await();

    h = true;
    l.unlock();
  }

  public void rel() {
    l.lock();
    h = false;
    heldC.signal();
    l.unlock();
  }
}
```

- Monitors typically implemented using **locks** and **condition variables**

# Implementing Monitors

```
class Mutex {
  private boolean h;

  private Lock l = new ...;
  private Condition heldC
      = l.newCond();

  public void acq() {
    l.lock();
    while (h)
      heldc.await();

    h = true;
    l.unlock();
  }

  public void rel() {
    l.lock();
    h = false;
    heldC.signal();
    l.unlock();
  }
}
```

- Monitors typically implemented using **locks** and **condition variables**

# Implementing Monitors

```
class Mutex {
  private boolean h;

  private Lock l = new ...;
  private Condition heldC
      = l.newCond();

  public void acq() {
    l.lock();
    while (h)
      heldc.await();

    h = true;
    l.unlock();
  }

  public void rel() {
    l.lock();
    h = false;
    heldC.signal();
    l.unlock();
  }
}
```

- Monitors typically implemented using **locks** and **condition variables**

# Implementing Monitors

```
class Mutex {
  private boolean h;

  private Lock l = new ...;
  private Condition heldC
      = l.newCond();

  public void acq() {
    l.lock();
    while (h)
      heldc.await();

    h = true;
    l.unlock();
  }

  public void rel() {
    l.lock();
    h = false;
    heldC.signal();
    l.unlock();
  }
}
```

- Monitors typically implemented using **locks** and **condition variables**

# Implementing Monitors

```
class Mutex {
  private boolean h;

  private Lock l = new ...;
  private Condition heldC
      = l.newCond();

  public void acq() {
    l.lock();
    while (h)
      heldc.await();

    h = true;
    l.unlock();
  }

  public void rel() {
    l.lock();
    h = false;
    heldC.signal();
    l.unlock();
  }
}
```

- Monitors typically implemented using **locks** and **condition variables**
- But hard to implement both correctly and efficiently
- Improper locking and signaling cause **atomicity violations** or **deadlocks**

# Our Goal

Simplify concurrent programming by making it much easier to implement ***correct and efficient*** monitors

# Our Goal

Simplify concurrent programming by making it much easier to implement ***correct and efficient*** monitors

## Approach:

(1) Use Conditional Critical Regions (CCRs) instead of locks and condition variables

- Provides higher level of abstraction — hence easier

(2) Synthesize efficient & correct low-level code with locks and condition variables

# Conditional Critical Region (CCRs)

```
waituntil(P) {S}
```

- Blocks when P is false and atomically executes S when P becomes true

# Conditional Critical Region (CCRs)

```
waituntil(P) {S}
```

- Blocks when P is false and atomically executes S when P becomes true

- Provides a higher level of abstraction:
  - No need to use locks to ensure atomicity of S
  - No need to explicitly signal using condition variables

# Conditional Critical Region (CCRs)

```
waituntil(P) {S}
```

- Blocks when P is false and atomically executes S when P becomes true

- Provides a higher level of abstraction:
  - No need to use locks to ensure atomicity of S
  - No need to explicitly signal using condition variables

Called *implicit monitor*

**Assumption**: Each monitor method is a sequence of CCRs

# Back to Monitors

```
class Mutex {
  private boolean h;

  private Lock l = new ...;
  private Condition heldC
       = l.newCond();

  public void acq() {
    l.lock();
    while (h)
      bCond.await();

    h = true;
    l.unlock();
  }

  public void rel() {
    l.lock();
    h = false;
    bCond.signal();
    l.unlock();
  }
}
```

***Explicit monitor***

```
class Mutex {
  private boolean h;

  public void atomic acq() {
    waituntil (!h) {
      h = true;
    }
  }

  public void atomic rel() {
    h = false;
  }
}
```

***Implicit monitor***

# Back to Monitors

```
class Mutex {
  private boolean h;

  private Lock l = new ...;
  private Condition heldC
       = l.newCond();

  public void acq() {
    l.lock();
    while (h)
      bCond.await();

    h = true;
    l.unlock();
  }

  public void rel() {
    l.lock();
    h = false;
    bCond.signal();
    l.unlock();
  }
}
```

***Explicit monitor***

Omit waituntil if predicate is true.

```
class Mutex {
  private boolean h;

  public void atomic acq() {
    waituntil (!h) {
      h = true;
    }
  }

  public void atomic rel() {
    h = false;
  }
}
```

***Implicit monitor***

# Back to Monitors

```
class Mutex {
  private boolean h;

  private Lock l = new ...;
  private Condition heldC
       = l.newCond();

  public void acq() {
    l.lock();
    while (h)
      bCond.await();

    h = true;
    l.unlock();
  }

  public void rel() {
    l.lock();
    h = false;
    bCond.signal();
    l.unlock();
  }
}
```

**Explicit monitor**

$Preds(M)$ →

```
class Mutex {
  private boolean h;

  public void atomic acq() {
    waituntil (!h) {
      h = true;
    }
  }

  public void atomic rel() {
    h = false;
  }
}
```

**Implicit monitor**

# Back to Monitors

```
class Mutex {
  private boolean h;

  private Lock l = new ...;
  private Condition heldC
       = l.newCond();

  public void acq() {
    l.lock();
    while (h)
      bCond.await();

    h = true;
    l.unlock();
  }

  public void rel() {
    l.lock();
    h = false;
    bCond.signal();
    l.unlock();
  }
}
```

***Explicit monitor***

Implicit Monitor = Explicit Monitor - Synchronization Code

```
class Mutex {
  private boolean h;

  public void atomic acq() {
    waituntil (!h) {
      h = true;
    }
  }

  public void atomic rel() {
    h = false;
  }
}
```

***Implicit monitor***

# Explicit vs Implicit Trade-offs

## Explicit

Pro: Efficient

Con: Hard to get right

## Implicit

Pro: Easy to get right

Con: Inefficient

# Explicit vs Implicit Trade-offs

## Explicit

Pro: Efficient

Con: Hard to get right

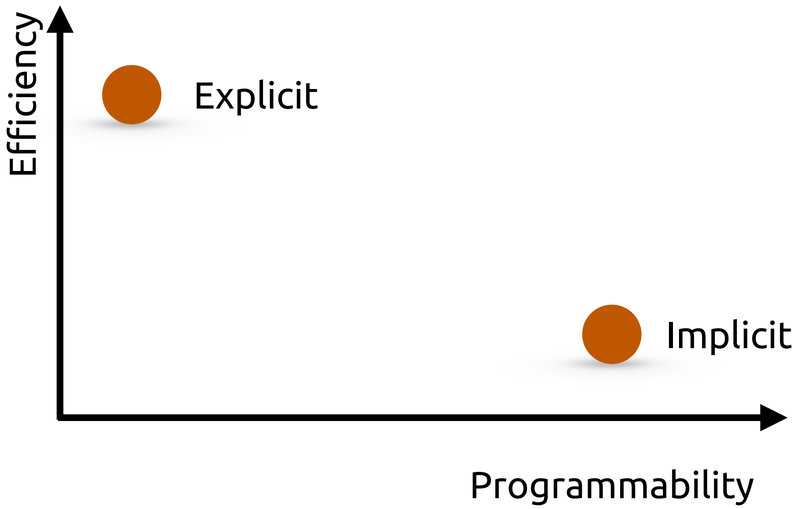
## Implicit

Pro: Easy to get right

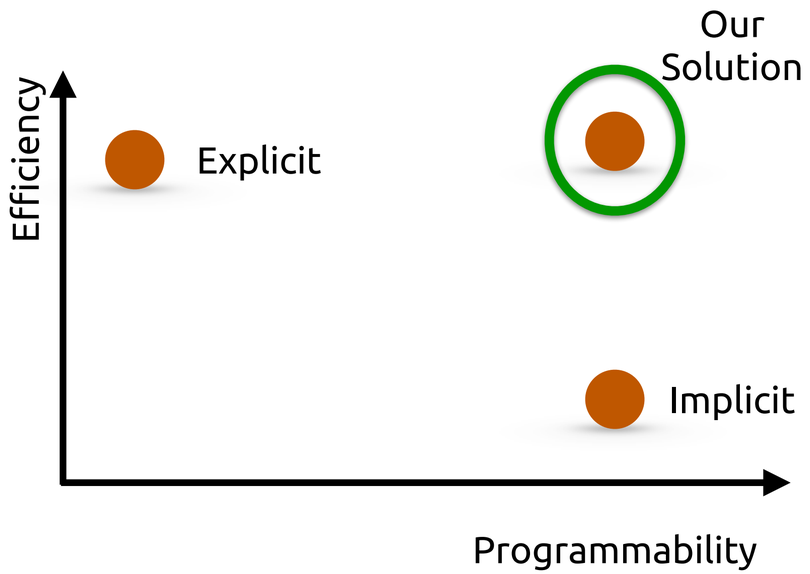
Con: Inefficient

All existing solutions require run-time support

# Synthesis-Based Solution



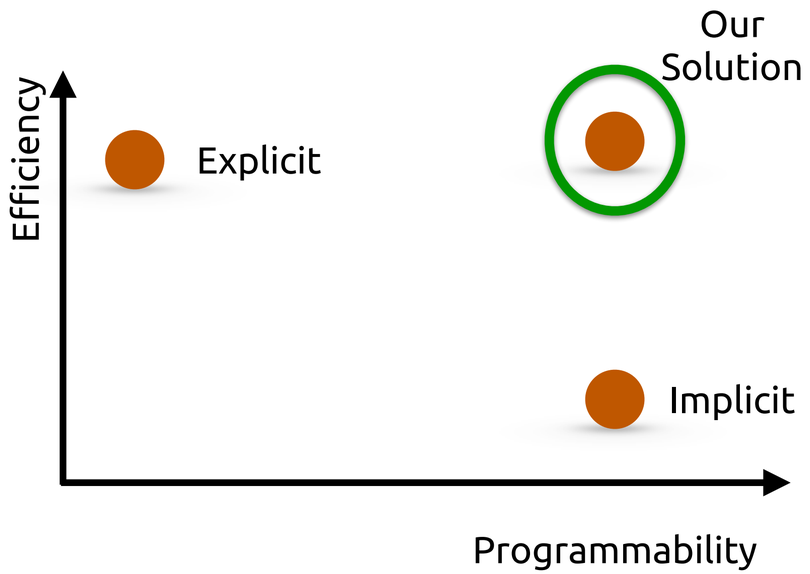
# Synthesis-Based Solution




Goal: *Synthesize* explicit monitor from implicit

- Correct-by-construction
- As efficient as possible

# Synthesis-Based Solution




Goal: *Synthesize* explicit monitor from implicit

- Correct-by-construction
- As efficient as possible

Provides best of both worlds in terms of efficiency & programmability!

# Road Map

**Part I:** How to automatically synthesize necessary signaling code

Symbolic Reasoning for Automatic Signal Placement.
Ferles et al, PLDI'18.

**Part II:** How to automatically synthesize efficient locking

Synthesizing Fine-grained Synchronization Protocols for Implicit Monitors.
Ferles et al, OOPSLA'22.

# Part 1: Synthesizing Signaling Code

**The task:**

Given an implicit monitor:

- instrument code with condition vars
- add all necessary wait/signal operations on these variables

# Part 1: Synthesizing Signaling Code

**The task:**

Given an implicit monitor:

- instrument code with condition vars
- add all necessary wait/signal operations on these variables

*Key difficulty:* Where and how to signal so that each thread can make progress and there are no spurious wake-ups?

# Signaling Decisions

For each CCR C and predicate p $\in Preds(M)$, decide *whether* to signal or not, and if so, *how*?

Is any signaling necessary?

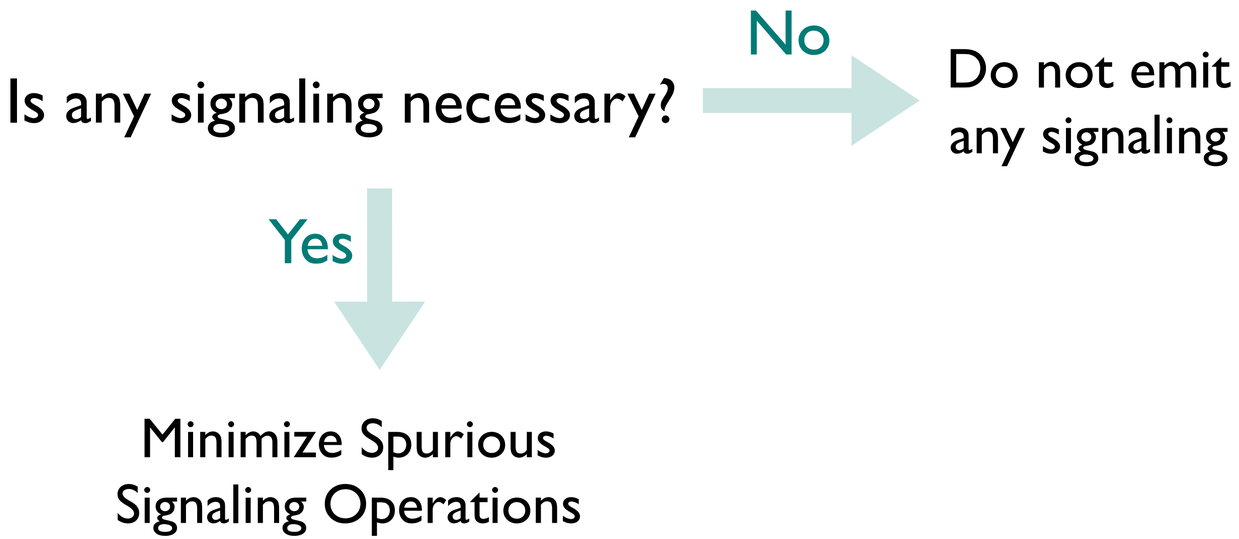
# Signaling Decisions

For each CCR C and predicate p $\in Preds(M)$, decide *whether* to signal or not, and if so, *how*?

Is any signaling necessary? —No→ Do not emit any signaling

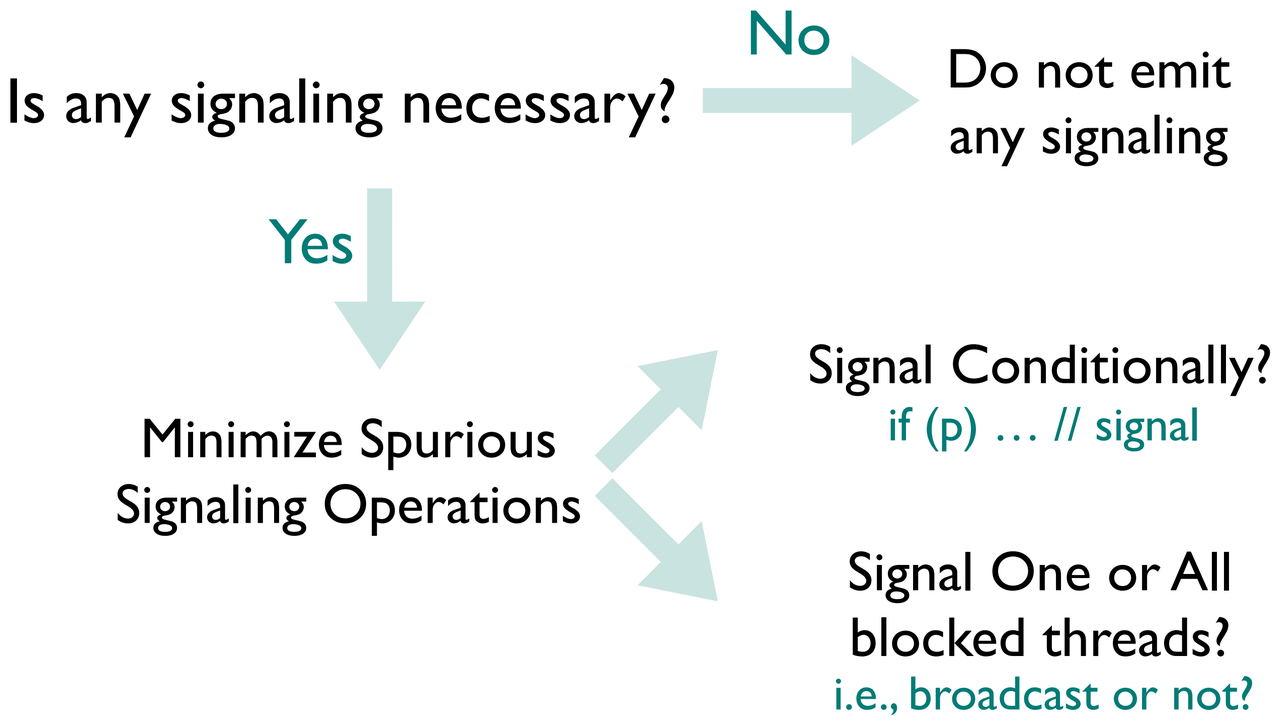
# Signaling Decisions

For each CCR C and predicate p $\in Preds(M)$, decide *whether* to signal or not, and if so, *how*?



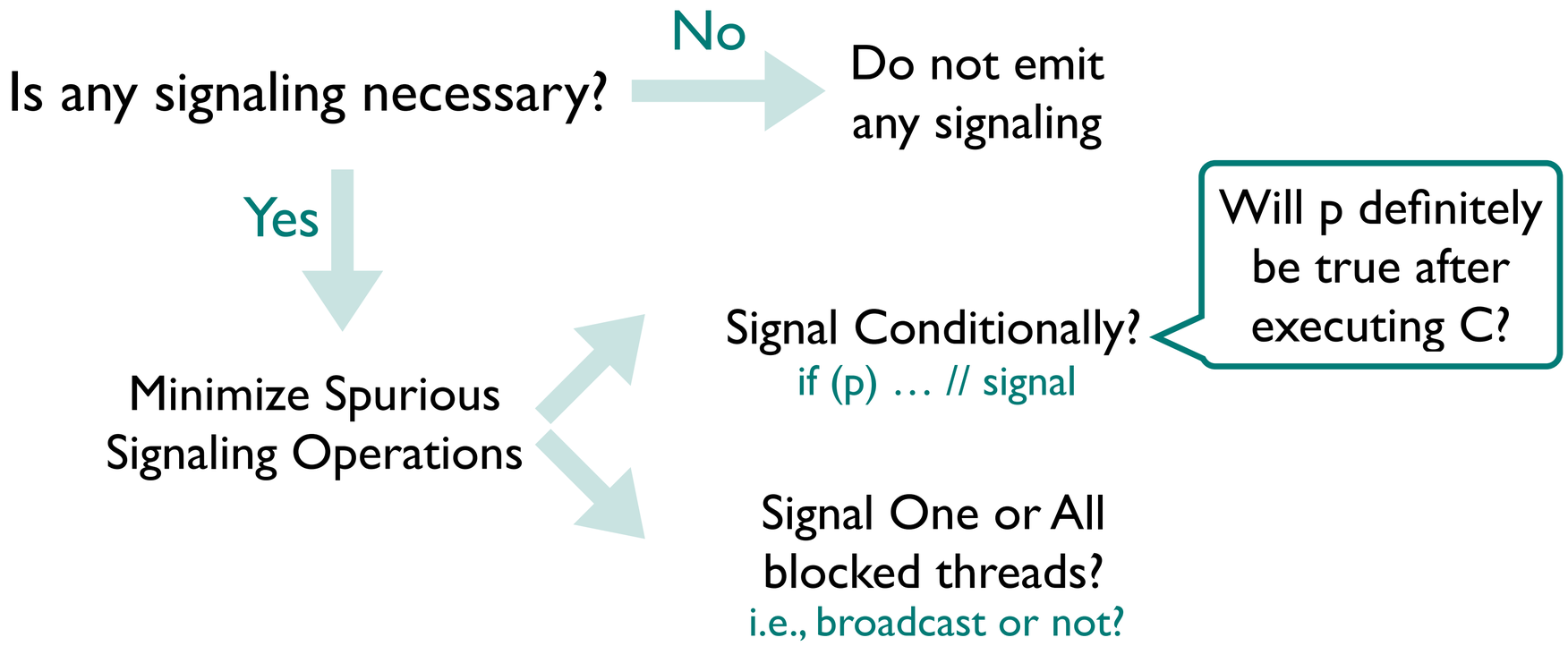
# Signaling Decisions

For each CCR C and predicate p $\in Preds(M)$, decide *whether* to signal or not, and if so, *how*?



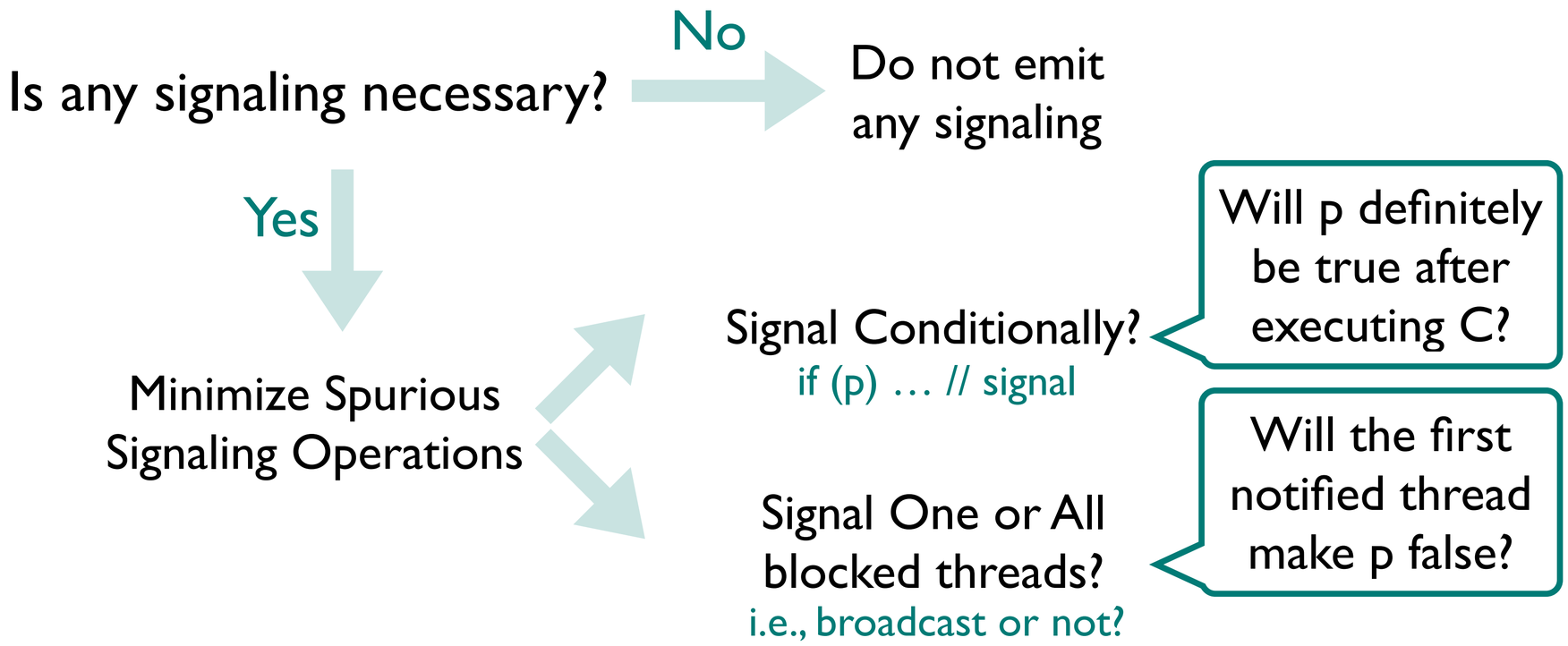
# Signaling Decisions

For each CCR C and predicate p $\in Preds(M)$, decide *whether* to signal or not, and if so, *how*?



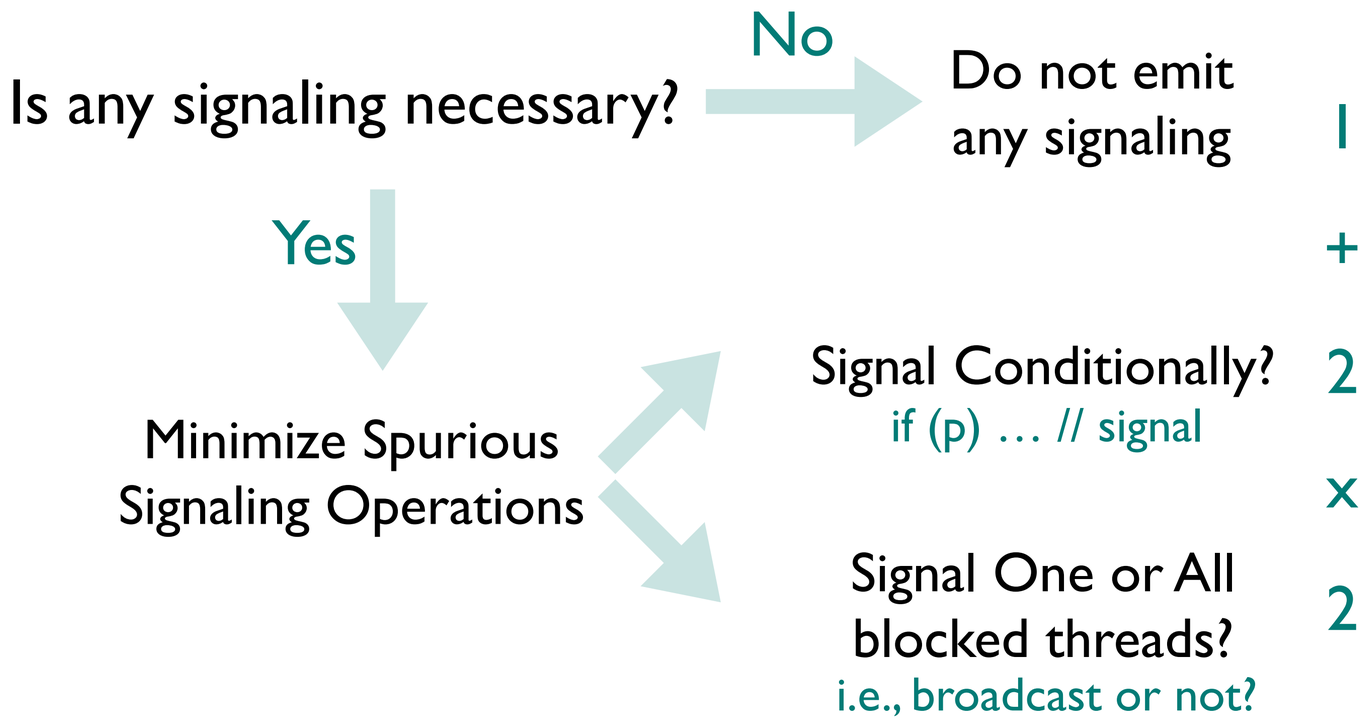
# Signaling Decisions

For each CCR C and predicate p $\in Preds(M)$, decide *whether* to signal or not, and if so, *how*?



# Signaling Decisions

For each CCR C and predicate p $\in Preds(M)$, decide *whether* to signal or not, and if so, *how*?




For each CCR and predicate, 5 possible outcomes

# Running Example

```
class RWLock {
  int readers = 0;
  boolean writerIn = false;

  atomic void enterReader() {
    waituntil(!writerIn) {
     readers++; }
  }

  atomic void exitReader() {
    if (readers > 0) readers—;
  }

  atomic void enterWriter() {
    waituntil(readers == 0 && !writerIn)
    { writerIn = true; }
  }

  atomic void exitWriter() {
    writerIn = false;
  }
}
```

# Running Example

```
class RWLock {
  int readers = 0;
  boolean writerIn = false;

  atomic void enterReader() {
    waituntil(!writerIn) {
     readers++; }
  }

  atomic void exitReader() {
    if (readers > 0) readers—;
  }

  atomic void enterWriter() {
    waituntil(readers == 0 && !writerIn)
    { writerIn = true; }
  }

  atomic void exitWriter() {
    writerIn = false;
  }
}
```

```
RWLock l = new …();
Readers:
l.enterReader();
… // Access Resource
l.exitReader();

Writers:
l.enterWriter();
… // Access Resource
l.exitWriter();
```

# Lots of Possible Implementations

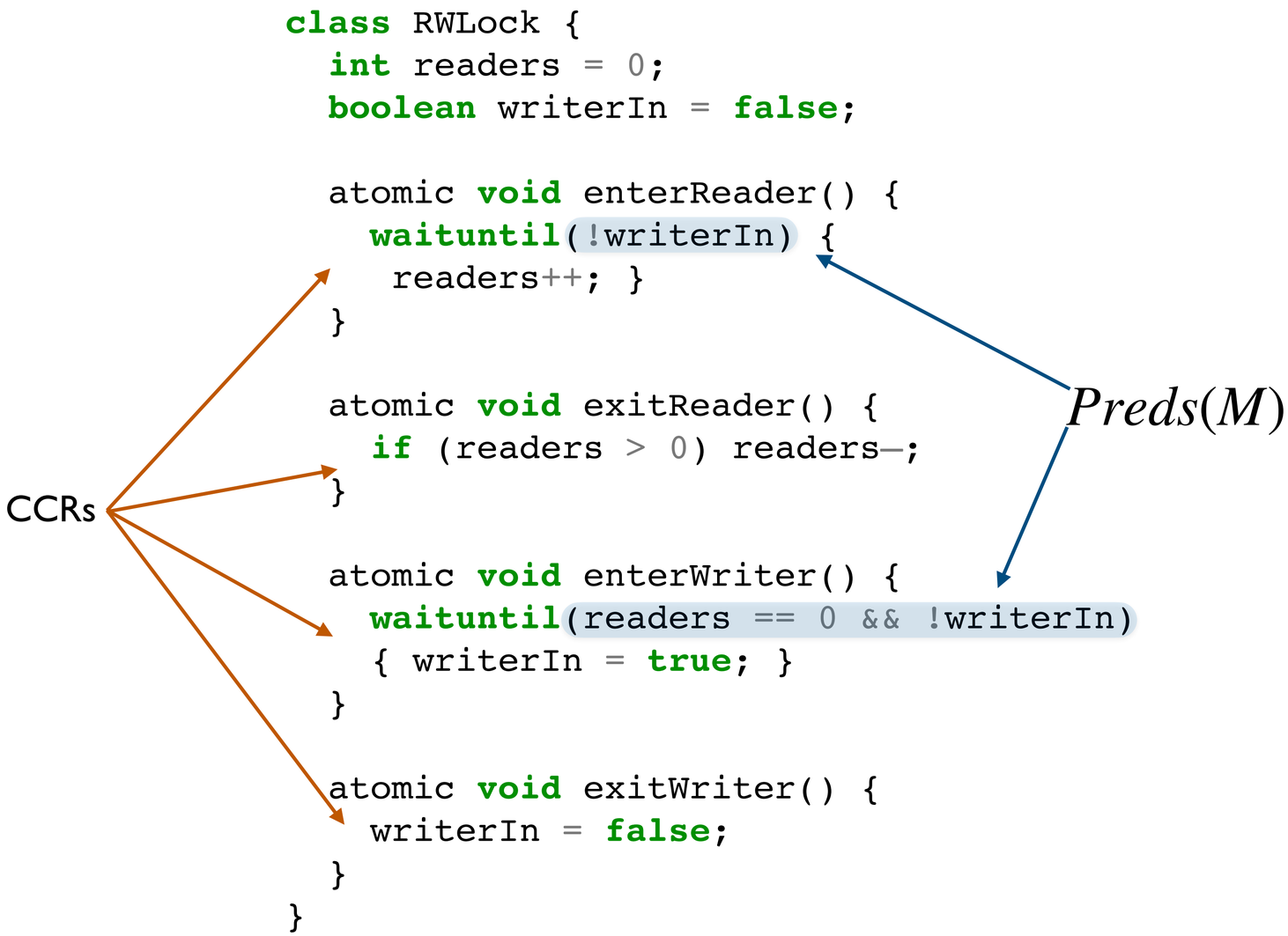
```
class RWLock {
  int readers = 0;
  boolean writerIn = false;

  atomic void enterReader() {
    waituntil(!writerIn) {
     readers++; }
  }

  atomic void exitReader() {
    if (readers > 0) readers–;
  }

  atomic void enterWriter() {
    waituntil(readers == 0 && !writerIn)
    { writerIn = true; }
  }

  atomic void exitWriter() {
    writerIn = false;
  }
}
```

CCRs

$Preds(M)$

Need to make signaling decisions for each combination of CCRs and predicates (8 pairs)

# Lots of Possible Implementations




Need to make signaling decisions for each combination of CCRs and predicates (8 pairs)

So $5^8$ different ways to implement explicit monitor! (ignoring locking)

# Optimal Signaling for this Example

```
class RWLock {
  int readers = 0;
  boolean writerIn = false;

  atomic void enterReader() {
    waituntil(!writerIn) {
     readers++; }
  }

  atomic void exitReader() {
    if (readers > 0) readers–;
  }

  atomic void enterWriter() {
    waituntil(readers == 0 && !writerIn)
    { writerIn = true; }
  }

  atomic void exitWriter() {
    writerIn = false;
  }
}
```

← No signaling necessary

# Optimal Signaling for this Example

```
class RWLock {
  int readers = 0;
  boolean writerIn = false;

  atomic void enterReader() {
    waituntil(!writerIn) {
     readers++; }
  }

  atomic void exitReader() {
    if (readers > 0) readers–;
  }

  atomic void enterWriter() {
    waituntil(readers == 0 && !writerIn)
    { writerIn = true; }
  }

  atomic void exitWriter() {
    writerIn = false;
  }
}
```

No signaling necessary

Condition var representing !writerIn

```
readC.signalAll();
```

Unconditional broadcast to readers

# Optimal Signaling for this Example

```
class RWLock {
  int readers = 0;
  boolean writerIn = false;

  atomic void enterReader() {
    waituntil(!writerIn) {
     readers++; }
  }

  atomic void exitReader() {
    if (readers > 0) readers--;
  }

  atomic void enterWriter() {
    waituntil(readers == 0 && !writerIn)
    { writerIn = true; }
  }

  atomic void exitWriter() {
    writerIn = false;
  }
}
```

Conditionally signal a single writer:

```
if(readers == 0) writeC.signal();
```

No signaling necessary

Condition var representing !writerIn

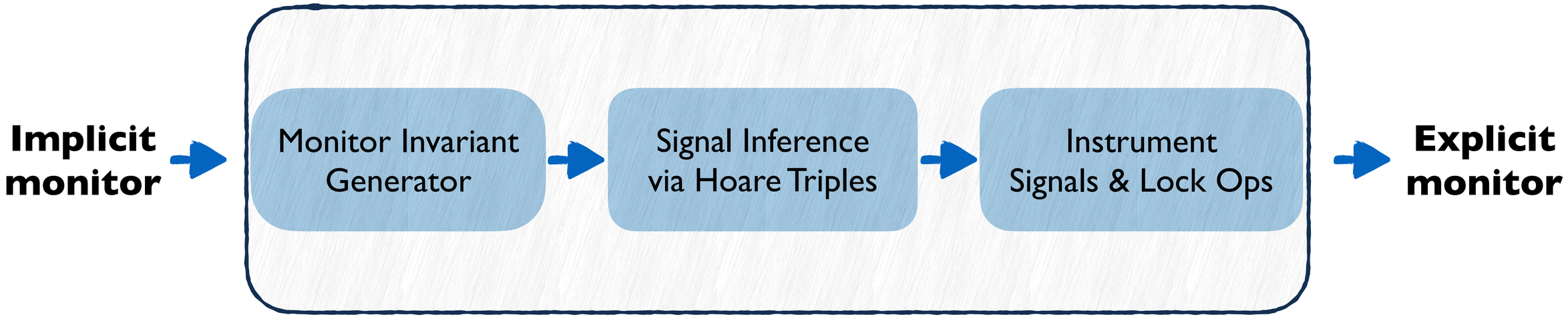
```
readC.signalAll();
```

Unconditional broadcast to readers

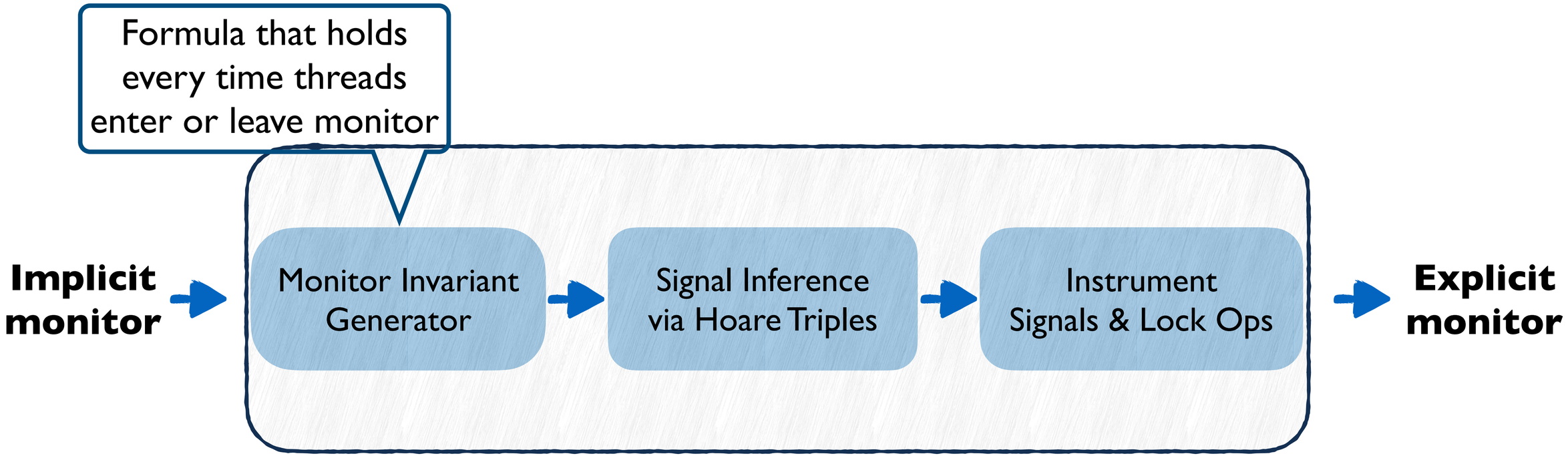
# Optimal Explicit Monitor

```
class RWLock {
  int readers = 0;
  boolean writerIn = false;

  Lock l = new ReentrantLock();
  Condition readC = l.newCondition(), writeC = l.newCondition();

  void enterReader() {
    l.lock();
    while(writerIn) readC.await();
    readers++;
    l.unlock();
  }

  void exitReader() {
    l.lock();
    if (readers > 0) readers--;
    if (readers == 0 && !writerIn) writeC.signal();
    l.unlock();
  }

  void enterWriter() {
    l.lock();
    while(readers != 0 || writerIn) writeC.await();
    writerIn = true;
    l.unlock();
  }

  void exitWriter() {
    l.lock();
    writerIn = false;
    if (readers == 0 && !writerIn) writeC.signal();
    readeC.signalAll();
    l.unlock();
  }}
```

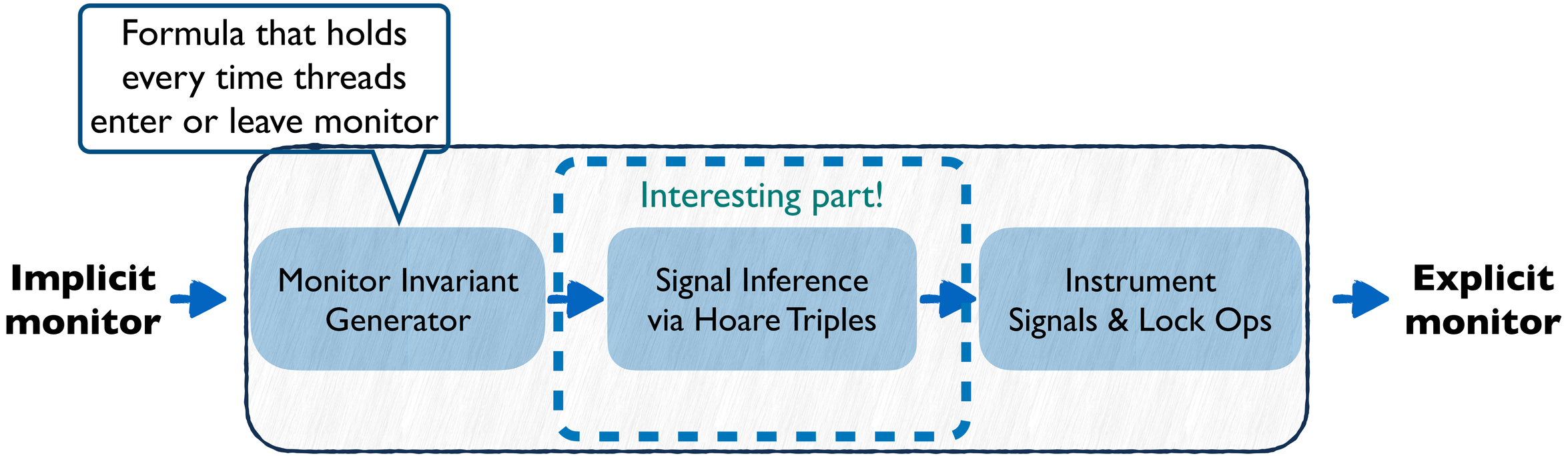
# Synthesis Workflow



# Synthesis Workflow



# Synthesis Workflow



# Primer on Hoare Triples

Hoare Triple: $\{ P \} S \{ Q \}$

# Primer on Hoare Triples

Is Q guaranteed to hold after executing S?

Hoare Triple: $\{ P \} S \{ Q \}$

# Primer on Hoare Triples

Starting in a state where P holds

Is Q guaranteed to hold after executing S?

Hoare Triple: $\{ P \} S \{ Q \}$

# Primer on Hoare Triples

Starting in a state where P holds

Is Q guaranteed to hold after executing S?

Hoare Triple: $\{ P \} S \{ Q \}$

If yes, we call a triple valid

# Primer on Hoare Triples

Starting in a state where P holds

Is Q guaranteed to hold after executing S?

Hoare Triple: $\{ P \} S \{ Q \}$

If yes, we call a triple valid

**Example of valid triple**

# Primer on Hoare Triples

Starting in a state where P holds

Is Q guaranteed to hold after executing S?

Hoare Triple: $\{ P \} S \{ Q \}$

If yes, we call a triple valid

**Example of valid triple**

$$\{ count \geq 0 \}$$

# Primer on Hoare Triples

Starting in a state where P holds

Is Q guaranteed to hold after executing S?

Hoare Triple: $\{ P \} S \{ Q \}$

If yes, we call a triple valid

**Example of valid triple**

$\{ count \geq 0 \}$ `count++;`

# Primer on Hoare Triples

Starting in a state where P holds

Is Q guaranteed to hold after executing S?

Hoare Triple: $\{ P \} S \{ Q \}$

If yes, we call a triple valid

**Example of valid triple**

$\{ count \geq 0 \}$ `count++;` $\{ count \geq 0 \}$

# Signaling Decisions as Hoare Triples


To Signal or Not:
Given CCR $C$, do we need any signaling for threads blocked on predicate $P \in Preds(M)$?

$$\{Inv \wedge \neg P\}\ C\ \{\neg P\}$$

# Signaling Decisions as Hoare Triples


To Signal or Not:
Given CCR $C$, do we need any signaling for threads blocked on predicate $P \in Preds(M)$?

If this triple is valid, **no need** to **signal** threads blocked on P!

$$\{Inv \wedge \neg P\}\ C\ \{\neg P\}$$

# Example: Signal or Not?

**CCR:**

```
atomic void enterReader() {

  waituntil(!writerIn){
    readers++;
  }

}
```

Need to signal writers after enterReader?

# Example: Signal or Not?

**Predicate:** $P_w \equiv readers = 0 \wedge \neg writerIn \quad \neg P_w \equiv readers \neq 0 \vee writerIn$

**CCR:**

```
atomic void enterReader() {

  waituntil(!writerIn){
    readers++;
  }

}
```

Need to signal writers after enterReader?

# Example: Signal or Not?

**Predicate:** $P_w \equiv readers = 0 \wedge \neg writerIn \quad \neg P_w \equiv readers \neq 0 \vee writerIn$ **Invariant:** $readers \geq 0$

**CCR:**

```
atomic void enterReader() {

  waituntil(!writerIn){
    readers++;
  }

}
```

Need to signal writers after enterReader?

# Example: Signal or Not?

**Predicate:** $P_w \equiv readers = 0 \wedge \neg writerIn \quad \neg P_w \equiv readers \neq 0 \vee writerIn$ **Invariant:** $readers \geq 0$

**CCR:**

```
atomic void enterReader() {
  {readers ≥ 0 ∧ ¬P_w}
  waituntil(!writerIn){
    readers++;
  }
  {¬P_w}
}
```

Need to signal writers after enterReader?

# Example: Signal or Not?

**Predicate:** $P_w \equiv readers = 0 \wedge \neg writerIn$ $\neg P_w \equiv readers \neq 0 \vee writerIn$ **Invariant:** $readers \geq 0$

**CCR:**

```
atomic void enterReader() {
  {readers ≥ 0 ∧ ¬P_w}
  waituntil(!writerIn){
     readers++;
  }
  { ¬P_w}
}
```

Need to signal writers after enterReader?

Triple is valid, no need to signal writers!

# Summary of Signaling

Key Idea: Reduce signaling decisions to validity of Hoare triples

- Once the signaling decisions are made, straightforward to instrument code with operations on condition vars

- To ensure atomicity and data-race-freedom, use one global lock acquired/released before/after each operation

# Implementation & Evaluation


- Evaluated on several monitors from popular GitHub repos.
  - Real-world applications: *Spring*, *Gradle*, etc.
  - Efficient explicit-signal implementations.
- Converted explicit monitor to implicit and pass it to Expresso

# Evaluation, cont

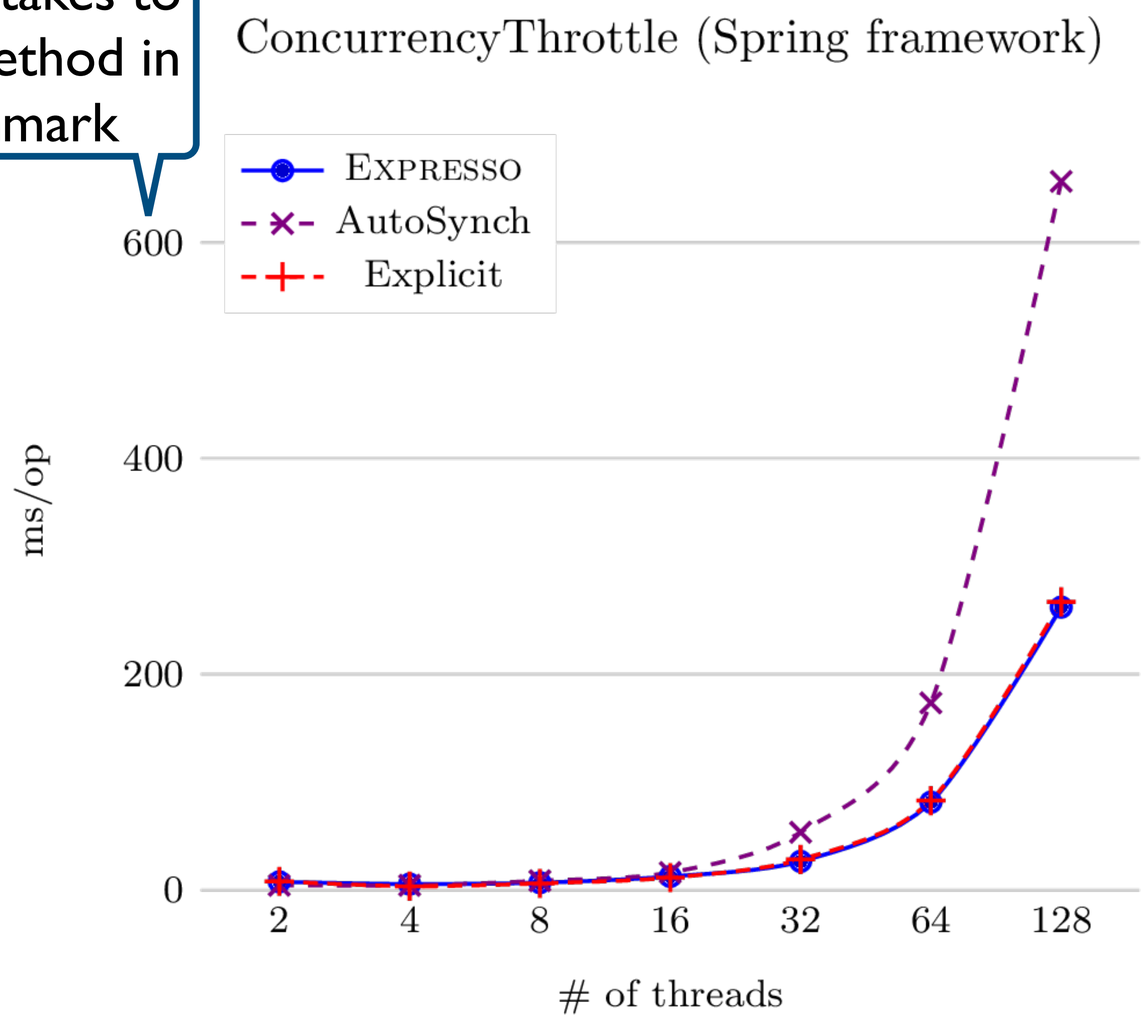
Compared ***EXPRESSO*** against:

- AutoSynch [1]: **state-of-the-art** runtime system for implicit monitors.
- Explicit: original implementation by expert programmer.

[1] Wei-Lun Hung et al. **AutoSynch**: An Automatic signal Monitor Based on Predicate Tagging **PLDI'13**

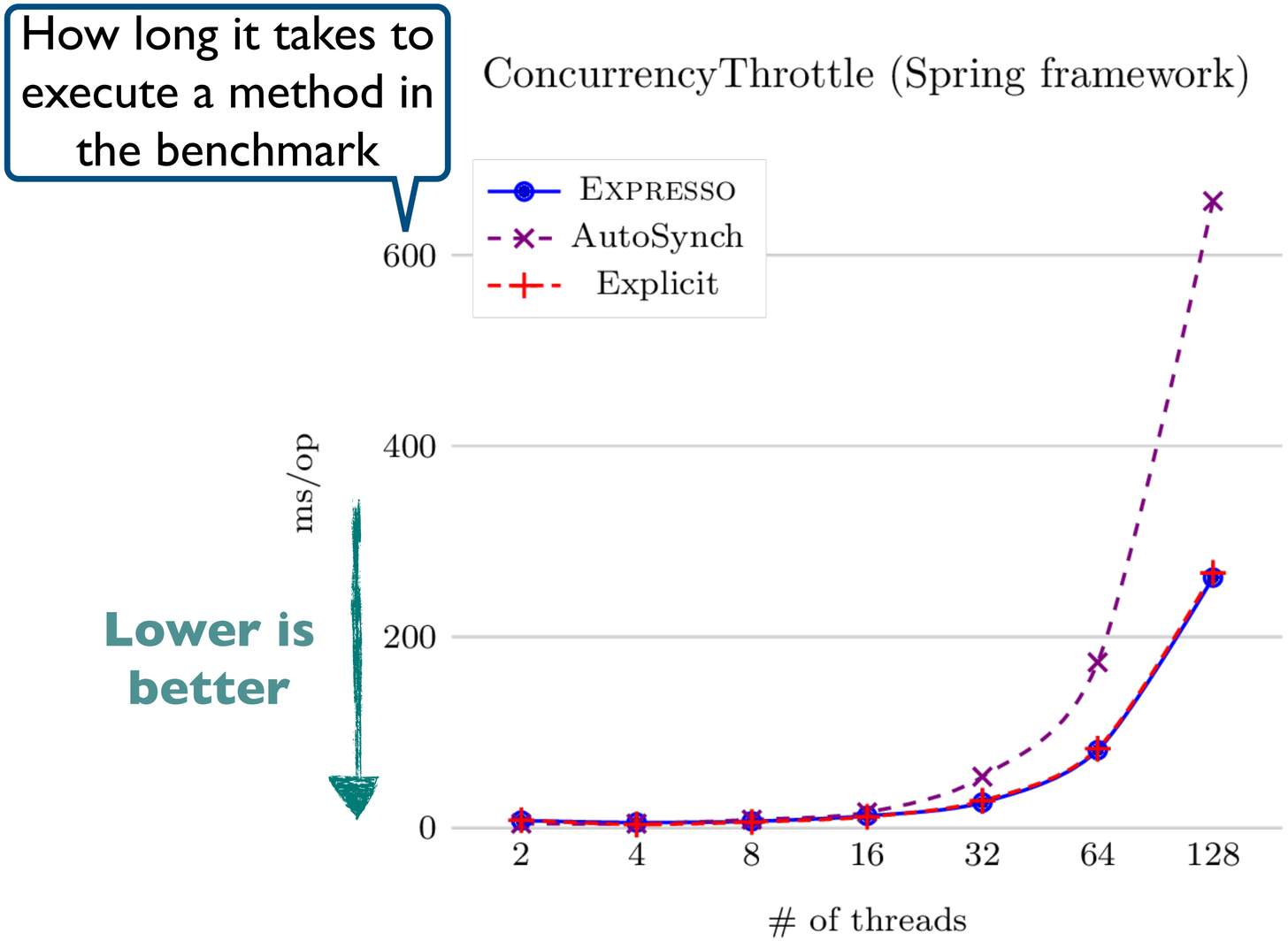
# Results

How long it takes to execute a method in the benchmark



# Results

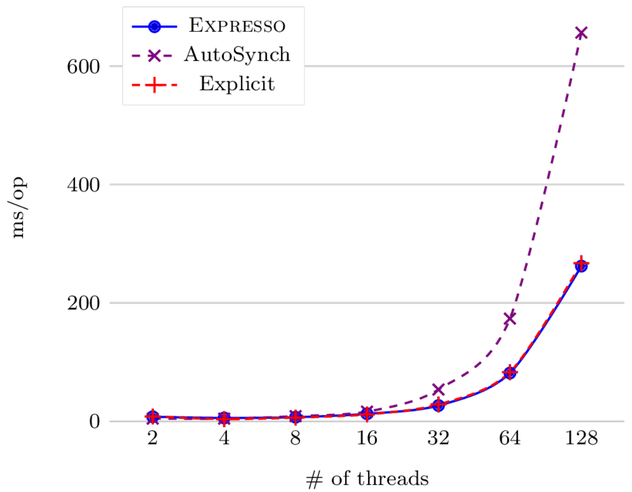
How long it takes to execute a method in the benchmark



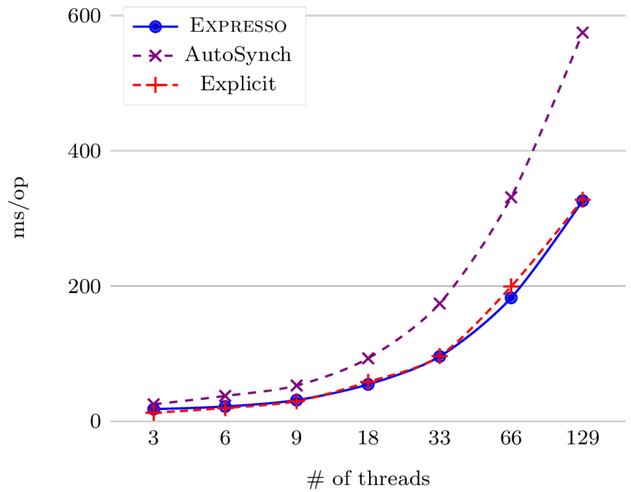

**Lower is better**

# Results, cont

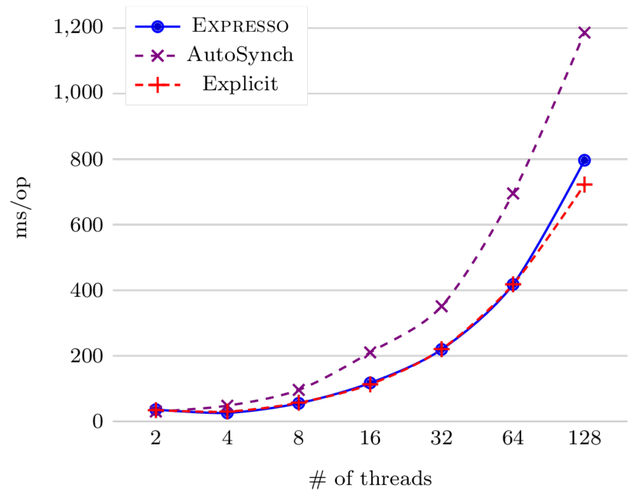
ConcurrencyThrottle (Spring framework)



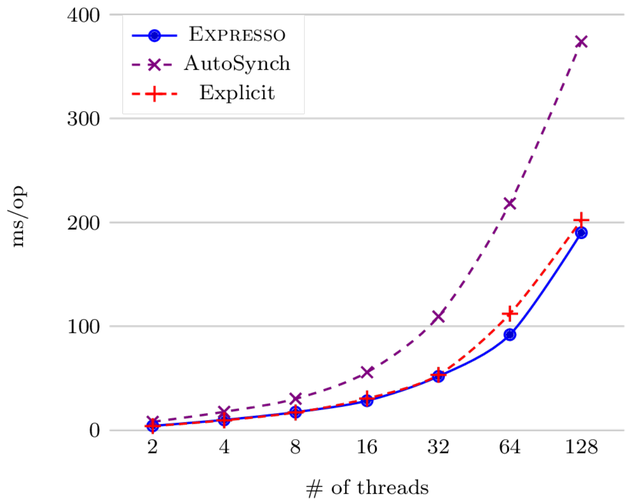

PendingPostQueue (EventBus)



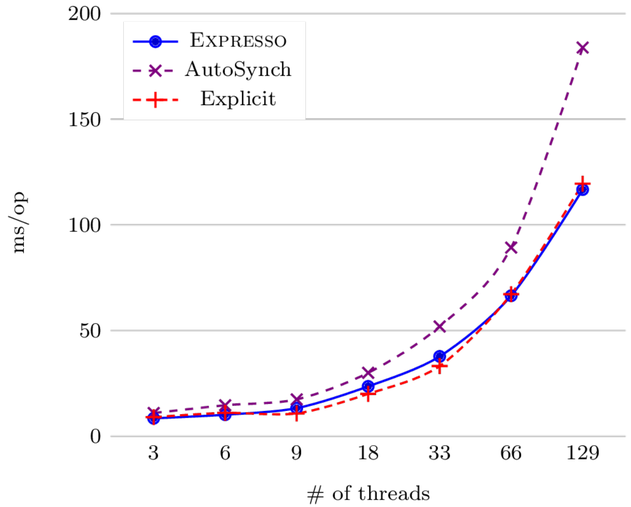

AsyncDispatch (Gradle)



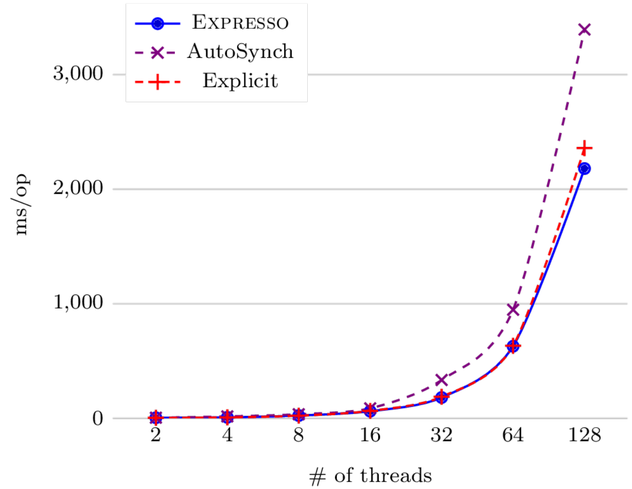

SimpleBlockingDeployment (Gradle)



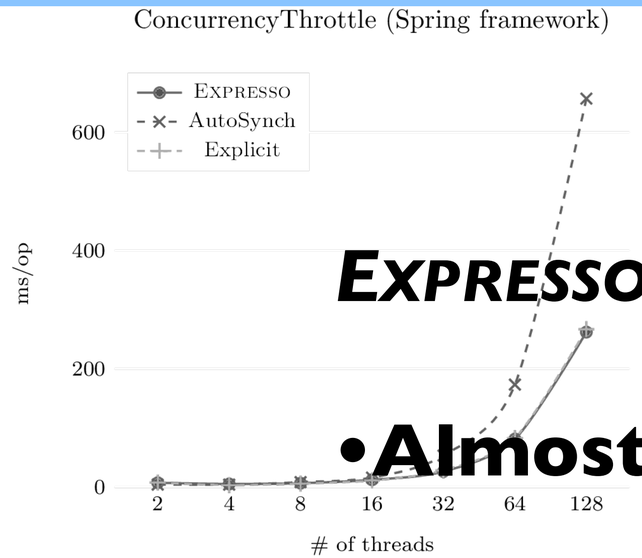

SimpleDecoder (ExoPlayer)

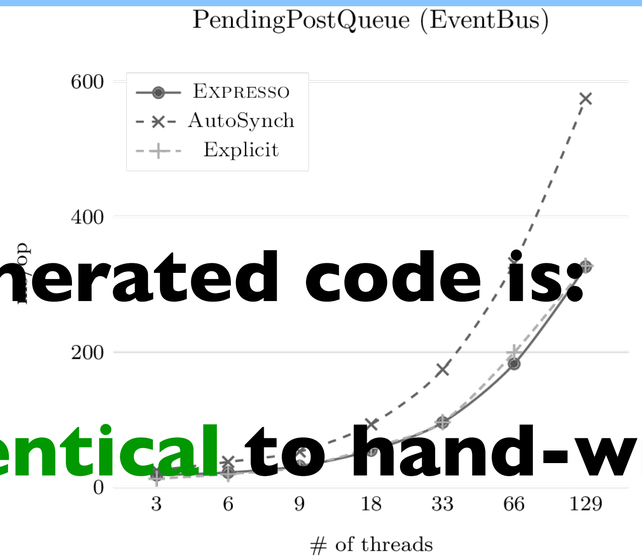


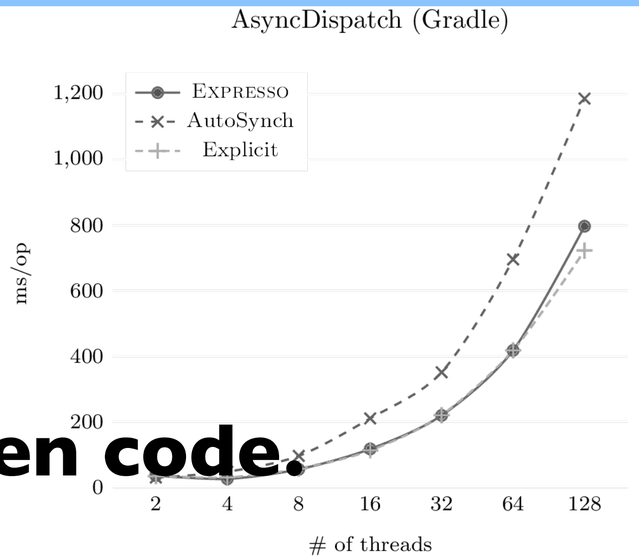

AsyncOperationExecutor (greenDAO)

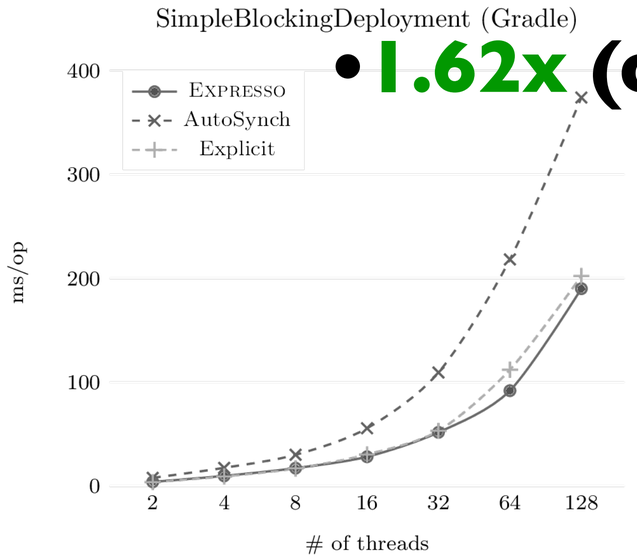


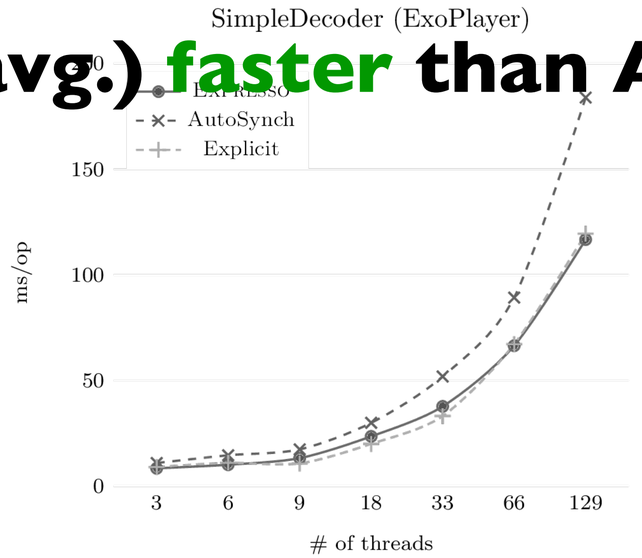
# Results, cont

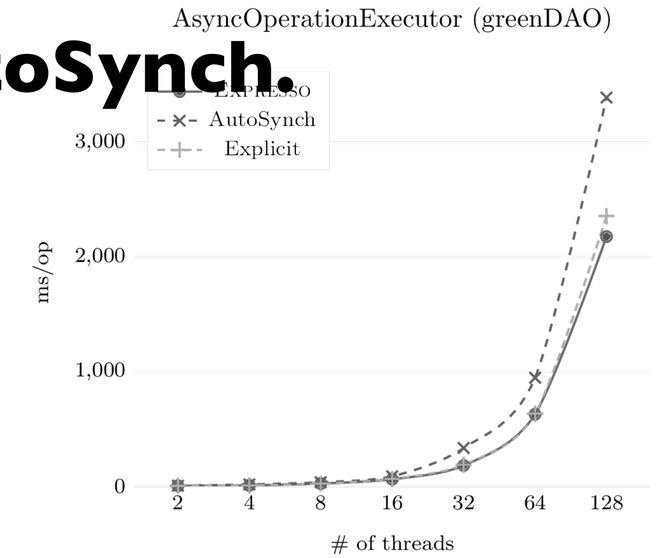



**EXPRESSO generated code is:**

- **Almost identical to hand-written code.**
- **1.62x (on avg.) faster than AutoSynch.**

# Road Map

**Part I:** How to automatically synthesize necessary signaling code

Symbolic Reasoning for Automatic Signal Placement.
Ferles et al, PLDI'18.

**Part II:** How to automatically synthesize efficient locking

Synthesizing Fine-grained Synchronization Protocols for Implicit Monitors.
Ferles et al, OOPSLA'22.

# Road Map

**Part I:** How to automatically synthesize necessary signaling code

Symbolic Reasoning for Automatic Signal Placement.
Ferles et al, PLDI'18.

**Part II:** How to automatically synthesize efficient locking

Synthesizing Fine-grained Synchronization Protocols for Implicit Monitors.
Ferles et al, OOPSLA'22.

# Recall

*Implicit monitor* → 

→ *Explicit monitor*

Expresso uses a *single global lock* acquired by all CCRs to ensure atomicity

***This can be suboptimal in many cases!***

# Part II: Synthesizing Fine-Grained Locking

In some cases, multiple threads can run ***concurrently*** in a monitor without violating the object's atomicity

# Part II: Synthesizing Fine-Grained Locking

In some cases, multiple threads can run ***concurrently*** in a monitor without violating the object's atomicity

Each method *appears* to execute atomically even though execution of methods can be interleaved

# Part II: Synthesizing Fine-Grained Locking

In some cases, multiple threads can run ***concurrently*** in a monitor without violating the object's atomicity

Each method *appears* to execute atomically even though execution of methods can be interleaved

Idea: Improve performance by synthesizing ***fine-grained locking*** in a way that will provide an *illusion* of atomicity

# Example

```
class ArrayBlockingQueue {
  int first = 0, last = 0, count = 0;
  Object[] queue;

  ArrayBlockingQueue(int capacity) {
    assert capacity < 1;
    this.queue = new Object[capacity];
  }

  atomic void put(Object o) {
    waituntil(count < queue.len){
      queue[last] = o;
      last = (last + 1) % queue.len;
      count++;
    }
  }

  atomic Object take() {
    waituntil(count > 0){
      Object r = queue[first];
      queue[first] = null;
      first = (first + 1) % queue.len;
      count--;
      return r;
    }
  }
}
```

Blocking FIFO Queue:

- put blocks if queue is full
- take blocks if queue is empty

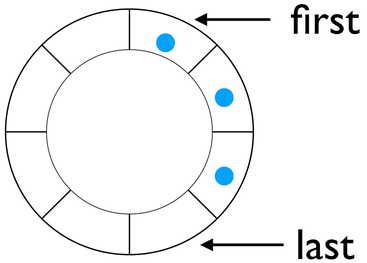
# Example

```
class ArrayBlockingQueue {
  int first = 0, last = 0, count = 0;
  Object[] queue;

  ArrayBlockingQueue(int capacity) {
    assert capacity < 1;
    this.queue = new Object[capacity];
  }

  atomic void put(Object o) {
    waituntil(count < queue.len){
      queue[last] = o;
      last = (last + 1) % queue.len;
      count++;
    }
  }

  atomic Object take() {
    waituntil(count > 0){
      Object r = queue[first];
      queue[first] = null;
      first = (first + 1) % queue.len;
      count--;
      return r;
    }
  }
}
```

Blocking FIFO Queue:

- put blocks if queue is full
- take blocks if queue is empty

Note: put and take can run *concurrently*



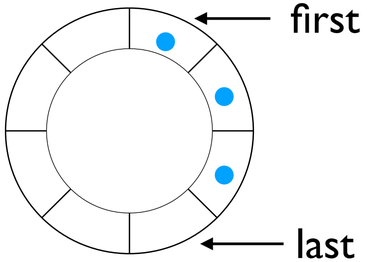
# Example

```
class ArrayBlockingQueue {
  int first = 0, last = 0, count = 0;
  Object[] queue;

  ArrayBlockingQueue(int capacity) {
    assert capacity < 1;
    this.queue = new Object[capacity];
  }

  atomic void put(Object o) {
    waituntil(count < queue.len){
      queue[last] = o;
      last = (last + 1) % queue.len;
      count++;
    }
  }

  atomic Object take() {
    waituntil(count > 0){
      Object r = queue[first];
      queue[first] = null;
      first = (first + 1) % queue.len;
      count--;
      return r;
    }
  }
}
```

Blocking FIFO Queue:

- put blocks if queue is full
- take blocks if queue is empty

Note: put and take can run ***concurrently***




! A global lock would prevent such parallelization opportunities

# Example, cont

```
class ArrayBlockingQueue {
  Lock putLock = new Lock(),
       takeLock = new Lock();
  Condition notFull = putLock.newCondition();
  Condition notEmpty = takeLock.newCondition();

  int first = 0, last = 0; Object[] queue;
  AtomicInteger count = new AtomicInteger(0);

  void put(Object o) {
    putLock.lock();
    while (count.get() == queue.length)
      notFull.await();
    queue[last] = o;
    last = (last + 1) % queue.length;
    count.getAndIncrement();
    putLock.unlock();
    // Signalling code …
  }

  Object take() {
    takeLock.lock();
    while (count.get() == 0)
      notEmpty.await();
    Object r = queue[first];
    queue[first] = null;
    first = (first + 1) % queue.length;
    count.getAndDecrement();
    takeLock.unlock();
    // Signalling code …
    return r;
  }
}
```

**Optimal**:
maximizes parallelism

# Example, cont

```
class ArrayBlockingQueue {
  Lock putLock = new Lock(),
       takeLock = new Lock();
  Condition notFull = putLock.newCondition();
  Condition notEmpty = takeLock.newCondition();

  int first = 0, last = 0; Object[] queue;
  AtomicInteger count = new AtomicInteger(0);

  void put(Object o) {
    putLock.lock();
    while (count.get() == queue.length)
      notFull.await();
    queue[last] = o;
    last = (last + 1) % queue.length;
    count.getAndIncrement();
    putLock.unlock();
    // Signalling code …
  }

  Object take() {
    takeLock.lock();
    while (count.get() == 0)
      notEmpty.await();
    Object r = queue[first];
    queue[first] = null;
    first = (first + 1) % queue.length;
    count.getAndDecrement();
    takeLock.unlock();
    // Signalling code …
    return r;
  }
}
```

two locks

**Optimal**:
maximizes parallelism

# Example, cont

```
class ArrayBlockingQueue {
  Lock putLock = new Lock(),
       takeLock = new Lock();
  Condition notFull = putLock.newCondition();
  Condition notEmpty = takeLock.newCondition();

  int first = 0, last = 0; Object[] queue;
  AtomicInteger count = new AtomicInteger(0);

  void put(Object o) {
    putLock.lock();
    while (count.get() == queue.length)
      notFull.await();
    queue[last] = o;
    last = (last + 1) % queue.length;
    count.getAndIncrement();
    putLock.unlock();
    // Signalling code …
  }

  Object take() {
    takeLock.lock();
    while (count.get() == 0)
      notEmpty.await();
    Object r = queue[first];
    queue[first] = null;
    first = (first + 1) % queue.length;
    count.getAndDecrement();
    takeLock.unlock();
    // Signalling code …
    return r;
  }
}
```

two locks

put uses putLock

**Optimal**:
maximizes parallelism

# Example, cont

```
class ArrayBlockingQueue {
  Lock putLock = new Lock(),
       takeLock = new Lock();
  Condition notFull = putLock.newCondition();
  Condition notEmpty = takeLock.newCondition();

  int first = 0, last = 0; Object[] queue;
  AtomicInteger count = new AtomicInteger(0);

  void put(Object o) {
    putLock.lock();
    while (count.get() == queue.length)
      notFull.await();
    queue[last] = o;
    last = (last + 1) % queue.length;
    count.getAndIncrement();
    putLock.unlock();
    // Signalling code …
  }

  Object take() {
    takeLock.lock();
    while (count.get() == 0)
      notEmpty.await();
    Object r = queue[first];
    queue[first] = null;
    first = (first + 1) % queue.length;
    count.getAndDecrement();
    takeLock.unlock();
    // Signalling code …
    return r;
  }
}
```

two locks

put uses putLock

take uses takeLock

**Optimal**:
maximizes parallelism

# Example, cont

```
class ArrayBlockingQueue {
  Lock putLock = new Lock(),
       takeLock = new Lock();
  Condition notFull = putLock.newCondition();
  Condition notEmpty = takeLock.newCondition();

  int first = 0, last = 0; Object[] queue;
  AtomicInteger count = new AtomicInteger(0);

  void put(Object o) {
    putLock.lock();
    while (count.get() == queue.length)
      notFull.await();
    queue[last] = o;
    last = (last + 1) % queue.length;
    count.getAndIncrement();
    putLock.unlock();
    // Signalling code …
  }

  Object take() {
    takeLock.lock();
    while (count.get() == 0)
      notEmpty.await();
    Object r = queue[first];
    queue[first] = null;
    first = (first + 1) % queue.length;
    count.getAndDecrement();
    takeLock.unlock();
    // Signalling code …
    return r;
  }
}
```

two locks

put uses putLock

take uses takeLock

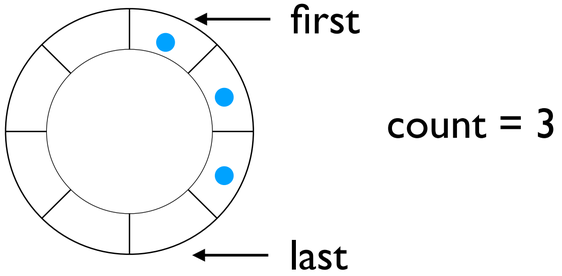
count is an AtomicInteger

**Optimal**:
maximizes parallelism

# Why is this correct?

## (1) Does not contain any races

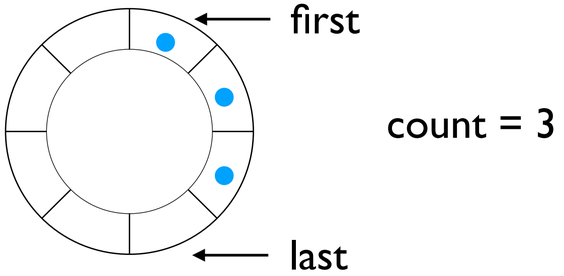
- No race between queue[first] and queue[last]
- Races on count are handled by AtomicInteger



# Why is this correct?

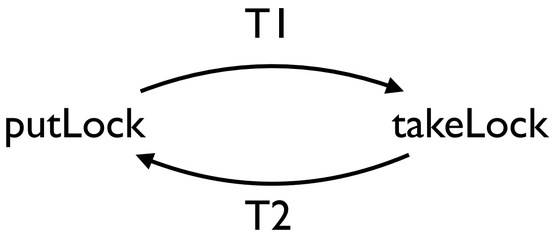
## (1) Does not contain any races

- No race between queue[first] and queue[last]
- Races on count are handled by AtomicInteger




## (2) Monitor is deadlock-free

- No cycles in the lock acquisition order



# Why is this correct? (cont.)

## (3) All operations *appear* to execute as one indivisible unit

- Executions where threads interleave are equivalent to sequential executions.

T1 `wait(count < len)` `queue[last] = o;…` `count++;`

T2 `wait(count > 0)` `r = queue[first];…` `count--;`

# Why is this correct? (cont.)

## (3) All operations *appear* to execute as one indivisible unit

- Executions where threads interleave are equivalent to sequential executions.

T1 `wait(count < len)` `queue[last] = o;…` `count++;`

T2 `wait(count > 0)` `r = queue[first];…` `count--;`

# Why is this correct? (cont.)

## (3) All operations *appear* to execute as one indivisible unit

Safe interleaving

- Executions where threads interleave are equivalent to sequential executions.

T1 `wait(count < len)` `queue[last] = o;…` `count++;`

T2 `wait(count > 0)` `r = queue[first];…` `count--;`

# Why is this correct? (cont.)

## (3) All operations *appear* to execute as one indivisible unit

Safe interleaving

- Executions where threads interleave are equivalent to sequential executions.

T1 `wait(count < len)` `queue[last] = o;…` `count++;`

T2 `wait(count > 0)` `r = queue[first];…` `count--;`

- The locking scheme only allows such *safe interleavings*

Gives an ***illusion*** of atomicity!

# Take-Away from Example

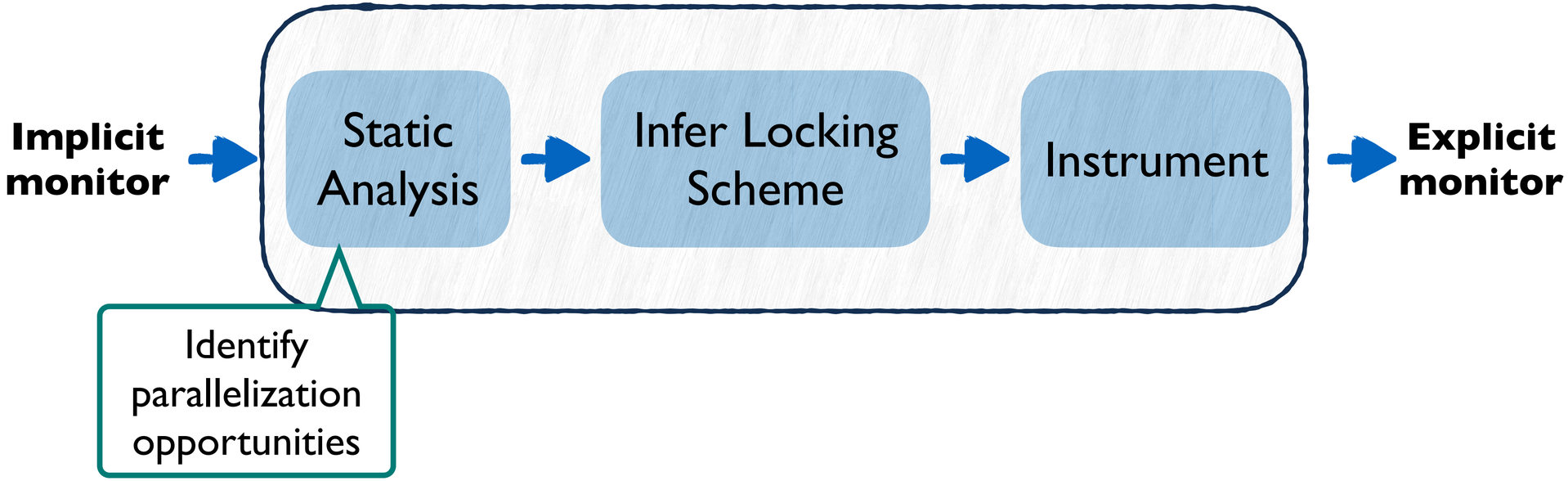
*Goal*: M*aximize* parallelization opportunities & obey correctness requirements:
data-race- and deadlock-freedom + atomicity

# Take-Away from Example

*Goal*: M*aximize* parallelization opportunities & obey correctness requirements: data-race- and deadlock-freedom + atomicity

# Take-Away from Example

*Goal*: M*aximize* parallelization opportunities & obey correctness requirements: data-race- and deadlock-freedom + atomicity

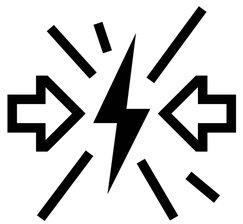


# Static Analysis

**First splits input monitor to disjoint code snippets**

# Static Analysis

**First splits input monitor to disjoint code snippets**

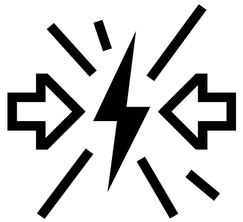


*Data races*

$\{(S, S') \mid \text{Racy}(S, S')\}$

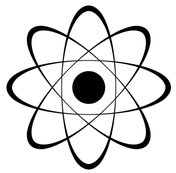
# Static Analysis

**First splits input monitor to disjoint code snippets**



*Data races*

$\{(S, S') \mid \text{Racy}(S, S')\}$

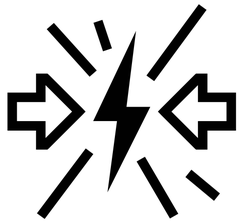


*Fields that can be implemented using atomic type*

$\{f \mid \text{AtomicTypeExists}(\text{Type}(f), \text{Updates}(f))\}$

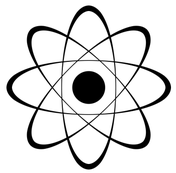
# Static Analysis

**First splits input monitor to disjoint code snippets**



*Data races*

$\{(S, S') \mid \text{Racy}(S, S')\}$



*Fields that can be implemented using atomic type*

$\{f \mid \text{AtomicTypeExists}(\text{Type}(f), \text{Updates}(f))\}$


*Safe interleaving opportunities*

$\{(S, S1, S2) \mid \text{SafeInterleaving}(S, S1, S2)\}$

# Detecting Safe Interleavings

**Theorem:** S can be interleaved in between S1 and S2 if it ***left-commutes*** with every ***predecessor*** of S1 and ***right-commutes*** with every ***successor*** of S2

# Detecting Safe Interleavings

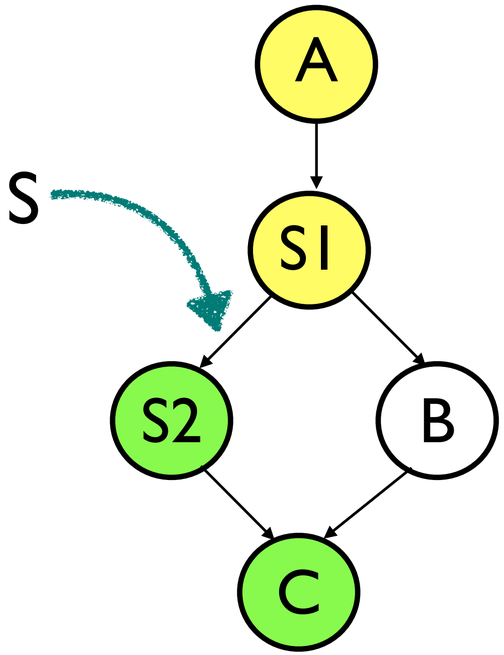
**Theorem:** S can be interleaved in between S1 and S2 if it ***left-commutes*** with every ***predecessor*** of S1 and ***right-commutes*** with every ***successor*** of S2

X left-commutes with Y, if whenever X executes right after Y, state is same as when X executes before Y

$Y; X \equiv X; Y$

# Detecting Safe Interleavings

**Theorem:** S can be interleaved in between S1 and S2 if it ***left-commutes*** with every ***predecessor*** of S1 and ***right-commutes*** with every ***successor*** of S2

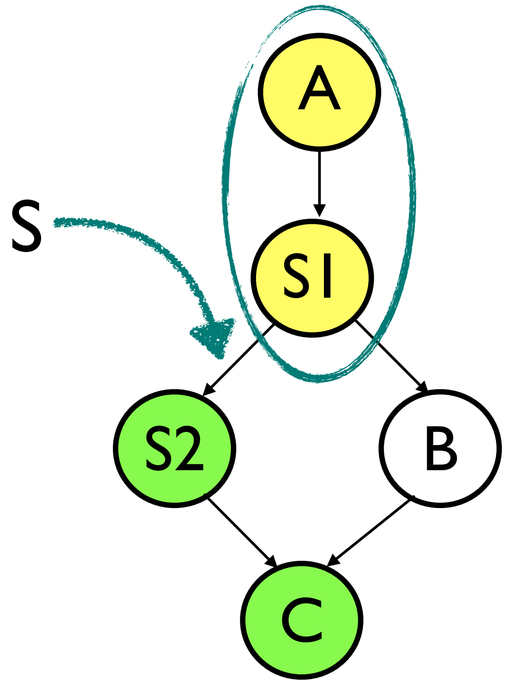



X left-commutes with Y, if whenever X executes right after Y, state is same as when X executes before Y

$Y; X \equiv X; Y$

# Detecting Safe Interleavings

**Theorem:** S can be interleaved in between S1 and S2 if it ***left-commutes*** with every ***predecessor*** of S1 and ***right-commutes*** with every ***successor*** of S2



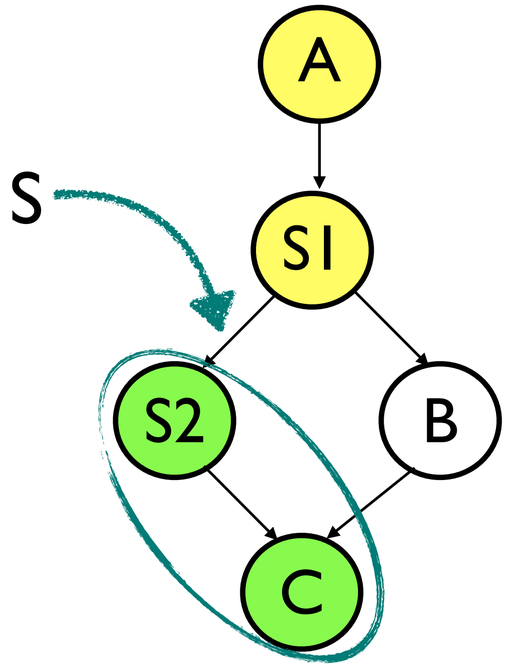

X left-commutes with Y, if whenever X executes right after Y, state is same as when X executes before Y

$Y; X \equiv X; Y$

$A;\ S \equiv S; A$
$S1;\ S \equiv S; S1$

# Detecting Safe Interleavings

**Theorem:** S can be interleaved in between S1 and S2 if it ***left-commutes*** with every ***predecessor*** of S1 and ***right-commutes*** with every ***successor*** of S2




X left-commutes with Y, if whenever X executes right after Y, state is same as when X executes before Y

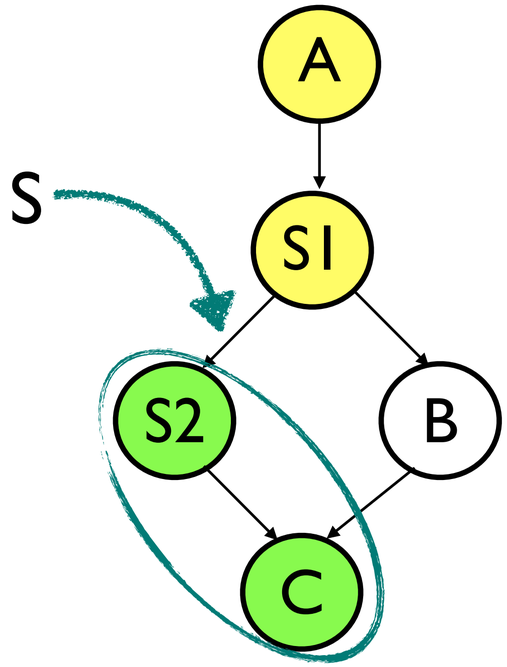
$Y; X \equiv X; Y$

$A;\ S \equiv S; A$
$S1;\ S \equiv S; S1$

$S;\ S2 \equiv S2; S$
$S;\ C \equiv C; S$

# Detecting Safe Interleavings

**Theorem:** S can be interleaved in between S1 and S2 if it ***left-commutes*** with every ***predecessor*** of S1 and ***right-commutes*** with every ***successor*** of S2




X left-commutes with Y, if whenever X executes right after Y, state is same as when X executes before Y

$Y; X \equiv X; Y$

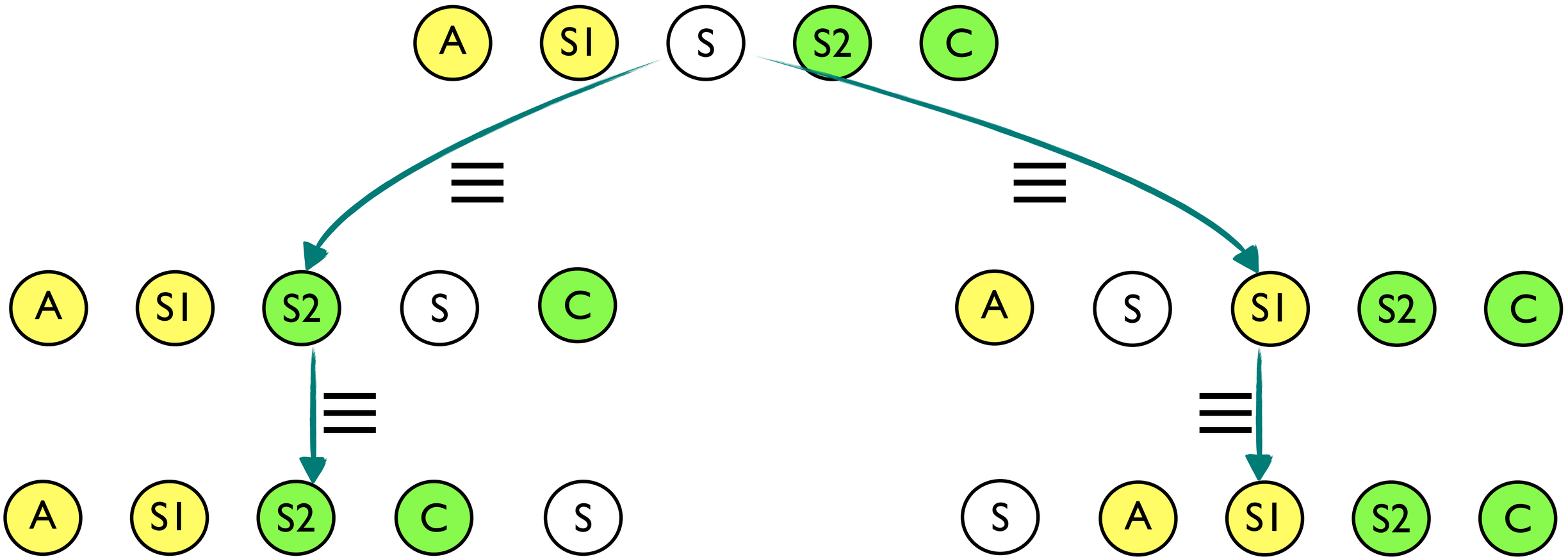
$A;\ S \equiv S; A$
$S1;\ S \equiv S; S1$

$S;\ S2 \equiv S2; S$
$S;\ C \equiv C; S$

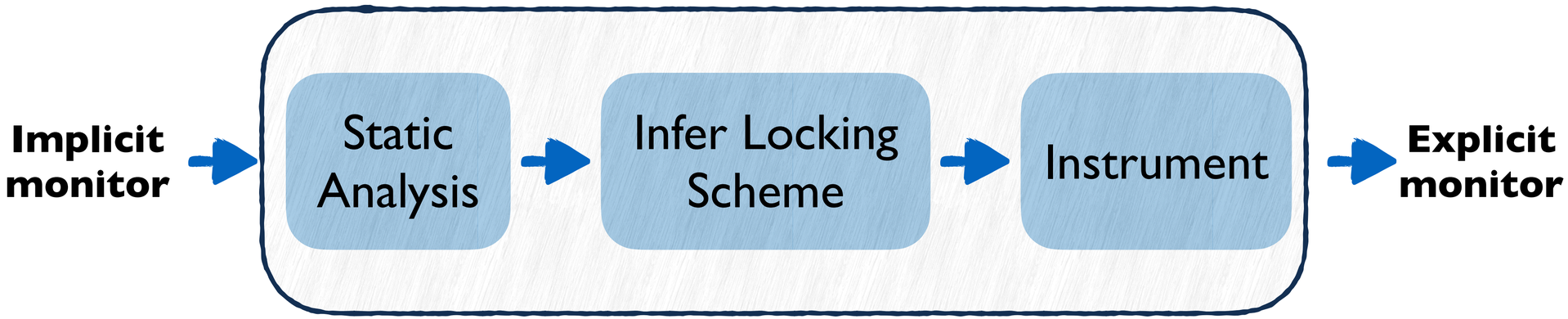
Program equivalence check

# Detecting Safe Interleavings

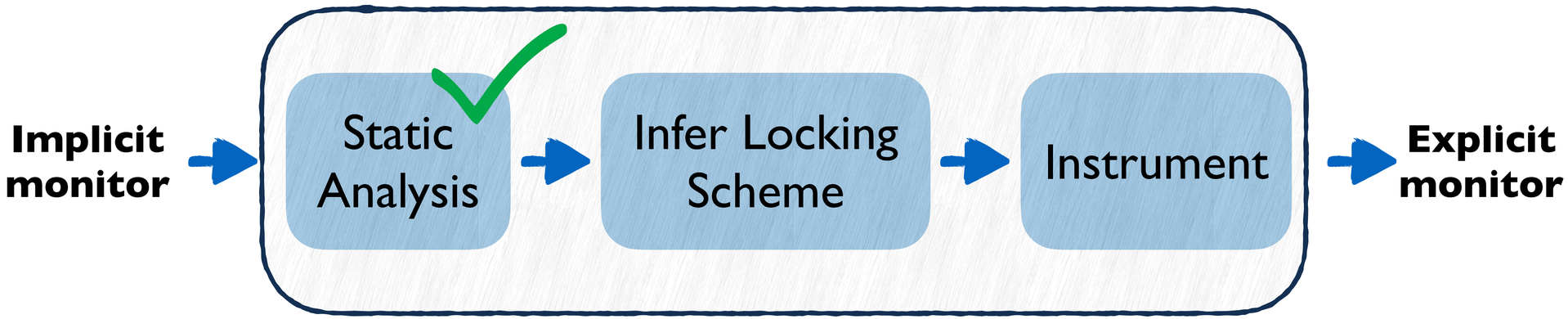
# Detecting Safe Interleavings



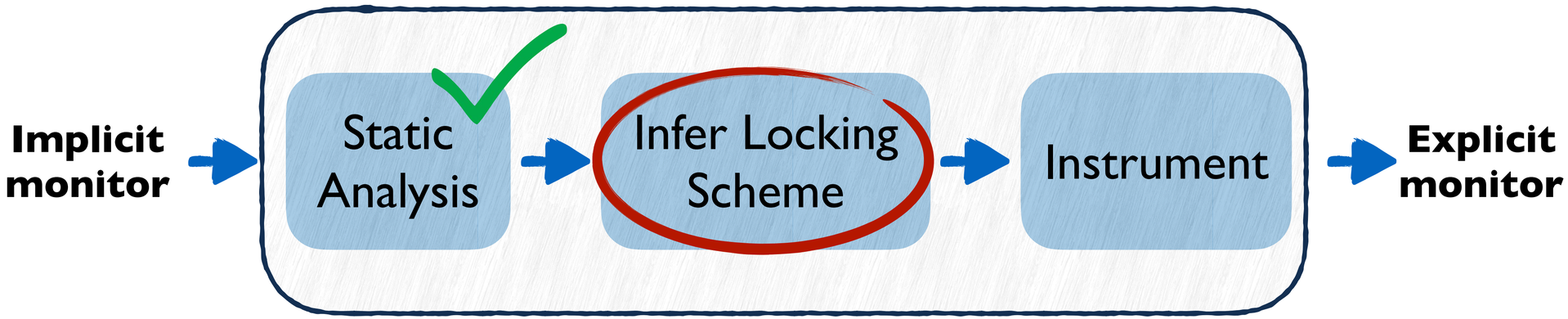
# Synthesis Framework



# Synthesis Framework



# Synthesis Framework



# Synthesizing Optimal Locking Scheme

**Locking Scheme Requirement**

# Synthesizing Optimal Locking Scheme

**Locking Scheme Requirement**

Maximize parallelization
among monitor methods

# Synthesizing Optimal Locking Scheme

**Locking Scheme Requirement**

Satisfy correctness requirements
(data race/deadlock freedom & atomicity)

+

Maximize parallelization
among monitor methods

# Synthesizing Optimal Locking Scheme

**Locking Scheme Requirement**

Satisfy correctness requirements
(data race/deadlock freedom & atomicity)

**+**

Maximize parallelization
among monitor methods

**Optimization problem**

# Synthesizing Optimal Locking Scheme

**Locking Scheme Requirement**

Satisfy correctness requirements
(data race/deadlock freedom & atomicity)

+

Maximize parallelization
among monitor methods

**Optimization problem** → **MaxSAT Encoding**

# Synthesizing Optimal Locking Scheme

**Locking Scheme Requirement**

Satisfy correctness requirements
(data race/deadlock freedom & atomicity)

+

Maximize parallelization
among monitor methods

**Optimization problem** → **MaxSAT Encoding**

Set of Boolean
Variables

# Synthesizing Optimal Locking Scheme

**Locking Scheme Requirement**

Satisfy correctness requirements
(data race/deadlock freedom & atomicity)

+

Maximize parallelization
among monitor methods

**Optimization problem** → **MaxSAT Encoding**

Set of Boolean Variables

+

Set of Hard Constraints

# Synthesizing Optimal Locking Scheme

**Locking Scheme Requirement**

Satisfy correctness requirements
(data race/deadlock freedom & atomicity)

+

Maximize parallelization
among monitor methods

**Optimization problem** → **MaxSAT Encoding**

Set of Boolean Variables

+

Set of
Hard Constraints

**All** Must be Satisfied

# Synthesizing Optimal Locking Scheme

**Locking Scheme Requirement**

Satisfy correctness requirements
(data race/deadlock freedom & atomicity)

+

Maximize parallelization
among monitor methods

**Optimization problem** → **MaxSAT Encoding**

Set of Boolean Variables

+

Set of
Hard Constraints

+

Set of
Soft Constraints

**All** Must be Satisfied

# Synthesizing Optimal Locking Scheme

**Locking Scheme Requirement**

Satisfy correctness requirements
(data race/deadlock freedom & atomicity)

+

Maximize parallelization
among monitor methods

**Optimization problem** → **MaxSAT Encoding**

Set of Boolean Variables

+

Set of
Hard Constraints

**All** Must be Satisfied

+

Set of
Soft Constraints

Max set that **can** be satisfied

# Synthesizing Optimal Locking Scheme

**Locking Scheme Requirement**

Satisfy correctness requirements
(data race/deadlock freedom & atomicity)

+

Maximize parallelization
among monitor methods

**Optimization problem** → **MaxSAT Encoding**

Map Scheme to Boolean variables

Set of Boolean Variables

+

Set of Hard Constraints

**All** Must be Satisfied

+

Set of Soft Constraints

Max set that **can** be satisfied

# Synthesizing Optimal Locking Scheme

**Locking Scheme Requirement**

Satisfy correctness requirements
(data race/deadlock freedom & atomicity)

+

Maximize parallelization
among monitor methods

**Optimization problem** → **MaxSAT Encoding**

Map Scheme to Boolean variables

Set of Boolean Variables

+

Set of Hard Constraints

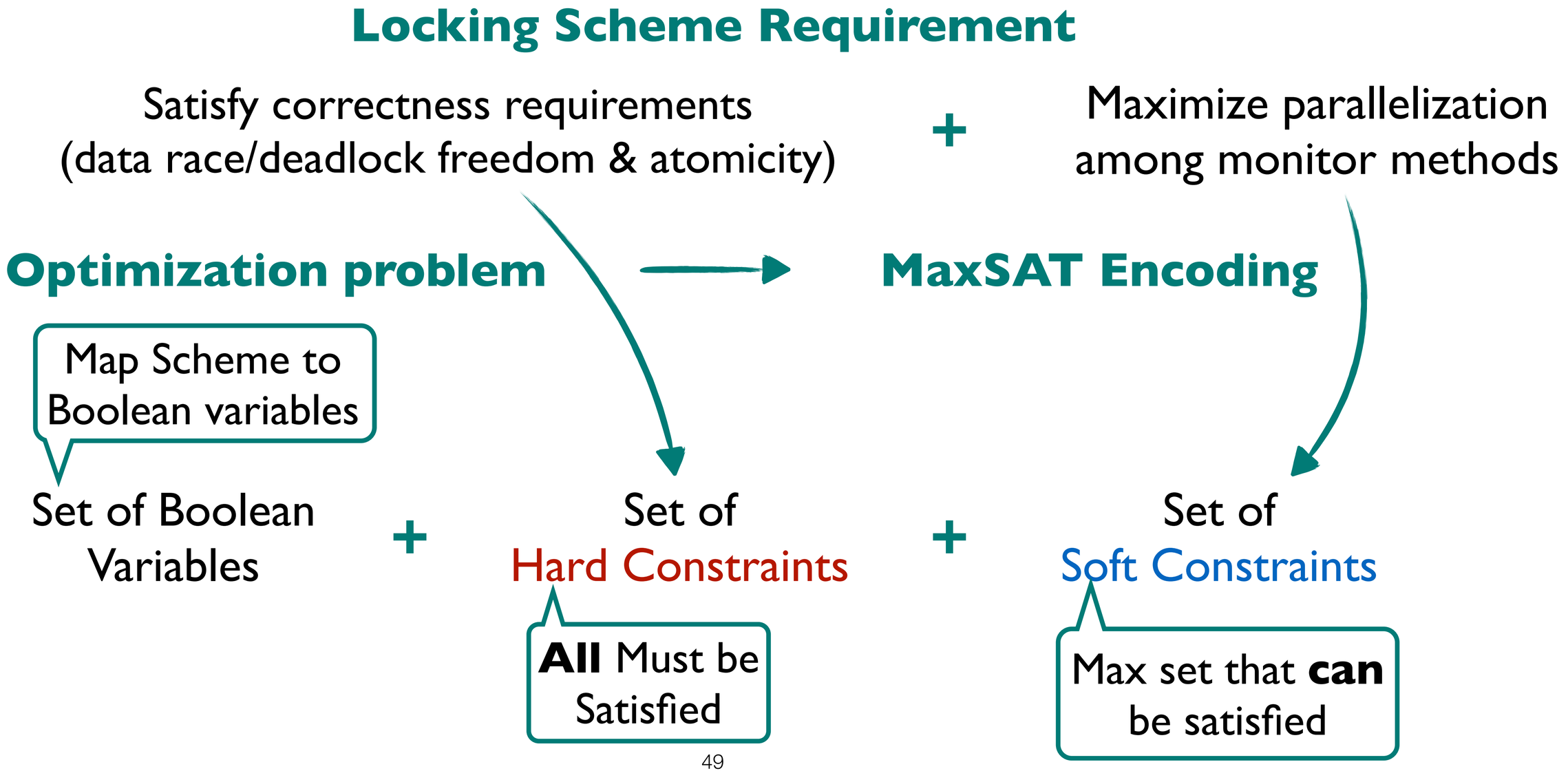
**All** Must be Satisfied

+

Set of Soft Constraints

Max set that **can** be satisfied

# Synthesizing Optimal Locking Scheme



# MaxSAT Encoding Variables

Introduce two types of boolean variables that represent the locking scheme.

# MaxSAT Encoding Variables

Introduce two types of boolean variables that represent the locking scheme.

$h_s^l$ Indicates that code snippet S must hold lock l

$$\mathscr{L}(S) = \{l \mid h_s^l\}$$

Lockset for snippet S

# MaxSAT Encoding Variables

Introduce two types of boolean variables that represent the locking scheme.

$h_s^l$ Indicates that code snippet S must hold lock l

$$\mathscr{L}(S) = \{l \mid h_s^l\}$$

Lockset for snippet S

$\alpha_f$ Indicates that field f must be implemented as atomic

$$f \in PossibleAtomicFields(M)$$

# Correctness Requirements
## (Hard constraints)

Data-race freedom

Atomicity

# Correctness Requirements (Hard constraints)

Data-race freedom

Resolve data races with locks

If S1 and S2 have a data race, then generate constraint:

$$\mathscr{L}(S1) \cap \mathscr{L}(S2) \neq \emptyset$$

Atomicity

# Correctness Requirements (Hard constraints)

## Data-race freedom

Resolve data races with locks

If S1 and S2 have a data race, then generate constraint:

$$\mathscr{L}(S1) \cap \mathscr{L}(S2) \neq \emptyset$$

## Atomicity

Only permit safe interleavings

If (S, S1, S2) is not a safe interleaving, then generate the following constraint

$$\mathscr{L}(S) \cap \mathscr{L}(S_1) \cap \mathscr{L}(S_2) \neq \emptyset$$

# Performance Objectives
## (Soft constraints)

Encode performance objective
using soft constraints

# Performance Objectives
## (Soft constraints)

Encode performance objective using soft constraints

Maximize number of code snippets that can run in parallel

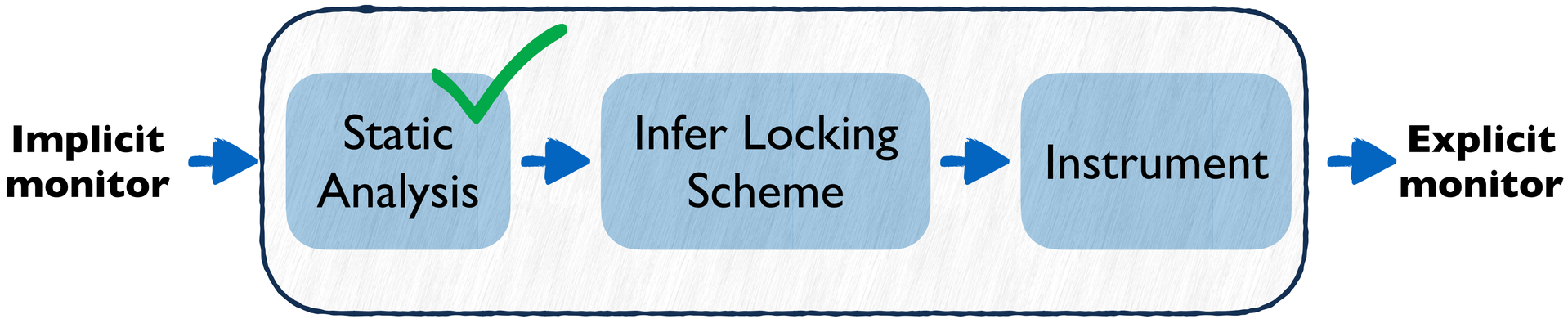
# Performance Objectives
## (Soft constraints)

Encode performance objective using soft constraints

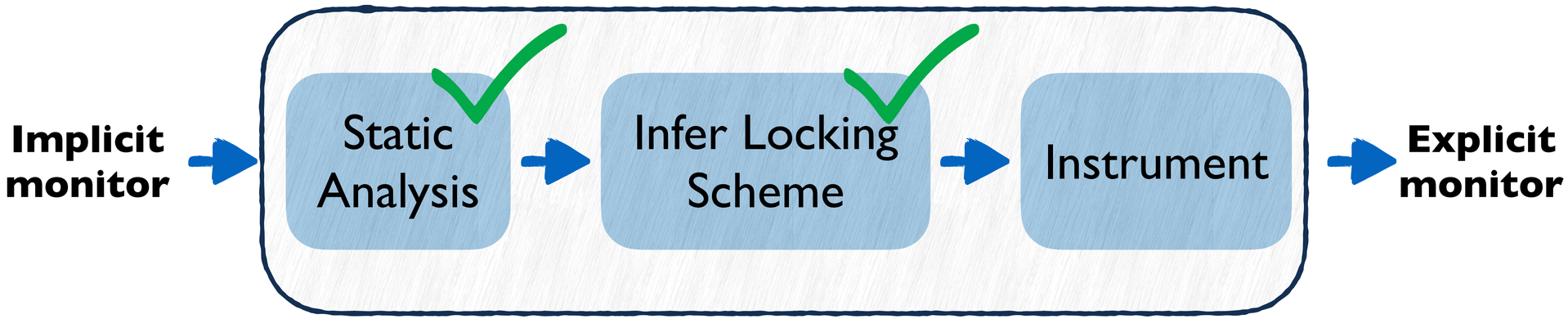
Maximize number of code snippets that can run in parallel

Minimize number of locks and atomic fields

# Synthesis Framework

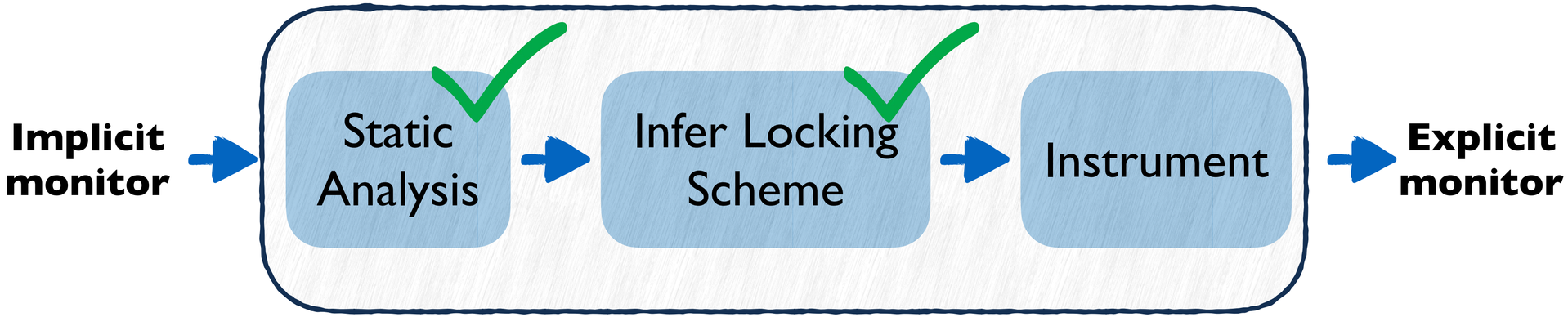


# Synthesis Framework

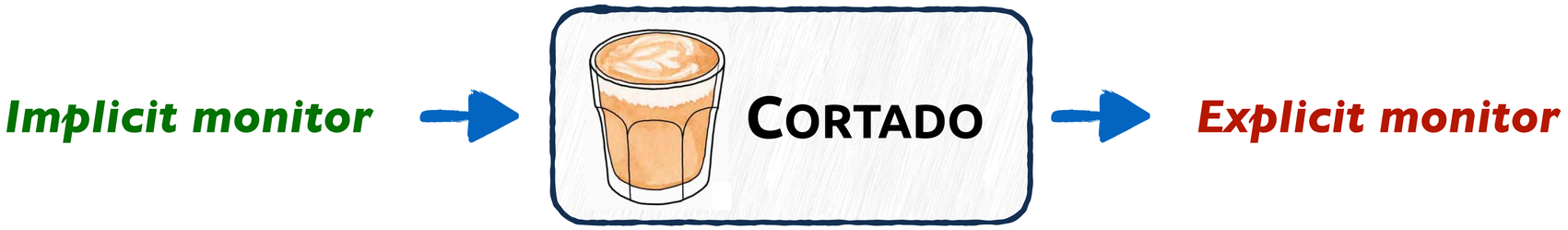


# Synthesis Framework

Key Idea: Statically detect parallelism and reduce locking scheme inference to MaxSAT

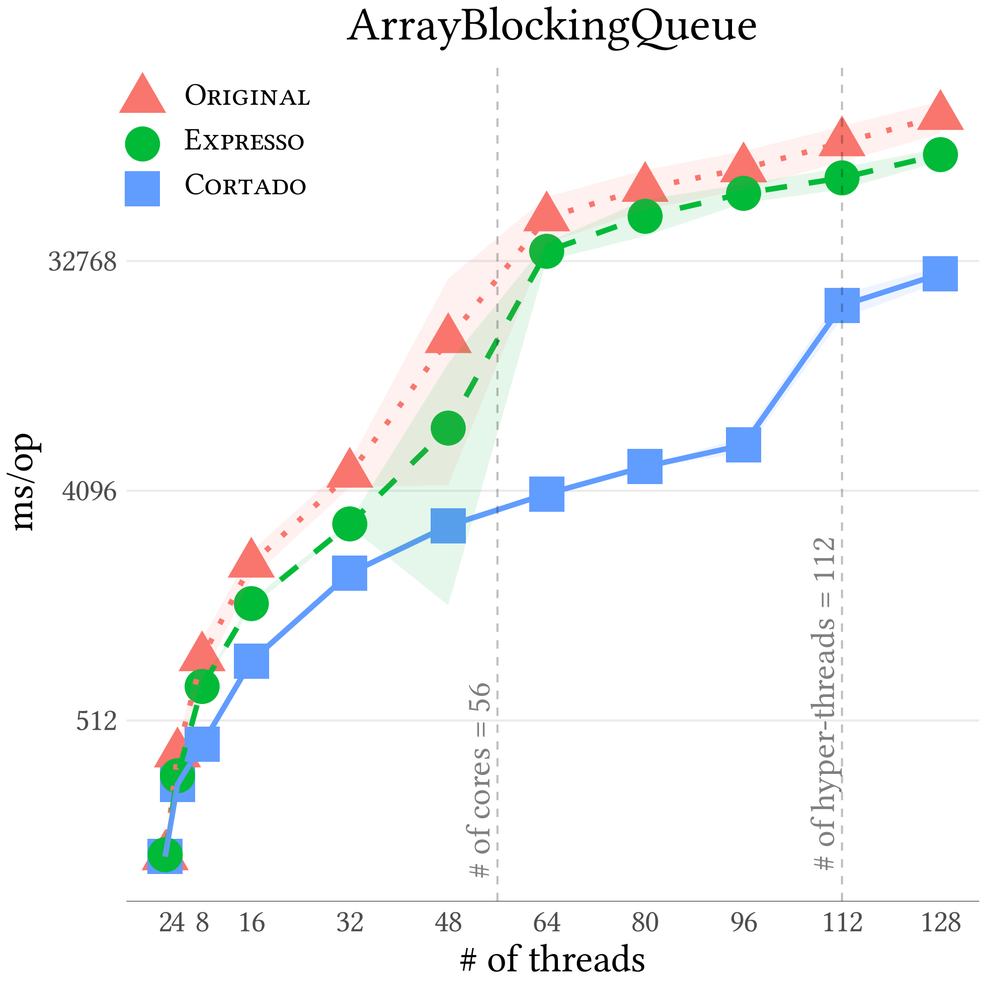


# Implementation & Evaluation



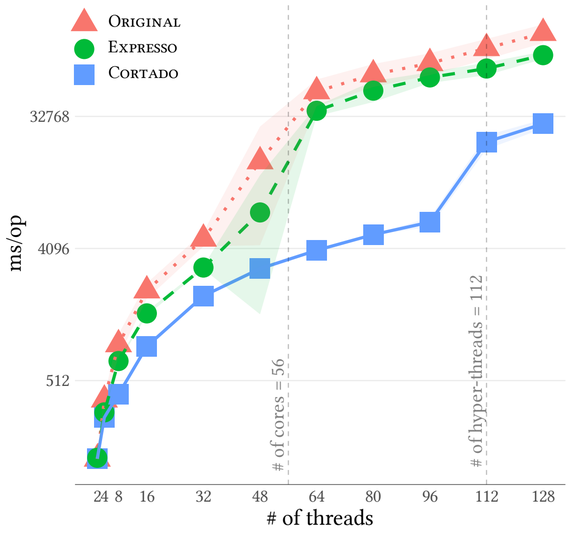

- Evaluated Cortado on monitors with parallelization opportunities
- Manually converted explicit monitor to implicit version
- Compared performance of Cortado-synthesized code against:
  - Manual version as collected from the project
  - The version generated by Expresso (uses a global lock)

# Results

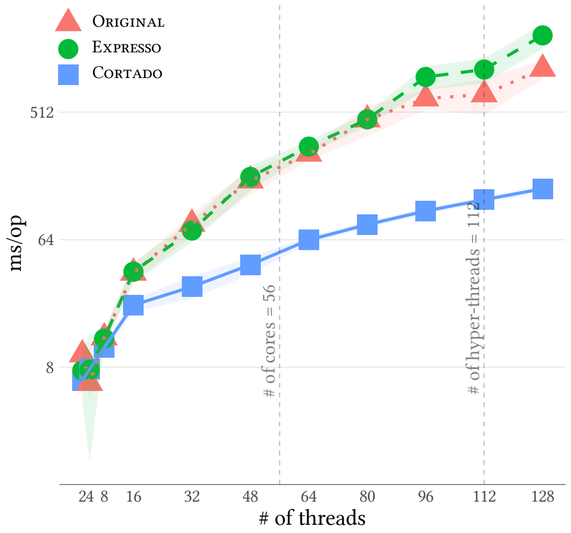

ArrayBlockingQueue
Original
Expresso
Cortado
32768
4096
512
ms/op
# of cores = 56
# of hyper-threads = 112
24 8 16 32 48 64 80 96 112 128
# of threads

# Results

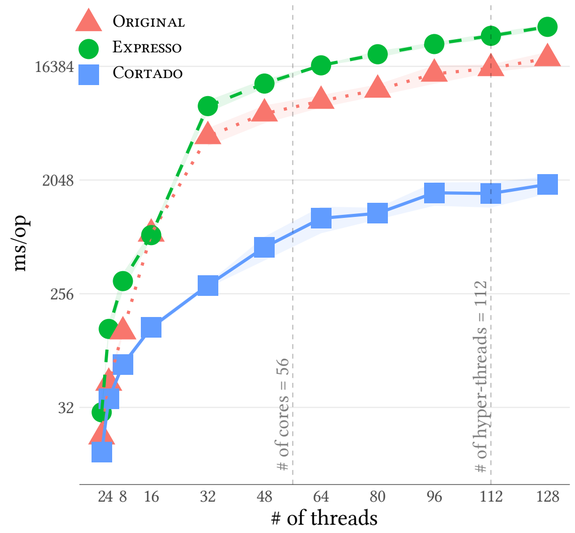
ArrayBlockingQueue



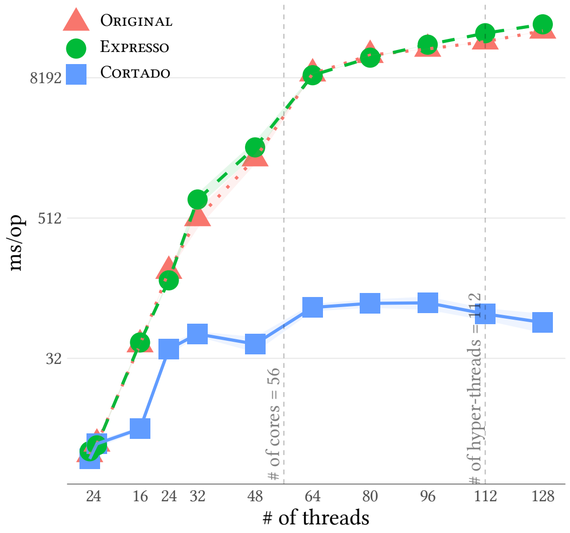

ConcurrencyThrottleSupport



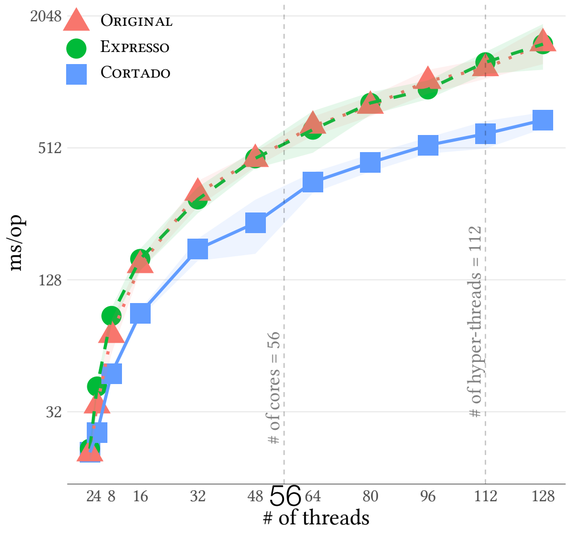

SinkQueue



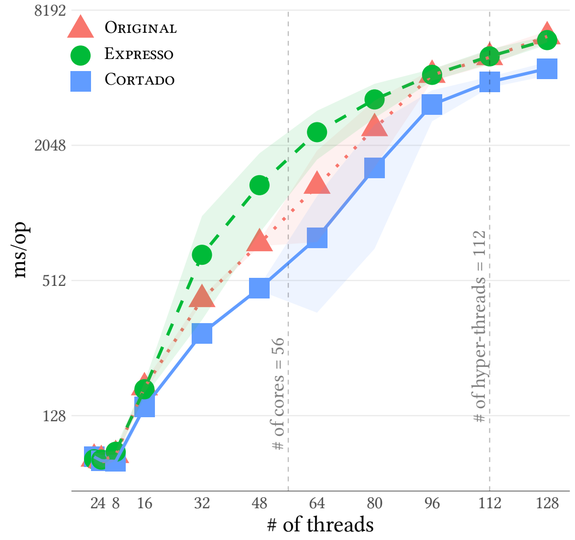

JobWrapper



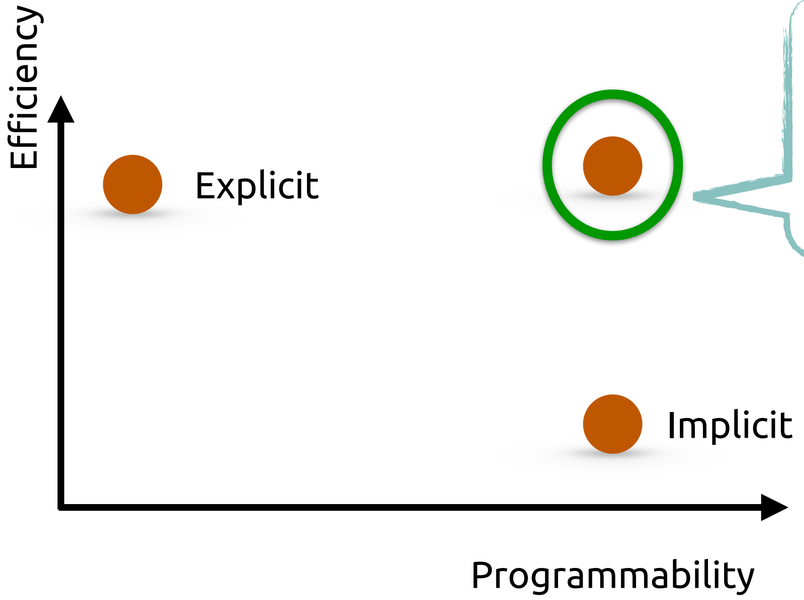

PausableThreadPoolExecutor



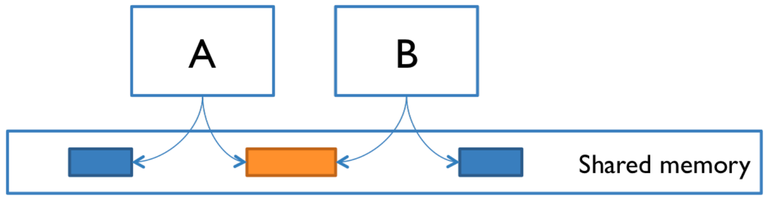

CountableThreadPool



# Summary




Provide best of both worlds wrt correctness & performance




*Implicit monitor* → EXPRESSO → *Explicit monitor*

*Implicit monitor* → CORTADO → *Explicit monitor*

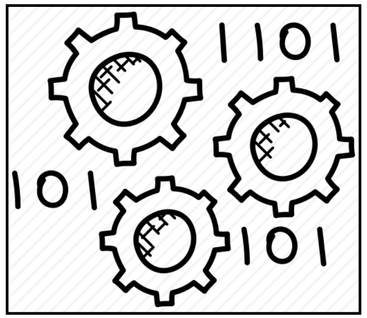
A set of formal methods techniques for simplifying concurrent programming

# Research Highlights

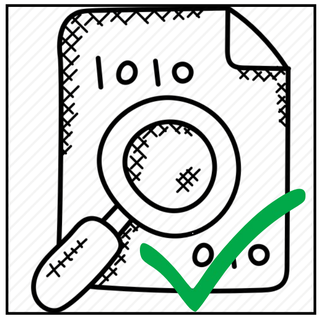
**Research Goal:** ***Formal methods for correct, efficient, and secure code!***

**Simplifying Concurrent Programming**
[PLDI'18,OOPSLA'22]

Synthesis



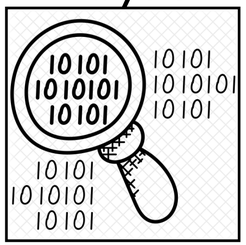

Verification




**Preventing Security Vulnerabilities**
[FSE'18,VSTTE'19,S&P'21]

**Verifying Correct Usage of APIs**
[POPL'21]

Core Program Analysis

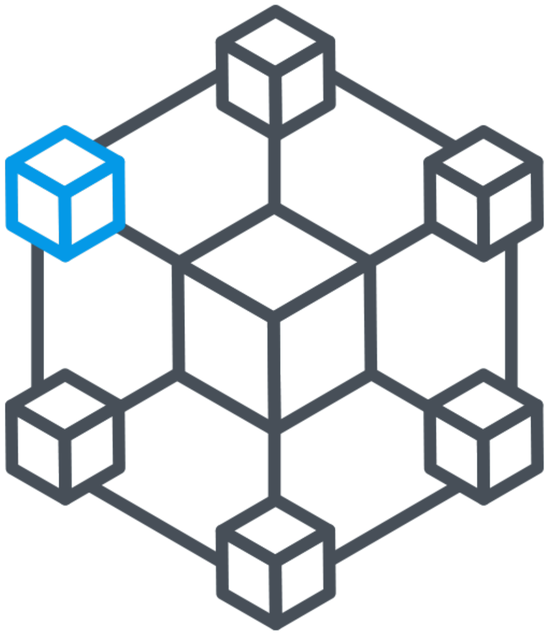



**Improve Scalability & Precision**
[FSE'17,SOAP'17,CC'18]

# Verification Techniques

## Preventing Security Vulnerabilities In Smart Contracts

**Programs running on a decentralized ledger**

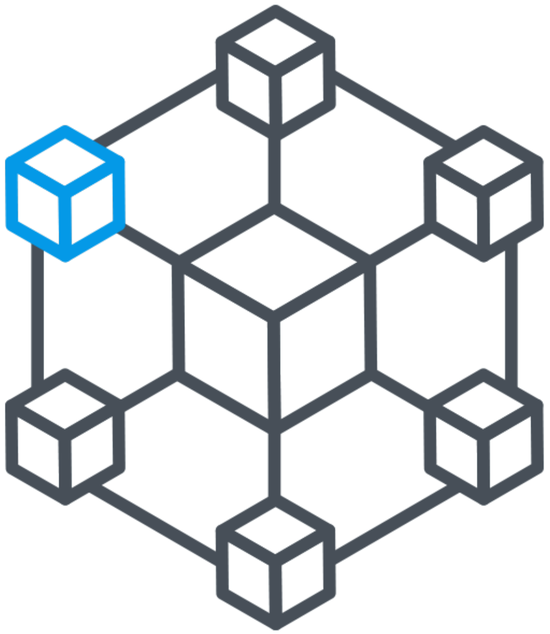


- *Motivation*: They manage a huge mount of user monetary funds
- Vulnerabilities in a contract can severely harm stakeholders

# Verification Techniques

## Preventing Security Vulnerabilities In Smart Contracts

**Programs running on a decentralized ledger**



- *Motivation*: They manage a huge mount of user monetary funds
- Vulnerabilities in a contract can severely harm stakeholders
- Verification to the rescue!
  - ***VeriSol[VSTTE'19]***: contract won't reach an undesired state
  - ***SmartPulse[S&P'21]***: contract will reach a desired state

# Verification Techniques

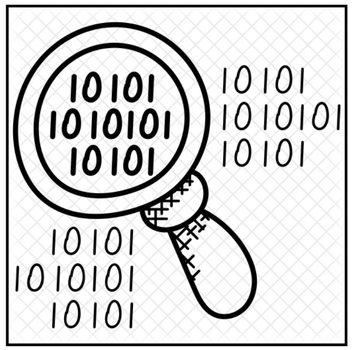
## Verifying Correct API Usage


- *Motivation*: Developers tend to misuse APIs written by others
- Prior work cannot handle important classes of APIs

  Reentrant lock API: $\texttt{l.acquire()}^n\ \texttt{l.release()}^n$

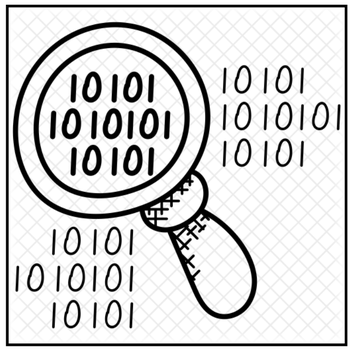
- ***CFPChecker[POPL'21]***: closes the theoretical gap for such APIs

# Core Program Analysis

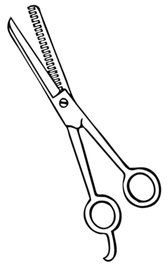



- *Motivation:* Verifiers and Synthesizers need to understand your code!
- Better program analysis tools imply better verifiers and synthesizers

# Core Program Analysis




- *Motivation*: Verifiers and Synthesizers need to understand your code!
- Better program analysis tools imply better verifiers and synthesizers



- ***Timmer[FSE'17]***: Program transformation that eliminates safe paths
- Transparently increases precision and scalability of other tools

# Future Directions

Can we push at the boundaries of formal method applications and further boost developers' confidence in their code?

**Direction I**

Verification & Synthesis Techniques in More Contexts

**Direction II**

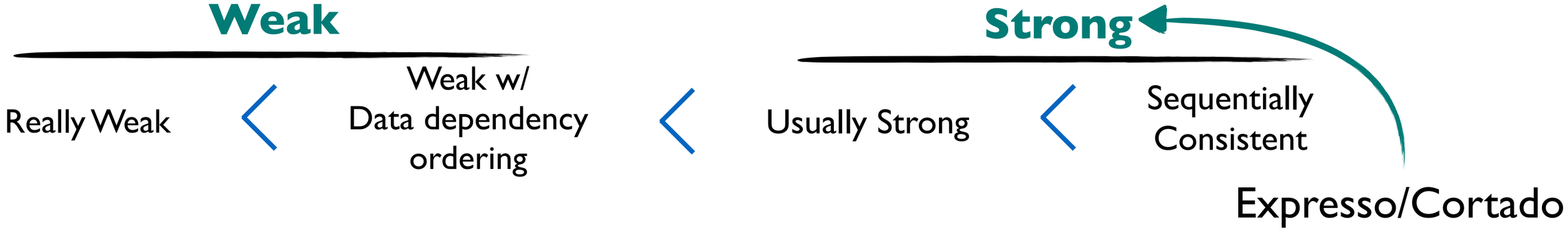
Make Formal Method Techniques More Accessible

# Simplify More Aspects of Shared Memory Concurrency

**Several Concurrent Programming Setups Based on Guarantees Provided by Underlying Memory System**

# Simplify More Aspects of Shared Memory Concurrency

**Several Concurrent Programming Setups Based on Guarantees Provided by Underlying Memory System**



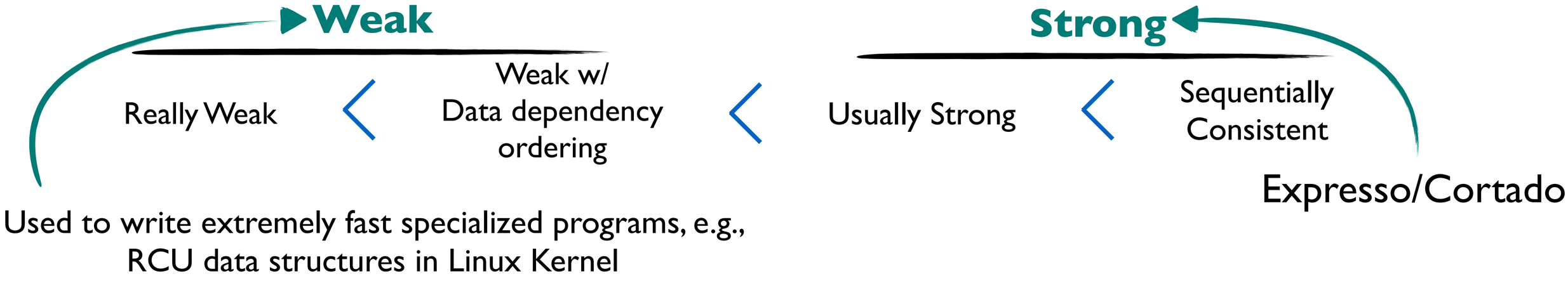
# Simplify More Aspects of Shared Memory Concurrency

**Several Concurrent Programming Setups Based on Guarantees Provided by Underlying Memory System**


Used to write extremely fast specialized programs, e.g., RCU data structures in Linux Kernel

# Simplify More Aspects of Shared Memory Concurrency

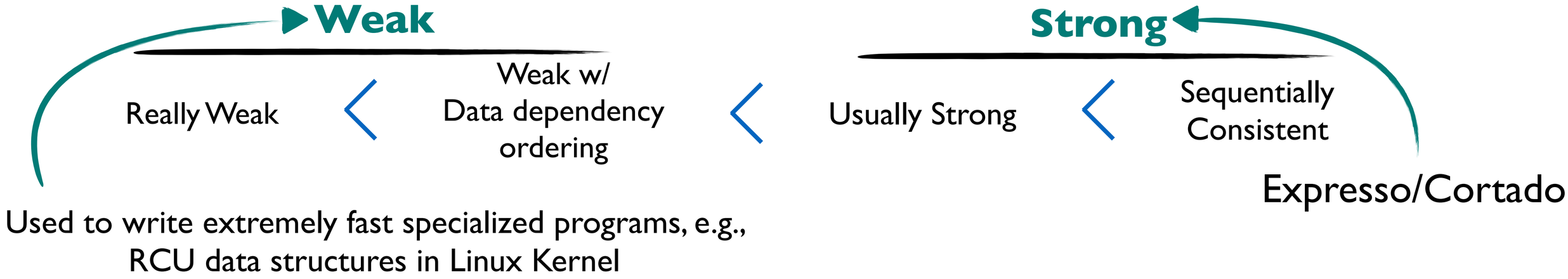
**Several Concurrent Programming Setups Based on Guarantees Provided by Underlying Memory System**




- Challenging to prove correctness of programs written with weak memory primitives
  - Many implementations have manually-written proofs

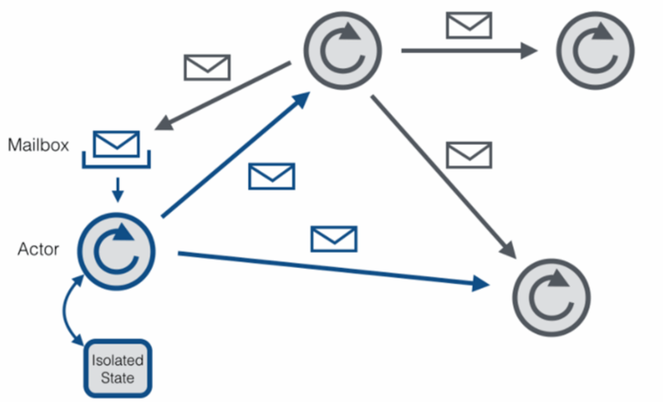
# Simplify More Aspects of Shared Memory Concurrency

**Several Concurrent Programming Setups Based on Guarantees Provided by Underlying Memory System**



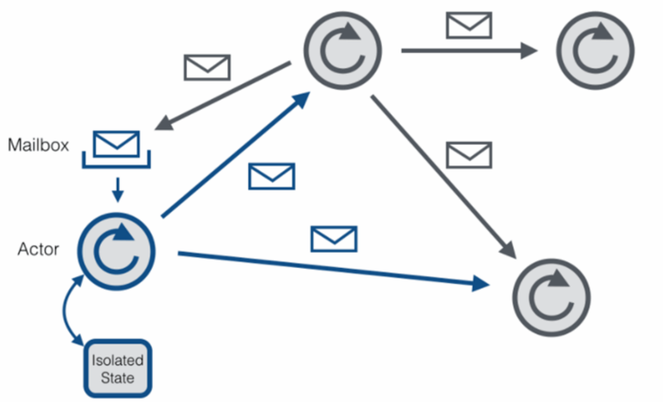

- Challenging to prove correctness of programs written with weak memory primitives
  - Many implementations have manually-written proofs
- ***Solutions:*** 1. More automation in verification
  2. Synthesize concurrent programs w/ weak memory primitives

# Simplify Other Concurrency Models




- Several other concurrency models based on message passing, e.g., MPI, Actor Model, etc.
- Nodes (i.e., threads) must conform to a communication protocol
  - Developers fail to implement the protocol correctly

# Simplify Other Concurrency Models




- Several other concurrency models based on message passing, e.g., MPI, Actor Model, etc.
- Nodes (i.e., threads) must conform to a communication protocol
  - Developers fail to implement the protocol correctly

- Automatically verify conformance to protocol given its formal description
- Automatically synthesize message passing programs from their specification

# Make Formal Methods More Accessible


- Users must be familiar with complex concepts and write specifications
- Even experts can write buggy specifications

# Make Formal Methods More Accessible

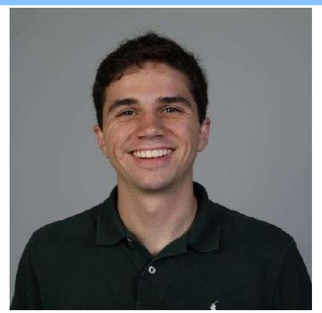

- Users must be familiar with complex concepts and write specifications
- Even experts can write buggy specifications

- Develop techniques that help developers fix wrong specifications
- Synthesize specifications from codebase artifacts (e.g., test cases)

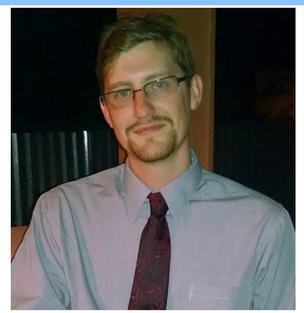
# Acknowledgements

Isil Dillig

Yannis Smaragdakis



Ben Sepanski



Jon Stephens

Jacob Van Geffen

Yuepeng Wang

James Bornholt

Shuvendu Lahiri

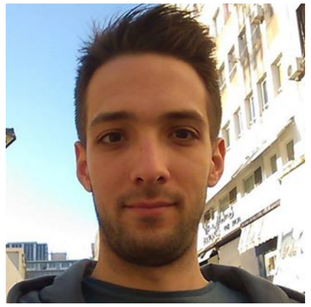
Yu Feng

Jiayi Wei

Ben Mariano

Rahul Krishnan

George Balatshouras



George Kastrinis

Maria Christakis

Valentin Wüstholz

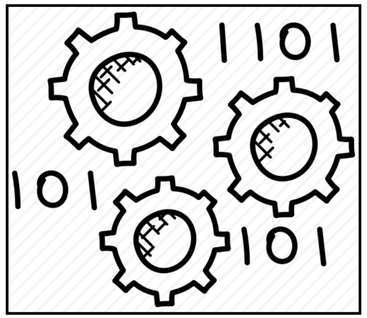
...

# Questions?

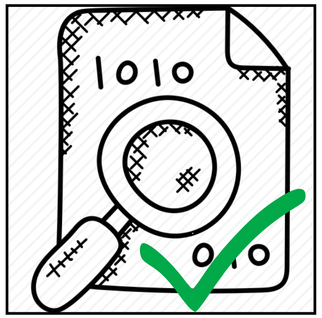
**Research Goal:** ***Formal methods for correct, efficient, and secure code!***

Synthesis

**Simplifying Concurrent Programming**
[PLDI'18,OOPSLA'22]



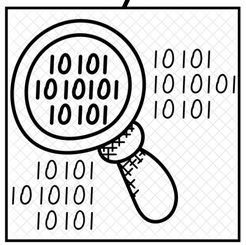

Verification



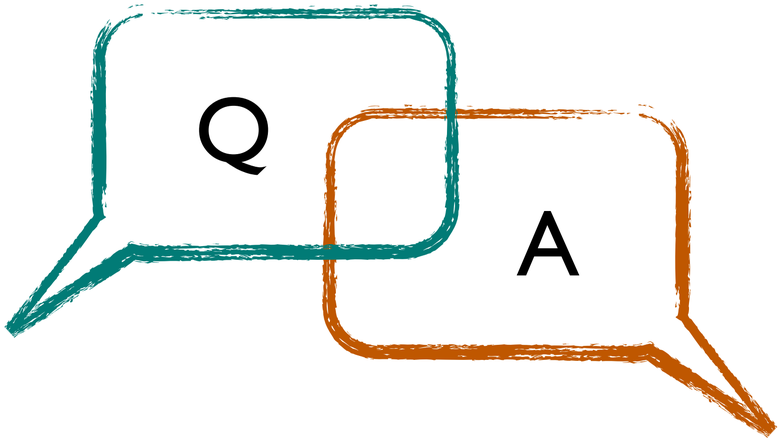

**Preventing Security Vulnerabilities**
[FSE'18,VSTTE'19,S&P'21]

**Verifying Correct Usage of APIs**
[POPL'21]

Core Program Analysis

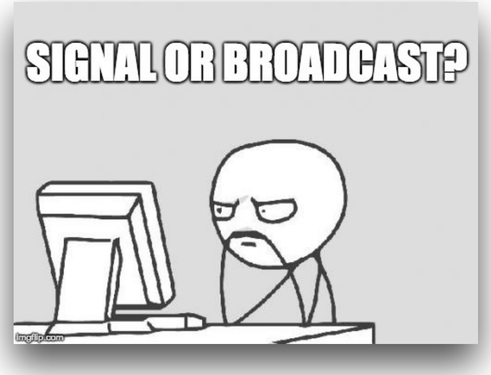



**Improve Scalability & Precision**
[FSE'17,SOAP'17,CC'18]



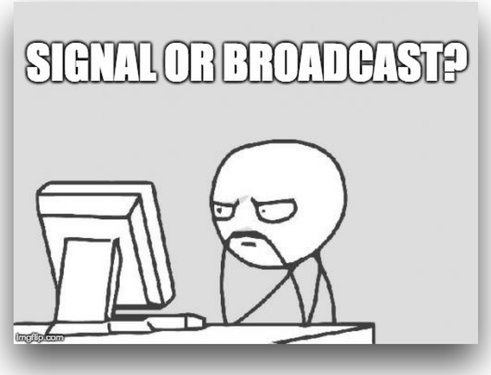
# Back Up Slides

# Signal or broadcast?




Do we need to wake up all threads block on P $\in Preds(M)$?

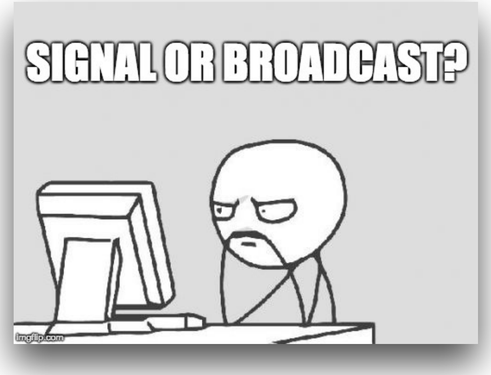
# Signal or broadcast?




Do we need to wake up all threads block on P $\in Preds(M)$?

Simulate what happens when a thread gets woken up

# Signal or broadcast?



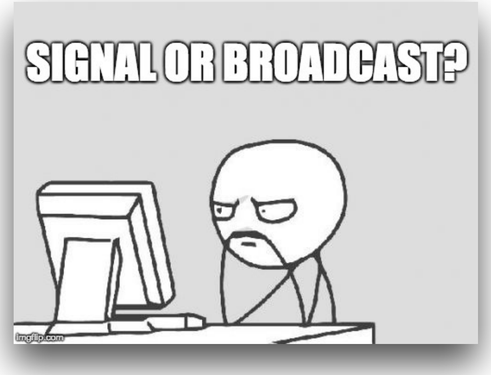

Do we need to wake up all threads block on P $\in Preds(M)$?

Simulate what happens when a thread gets woken up

For each CCR C of the form `waituntil(P) S`, check validity of Hoare triple:

$$\{Inv\}\ C\ \{\neg P\}$$

# Signal or broadcast?




Do we need to wake up all threads block on P $\in Preds(M)$?

Simulate what happens when a thread gets woken up

For each CCR C of the form `waituntil(P) S`, check validity of Hoare triple:

$$\{Inv\}\ C\ \{\neg P\}$$

If ***all*** of them are valid, no need to broadcast!

# Example: Signal or broadcast?

**Predicate:** $P_w \equiv readers = 0 \wedge \neg writerIn \quad \neg P_w \equiv readers \neq 0 \vee writerIn$ **Invariant:** $readers \geq 0$

**CCR:**

```
atomic void enterWriter() {

  waituntil(readers == 0 && !writerIn);
  writerIn = true;

}
```

# Example: Signal or broadcast?

**Predicate:** $P_w \equiv readers = 0 \wedge \neg writerIn \quad \neg P_w \equiv readers \neq 0 \vee writerIn$ **Invariant:** $readers \geq 0$

**CCR:**

```
atomic void enterWriter() {
  { readers ≥ 0 }
  waituntil(readers == 0 && !writerIn);
  writerIn = true;
  { ¬P_w }
}
```

# Example: Signal or broadcast?

**Predicate:** $P_w \equiv readers = 0 \wedge \neg writerIn \quad \neg P_w \equiv readers \neq 0 \vee writerIn$ **Invariant:** $readers \geq 0$

**CCR:**

```
atomic void enterWriter() {
  { readers ≥ 0 }
  waituntil(readers == 0 && !writerIn);
  writerIn = true;
  { ¬P_w }
}
```

Triple is valid, safe to notify a single writer!

# Conditional or unconditional signal?

Decision 2: Given CCR `waituntil(C) S` and predicate P, do we need to check P before signaling threads blocked on it?

Needed to avoid unnecessary context-switch

# Conditional or unconditional signal?

Decision 2: Given CCR `waituntil(C) S` and predicate P, do we need to check P before signaling threads blocked on it?

Needed to avoid unnecessary context-switch

$$\{Inv \wedge \neg P\}\ S\ \{P\}$$

# Conditional or unconditional signal?

Decision 2: Given CCR `waituntil(C) S` and predicate P, do we need to check P before signaling threads blocked on it?

Needed to avoid unnecessary context-switch

$$\{Inv \wedge \neg P\}\ S\ \{P\}$$

If this triple is valid, no need to test P at run-time!

# Example: Conditional Signal?

Can we signal readers unconditionally?

**CCR:**

```
atomic void exitWriter() {

    writerIn = false;

}
```

# Example: Conditional Signal?

**Invariant:** $readers \geq 0$

**Predicate:** $P_r \equiv \neg writerIn$

Can we signal readers unconditionally?

**CCR:**

```
atomic void exitWriter() {

    writerIn = false;

}
```

# Example: Conditional Signal?

**Invariant:** $readers \geq 0$

**Predicate:** $P_r \equiv \neg writerIn$

Can we signal readers unconditionally?

**CCR:**

```
atomic void exitWriter() {
    {readers ≥ 0 ∧ ¬P_r}
    writerIn = false;

}
```

# Example: Conditional Signal?

**Invariant:** $readers \geq 0$

**Predicate:** $P_r \equiv \neg writerIn$

Can we signal readers unconditionally?

**CCR:**

```
atomic void exitWriter() {
    {readers ≥ 0 ∧ ¬P_r}
    writerIn = false;
    {P_r}
}
```

# Example: Conditional Signal?

**Invariant:** $readers \geq 0$

**Predicate:** $P_r \equiv \neg writerIn$

Can we signal readers unconditionally?

**CCR:**

```
atomic void exitWriter() {
    {readers ≥ 0 ∧ ¬P_r}
    writerIn = false;
    {P_r}
}
```

Triple is valid, signal readers unconditionally!

# Hard constraints

Data-race freedom

Atomicity

Deadlock-freedom

# Hard constraints

Data-race freedom

Atomicity

Deadlock-freedom

Resolve race through locks or atomic fields

$$\mathsf{Racy}(S1,S2)$$
$$\Rightarrow$$
$$\mathscr{L}(S1) \cap \mathscr{L}(S2) \neq \varnothing$$
$$\vee$$
$$\exists f.\ (\mathit{Races}(S1,S2) = \{f\} \wedge a_f)$$

# Hard constraints

## Data-race freedom

Resolve race through locks or atomic fields

$$\text{Racy}(S1,S2)$$
$$\Rightarrow$$
$$\mathscr{L}(S1) \cap \mathscr{L}(S2) \neq \emptyset$$
$$\vee$$
$$\exists f.\ (Races(S1,S2) = \{f\} \wedge a_f)$$

## Atomicity

Only permit safe interleavings

$$\neg\text{Safe}(S, S1,S2)$$
$$\Rightarrow$$
$$\left(\mathscr{L}(S) \cap \mathscr{L}(S_1) \cap \mathscr{L}(S_2) \neq \emptyset\right)$$

## Deadlock-freedom

# Hard constraints

## Data-race freedom

Resolve race through locks or atomic fields

$$\text{Racy}(S1,S2)$$
$$\Rightarrow$$
$$\mathscr{L}(S1) \cap \mathscr{L}(S2) \neq \emptyset$$
$$\vee$$
$$\exists f.\ (Races(S1,S2) = \{f\} \wedge a_f)$$

## Atomicity

Only permit safe interleavings

$$\neg\text{Safe}(S, S1,S2)$$
$$\Rightarrow$$
$$\left(\mathscr{L}(S) \cap \mathscr{L}(S_1) \cap \mathscr{L}(S_2) \neq \emptyset\right)$$

## Deadlock-freedom

Impose a static lock acquisition order

$$\text{Edge}(S1,S2)$$
$$\Rightarrow$$
$$\left(l_1 \in \mathscr{L}(S1) \cap \mathscr{L}(S2)\right.$$
$$\wedge\, l_2 \in \mathscr{L}(S2) \setminus \mathscr{L}(S1)$$
$$\left.\Rightarrow l_1 \prec l_2\ \right)$$

# Soft constraints

Maximize parallelism

Minimize locks

Minimize atomics

# Soft constraints

Maximize parallelism

Minimize locks

Minimize atomics

Prefer: Non-racy snippet pairs **not** sharing a lock.

$$\neg \text{Racy}(S1, S2)$$

$$\Rightarrow$$

$$\mathscr{L}(s1) \cap \mathscr{L}(s2) = \varnothing$$

# Soft constraints

Maximize parallelism

Prefer: Non-racy snippet pairs **not** sharing a lock.

$$\neg \text{Racy}(S1, S2)$$

$$\Rightarrow$$

$$\mathscr{L}(s1) \cap \mathscr{L}(s2) = \varnothing$$

Minimize locks

Prefer: Lock not being used

$$\bigwedge_{s} \neg h_s^l$$

Minimize atomics

# Soft constraints

Maximize parallelism

Prefer: Non-racy snippet pairs **not** sharing a lock.

$$\neg \text{Racy}(S1, S2)$$

$$\Rightarrow$$

$$\mathscr{L}(s1) \cap \mathscr{L}(s2) = \varnothing$$

Minimize locks

Prefer: Lock not being used

$$\bigwedge_{s} \neg h_s^l$$

Minimize atomics

Prefer: field not converted to atomic

$$\neg a_f$$