IMUnit:
Improved Multithreaded Unit Testing

Vilas Jagannath, Milos Gligoric, Dongyun Jin, Qingzhou Luo
Grigore Rosu, **Darko Marinov**

ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

January 31, 2012
CREST Open Workshop (COW 17)
London, UK

UPCRC Illinois
Universal Parallel Computing Research Center
NSF
NASA
NATIONAL SECURITY AGENCY
UNITED STATES OF AMERICA
SAMSUNG

Project Team

Vilas Jagannath

Milos Gligoric

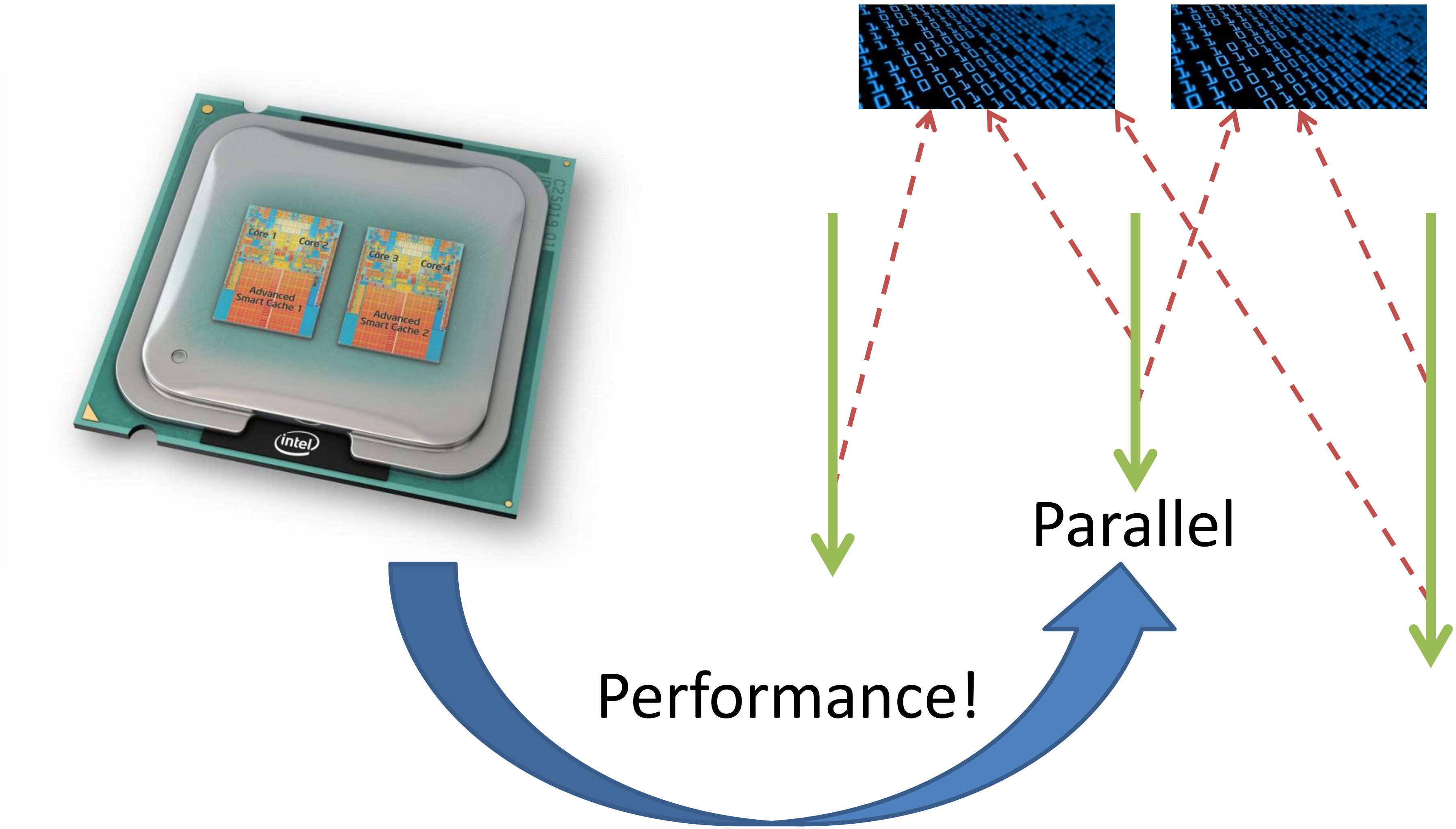
Dongyun Jin

Qingzhou Luo

Grigore Rosu

Darko Marinov

Multicore World



Shared Memory Multithreaded

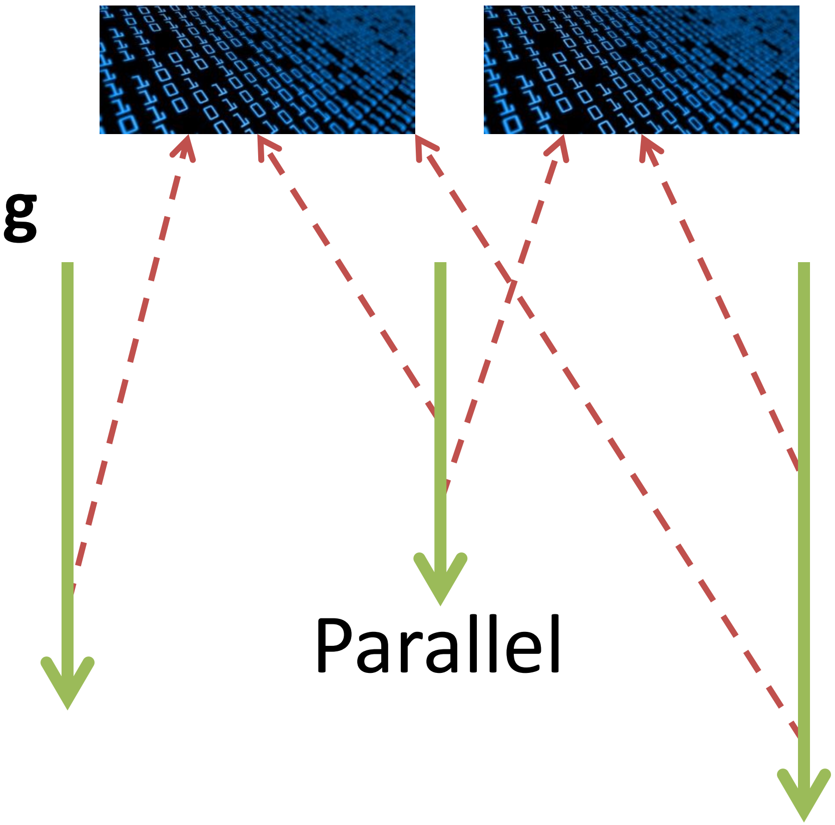
Parallel

Performance!

Difficult to Develop Correct Multithreaded Code

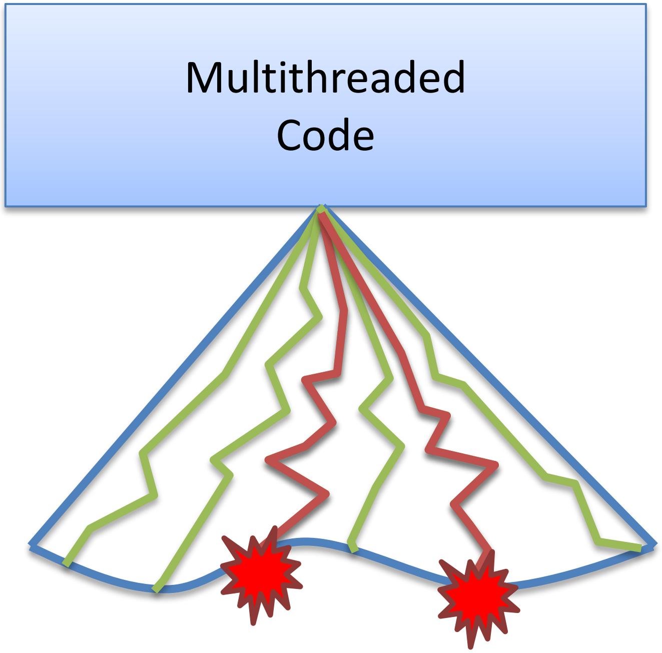
- **Non-deterministic scheduling**
- Data races
- Deadlocks
- Atomicity violations
- …

Shared Memory Multithreaded



Parallel

Difficult to Test Multithreaded Code



- Failures triggered by specific schedules
- Most research focuses on exploring schedules for given manually written tests on one given code version

Challenges in Unit Testing MT Code

1. How to **write** multithreaded unit tests?
   - Developers often want to test specific schedules
   - How to **express schedules** in unit tests?
2. How to **explore** multithreaded unit tests?
   - Current techniques focus on one code version
   - Code evolves, need **efficient regression testing**
3. How to **generate** multithreaded unit tests?
   - How to **automatically generate test code**?
   - How to **automatically generate schedules**?

Our Work on All Three Topics

1. **Writing multithreaded unit tests (this talk)**
   - IMUnit: Illinois/improved multithreaded unit testing [ESEC/FSE’11]
     - Read “immunity”: isolate code from bugs

2. Regression testing
   - Prioritizing exploration of change-impacted schedules [ISSTA’11]
   - Selecting schedules under changes [ICST’10, STVR’12?]

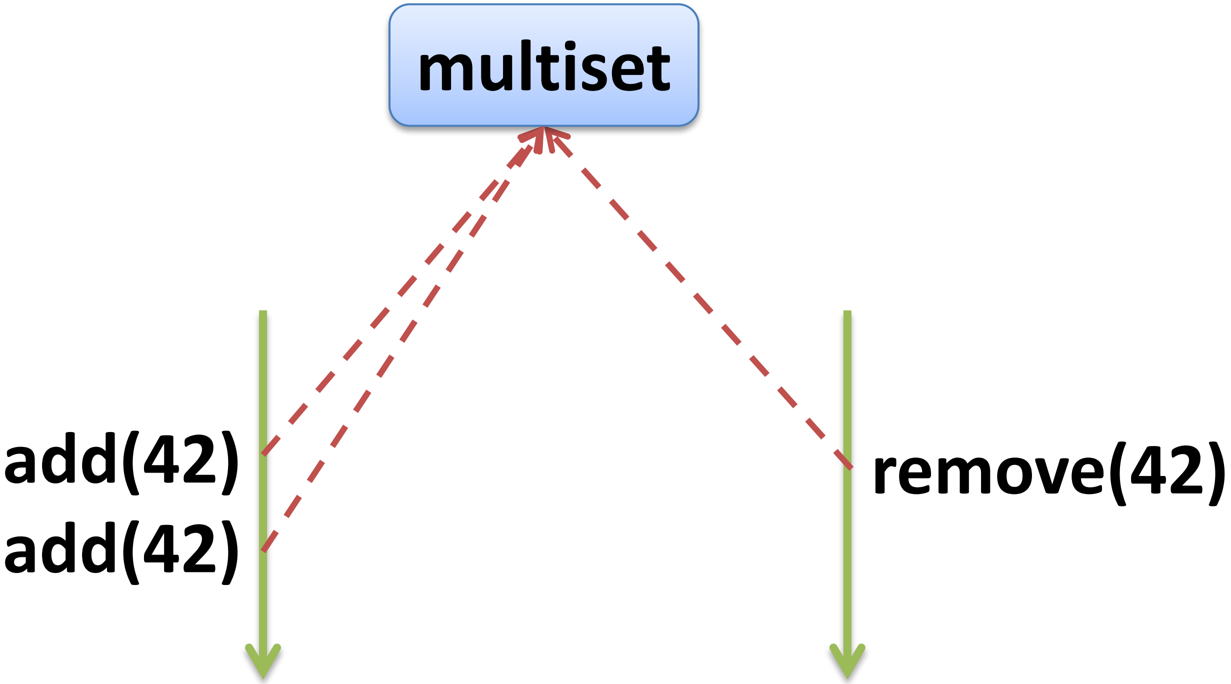
3. Generating tests
   - Generating schedules [ICSE’08]
   - Generating code [ICSE’12]

Example: ConcurrentHashMultiSet

- Thread-safe Multiset aka Bag implementation
- Provided by Guava (Google Collections)
- Consider testing these three methods

```
package com.google.common.collect;
public class ConcurrentHashMultiSet<E>
{
    boolean add(E element) …
    boolean remove(Object element) …
    int count(Object element) …
    …
}
```

Testing Adds and Remove



multiset

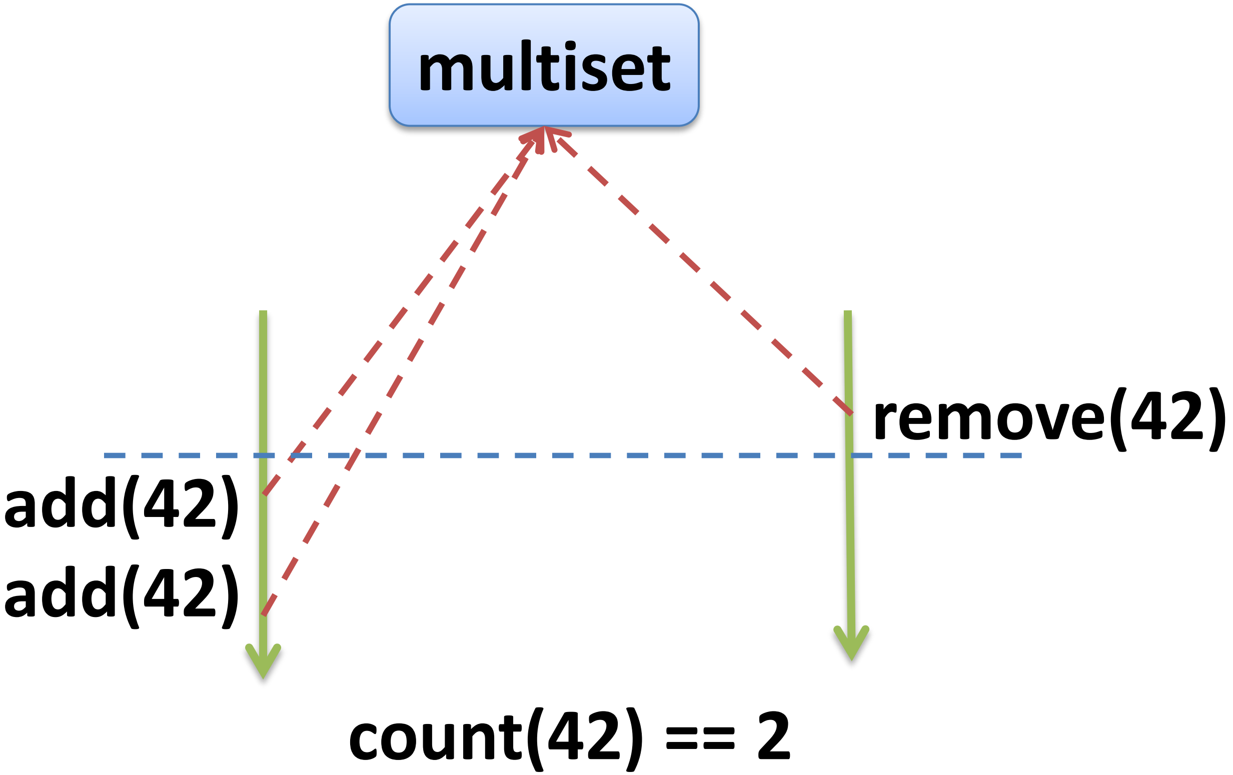
add(42)

add(42)

remove(42)

count(42) is schedule dependent

Testing Remove Before Adds

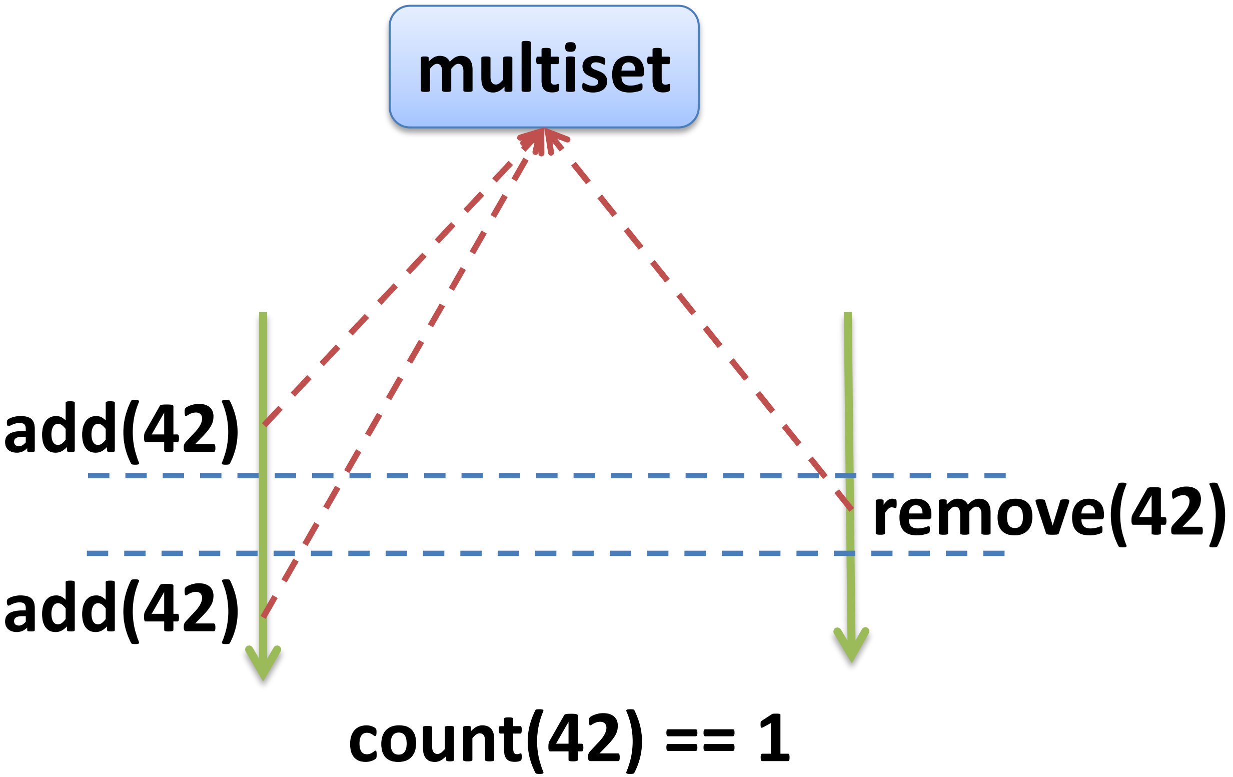


Sleep-Based Test: Remove Before Adds

```
@Test
 public void testRemoveBeforeAdds() … {
   …
   multiset= ConcurrentHashMultiset.create();
   Thread addThread = new Thread(new Runnable() {
     public void run() {
       Thread.sleep(60);
       multiset.add(42);
       multiset.add(42);
     }
   });
   addThread.start();
   multiset.remove(42);
   addThread.join();
   assertEquals(2, multiset.count(42));
 }
```

Sleep used to express and enforce schedule

Testing Remove Between Adds



multiset

add(42)

remove(42)

add(42)

count(42) == 1

Sleep-Based Test: Remove Between Adds

```
@Test
 public void testRemoveBetweenAdds() … {
   …
   multiset = ConcurrentHashMultiset.create();
   Thread addThread = new Thread(new Runnable() {
     public void run() {
       multiset.add(42);
       Thread.sleep(80);
       multiset.add(42);
     }
   });
   addThread.start();
   Thread.sleep(40);
   multiset.remove(42);
   addThread.join();
   assertEquals(1, multiset.count(42));
 }
```

Sleeps used to express and enforce schedule

Sleep-Based Tests : Issues

```
@Test
 public void testRemoveBetweenAdds() … {
   …
   multiset = ConcurrentHashMultiset.create();
   Thread addThread = new Thread(new Runnable() {
     public void run() {
       multiset.add(42);
       Thread.sleep(80);
       multiset.add(42);
     }
   });
   addThread.start();
   Thread.sleep(40);
   multiset.remove(42);
   addThread.join();
   assertEquals(1, multiset.count(42));
 }
```

- **Fragile**
- **Inefficient**
- **Non modular**
- **Implicit schedule**

	Not buggy	**Buggy**
Pass	True Negative	**False Negative**
Fail	**False Positive**	True Positive

Others have also recognized issues…

- Previous solutions:
  - ConAn: Long, Hoffman and Strooper
  - ConcJUnit: Ricken and Cartwright
  - ThreadControl: Dantas, Brasileiro and Cirne
- Latest solution:
  - MultithreadedTC: Pugh and Ayewah
  - Tick-based tests
  - + Robust, Efficient
  - —But Non modular, Implicit schedule
  - —Different from traditional tests
- **IMUnit: Event-based tests**

IMUnit Test: Remove Before Adds

```
@Test
@Schedule("finishRemove->startingAdd1")
public void testRemoveBeforeAdds() … {
  …
  multiset = ConcurrentHashMultiset.create();
  Thread addThread = new Thread(new Runnable() {
    public void run() {
      @Event("startingAdd1");
      multiset.add(42);
      multiset.add(42);
    }
  });
  addThread.start();
  multiset.remove(42);
  @Event("finishRemove");
  addThread.join();
  assertEquals(2, multiset.count(42));
}
```

Event orderings used to specify schedules

IMUnit Test: Remove Before Adds

```
@Test
@Schedule("finishRemove->startingAdd1")
public void testRemoveBeforeAdds() … {
  …
  multiset = ConcurrentHashMultiset.create();
  Thread addThread = new Thread(new Runnable() {
    public void run() {
      @Event("startingAdd1");
      multiset.add(42);
      multiset.add(42);
    }
  });
  addThread.start();
  multiset.remove(42);
  @Event("finishRemove");
  addThread.join();
  assertEquals(2, multiset.count(42));
}
```

@Event: interesting point in execution of a thread

IMUnit Test: Remove Before Adds

```
@Test
@Schedule("finishRemove->startingAdd1")
public void testRemoveBeforeAdds() … {
  …
  multiset = ConcurrentHashMultiset.create();
  Thread addThread = new Thread(new Runnable() {
    public void run() {
      @Event("startingAdd1");
      multiset.add(42);
      multiset.add(42);
    }
  });
  addThread.start();
  multiset.remove(42);
  @Event("finishRemove");
  addThread.join();
  assertEquals(2, multiset.count(42));
}
```

@Schedule: set of event orderings

- **e -> e’ ≡ e before e’**
- **Partial order**

IMUnit Test: Remove Between Adds

```
@Test
@Schedule("finishAdd1->startingRemove, finishRemove->startingAdd2")
public void testRemoveBetweenAdds() … {
  …
  multiset = ConcurrentHashMultiset.create();
  Thread addThread = new Thread(new Runnable() {
    public void run() {
      multiset.add(42);
      @Event("finishAdd1");
      @Event("startingAdd2");
      multiset.add(42);
    }
  });
  addThread.start();
  @Event("startingRemove");
  multiset.remove(42);
  @Event("finishRemove");
  addThread.join();
  assertEquals(1, multiset.count(42));
}
```

@Events

IMUnit Test: Both Schedules

```
@Test
@Schedule("finishRemove->startingAdd1")
@Schedule("finishAdd1->startingRemove, finishRemove->startingAdd2")
public void testAddRemove() … {
  …
  multiset = ConcurrentHashMultiset.create();
  Thread addThread = new Thread(new Runnable() {
    public void run() {
      @Event("startingAdd1");
      multiset.add(42);
      @Event("finishAdd1");
      @Event("startingAdd2");
      multiset.add(42);
    }
  });
  addThread.start();
  @Event("startingRemove");
  multiset.remove(42);
  @Event("finishRemove");
  addThread.join();
  …
}
```

Schedule Language

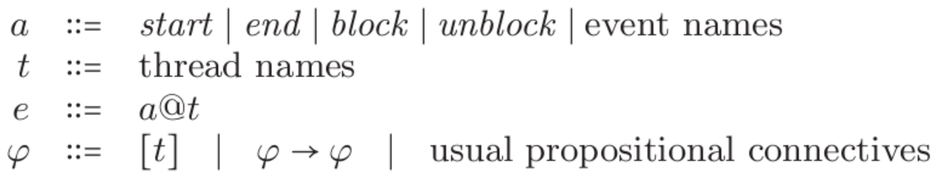
```
<Schedule>    ::= { <Ordering> "," } <Ordering>
<Ordering>    ::= <Condition> "->" <Basic Event>
<Condition>   ::= <Basic Event> | <Block Event> | <Condition> "||" <Condition>
                  | <Condition> "&&" <Condition> | "(" <Condition> ")"
<Block Event> ::= "[" <Basic Event> "]"
<Basic Event> ::= <Event Name> ["@" <Thread Name>]
                  | "start" "@" <Thread Name> | "end" "@" <Thread Name>
<Event Name>  ::= { <Id> "." } <Id>
<Thread Name> ::= <Id>
```

- Events
  - Two types: non-blocking-event and [blocking-event]
  - Can be parameterized by thread-name: event@threadName
  - Can also be combined into conditions using “||” and “&&”
- Ordering specifies order between a condition and event
  - “->” is the ordering operator
  - before-condition -> after-event
- Schedule is a comma-separated list of orderings

Schedule Logic

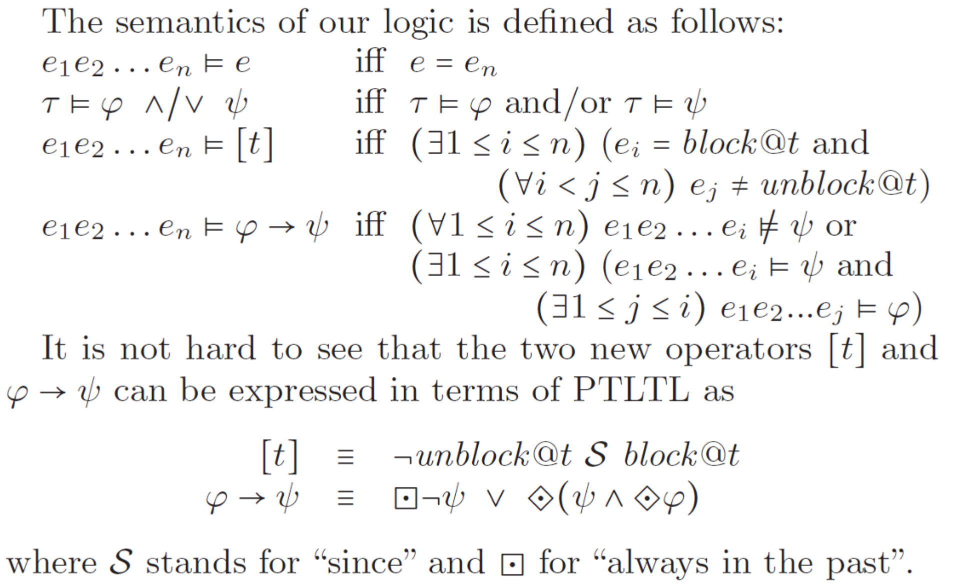
- Fragment of PTLTL
  - Over finite well formed multithreaded unit test execution traces
  - Two temporal operators
    - Block
    - Ordering
- Guided by practical requirements
  - Over 200 existing multithreaded unit tests
- Details in paper

Logic Syntax:



$$\begin{array}{rcl} a & ::= & \mathit{start} \mid \mathit{end} \mid \mathit{block} \mid \mathit{unblock} \mid \text{event names} \\ t & ::= & \text{thread names} \\ e & ::= & a@t \\ \varphi & ::= & [t] \quad \mid \quad \varphi \rightarrow \varphi \quad \mid \quad \text{usual propositional connectives} \end{array}$$

Logic Semantics:



The semantics of our logic is defined as follows:

$$\begin{array}{ll} e_1e_2\ldots e_n \models e & \text{iff} \;\; e = e_n \\ \tau \models \varphi \;\; \wedge/\vee \;\; \psi & \text{iff} \;\; \tau \models \varphi \text{ and/or } \tau \models \psi \\ e_1e_2\ldots e_n \models [t] & \text{iff} \;\; (\exists 1 \leq i \leq n)\ (e_i = \mathit{block}@t \text{ and} \\ & \qquad\qquad (\forall i < j \leq n)\ e_j \neq \mathit{unblock}@t) \\ e_1e_2\ldots e_n \models \varphi \rightarrow \psi & \text{iff} \;\; (\forall 1 \leq i \leq n)\ e_1e_2\ldots e_i \not\models \psi \text{ or} \\ & \;\;\;\; (\exists 1 \leq i \leq n)\ (e_1e_2\ldots e_i \models \psi \text{ and} \\ & \qquad\qquad (\exists 1 \leq j \leq i)\ e_1e_2 ... e_j \models \varphi) \end{array}$$

It is not hard to see that the two new operators $[t]$ and $\varphi \rightarrow \psi$ can be expressed in terms of PTLTL as

$$\begin{array}{rcl} [t] & \equiv & \neg \mathit{unblock}@t \;\mathcal{S}\; \mathit{block}@t \\ \varphi \rightarrow \psi & \equiv & \boxdot \neg \psi \;\vee\; \Diamonddot(\psi \wedge \Diamonddot \varphi) \end{array}$$

where $\mathcal{S}$ stands for "since" and $\boxdot$ for "always in the past".

Schedule Enforcement

- Two implementations: **original** and **light**
- Original implemented using JavaMOP
- Schedule logic implemented as JavaMOP logic plugin
- Takes as input a schedule and outputs a monitor
- Monitor aspects are weaved into test code
- Different monitor for each test, schedule pair
- Monitor can work in two modes:
  - Active mode enforces schedules
  - Passive mode prints error if execution deviates from schedule

IMUnit Light

- Original implementation:
  - Preprocessing for @Event
  - Instrumentation to weave in monitor
  - Dependency on AspectJ etc
- IMUnit light
  - Just need imunit.jar on classpath
  - fireEvent (“eventName”) instead of @Event
  - Centralized monitor provided by library
  - Even more efficient

IMUnit Event-Based Tests: Features

```
@Test
@Schedule("finishRemove->startingAdd1")
@Schedule("finishAdd1->startingRemove, finishRemove->startingAdd2")
public void testAddRemove() … {
  …
  multiset = ConcurrentHashMultiset.create();
  Thread addThread = new Thread(new Runnable() {
    public void run() {
      @Event("startingAdd1");
      multiset.add(42);
      @Event("finishAdd1");
      @Event("startingAdd2");
      multiset.add(42);
    }
  });
  addThread.start();
  @Event("startingRemove");
  multiset.remove(42);
  @Event("finishRemove");
  addThread.join();
  …
}
```

- + **Robust**
- + **Efficient**
- + **Modular**
- + **Explicit schedule**

Manual Migration

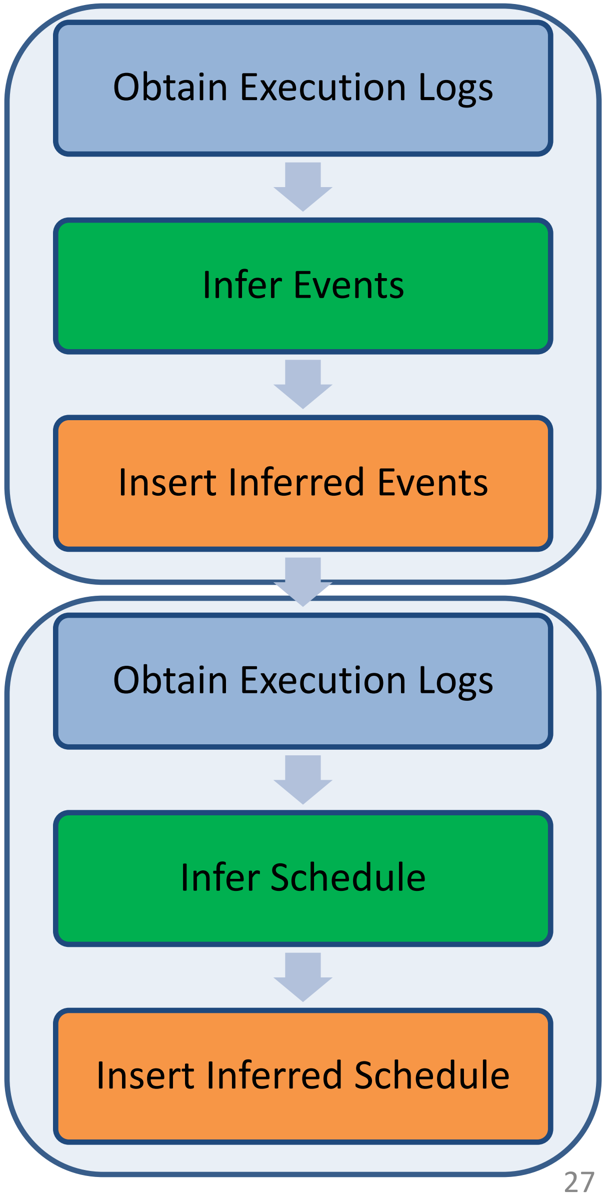
We manually migrated over 200 sleep-based tests to IMUnit

Migration typically involved the following steps:

1. Optionally name threads (default names non-deterministic)
2. Introduce events using @Event annotations
   - Need to identify interesting points
3. Introduce schedule using @Schedule annotation
   - Need to understand intended sleep-based schedule
   - Specify the orderings required by intended schedule
   - Also identify blocking vs. non-blocking events
4. Check that added schedule is the intended schedule
5. Remove sleeps
6. Optionally merge tests with different schedules but similar code

Automated Migration

- Introducing events and schedules most challenging
- Inferred using execution logs of sleep-based tests
- Two phase process:
  - Inferring likely events
    - Precision: 75% , Recall: 79%
  - Inferring likely schedules
    - Precision: 96%, Recall: 94%
- More details in paper



Evaluation

- Expressiveness of schedule language
- Efficiency of schedule enforcement

Expressiveness of Schedule Language

- Experience with migrating over 200 sleep-based unit tests
  - 7 different open source projects
- Evolved language using migration experience
  - Blocking events added because they were required by many tests
  - Events in loops were only required for 5 tests so not added yet
- Replaced sleeps with events and schedules in 198 tests

Subject	Tests	Events	Orderings
Commons Collections	18	51	32
JBoss-Cache	27	105	47
Lucene	2	3	4
Mina	1	2	1
Pool	2	8	3
Sysunit	9	33	34
JSR-166 TCK	139	577	277
Σ	198	779	398

Efficiency of Schedule Enforcement

- IMUnit test execution vs. sleep-based test execution
- IMUnit test execution more than 3X faster
  - Schedule enforcement is efficient
- Also demonstrates the over estimation of sleep delays
  - Sleeps are inefficient

Subject	Original [s]	IMUnit [s]	Speedup
Commons Collections	4.96	1.06	4.68
JBoss-Cache	65.58	31.25	2.10
Lucene	11.02	3.57	3.09
Mina	0.26	0.17	1.53
Pool	1.43	1.04	1.38
Sysunit	17.67	0.35	50.49
JSR-166 TCK	15.20	9.56	1.59
		Geometric Mean	3.39

Writing Multithreaded Unit Tests…

- Dominant solution: sleep-based
  - Fragile, Inefficient, Non-Modular, Implicit
- **IMUnit: event-based**
  - Robust, Efficient, Modular, Explicit
  - Schedule language is expressive
  - Schedule enforcement is efficient
  - Automated migration
  - More details in paper

http://mir.cs.illinois.edu/imunit