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

**Programmer**
- Write Multi-threaded Program
- Profile for Bottlenecks
- Implement Optimization
- Release Program

**Systems Researcher**
- Change Kernel
- Test Implementation
- Modify Implementation
- Release Modifications

**Architect**
- Design Multi-processor
- Simulate with Benchmarks
- Optimize Design
- Release Chip

One architecture?  
Multiple architectures?  
Architectures that don’t exist?  
One application?  
Application input?  
Simulation time?
Cross-Stack Interactions for Synchronization

Application
Thread Library/Application Runtime
Operating System
Architecture
Modelling Multithreaded Applications

- Representation of the Application

  Thread Model

  Runtime/OS Model
  Architecture Model

  Architectural Configuration
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

- $t_1$
- $t_2$
- $t_3$
- $t_4$
The Impact of Synchronization

\[ t_1 \quad t_2 \quad t_3 \quad t_4 \]
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. \texttt{lock} (mutex);
2. while (queue.full()) {
3. \hspace{1em} \texttt{wait} (condition\_dequeue, mutex);
4. }
5. queue.push(work);
6. \texttt{signal} (condition\_enqueue);
7. \texttt{unlock} (mutex);

Removing Work from a Queue

1. \texttt{lock} (mutex);
2. while (queue.empty()) {
3. \hspace{1em} \texttt{wait} (condition\_enqueue, mutex);
4. }
5. work = queue.pop();
6. \texttt{signal} (condition\_dequeue);
7. \texttt{unlock} (mutex);
## Representing an Application

### Synchronization Trace

<table>
<thead>
<tr>
<th>Thread</th>
<th>Event</th>
<th>Primitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer</td>
<td>Lock</td>
<td>mutex</td>
</tr>
<tr>
<td>Producer</td>
<td>Lock</td>
<td>mutex</td>
</tr>
<tr>
<td>Consumer</td>
<td>Wait</td>
<td>enqueue</td>
</tr>
<tr>
<td>Producer</td>
<td>Signal</td>
<td>enqueue</td>
</tr>
<tr>
<td>Producer</td>
<td>Unlock</td>
<td>mutex</td>
</tr>
<tr>
<td>Consumer</td>
<td>Signal</td>
<td>dequeue</td>
</tr>
<tr>
<td>Consumer</td>
<td>Unlock</td>
<td>mutex</td>
</tr>
</tbody>
</table>

### Task Graph

[Diagram showing the task graph with nodes labeled Producer, Consumer, Lock (L), Signal (S), Unlock (U), and Wait (wait).]
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)
<table>
<thead>
<tr>
<th>Thread</th>
<th>Event</th>
<th></th>
<th>Thread</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer</td>
<td>Lock</td>
<td>Consumer locks mutex first (original trace)</td>
<td>Producer</td>
<td>Lock</td>
</tr>
<tr>
<td>Producer</td>
<td>Lock</td>
<td></td>
<td>Consumer</td>
<td>Lock</td>
</tr>
<tr>
<td>Consumer</td>
<td>Wait</td>
<td></td>
<td>Producer</td>
<td>Signal</td>
</tr>
<tr>
<td>Producer</td>
<td>Signal</td>
<td>Producer locks mutex first</td>
<td>Producer</td>
<td>Unlock</td>
</tr>
<tr>
<td>Producer</td>
<td>Unlock</td>
<td>Consumer is much faster than producer</td>
<td>Consumer</td>
<td>Wait</td>
</tr>
<tr>
<td>Consumer</td>
<td>Signal</td>
<td></td>
<td>Consumer</td>
<td>Signal</td>
</tr>
<tr>
<td>Consumer</td>
<td>Unlock</td>
<td></td>
<td>Consumer</td>
<td>Unlock</td>
</tr>
</tbody>
</table>

**One Trace, Multiple Total Orders – Captures Non-Determinism**
Modelling Locks and Condition Variables

Per-Lock Thread Queues

- t₁ t₂
- t₃ t₄
- t₁

Condition Variable Counters

- On wait
  - Decrease counter by 1
- On signal
  - Increase counter by 1
- On broadcast
  - Increase 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

Thread Model – A sequence of events

Synchronization Model

Scheduler

Controller

Inter-thread dependencies

each core has its own frequency and the CPI for each thread

each thread to core map

executing threads

sleep

schedule

queue

trace

TID(1) Acquire(A) 100
TID(3) Acquire(A) 342
TID(2) Barrier(B) 612
TID(1) Release(A) 30
TID(3) Release(A) 34
TID(1) Barrier(B) 843
TID(3) Barrier(B) 702

the synchronization event and the object it is acting on

the instruction count between events for a given thread ID (TID)

...
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

Average of measured vs. Average of estimated time for various benchmarks.
Water (nsquared): 8 Cores

Estimated with Our Model

Estimated with Vtune™
## Model Runtime

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input Set</th>
<th>Input Size</th>
<th>Trace Size</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>Native</td>
<td>603 MB</td>
<td>1.1 KB</td>
<td>4 ms</td>
</tr>
<tr>
<td>bodytrack</td>
<td>Native</td>
<td>616 MB</td>
<td>31 MB</td>
<td>4.9 minutes</td>
</tr>
<tr>
<td>water (nsquared)</td>
<td>Native</td>
<td>3.6 MB</td>
<td>53 MB</td>
<td>7.5 minutes</td>
</tr>
<tr>
<td>average</td>
<td></td>
<td>2 MB</td>
<td></td>
<td>32 seconds</td>
</tr>
</tbody>
</table>

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?
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

<table>
<thead>
<tr>
<th>Thread</th>
<th>Event</th>
<th>Primitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer</td>
<td>Lock</td>
<td>mutex</td>
</tr>
<tr>
<td>Producer</td>
<td>Lock</td>
<td>mutex</td>
</tr>
<tr>
<td>Consumer</td>
<td>Wait</td>
<td>enqueue</td>
</tr>
<tr>
<td>Producer</td>
<td>Signal</td>
<td>enqueue</td>
</tr>
<tr>
<td>Producer</td>
<td>Unlock</td>
<td>mutex</td>
</tr>
<tr>
<td>Consumer</td>
<td>Signal</td>
<td>dequeue</td>
</tr>
<tr>
<td>Consumer</td>
<td>Unlock</td>
<td>mutex</td>
</tr>
</tbody>
</table>

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

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

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

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

<table>
<thead>
<tr>
<th>Thread</th>
<th>Event</th>
<th>Primitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer</td>
<td>Lock</td>
<td>mutex</td>
</tr>
<tr>
<td>Producer</td>
<td>Signal</td>
<td>enqueue</td>
</tr>
<tr>
<td>Producer</td>
<td>Unlock</td>
<td>mutex</td>
</tr>
<tr>
<td>Consumer</td>
<td>Lock</td>
<td>mutex</td>
</tr>
<tr>
<td>Consumer</td>
<td>Wait</td>
<td>enqueue</td>
</tr>
<tr>
<td>Consumer</td>
<td>Signal</td>
<td>dequeue</td>
</tr>
<tr>
<td>Consumer</td>
<td>Unlock</td>
<td>mutex</td>
</tr>
</tbody>
</table>