Cooperari

A tool for cooperative testing of multithreaded Java programs

Eduardo R. B. Marques, Francisco Martins, Miguel Simões

LASIGE/Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa

2014 International Conference on Principles and Practices of Programming on the Java Platform, Krakow, Poland, September 25, 2014

Outline

- Background
  - Why is multithreaded (Java) code hard to test?
  - How does cooperative semantics help ?
- The Cooperari tool
  - Design & Implementation
  - Results
  - Future work
  - Brief demo (if I have time)

A simple Java semaphore

```
package dijkstra;
class Semaphore {
  int value;
  Semaphore(int initial) { value = initial; }
  int getValue() { return value; }
  void down() throws InterruptedException {
    synchronized (this) {
      while (value == 0) { wait(); }
      value = value - 1;
    }
  }
  void up() {
    synchronized (this) {
      value = value + 1;
      if (value == 1) { notify(); }
    }
  }
}
```

Semaphore test

```
private static int N = . . .;
@Test
public final void test() {
  Semaphore s = new Semaphore(N-1);
  Runnable[] c = new Runnable[N];
  for (int i=0; i < N; i++) c[i] = new Client(s);
  runThreads(c);
  assertEquals(N-1, s.getCount());
}

static class Client implements Runnable {
  private Semaphore sem;
  Client(Semaphore s) { sem = s; }
  public void run() {
    try {
      sem.down();
      . . .
      sem.up();
    } catch(InterruptedException e) { . .. }
  }
}
```

Bugs are easy to come by …

```
void up() {
  value = value + 1;
  if (value == 1) notify();
}
```

```
void up() {
  value = value + 1;
  synchronized (this) {
   if (value == 1) notify();
  }
}
```

```
void up() {
  synchronized (this) {
    value = value + 1;
  }
  synchronized (this) {
    if (value == 1) notify();
  }
}
```

```
void up() {
  synchronized (this) {
    value = value + 1;
  }
  if (value == 1)
    synchronized(this) {
      notify();
    }
}
```

Bad use of monitors/shared data may lead to data races, deadlocks etc
An error in up() may deadlock a thread that is trying to down() the semaphore

Standard testing fails to expose bugs

- Even simple bug patterns are elusive to detect and reproduce precisely.
- Bugs may be exposed only for (sometimes very) particular thread schedules.
- Scheduler operates preemptively, non-deterministically, and context switches are too coarse-grained.
- Code is also hard or impossible to debug. Heisenbugs are common.

Cooperative semantics

- Key observation: relevant context switches happen at thread interference points
  - shared data access (read/write)
  - multithreading primitives (locks, monitor notifications, barriers, etc)
- Cooperative semantics
  - **Let threads yield voluntarily at interference points.**
  - **And let code between two yield points run serially as a transaction**, without interference from other threads
  - The semantics of a program is fully preserved as long as yield points are fully identified [Yi et al., ISSTA’12, PPoPP’11]
  - Potential for deterministic testing + custom state-space exploration (eventually an exhaustive one)

Yield points

```
void down() … {
  synchronized (this) {
    while (value == 0) {
      wait();
    }
    value = value - 1;
  }
}
```

① lock acquisition

② field read

③ wait() call

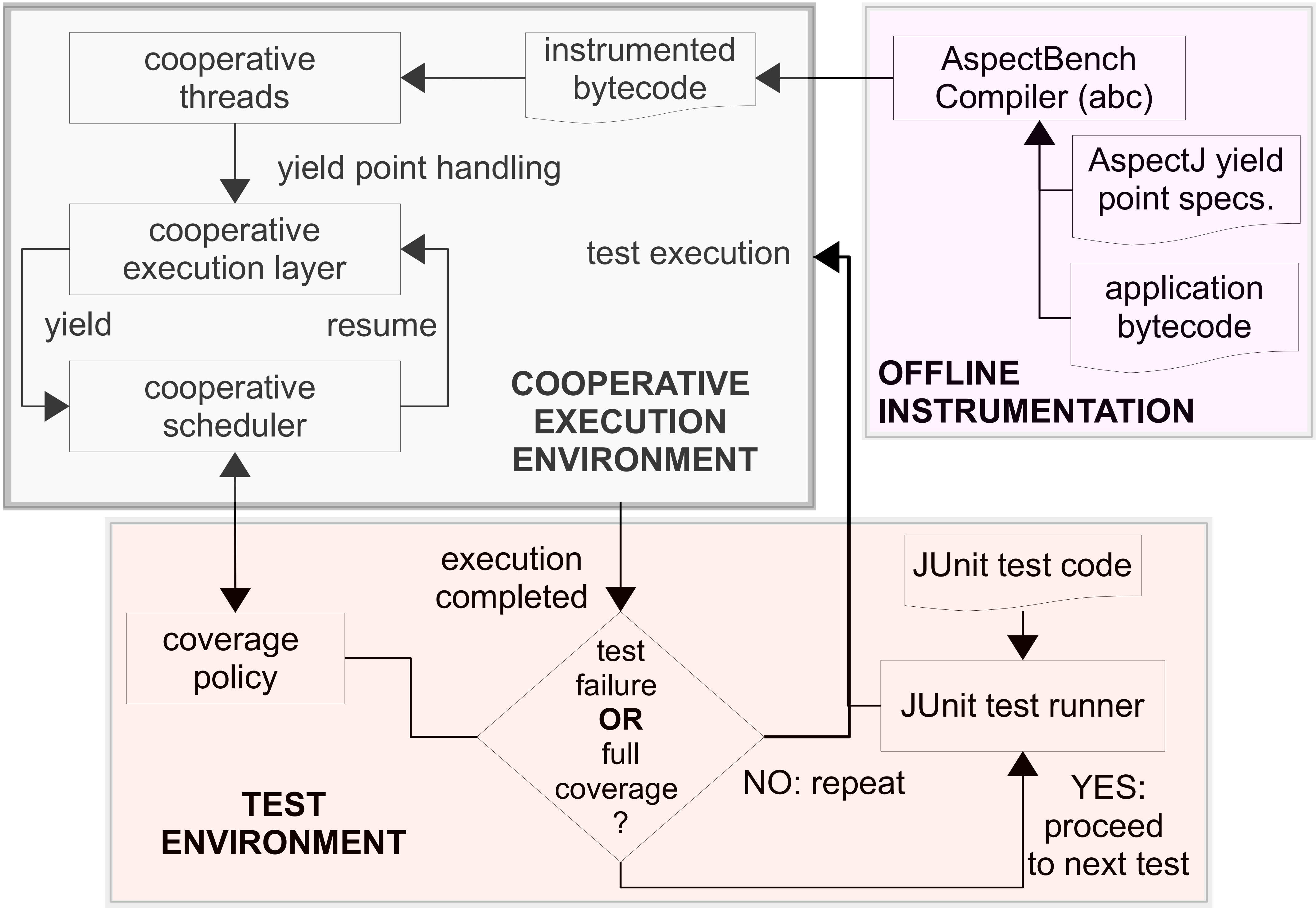
④ ⑤ field read + write

⑥ lock release

Cooperari in a nutshell

- AspectJ-based bytecode instrumentation intercepts yield points
  - no changes to JVM itself (a standard JVM can be used)
  - yield point support: data access, monitor operations, and most java.lang.Thread methods
- Yield point interception delegates full control to
  - cooperative scheduler that rules out interference from JVM scheduler
  - “virtualization layer” for deterministic execution of multithreading primitives
- Coverage policies
  - for custom exploration of the state-space of thread schedules
- Deadlock & race detection mechanisms
- Integrates with the JUnit framework.

Cooperari in a nutshell



Test execution

data races reported

```
Race: T0 at Semaphore.java:12 over Semaphore.value
Race: T1 at Semaphore.java:13 over Semaphore.value
Failure trace for test written to
    'log/examples.semaphore.TestSemaphore_test.trace.txt'
test: executed 36 times in 578 ms [failed]
1) test(examples.semaphore.TestSemaphore)
   WaitDeadlockError: { T2/Semaphore.java:7 }
```

cooperative trace log

thread T2 cannot complete wait() call within down()

(semaphore example from the paper)

Cooperative trace log

```
<step> <thread> <yield point>                   # r/w
15 0 monitorenter(L0) Semaphore.java:12         # {}/{}
...
22 1 monitorenter(L0) Semaphore.java:12
23 2 wait(L0) Semaphore.java:7
24 1 get(Semaphore.value) Semaphore.java:12
25 1 set(Semaphore.value) Semaphore.java:12
26 1 monitorexit(L0) Semaphore.java:12
27 1 get(Semaphore.value) Semaphore.java:12 # 1/{}
28 0 get(Semaphore.value) Semaphore.java:12 # 0,1/{}
29 0 set(Semaphore.value) Semaphore.java:12 # 1/0 [race]
30 0 monitorexit(L0) Semaphore.java:12      # [value=2]
31 0 get(Semaphore.value) Semaphore.java:13 # 0,1/{}
32 0 <end> # {}/{};read 2; no call to notify()
33 1 <end> # read 2; no call to notity()
```

thread T2 cannot complete wait() call within down()

(semaphore example from the paper)

Instrumentation pattern

```
void around (Object o) : lock() && args(o) {
  CThread t = CThread.get(thisJoinPoint, o.getClass());
  if (t != null) {
    t.cMonitorEnter(o);
  } else {
    proceed(o);
  }
}
```

Lock acquisition is diverted to the cooperative execution layer.

- Example: interception / instrumentation of lock acquisition
  - the JVM `monitorenter` instruction.
  - we do something similar for lock release, Thread methods, data access
  - Note: AspectBench Compiler (abc) incorporates AspectJ pointcut extensions for lock and unlock operations (from the RacerAJ tool)
- If a thread is cooperative, i.e., launched via `runThreads`(), the execution will be *diverted* to the Cooperari runtime
  - monitorenter will not execute at all in that case

Thread yield & resumption

thread yields …

```
public void cYield() {
   ... // yielding
   yield = true;
   syncYield();
   while (yield) {
     LockSupport.park();
   }
   syncResume();
   ... // resumed
 }
```

`LockSupport.park()` Java API call disables execution for the thread by the JVM scheduler.

Invariant: only one cooperative thread can be picked up for execution by the JVM scheduler at any given time.

… and is later resumed

```
 public void cResume() {
   ...
   yield = false;
   LockSupport.unpark(this);
 }
```

`LockSupport.unpark()` lets the JVM scheduler pick up the thread again

Cooperative scheduler

- On a thread yield:
  - the cooperative scheduler must pick the thread to run next
  - the decision is up to the coverage policy that is set
  - scheduler then resumes the chosen thread

```
CoveragePolicy policy;
List<CThread> ready;
...
public void cStep() {
  ...
  syncYield();
  CThread t = policy.decision(ready);
  t.cResume();
  syncResume();
  ...
}
```

thread selection
& resumption

Coverage policies

- Pseudo-random policy
  - Memory-less
  - Chooses thread at random
  - Uses fixed seed for repeatable test sessions
- History-dependent policy (see paper for details)
  - Keeps a track of past decisions across multiple test trials
  - program state abstraction + partial order reduction technique to limit search space
  - No backtracking though …

Multithreading primitives

virtual monitor datum

```
class CMonitor {
  // Reference count.
  int refCount;
  // Owner thread
  CThread owner;
    // Wait count
  int waitCount;
  // Notification epoch
  long nEpoch;
  // Notification queue
  Queue<Integer> nQueue;
}
```

lock acquisition example

```
  CMonitor m;
  void init(Object o) {
    m = getMonitor(o);
    m.refCount++;
  }
  CState getState() {
   return
     m.owner == null ?
       CREADY : CBLOCKED;
  }
  void complete(CThread t) {
    m.owner = t;
  }
```

on yield

ready-state report

on resumption

Cooperative implementation is required for deterministic operation.

Deadlock & race detection

- Cooperative execution naturally exposes deadlocks and races …
    - we merely need to observe the execution step-by-step
    - simple detection mechanisms may be employed
- Deadlock detection
    - resource graph for lock-acquisition cycles
    - + plain check to see if no thread can progress
- Race detection
    - Pending read/write counter for each active data accesses
    - At most one pending read/write access per thread
- More details in the paper

Benchmark	Hist. dep. coverage					Random coverage					Unconstrained execution				
	2	4	8	16	32	2	4	8	16	32	2	4	8	16	32
Alarm Clock	*1000*	7	3	1	2	*1000*	8	13	8	6	*1000*	——— **1000** ———			
	64.9	1.4	1.1	0.7	1.2	62.3	1.4	2.2	1.9	1.8	51.0	51.4	51.6	52.0	52.7
Apache Common Lang					1	1	1	1	1	1	——— **1000** ———				
					0.8	0.2	0.2	0.4	0.5	0.8	0.3	0.6	0.9	1.4	2.2
Bank	1	1	1	3	1	1	1	1	2	1	——— **1000** ———				
	0.1	0.2	0.4	1.0	1.0	0.1	0.2	0.3	1.1	1.0	1.4	1.5	1.9	2.6	3.7
Clean	3	4	2	1	1	21	2	1	1	1	25	3	1	1	1
	0.4	2.0	1.7	1.4	1.6										
Dining Philosophers	2	9	48	581	**1000**										
	0.1	0.2	1.0	18.8	189.4										
Linked List															
Merge Sort															
	4.6	6.8	17.7	26.9	3.2										
Piper	*1000*	2	2	1	1										
	7.7	0.3	0.3	0.6	0.8	7.2	0.1	0.2	0.4	0.7	0.7	0.1	0.1	0.1	0.1
Reorder	50	13	4	7	20	2	23	26	17	10	——— **1000** ———				
	0.4	0.2	0.2	0.5	1.5	0.1	0.3	0.7	0.8	0.9	0.3	0.4	0.8	1.4	2.3
Semaphore	*1000*	37	137	**1000**	**1000**	*1000*	8	249	**1000**	**1000**	*1000*	——— **1000** ———			
	6.5	0.7	2.7	24.4	73.1	6.0	0.2	2.4	32.1	63.0	0.3	0.5	0.8	1.2	1.9
Two Stage					28	324	141	157	92	46	——— **1000** ———				
					3.2	3.6	2.5	3.6	0.3	4.0	0.3	0.4	0.7	1.3	2.4
Wrong Lock					1	5	1	2	3	1	——— **1000** ———				
	0.1	0.1	0.2	0.9	0.5	0.1	0.1	0.2	0.5	0.5	0.3	0.4	0.8	1.3	2.2

thread count

test trials till bug found (up to 1000)

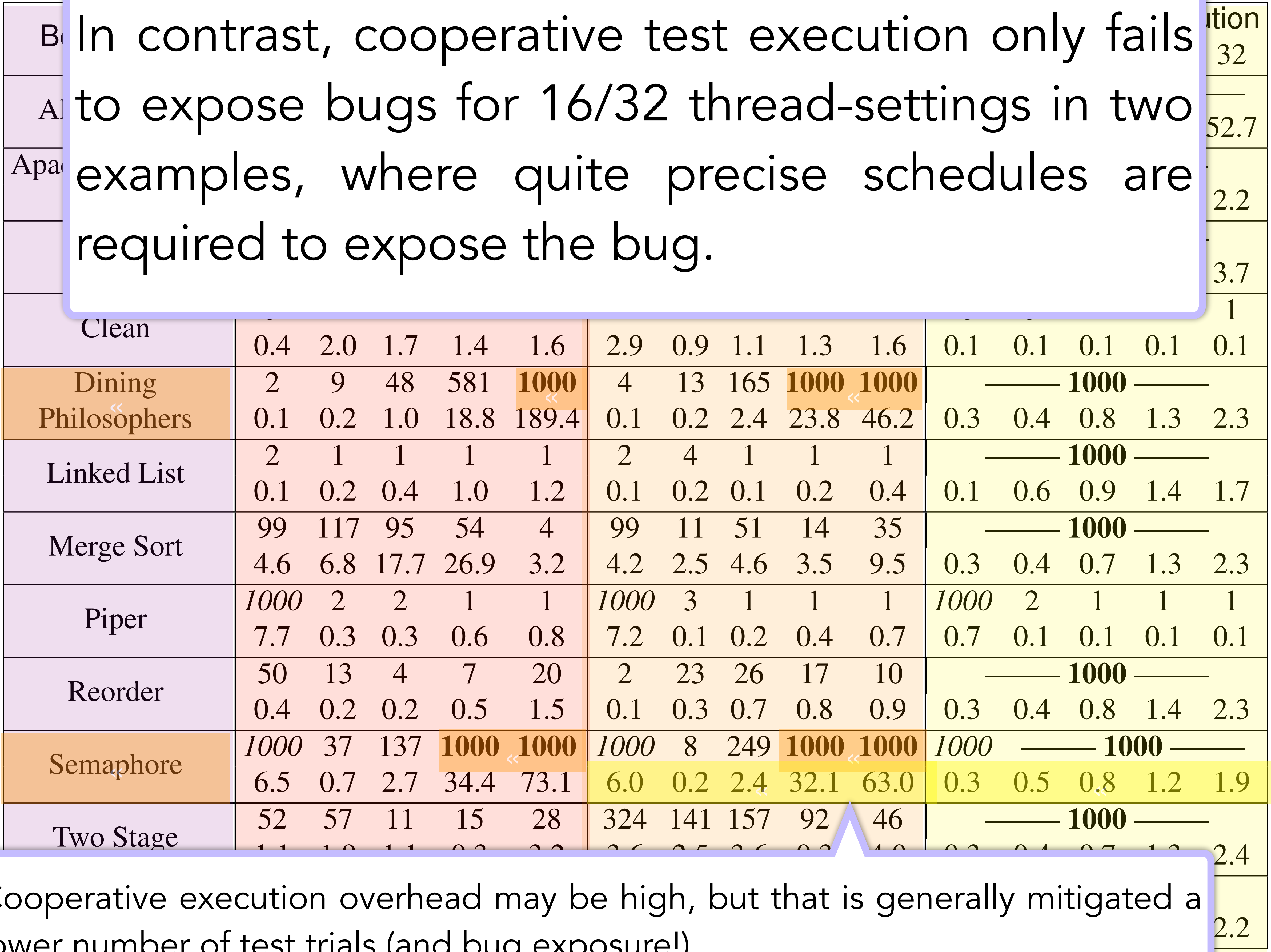
execution time for all test trials

Results for 12 benchmarks from the ConTest + SIR suites.

Benchmark	Hist. dep. coverage					Random coverage					Unconstrained execution				
	2	4	8	16	32	2	4	8	16	32	2	4	8	16	32
Alarm Clock	*10*										*1000*	——— **1000** ———			
	64									8	51.0	51.4	51.6	52.0	52.7
Apache Common Lang											——— **1000** ———				
	0									8	0.3	0.6	0.9	1.4	2.2
Bank											——— **1000** ———				
	0									0	1.4	1.5	1.9	2.6	3.7
Clean											25	3	1	1	1
	0									6	0.1	0.1	0.1	0.1	0.1
Dining Philosophers										**00**	——— **1000** ———				
	0.1	0.2	1.0	18.8	189.4	0.1	0.2	2.4	23.8	46.2	0.3	0.4	0.8	1.3	2.3
Linked List	2	1	1	1	1	2	4	1	1	1	——— **1000** ———				
	0.1	0.2	0.4	1.0	1.2	0.1	0.2	0.1	0.2	0.4	0.1	0.6	0.9	1.4	1.7
Merge Sort	99	117	95	54	4	99	11	51	14	35	——— **1000** ———				
	4.6	6.8	17.7	26.9	3.2	4.2	2.5	4.6	3.5	9.5	0.3	0.4	0.7	1.3	2.3
Piper	*1000*	2	2	1	1	*1000*	3	1	1	1	*1000*	2	1	1	1
	7.7	0.3	0.3	0.6	0.8	7.2	0.1	0.2	0.4	0.7	0.7	0.1	0.1	0.1	0.1
Reorder	50	13	4	7	20	2	23	26	17	10	——— **1000** ———				
	0.4	0.2	0.2	0.5	1.5	0.1	0.3	0.7	0.8	0.9	0.3	0.4	0.8	1.4	2.3
Semaphore	*1000*	37	137	**1000**	**1000**	*1000*	8	249	**1000**	**1000**	*1000*	——— **1000** ———			
	6.5	0.7	2.7	34.4	73.1	6.0	0.2	2.4	32.1	63.0	0.3	0.5	0.8	1.2	1.9
Two Stage	52	57	11	15	28	324	141	157	92	46	——— **1000** ———				
	1.1	1.9	1.1	0.3	3.2	3.6	2.5	3.6	0.3	4.0	0.3	0.4	0.7	1.3	2.4
Wrong Lock	5	1	2	7	1	5	1	2	3	1	——— **1000** ———				
	0.1	0.1	0.2	0.9	0.5	0.1	0.1	0.2	0.5	0.5	0.3	0.4	0.8	1.3	2.2

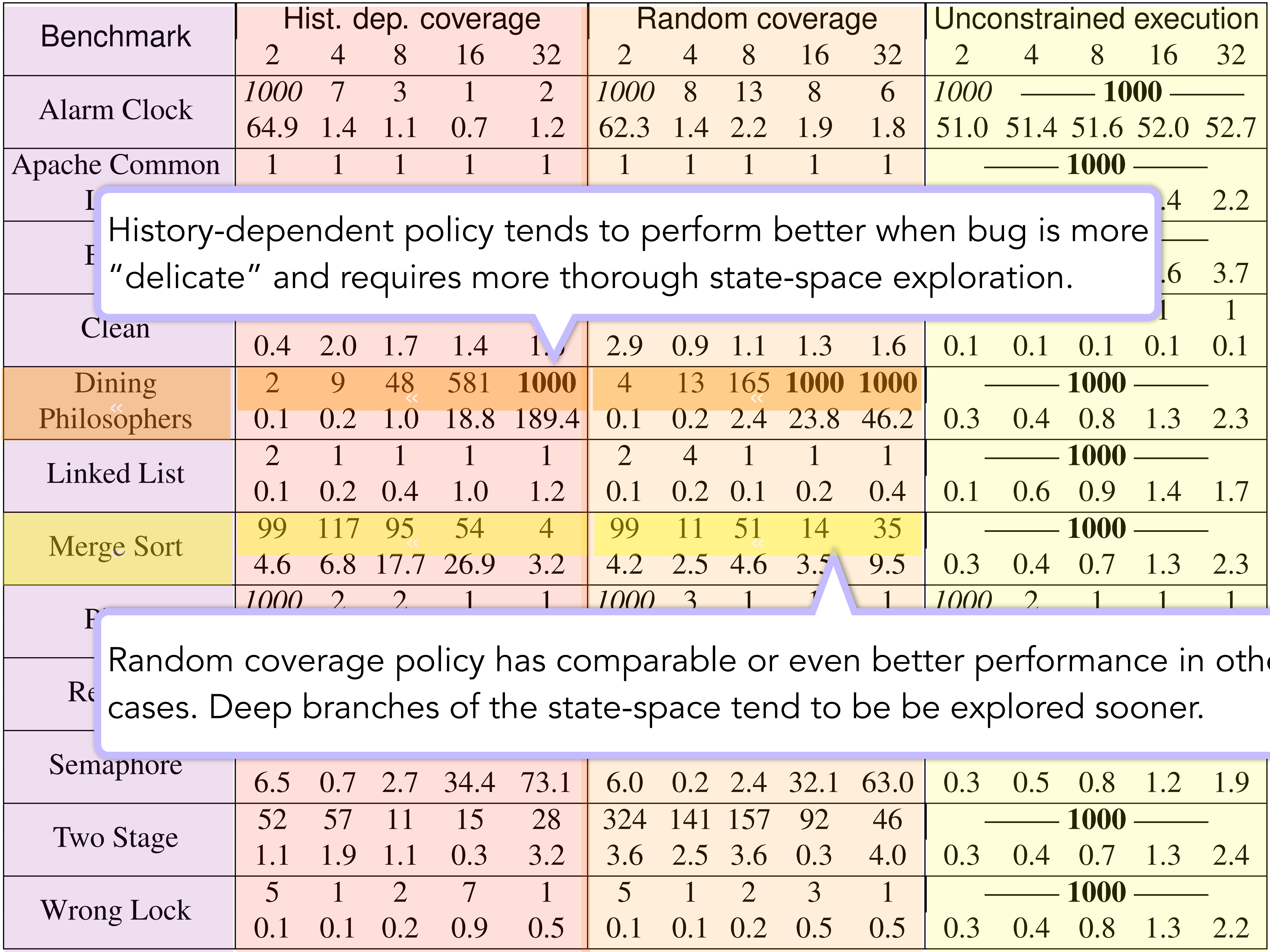
Unconstrained test execution fails to expose bugs except in two cases.

In contrast, cooperative test execution only fails to expose bugs for 16/32 thread-settings in two examples, where quite precise schedules are required to expose the bug.



															tion 32
															52.7
															2.2
															3.7
Clean															1
	0.4	2.0	1.7	1.4	1.6	2.9	0.9	1.1	1.3	1.6	0.1	0.1	0.1	0.1	0.1
Dining Philosophers	2	9	48	581	**1000**	4	13	165	**1000**	**1000**	**1000**				
	0.1	0.2	1.0	18.8	189.4	0.1	0.2	2.4	23.8	46.2	0.3	0.4	0.8	1.3	2.3
Linked List	2	1	1	1	1	2	4	1	1	1	**1000**				
	0.1	0.2	0.4	1.0	1.2	0.1	0.2	0.1	0.2	0.4	0.1	0.6	0.9	1.4	1.7
Merge Sort	99	117	95	54	4	99	11	51	14	35	**1000**				
	4.6	6.8	17.7	26.9	3.2	4.2	2.5	4.6	3.5	9.5	0.3	0.4	0.7	1.3	2.3
Piper	*1000*	2	2	1	1	*1000*	3	1	1	1	*1000*	2	1	1	1
	7.7	0.3	0.3	0.6	0.8	7.2	0.1	0.2	0.4	0.7	0.7	0.1	0.1	0.1	0.1
Reorder	50	13	4	7	20	2	23	26	17	10	**1000**				
	0.4	0.2	0.2	0.5	1.5	0.1	0.3	0.7	0.8	0.9	0.3	0.4	0.8	1.4	2.3
Semaphore	*1000*	37	137	**1000**	**1000**	*1000*	8	249	**1000**	**1000**	*1000*	**1000**			
	6.5	0.7	2.7	34.4	73.1	6.0	0.2	2.4	32.1	63.0	0.3	0.5	0.8	1.2	1.9
Two Stage	52	57	11	15	28	324	141	157	92	46	**1000**				
															2.4
															2.2

Cooperative execution overhead may be high, but that is generally mitigated a lower number of test trials (and bug exposure!)

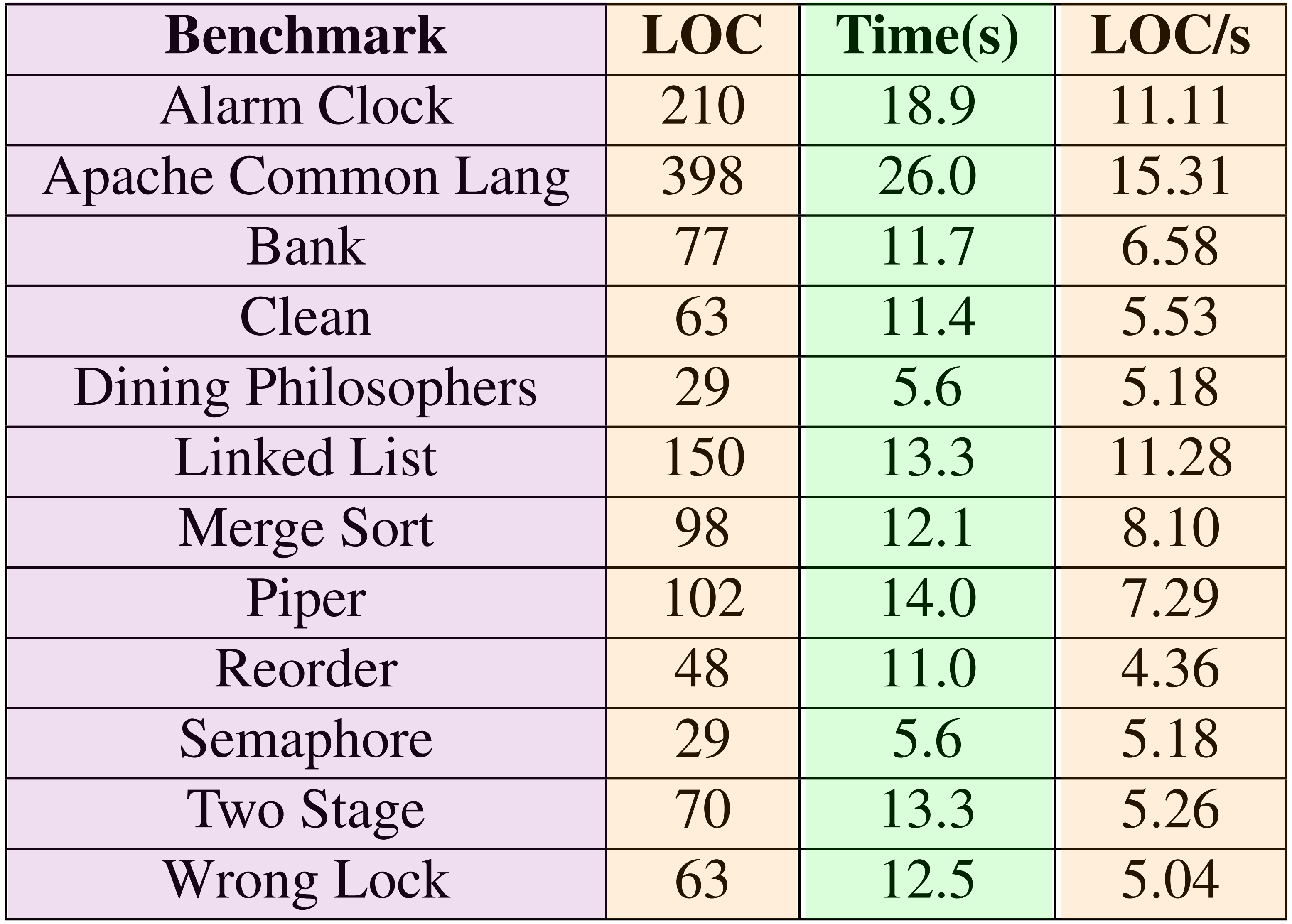


Benchmark	Hist. dep. coverage					Random coverage					Unconstrained execution				
	2	4	8	16	32	2	4	8	16	32	2	4	8	16	32
Alarm Clock	*1000*	7	3	1	2	*1000*	8	13	8	6	*1000*	**1000**			
	64.9	1.4	1.1	0.7	1.2	62.3	1.4	2.2	1.9	1.8	51.0	51.4	51.6	52.0	52.7
Apache Common I	1	1	1	1	1	1	1	1	1	1	**1000**				
														.4	2.2
F															
														.6	3.7
Clean														1	1
	0.4	2.0	1.7	1.4	1.	2.9	0.9	1.1	1.3	1.6	0.1	0.1	0.1	0.1	0.1
Dining Philosophers	2	9	48	581	**1000**	4	13	165	**1000**	**1000**	**1000**				
	0.1	0.2	1.0	18.8	189.4	0.1	0.2	2.4	23.8	46.2	0.3	0.4	0.8	1.3	2.3
Linked List	2	1	1	1	1	2	4	1	1	1	**1000**				
	0.1	0.2	0.4	1.0	1.2	0.1	0.2	0.1	0.2	0.4	0.1	0.6	0.9	1.4	1.7
Merge Sort	99	117	95	54	4	99	11	51	14	35	**1000**				
	4.6	6.8	17.7	26.9	3.2	4.2	2.5	4.6	3.5	9.5	0.3	0.4	0.7	1.3	2.3
P	*1000*	2	2	1	1	*1000*	3	1	1	1	*1000*	2	1	1	1
Re															
Semaphore															
	6.5	0.7	2.7	34.4	73.1	6.0	0.2	2.4	32.1	63.0	0.3	0.5	0.8	1.2	1.9
Two Stage	52	57	11	15	28	324	141	157	92	46	**1000**				
	1.1	1.9	1.1	0.3	3.2	3.6	2.5	3.6	0.3	4.0	0.3	0.4	0.7	1.3	2.4
Wrong Lock	5	1	2	7	1	5	1	2	3	1	**1000**				
	0.1	0.1	0.2	0.9	0.5	0.1	0.1	0.2	0.5	0.5	0.3	0.4	0.8	1.3	2.2

History-dependent policy tends to perform better when bug is more “delicate” and requires more thorough state-space exploration.

Random coverage policy has comparable or even better performance in other cases. Deep branches of the state-space tend to be be explored sooner.

Bytecode instrumentation using abc is slow, but it is performed only when required, driven by code (bytecode) changes.



Benchmark	LOC	Time(s)	LOC/s
Alarm Clock	210	18.9	11.11
Apache Common Lang	398	26.0	15.31
Bank	77	11.7	6.58
Clean	63	11.4	5.53
Dining Philosophers	29	5.6	5.18
Linked List	150	13.3	11.28
Merge Sort	98	12.1	8.10
Piper	102	14.0	7.29
Reorder	48	11.0	4.36
Semaphore	29	5.6	5.18
Two Stage	70	13.3	5.26
Wrong Lock	63	12.5	5.04

Future work

- More work on coverage policies
  - e.g., employ stateless model checker for exhaustive state-space exploration (like more robust systems such as CHESS, Cloud9, CONCURRIT)
- More in-depth analysis using larger real-world programs
- Cover a larger set of multithreaded Java primitives
  - e.g., atomic operations, barriers amongst others in java.util.concurrent
- Yield point inference
  - code is over-instrumented in many cases
- Misc. technical improvements
  - replace bytecode instrumentation framework (abc no longer actively maintained and has a few bugs)
  - develop an Eclipse IDE plugin

Thank you

For more info check out

bitbucket.org/edrdo/cooperari