UNIVERSITY OF TORONTO
FACULTY OF APPLIED SCIENCE & ENGINEERING

# Fast and Accurate Performance Analysis of Synchronization

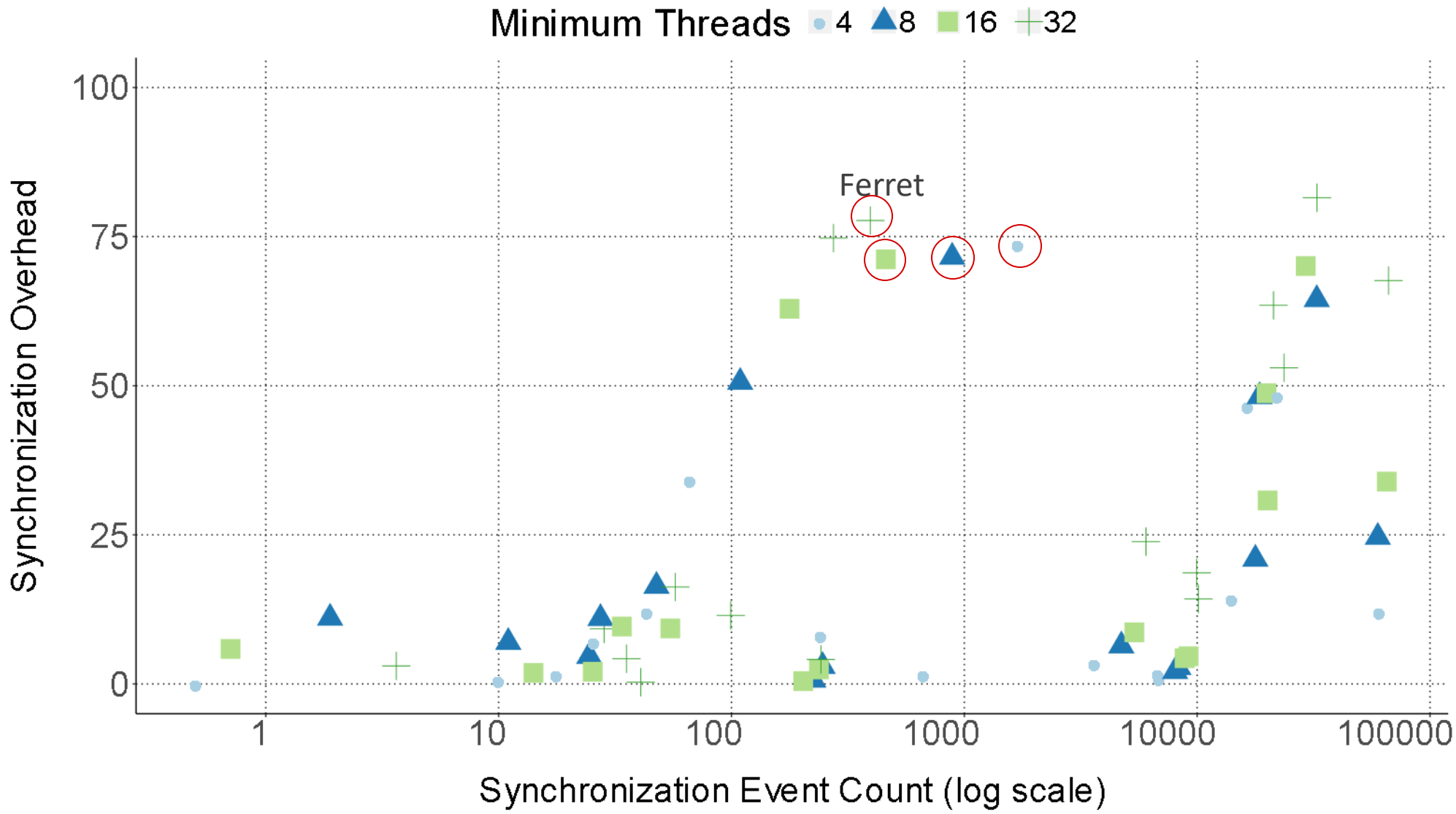
Mario Badr and Natalie Enright Jerger

# Evaluating Multi-Threaded Performance

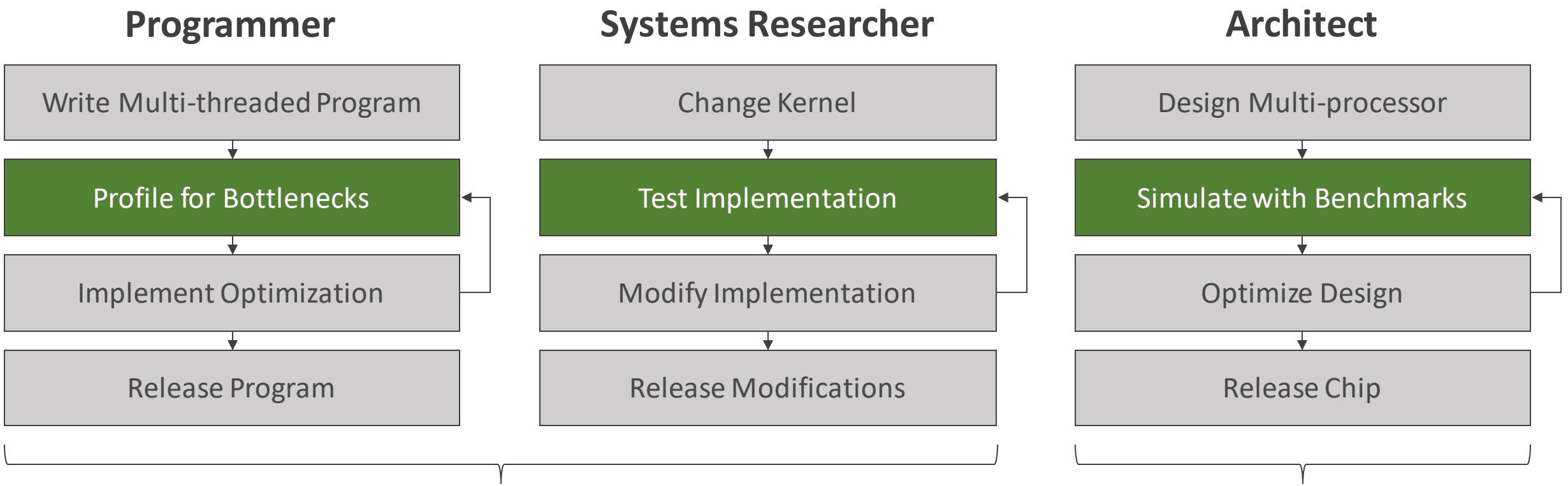
- Difficult and Time Consuming
  - Non-Determinism
  - Cross-stack effects
  - Different Architectures

- Goal: Make it Straightforward and Fast
  - One trace, many total orders
  - High level of abstraction

# More Synchronization != More Overhead



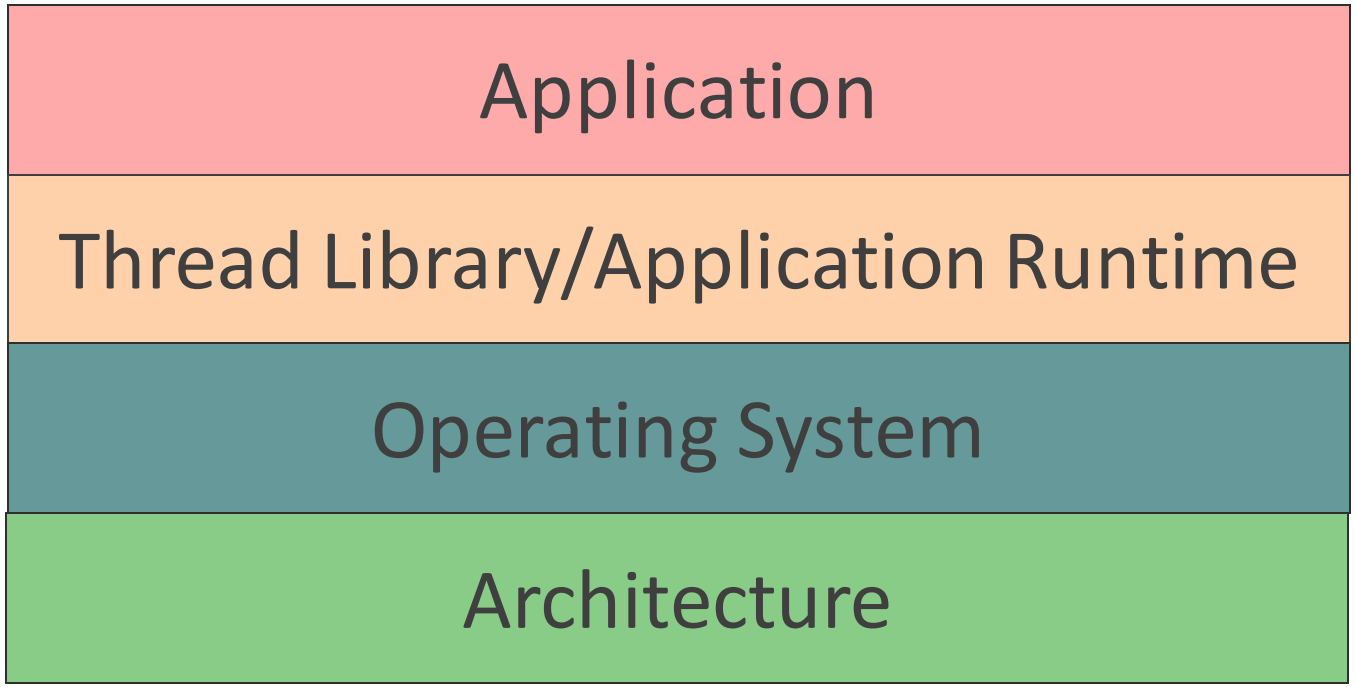
# Multi-threaded, Multi-core Workflows



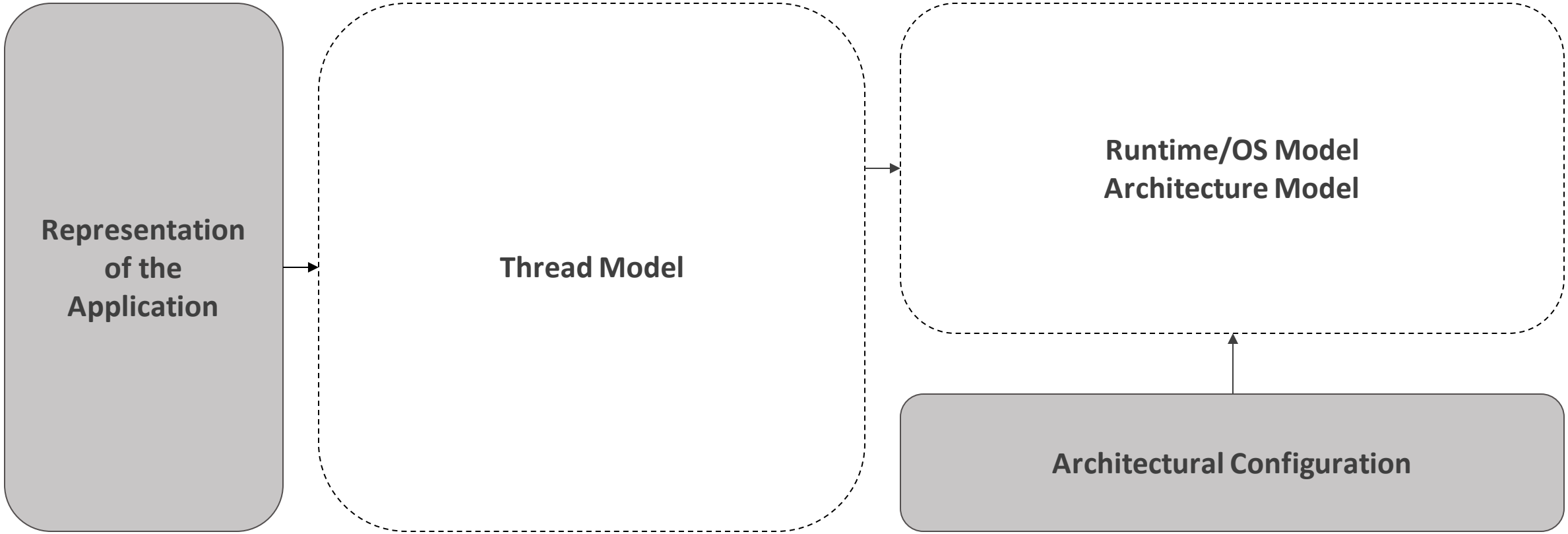

One architecture?
Multiple architectures?
Architectures that don't exist?

One application?
Application input?
Simulation time?

# Cross-Stack Interactions for Synchronization



# Modelling Multithreaded Applications

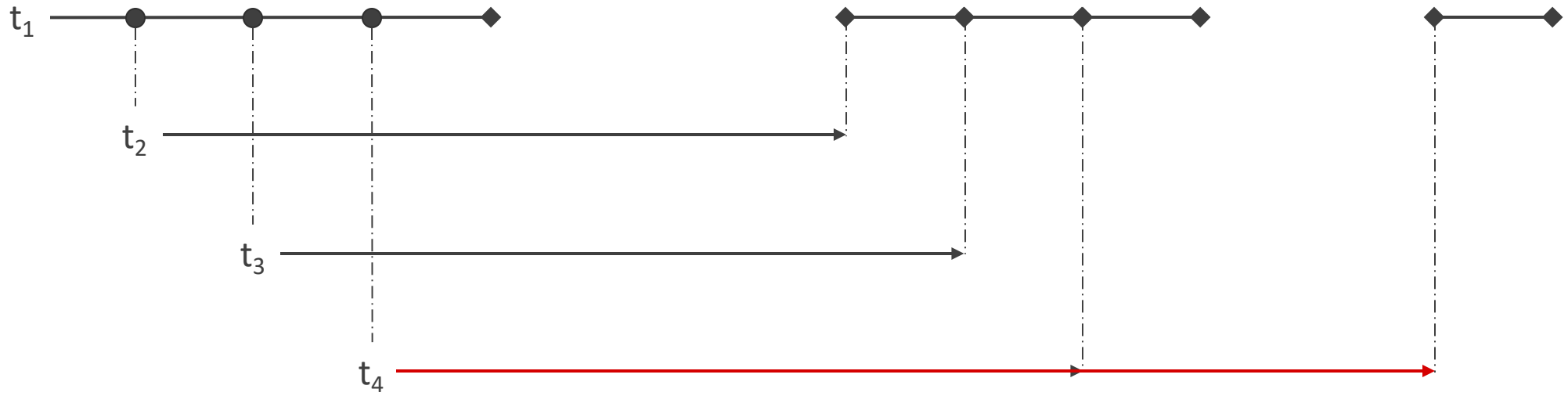


# Execution of a Parallel Program

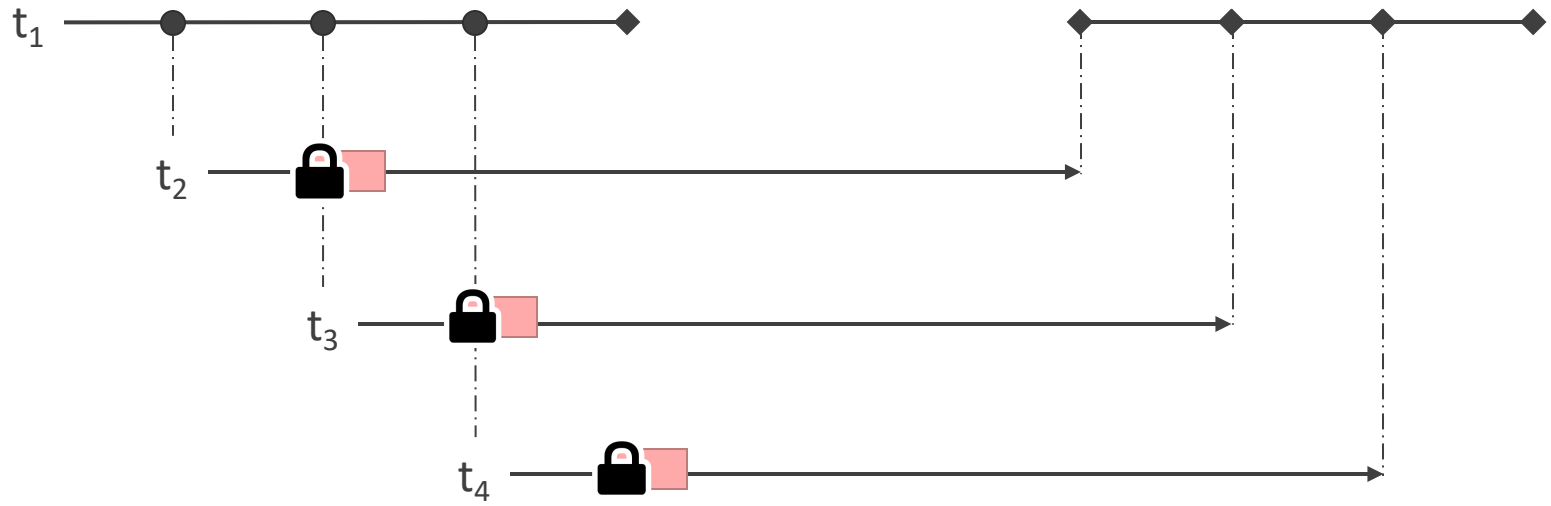
# What impacts a thread's execution time?

- Heterogeneity
  - Architectures (e.g., big.LITTLE)
  - Dynamic Voltage and Frequency Scaling (DVFS)
- Contention
  - Synchronization
- Many other things

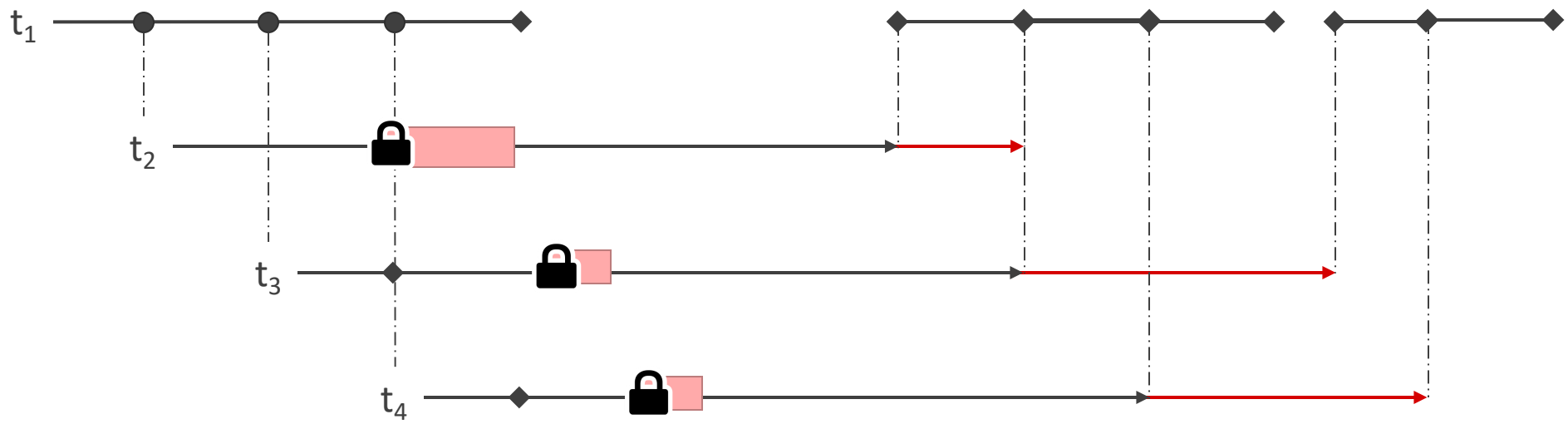
# The Impact of Heterogeneity



# The Impact of Synchronization



# Heterogeneity and Synchronization




The order and time of synchronization events impacts performance.

# Modelling Cross-Stack Interactions

- How to represent a multi-threaded application?
  - Task Graph
  - Trace
- How to model the operating system and runtime?
  - Thread scheduling
  - Synchronization
- How to model the architecture?
  - Rate of execution (e.g., cycles per instruction)

# The Producer Consumer Example

**Adding Work to a Queue**

```
1   lock(mutex);
2   while(queue.full()) {
3     wait(condition_dequeue, mutex);
4   }
5   queue.push(work);
6   signal(condition_enqueue);
7   unlock(mutex);
```

**Removing Work from a Queue**

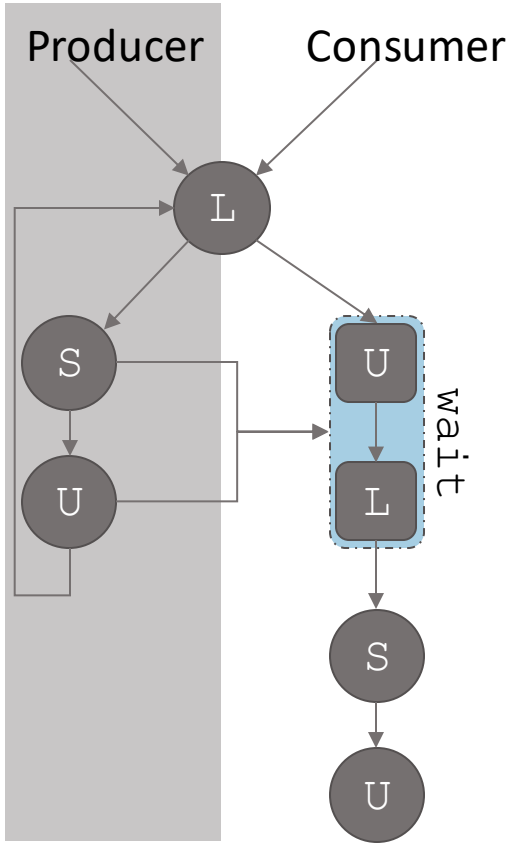
```
1   lock(mutex);
2   while(queue.empty()) {
3     wait(condition_enqueue, mutex);
4   }
5   work = queue.pop();
6   signal(condition_dequeue);
7   unlock(mutex);
```

# Representing an Application

**Synchronization Trace**

| Thread | Event | Primitive |
|---|---|---|
| Consumer | **L**ock | mutex |
| Producer | **L**ock | mutex |
| Consumer | Wait | enqueue |
| Producer | **S**ignal | enqueue |
| Producer | **U**nlock | mutex |
| Consumer | **S**ignal | dequeue |
| Consumer | **U**nlock | mutex |

**Task Graph**



# The order of synchronization events

- A synchronization trace gives us the *program order* of each thread
- We want to determine the *total order* of all synchronization events
- The *total order* must be *correct*
  - Safety (e.g., no two threads in the same critical section)
  - Liveness (e.g., all threads make progress eventually)

| Thread | Event |
|---|---|
| Consumer | Lock |
| Producer | Lock |
| Consumer | Wait |
| Producer | Signal |
| Producer | Unlock |
| Consumer | Signal |
| Consumer | Unlock |

Consumer locks mutex first (original trace)

| Thread | Event |
|---|---|
| Consumer | Lock |
| Consumer | Wait |
| Producer | Lock |
| Producer | Signal |
| Producer | |
| Consume | |
| Consumer | Unlock |

Consumer is much faster than producer

| Thread | Event |
|---|---|
| Producer | Lock |
| Consumer | Lock |
| Producer | Signal |
| Producer | Unlock |
| Consumer | Wait |
| Consumer | Signal |
| Consumer | Unlock |

Producer locks mutex first

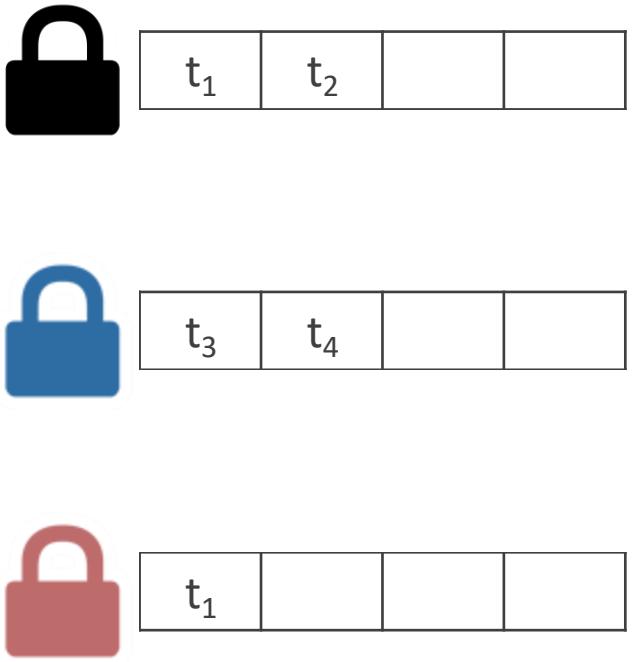
| Thread | Event |
|---|---|
| Producer | Lock |
| Producer | Signal |
| Producer | Unlock |
| Consumer | Lock |
| | |
| | l |
| Consumer | Unlock |

consumer

One Trace, Multiple Total Orders – Captures Non-Determinism

# Modelling Locks and Condition Variables

**Per-Lock Thread Queues**




**Condition Variable Counters**

- On wait
  - Decrement counter by 1
- On signal
  - Increment counter by 1
- On broadcast
  - Increment counter by number of consumers

# Estimating the Time Between Events

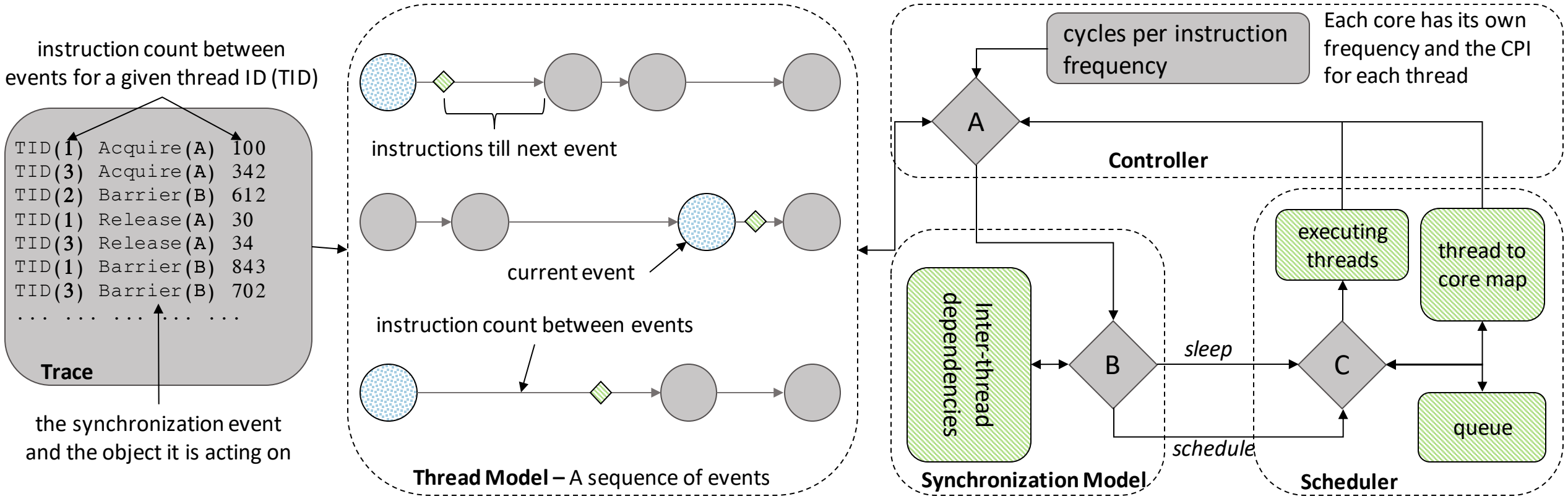
1. Dynamic Instructions
   - The *distance* between events

2. Core Frequency and Microarchitecture
   - The *rate* between events

3. The Scheduling of Threads
   - The opportunity to *execute* dynamic instructions

4. The Timing of Prior Events
   - The *dependencies* between threads

# Our High Level Abstraction



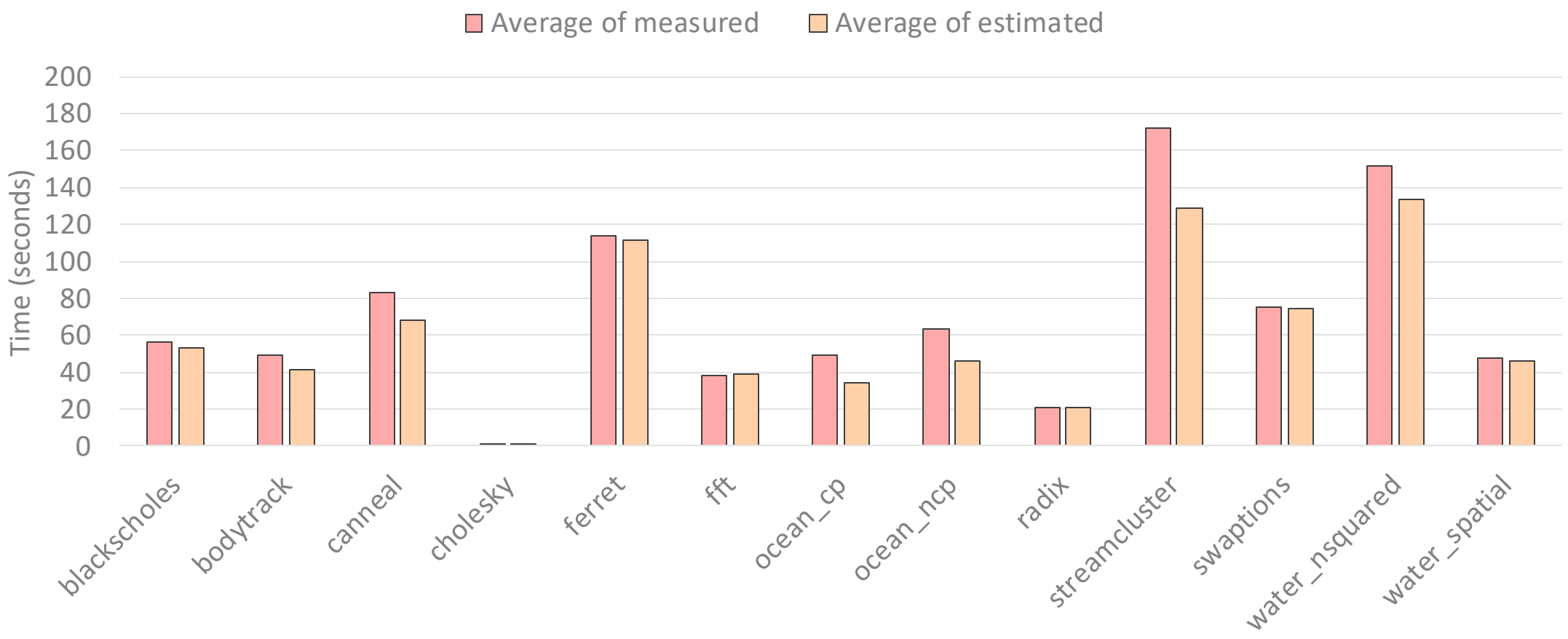
# Validation Methodology

- Benchmarks: PARSEC 3.0, Splash-3
  - Execution time measured with GNU time
  - Traces generated with Pin
  - Cycles-per-instruction profiled with VTune™
- Architecture: Intel Xeon E5-2650 v2
  - 2 sockets, 8 cores per socket, 2 threads per core
  - 20 MB L3 Cache
  - 2.6 GHz
- Three runs for each experiment

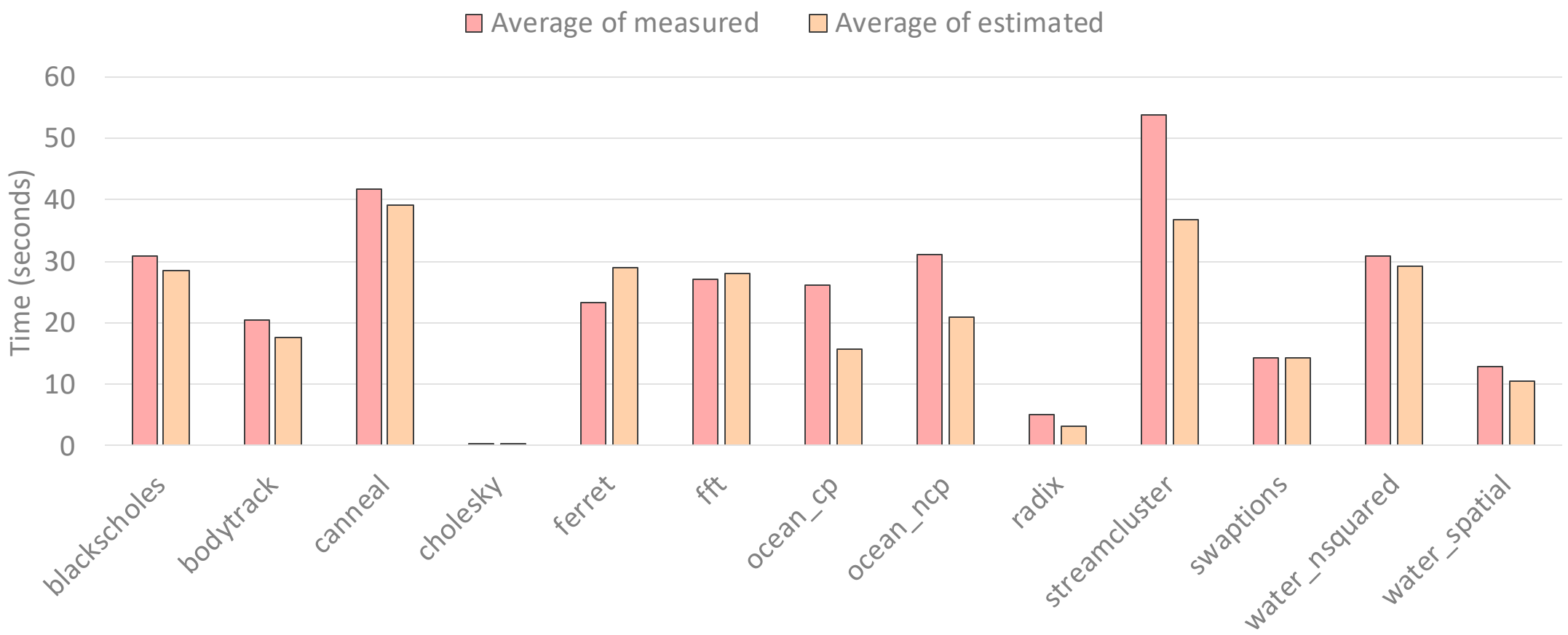
# Assumptions and Approximations

- Cycles-Per-Instruction encompasses microarchitecture and memory hierarchy performance
- Synchronization events have zero latency
- Context switches have zero latency
- Synchronization model approximates application state
  - i.e., for condition variables

# Model Validation: 4 Cores, Single Socket

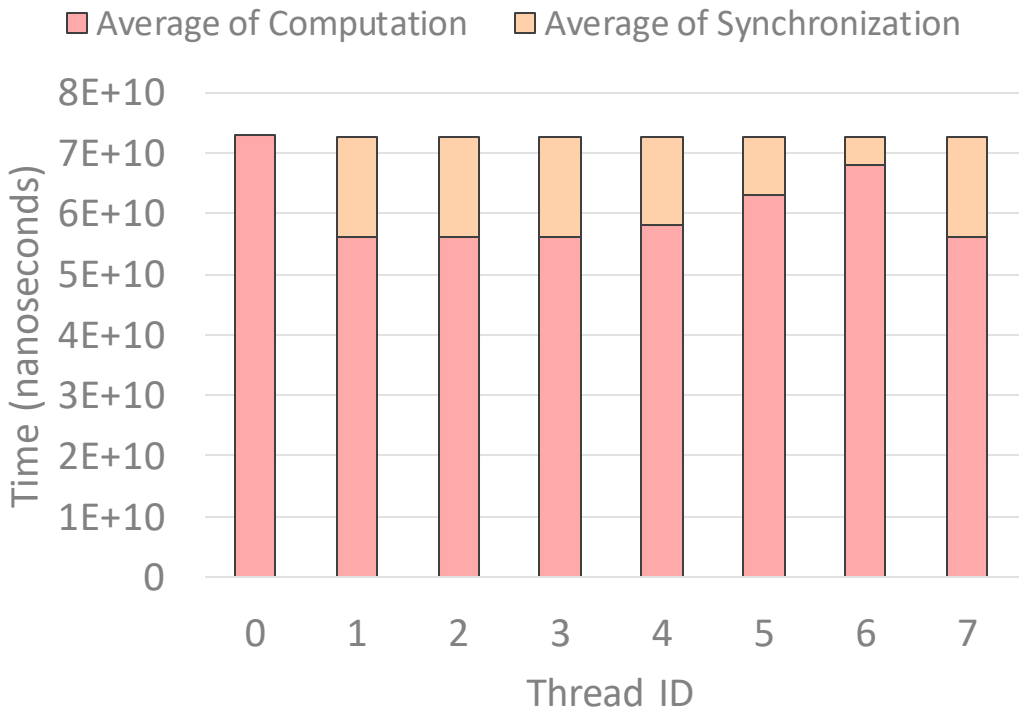


# Model Validation: 32 Cores, Dual Socket

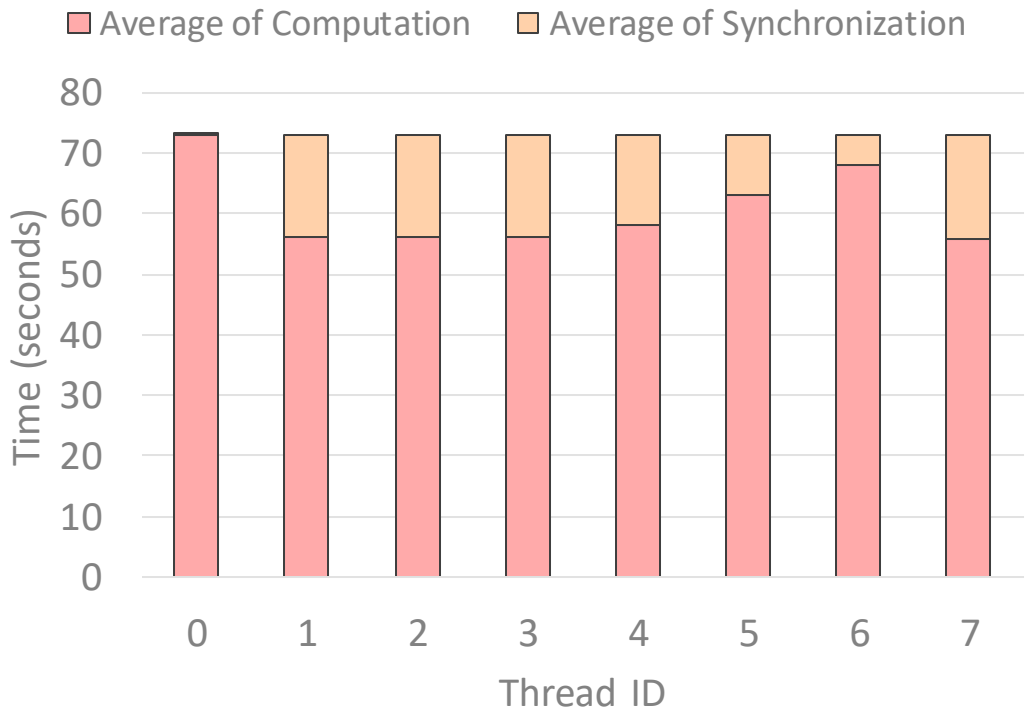


# Water (nsquared): 8 Cores

## Estimated with Our Model




## Estimated with Vtune™



# Model Runtime

| Benchmark | Input Set | Input Size | Trace Size | Runtime |
|---|---|---|---|---|
| blackscholes | Native | 603 MB | 1.1 KB | 4 ms |
| bodytrack | Native | 616 MB | 31 MB | 4.9 minutes |
| water (nsquared) | Native | 3.6 MB | 53 MB | 7.5 minutes |
| **average** | | | **2 MB** | **32 seconds** |

Orders of magnitude faster than simulation of smaller input sets.

# Conclusion

- A very high level of abstraction can accurately and quickly estimate the performance of a multi-threaded application on a multi-core processor.
  - Average 7.2% error in total execution time
  - Average 32 seconds to generate an estimate
- Programmers and Systems Researchers can evaluate on many architectures
- Architects can evaluate with native inputs and many applications

# Future Work

- How much non-determinism is there across multiple traces of an application?

- How can a {memory, network} contention model be added to improve error without significantly increasing model complexity?

UNIVERSITY OF TORONTO
FACULTY OF APPLIED SCIENCE & ENGINEERING

# Our Work is Open Source

https://github.com/mariobadr/simsync-pmam

License: Apache 2.0

Mario Badr and Natalie Enright Jerger

# Scenario A – Consumer locks mutex first

| Thread | Event | Primitive |
|---|---|---|
| Consumer | **L**ock | mutex |
| Producer | **L**ock | mutex |
| Consumer | Wait | enqueue |
| Producer | **S**ignal | enqueue |
| Producer | **U**nlock | mutex |
| Consumer | **S**ignal | dequeue |
| Consumer | **U**nlock | mutex |

1. Consumer locks mutex
2. Producer attempts lock
   - Producer blocked
3. Consumer waits for enqueue
   - Consumer blocked, silent unlock
   - Producer unblocked, silent lock
4. Producer signals enqueue
   - Consumer tries to lock, remains blocked
5. Producer unlocks mutex
   - Consumer unblocked, silent lock
6. Consumer signals dequeue
7. Consumer unlocks mutex

# Scenario B – Consumer is much faster

| Thread | Event | Primitive |
|---|---|---|
| Consumer | **L**ock | mutex |
| Consumer | Wait | enqueue |
| Producer | **L**ock | mutex |
| Producer | **S**ignal | enqueue |
| Producer | **U**nlock | mutex |
| Consumer | **S**ignal | dequeue |
| Consumer | **U**nlock | mutex |

1. Consumer locks mutex
2. Consumer waits for enqueue
   - Consumer blocked, silent unlock
3. Producer locks mutex
4. Producer signals enqueue
   - Consumer tries lock, remains blocked
5. Producer unlocks mutex
   - Consumer unblocked, silent lock
6. Consumer signals dequeue
7. Consumer unlocks mutex

# Scenario C – Producer locks mutex first

| Thread | Event | Primitive |
|---|---|---|
| Producer | **L**ock | mutex |
| Consumer | **L**ock | mutex |
| Producer | **S**ignal | enqueue |
| Producer | **U**nlock | mutex |
| Consumer | Wait | enqueue |
| Consumer | **S**ignal | dequeue |
| Consumer | **U**nlock | mutex |

1. Producer locks mutex
2. Consumer attempts lock
   - Consumer blocked
3. Producer signals enqueue
4. Producer unlocks mutex
   - Consumer unblocked
5. Consumer locks mutex
6. Consumer does not have to wait
7. Consumer signals dequeue
8. Consumer unlocks mutex

# Scenario D – Producer is much faster

| Thread | Event | Primitive |
|---|---|---|
| Producer | **L**ock | mutex |
| Producer | **S**ignal | enqueue |
| Producer | **U**nlock | mutex |
| Consumer | **L**ock | mutex |
| Consumer | Wait | enqueue |
| Consumer | **S**ignal | dequeue |
| Consumer | **U**nlock | mutex |

1. Producer locks mutex
2. Producer signals enqueue
3. Producer unlocks mutex
4. Consumer locks mutex
5. Consumer does not have to wait
6. Consumer signals dequeue
7. Consumer unlocks mutex