Effective Static Deadlock Detection

Mayur Naik*

Chang-Seo Park+, Koushik Sen+, David Gay*

*Intel Research, Berkeley   + UC Berkeley

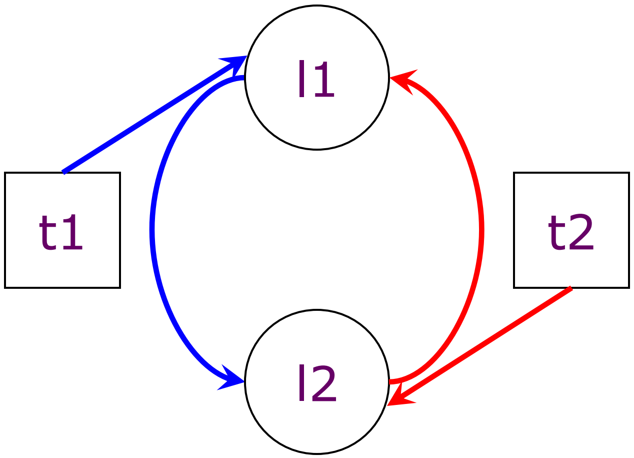
What is a Deadlock?

- An unintended condition in a shared-memory, multi-threaded program in which:
  - a set of threads blocks forever
  - because each thread in the set waits to acquire a lock being held by another thread in the set
    - This work: ignore other causes (e.g., wait/notify)

- Example

```
// thread t1
sync (l1) {
    sync (l2) {
        …
    }
}
```

```
// thread t2
sync (l2) {
    sync (l1) {
        …
    }
}
```

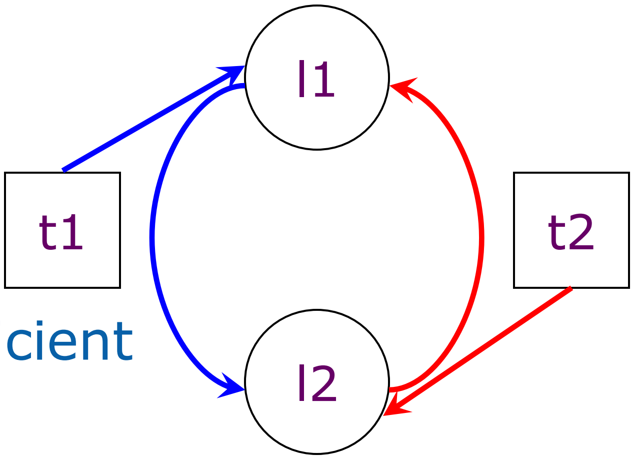


Motivation

- Today's concurrent programs are rife with deadlocks
  - 6,500/198,000 (~ 3%) of bug reports in Sun's bug database at http://bugs.sun.com are deadlocks
- Deadlocks are difficult to detect
  - Usually triggered non-deterministically, on specific thread schedules
  - Fail-stop behavior not guaranteed (some threads may be deadlocked while others continue to run)
- Fixing other concurrency bugs like races can introduce new deadlocks
  - Our past experience with reporting races: developers often ask for deadlock checker

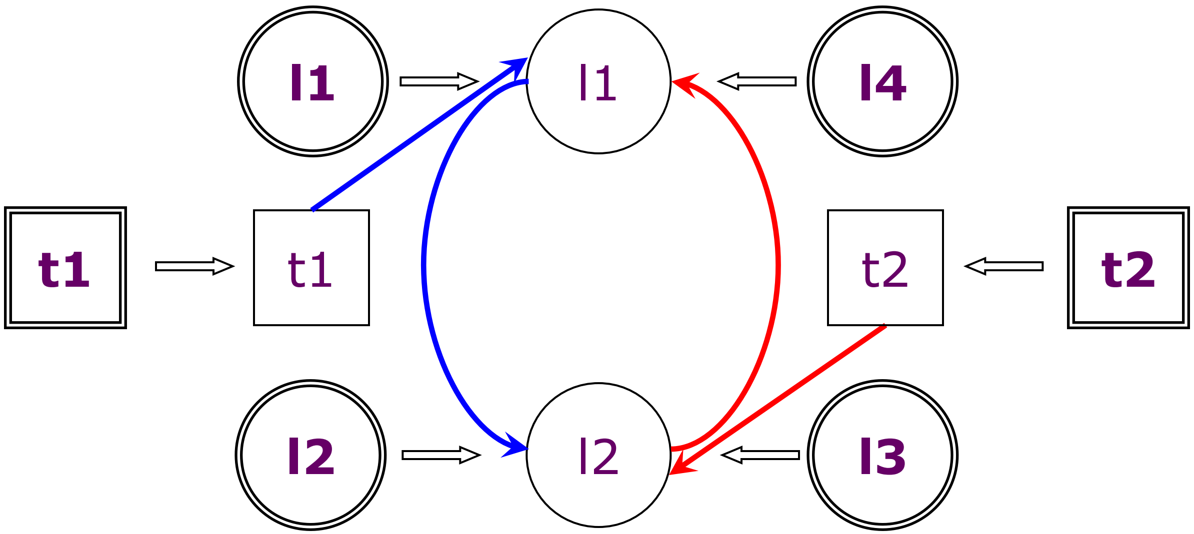
Previous Work

- Based on detecting cycles in program's dynamic or static lock order graph
- Dynamic approaches
  - Inherently unsound
  - Inapplicable to open programs
  - Can be ineffective without sufficient test input data

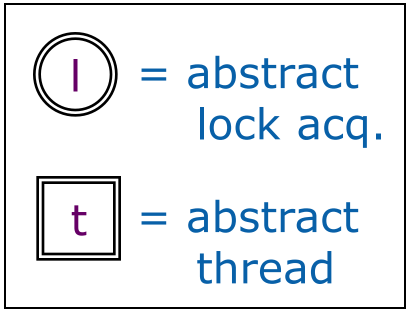


- Static approaches
  - Type systems (e.g., Boyapati-Lee-Rinard OOPSLA'02)
    - Annotation burden often significant
  - Model checking (e.g., SPIN)
    - Does not currently scale beyond few KLOC
  - Dataflow analysis (e.g., Engler & Ashcraft SOSP'03; Williams-Thies-Ernst ECOOP'05)
    - Scalable but highly imprecise

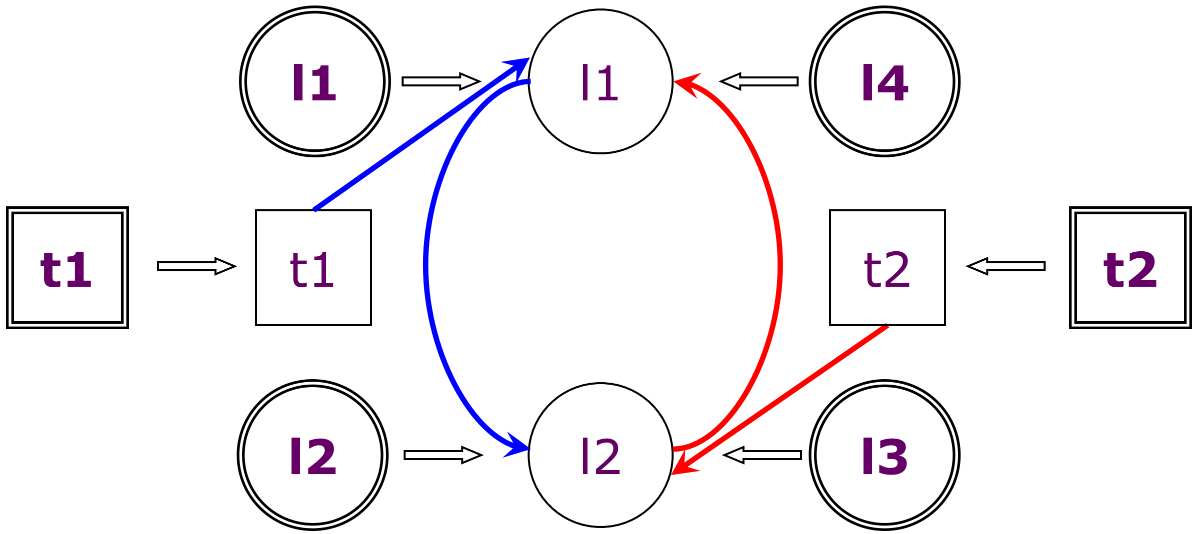
Challenges to Static Deadlock Detection



- Deadlock freedom is a complex property
  - can **t1**,**t2** denote different threads?
  - can **l1**,**l4** denote same lock?
  - can **t1** acquire locks **l1**->**l2**?
  - some more …



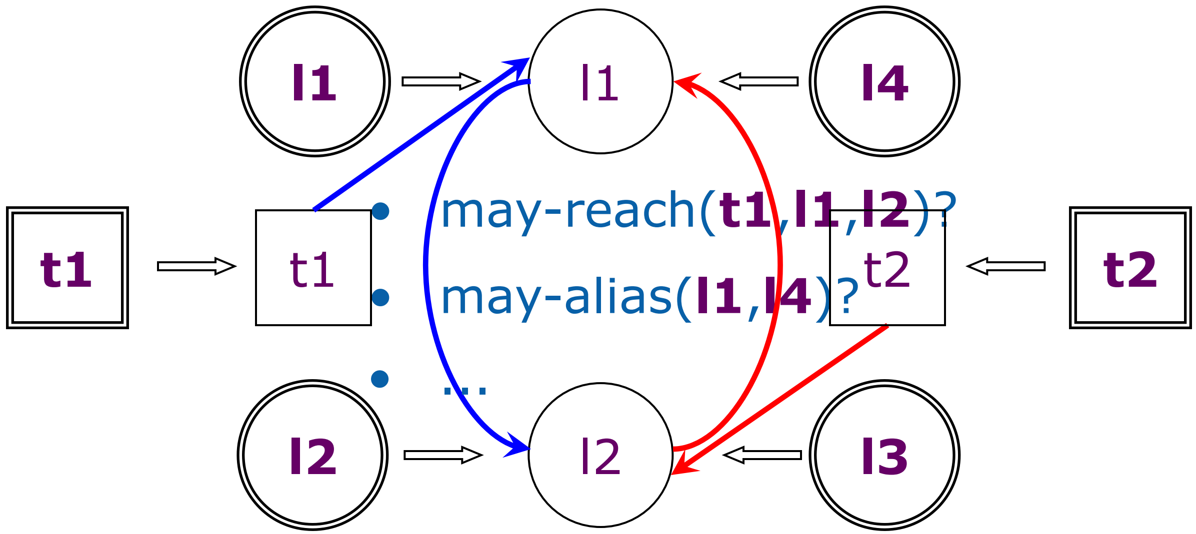
Our Rationale



- Deadlock freedom is a complex property
  - can **t1**,**t2** denote different threads?
  - can **l1**,**l4** denote same lock?
  - can **t1** acquire locks **l1**->**l2**?
  - some more …

- Existing static deadlock detectors cannot check all conditions *simultaneously* and *effectively*
- But each condition can be checked *separately* and *effectively* using existing static analyses

Our Approach



- Consider all candidate deadlocks in closed program
- Check each of six necessary conditions for each candidate to be a deadlock
- Report candidates that satisfy all six conditions
- Note: Finds only deadlocks involving 2 threads/locks
  - Deadlocks involving > 2 threads/locks rare in practice

Example: jdk1.4 java.util.logging

```
class LogManager {
        static LogManager manager =              l4   l1
            new LogManager();
l3
155: Hashtable loggers = new Hashtable();
280: sync boolean addLogger(Logger l) {
            String name = l.getName();
            if (!loggers.put(name, l))
                return false;
            // ensure l's parents are instantiated
            for (...) {
                String pname = ...;
314:            Logger.getLogger(pname);
            }
            return true;
        }
l2
420: sync Logger getLogger(String name) {
            return (Logger) loggers.get(name);
        }
}
```

```
class Logger {
226: static sync Logger getLogger(String name) {
            LogManager lm = LogManager.manager;
228:        Logger l = lm.getLogger(name);
            if (l == null) {
                l = new Logger(...);
231:            lm.addLogger(l);
            }
            return l;
        }
}
class Harness {
    static void main(String[] args) {
t1
11:     new Thread() { void run() {
13:             Logger.getLogger(...);
        }}.start();
t2
16:     new Thread() { void run() {
18:             LogManager.manager.addLogger(...);
        }}.start();
    }
}
```

Example Deadlock Report

```
*** Stack trace of thread <Harness.java:11>:
LogManager.addLogger (LogManager.java:280)
    - this allocated at <LogManager.java:155>
    - waiting to lock {<LogManager.java:155>}
Logger.getLogger (Logger.java:231)
    - holds lock {<Logger.java:0>}
Harness$1.run (Harness.java:13)

*** Stack trace of thread <Harness.java:16>:
Logger.getLogger (Logger.java:226)
    - waiting to lock {<Logger.java:0>}
LogManager.addLogger (LogManager.java:314)
    - this allocated at <LogManager.java:155>
    - holds lock {<LogManager.java:155>}
Harness$2.run (Harness.java:18)
```

Our Approach

- Six necessary conditions identified experimentally

1. Reachable
2. Aliasing
3. Escaping
4. Parallel
5. Non-reentrant
6. Non-guarded

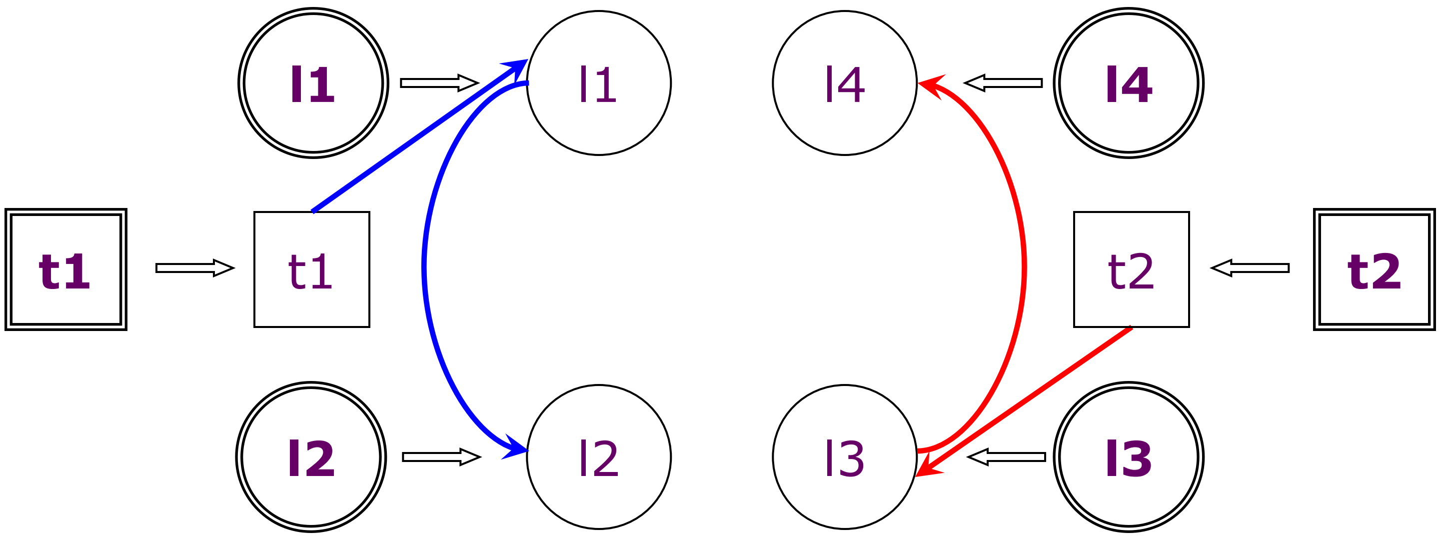
(Conditions 1–4:)
- Relatively language independent
- Incomplete but sound checks

(Conditions 5–6:)
- Widely-used Java locking idioms
- Incomplete and unsound checks
  - sound needs must-alias analysis

- Checked using four incomplete but sound whole-program static analyses

1. Call-graph analysis
2. May-alias analysis
3. Thread-escape analysis
4. May-happen-in-parallel analysis

Condition 1: Reachable



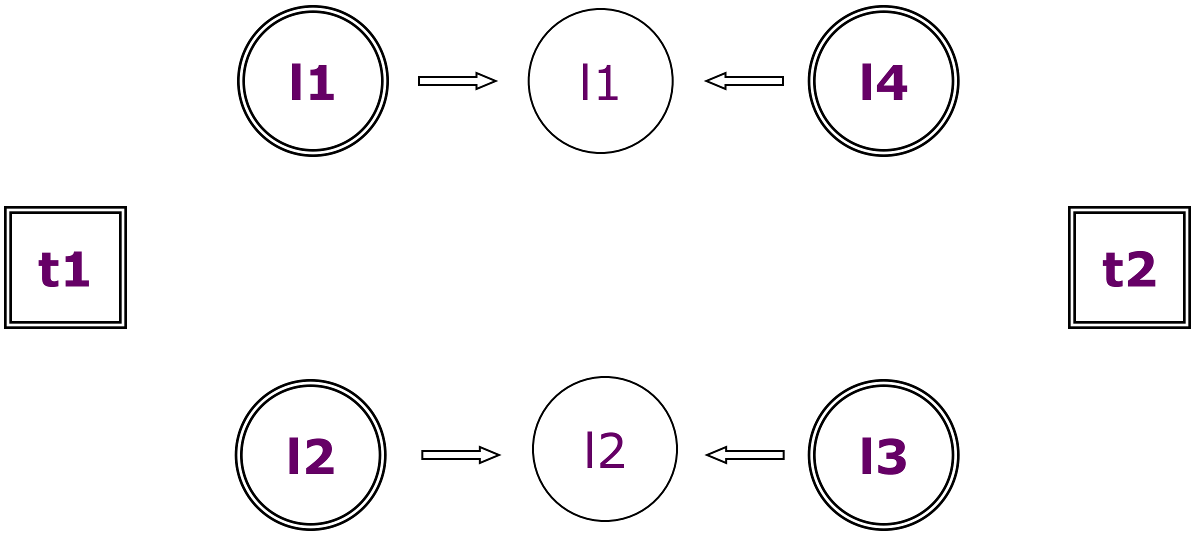
- Property: In some execution:
  - can a thread abstracted by **t1** reach **l1**
  - and after acquiring lock at **l1**, proceed to reach **l2** while holding that lock?
  - and similarly for **t2**, **l3**, **l4**
- Solution: Use call-graph analysis
  - k-object-sensitive [Milanova-Rountev-Ryder ISSTA'03]

Example: jdk1.4 java.util.logging

```
class LogManager {                                          l4  l1
    static LogManager manager =
        new LogManager();
l3
155: Hashtable loggers = new Hashtable();
280: sync boolean addLogger(Logger l) {
        String name = l.getName();
        if (!loggers.put(name, l))
            return false;
        // ensure l's parents are instantiated
        for (...) {
            String pname = ...;
314:        Logger.getLogger(pname);
        }
        return true;
    }
l2
420: sync Logger getLogger(String name) {
        return (Logger) loggers.get(name);
    }
}
```

```
class Logger {
226: static sync Logger getLogger(String name) {
        LogManager lm = LogManager.manager;
228:    Logger l = lm.getLogger(name);
        if (l == null) {
            l = new Logger(...);
231:        lm.addLogger(l);
        }
        return l;
    }
}
class Harness {
    static void main(String[] args) {
t1
11:     new Thread() { void run() {
13:         Logger.getLogger(...);
        }}.start();
t2
16:     new Thread() { void run() {
18:         LogManager.manager.addLogger(...);
        }}.start();
    }
}
```

Condition 2: Aliasing



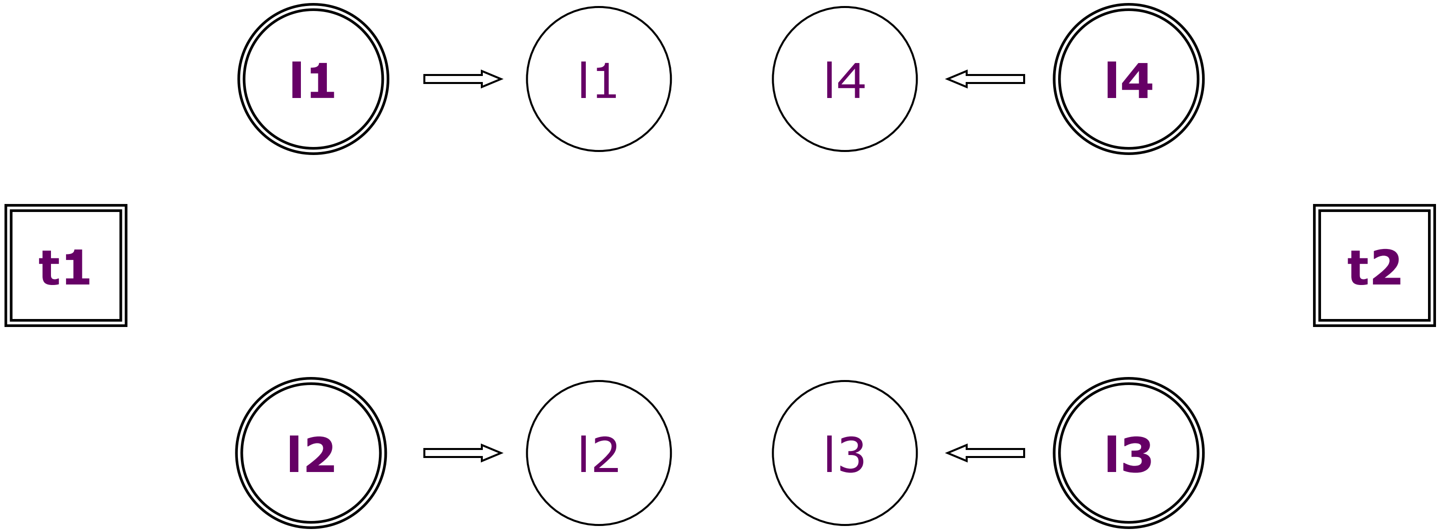
- Property: In some execution:
  - can a lock acquired at **l1** be the same as a lock acquired at **l4**?
  - and similarly for **l2**, **l3**
- Solution: Use may-alias analysis
  - k-object-sensitive [Milanova-Rountev-Ryder ISSTA’03]

Example: jdk1.4 java.util.logging

```
class LogManager {
        static LogManager manager =              l4
            new LogManager();
l3
155: Hashtable loggers = new Hashtable();
280: sync boolean addLogger(Logger l) {
            String name = l.getName();
            if (!loggers.put(name, l))
                return false;
            // ensure l's parents are instantiated
            for (...) {
                String pname = ...;
314:            Logger.getLogger(pname);
            }
            return true;
        }
l2
420: sync Logger getLogger(String name) {
            return (Logger) loggers.get(name);
        }
}
```

```
l1
class Logger {
226: static sync Logger getLogger(String name) {
            LogManager lm = LogManager.manager;
228:        Logger l = lm.getLogger(name);
            if (l == null) {
                l = new Logger(...);
231:            lm.addLogger(l);
            }
            return l;
        }
}
class Harness {
    static void main(String[] args) {
t1
11:     new Thread() { void run() {
13:         Logger.getLogger(...);
        }}.start();
t2
16:     new Thread() { void run() {
18:         LogManager.manager.addLogger(...);
        }}.start();
    }
}
```

Condition 3: Escaping



- Property: In some execution:
  - can a lock acquired at **l1** be thread-shared?
  - and similarly for each of **l2**, **l3**, **l4**
- Solution: Use thread-escape analysis

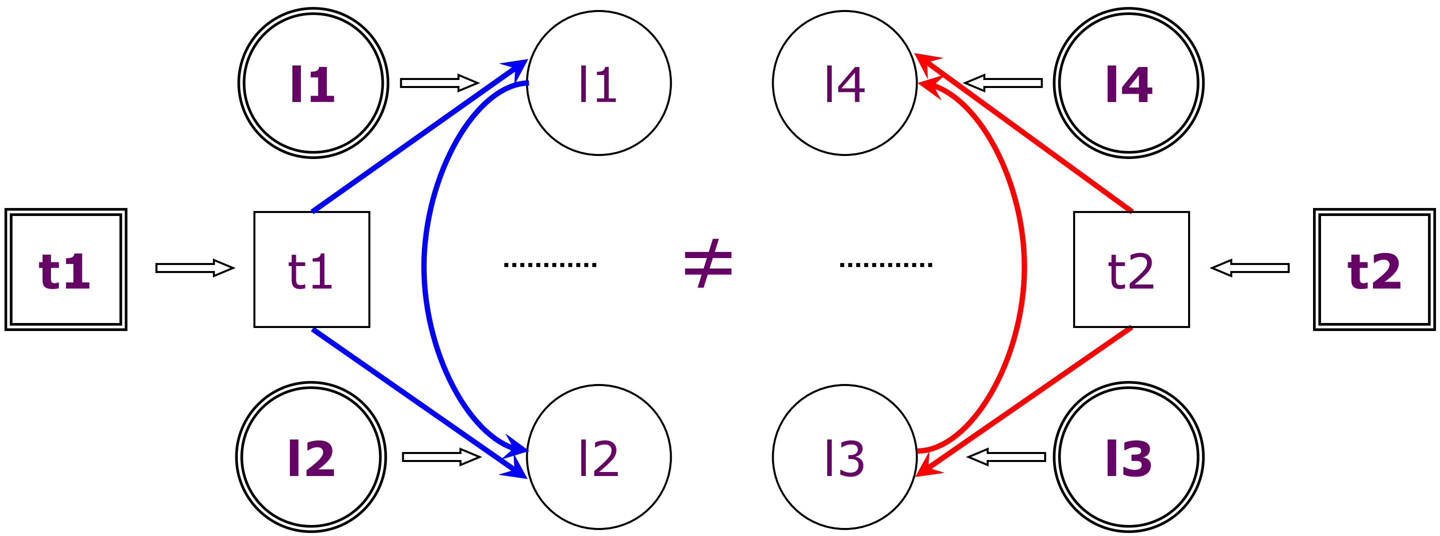
Example: jdk1.4 java.util.logging

```
class LogManager {
        static LogManager manager =
            new LogManager();
l3  155: Hashtable loggers = new Hashtable();
    280: sync boolean addLogger(Logger l) {
            String name = l.getName();
            if (!loggers.put(name, l))
                return false;
            // ensure l's parents are instantiated
            for (...) {
                String pname = ...;
    314:        Logger.getLogger(pname);
            }
            return true;
        }
l2  420: sync Logger getLogger(String name) {
            return (Logger) loggers.get(name);
        }
}
```

l4 l1

```
class Logger {
226: static sync Logger getLogger(String name) {
            LogManager lm = LogManager.manager;
228:        Logger l = lm.getLogger(name);
            if (l == null) {
                l = new Logger(...);
231:            lm.addLogger(l);
            }
            return l;
        }
}
class Harness {
    static void main(String[] args) {
t1  11:     new Thread() { void run() {
    13:         Logger.getLogger(...);
            }}.start();
t2  16:     new Thread() { void run() {
    18:         LogManager.manager.addLogger(...);
            }}.start();
    }
}
```

Condition 4: Parallel



- Property: In some execution:
  - can different threads abstracted by **t1** and **t2**
  - simultaneously reach **l2** and **l4**?
- Solution: Use may-happen-in-parallel analysis
  - Does not model full happens-before relation
  - Models only thread fork construct
  - Other conditions model other constructs

Example: jdk1.4 java.util.logging

l4 **l1**

l3

l2

```
class LogManager {
    static LogManager manager =
        new LogManager();
155: Hashtable loggers = new Hashtable();
280: sync boolean addLogger(Logger l) {            // l3
        String name = l.getName();
        if (!loggers.put(name, l))
            return false;
        // ensure l's parents are instantiated
        for (...) {
            String pname = ...;
314:        Logger.getLogger(pname);
        }
        return true;
    }
420: sync Logger getLogger(String name) {          // l2
        return (Logger) loggers.get(name);
    }
}
```

```
class Logger {
226: static sync Logger getLogger(String name) {
        LogManager lm = LogManager.manager;
228:    Logger l = lm.getLogger(name);
        if (l == null) {
            l = new Logger(...);
231:        lm.addLogger(l);
        }
        return l;
    }
}
```

t1 **t2**

```
class Harness {
    static void main(String[] args) {
11:     new Thread() { void run() {              // t1
13:         Logger.getLogger(...);
        }}.start();
16:     new Thread() { void run() {              // t2
18:         LogManager.manager.addLogger(...);
        }}.start();
    }
}
```

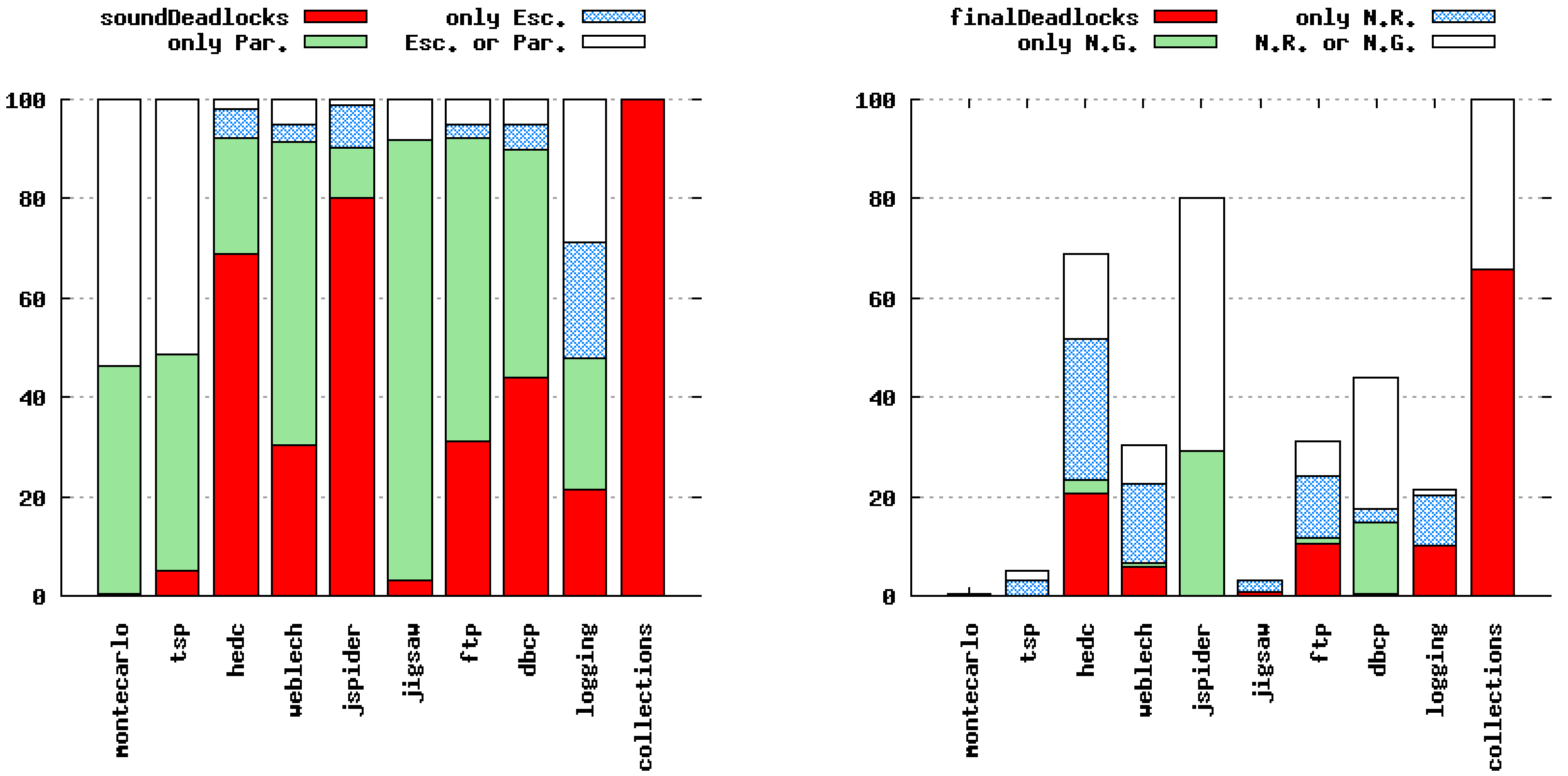
Benchmarks

Benchmark	LOC	Classes	Methods	Syncs	Time
moldyn	31,917	63	238	12	4m48s
montecarlo	157,098	509	3447	190	7m53s
raytracer	32,576	73	287	16	4m51s
tsp	154,288	495	3335	189	7m48s
sor	32,247	57	208	5	4m48s
hedc	160,071	530	3552	204	21m15s
weblech	184,098	656	4620	238	32m02s
jspider	159,494	557	3595	205	15m34s
jigsaw	154,584	497	3346	184	15m23s
ftp	180,904	642	4383	252	35m55s
dbcp	168,018	536	3602	227	16m04s
cache4j	34,603	72	218	7	4m43s
logging	167,923	563	3852	258	9m01s
collections	38,961	124	712	55	5m42s

Experimental Results

Benchmark	Deadlocks (0-cfa)	Deadlocks (k-obj.)	Lock type pairs (total)	Lock type pairs (real)
moldyn	0	0	0	0
montecarlo	0	0	0	0
raytracer	0	0	0	0
tsp	0	0	0	0
sor	0	0	0	0
hedc	7,552	2,358	22	19
weblech	4,969	794	22	19
jspider	725	4	1	0
jigsaw	23	18	3	3
ftp	16,259	3,020	33	24
dbcp	320	16	4	3
cache4j	0	0	0	0
logging	4,134	4,134	98	94
collections	598	598	16	16

Individual Analysis Contributions



Conclusion

- Novel approach to static deadlock detection for Java
  - Checks six necessary conditions for a deadlock
  - Uses four off-the-shelf static analyses

- Neither sound nor complete, but effective in practice
  - Applied to suite of 14 multi-threaded Java programs comprising over 1.5 MLOC
  - Found all known deadlocks as well as previously unknown ones, with few false alarms