A New Approach to Concurrency and Parallelism (Part 2)

Tom Ball (tball@microsoft.com)
Sebastian Burckhardt (sburckha@microsoft.com)
Madan Musuvathi (madanm@microsoft.com)
Microsoft Research
<table>
<thead>
<tr>
<th><strong>Performance</strong></th>
<th><strong>Parallelism</strong></th>
<th><strong>Concurrency</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td></td>
<td>Responsiveness</td>
</tr>
<tr>
<td><strong>Correctness</strong></td>
<td>Atomicity, Determinism, Deadlock, Livelock, Linearizability, Data races, ...</td>
<td></td>
</tr>
</tbody>
</table>
• **What:** 16 weeks (8 units) of material  
  • Slides  
  • Notes  
  • Exercises, quizzes  
  • Sample programs and applications  
  • Tests and tools  

• **Who:** beginning graduates, senior undergraduates  

• **Prerequisites:** OO programming, systems, data structures  

• **Dependencies:**  
  • .NET 4  
  • C# and F# languages
PPCP Units: Breadth with Correctness Concepts

- Unit 1: Imperative Data Parallel Programming
- Unit 2: Shared Memory
- Unit 3: Concurrent Components
- Unit 4: Functional Data Parallel Programming
- Unit 5: Scheduling and Synchronization
- Unit 6: Interactive/Reactive Systems
- Unit 7: Message Passing
- Unit 8: Advanced Topics
Source code release
- [chesstool.codeplex.com](http://chesstool.codeplex.com)

Preemption bounding [PLDI07]
- speed search for bugs
- simple counterexamples

Fair stateless exploration [PLDI08]
- scales to large programs

Architecture [OSDI08]
- Tasks and SyncVars
- API wrappers

LineUp: automatic linearizability checking [PLDI10]

Data race detection

Memory model issues

Coming:
- Concurrency unit tests
- Determinism checking
Some Correctness Concepts Featured in PPCP

- Data race free discipline and happens-before data race detection
- Automated linearizability checking of concurrent components
- Supported by CHESS
Data Race Free (DRF) Discipline Happens - Before Race Detection
Why Care About Data Races?

- **Data races may reveal synchronization errors**
  - Many errors (from simple omissions to algorithmic mistakes) can manifest as data races.
  - **Data race detectors can often help to find & fix concurrency bugs very efficiently.**
  - But: some data races may appear “benign”, watering down the utility of such detectors (false alarms)

- **Data races are not portable**
  - Behavior of program with data races depends on memory model
  - Relaxations in compiler or hardware may introduce strange & platform-dependent effects
What is a Data Race, Traditionally?

• Long history, many definitions

• Sometimes linked to specific programming idioms
  • “shared variables must be lock-protected”

• Often unclear terminology
  • “Races” vs. “Data Races”: Is it a race if two threads try to acquire the same lock?
  • “Ordered by synchronization”: What counts as synchronization?

• Recently: *Convergence of Definition*
  • Motivated by research on memory models and recent proposals for language-level memory models (Java, C++)
What is a Data Race, *Today/Tomorrow*?

- If two *conflicting* memory accesses happen *concurrently*, we have a data race.

- Two memory accesses *conflict* if
  - They target the same location
  - They are not both reads
  - They are not both synchronization operations
Data-Race-Free (DRF) Discipline means we write programs that have NO data races (not even “benign” ones).

Already “best practice” for many, but not all programmers.
• Answer A: I have to protect everything with locks and must not use lock-free synchronization techniques

• Answer B: I have to properly declare racy accesses using type qualifiers (atomic, volatile) or special operations (interlocked, compare-and-swap)
DRF Discipline Pros & Cons

• Pros
  • Code is more declarative (easier to see intentions)
  • Code is immune against memory model relaxations (= why DRF invented in the first place).
  • All data races are bugs, no benign races.
  • Code is easier to verify and debug.

• Cons
  • Have to learn how to use type qualifiers correctly
  • Annotation overhead (not much)
  • Some qualifiers not efficient on some platforms
• Test for concurrent conflicting accesses
  • Problem: schedule varies from run to run
  • Probability of making potentially concurrent accesses actually simultaneous often not very good.

• Idea: happens-before race detector
  • Check for conflicting accesses that could have been concurrent in a slightly different schedule
• Use **logical clocks** and **timestamps** to define a partial order called *happens-before* on events in a concurrent system.

• States *precisely* when two events are *logically* concurrent (abstracting away real time).

- Cross-edges from send events to receive events
- \((a_1, a_2, a_3)\) happens before \((b_1, b_2, b_3)\) iff \(a_1 \leq b_1\) and \(a_2 \leq b_2\) and \(a_3 \leq b_3\)
Happens-Before for Shared Memory

- **Distributed Systems**
  Cross-edges from send to receive events

- **Shared Memory systems**
  Cross-edges represent ordering effect of synchronization
  - Edges from lock release to subsequent lock acquire
  - Edges from volatile writes to subsequent volatile reads
  - Long list of primitives that may create edges
    - Semaphores, Waithandles, Rendezvous, system calls (asynchronous IO), ...
### Example

<table>
<thead>
<tr>
<th>Static Program</th>
<th>Dynamic Execution Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>int data;</td>
<td>data = 1;</td>
</tr>
<tr>
<td>volatile bool flag;</td>
<td>flag = true;</td>
</tr>
<tr>
<td>Thread 1</td>
<td>while (!flag) yield(); int x = data;</td>
</tr>
<tr>
<td>Thread 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(!flag)-&gt;true</td>
</tr>
<tr>
<td></td>
<td>yield()</td>
</tr>
<tr>
<td></td>
<td>(!flag)-&gt;false</td>
</tr>
<tr>
<td></td>
<td>x = data</td>
</tr>
</tbody>
</table>

- Not a data race because \((1,0) \leq (1,4)\)
- If flag were not declared volatile, we would not add a cross-edge, and this would be a data race.
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Runtime.InteropServices;
using System.Runtime.ConstrainedExecution;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Diagnostics;  
namespace BadSystemCollectionsConcurrent {
    /// 
    /// Represents a thread-safe, concurrent queue primitive, and its associated debugger view type.  
    /// A lock-free, concurrent queue primitive, and its associated debugger view type.
    /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
    /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
    /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
    /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
    /// </summary>
    public class ConcurrentQueue<T> : IEnumerable<T>, IEnumerable, IProducerConcurrentCollection<T> {
        // 
        /// Represents a thread-safe first-in, first-out collection of objects.
        /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
        /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
        /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
        /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
        /// 
        /// </summary>
        public class ConcurrentQueue<T> : IEnumerable<T>, IEnumerable, IProducerConcurrentCollection<T> {
            // 
            /// Represents a thread-safe first-in, first-out collection of objects.
            /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
            /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
            /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
            /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
            /// 
            /// </summary>
            public class ConcurrentQueue<T> : IEnumerable<T>, IEnumerable, IProducerConcurrentCollection<T> {
                // 
                /// Represents a thread-safe first-in, first-out collection of objects.
                /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                /// 
                /// </summary>
                public class ConcurrentQueue<T> : IEnumerable<T>, IEnumerable, IProducerConcurrentCollection<T> {
                    // 
                    /// Represents a thread-safe first-in, first-out collection of objects.
                    /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                    /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                    /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                    /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                    /// 
                    /// </summary>
                    public class ConcurrentQueue<T> : IEnumerable<T>, IEnumerable, IProducerConcurrentCollection<T> {
                        // 
                        /// Represents a thread-safe first-in, first-out collection of objects.
                        /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                        /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                        /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                        /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                        /// 
                        /// </summary>
                        public class ConcurrentQueue<T> : IEnumerable<T>, IEnumerable, IProducerConcurrentCollection<T> {
                            // 
                            /// Represents a thread-safe first-in, first-out collection of objects.
                            /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                            /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                            /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                            /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                            /// 
                            /// </summary>
                            public class ConcurrentQueue<T> : IEnumerable<T>, IEnumerable, IProducerConcurrentCollection<T> {
                                // 
                                /// Represents a thread-safe first-in, first-out collection of objects.
                                /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                                /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                                /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                                /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                                /// 
                                /// </summary>
                                public class ConcurrentQueue<T> : IEnumerable<T>, IEnumerable, IProducerConcurrentCollection<T> {
                                    // 
                                    /// Represents a thread-safe first-in, first-out collection of objects.
                                    /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                                    /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                                    /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                                    /// <param name="index">The zero-based index in <paramref name="array"> at which copying begins.</param>
                                    /// 
                                    /// </summary>
Let's write a test

```java
q = new ConcurrentQueue();
q.push(10);
t = q.pop();
Assert( ? )
```
Let's write a test

```java
q = new ConcurrentQueue();
q.push(10);
t = q.pop();

Assert:
q.size() is 0 or 1
```
Let's write a test

```java
q = new ConcurrentQueue();
q.push(10);
t = q.pop();

Assert:
  q.size() is 0 or 1
  and t is 10 or <fail>
```
Let's write a test

```java
q = new ConcurrentQueue();
q.push(10);
t = q.pop();

Assert:
 t = fail && q.size() = 1 &&
   q.peek() == 10 ||
 t = 10 && q.size() = 0
```
Let's write a test

\[ q = \text{new} \ \text{ConcurrentQueue}(); \]

\[ q.\text{push}(10); \]
\[ t = q.\text{pop}(); \]

\[ q.\text{push}(20); \]
\[ u = q.\text{pop}(); \]

\[ \text{Assert} \ (\ ? \ ) \]
Let's write a test

```java
q = new ConcurrentQueue();
q.push(10);
t = q.pop();
q.push(20);
u = q.pop();

Assert:
q.size() == 0 &&
(t = 10 || t = 20) &&
(u = 10 || t = 20) &&
u != t
```
```java
q = new ConcurrentQueue();
q.push(10);
t1 = q.pop();
t2 = q.peek();
q.push(20);
q.push(30);
u1 = q.peek();
q.push(40);
u2 = q.pop();
v1 = q.pop();
q.push(50);
v2 = q.peek();
q.push(60);
```

Assert (? )
q = new ConcurrentQueue();

q.push(10);
t1 = q.pop();
t2 = q.peek();
q.push(20);

Assert:
ConcurrentQueue behaves like a queue
Informally, this is “thread safety”

ConcurrentQueue behaves like a queue

A piece of code is thread-safe if it functions correctly during simultaneous execution by multiple threads.
ConcurrentQueue behaves like a queue

Concurrent behaviors of ConcurrentQueue

are consistent with

a sequential specification of a queue

Every operation appears to occur atomically at some point between the call and return
q = new ConcurrentQueue();

q.push(10);
t1 = q.pop();
t2 = q.peek();
q.push(20);

Assert:
Linearizability wrt a given sequential specification

q.push(30);
u1 = q.peek();
q.push(40);
u2 = q.pop();
q.push(50);

v1 = q.pop();
q.push(50);
v2 = q.peek();
q.push(60);
```java
q = new ConcurrentQueue();
q.push(10);
t1 = q.pop();
t2 = q.peek();
q.push(20);
q.push(30);
u1 = q.peek();
q.push(40);
u2 = q.pop();
q.push(50);
v2 = q.peek();
q.push(60);

Assert:
Exists some deterministic spec wrt which q is Linearizable
```
• Automatically synthesize a sequential specification
  • By observing sequential behaviors of a component

• Check linearizability with respect to this spec

• Completeness
  • LineUp failure $\Rightarrow$ Component is not linearizable wrt any deterministic spec

• Restricted Soundness
  • Component is not linearizable $\Rightarrow$ Exists a test case for which LineUp fails
• Thread safety == Generalized linearizability

• Linearizability does not check against incorrect blocking
  • An implementation that blocks on all operations is vacuously linearizable
# Practical Parallel and Concurrent Programming (PP&CP)

<table>
<thead>
<tr>
<th>P&amp;C</th>
<th>Parallelism</th>
<th>Concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Performance</strong></td>
<td>Speedup</td>
<td>Responsiveness</td>
</tr>
<tr>
<td><strong>Correctness</strong></td>
<td>Atomicity, Determinism, Deadlock, Livelock, Linearizability, Data races, ...</td>
<td></td>
</tr>
</tbody>
</table>