Effective Static Deadlock Detection

Mayur Naik, Chang-Seo Park, Koushik Sen and David Gay
ICSE'09

Presented by Alex Kogan
27.04.2014

Outline

- Introduction
- Suggested Solution
- Evaluation
- Related work
- Conclusions
- Discussion

Introduction

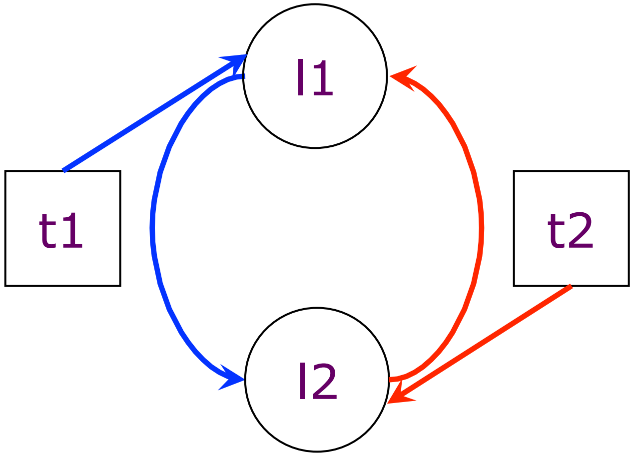
What is a Deadlock?

- An unintended condition in a shared-memory, multi-threaded program in which:
  - a set of threads block forever
  - because each thread in the set waits to acquire a lock being held by another thread in the set

Example

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


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



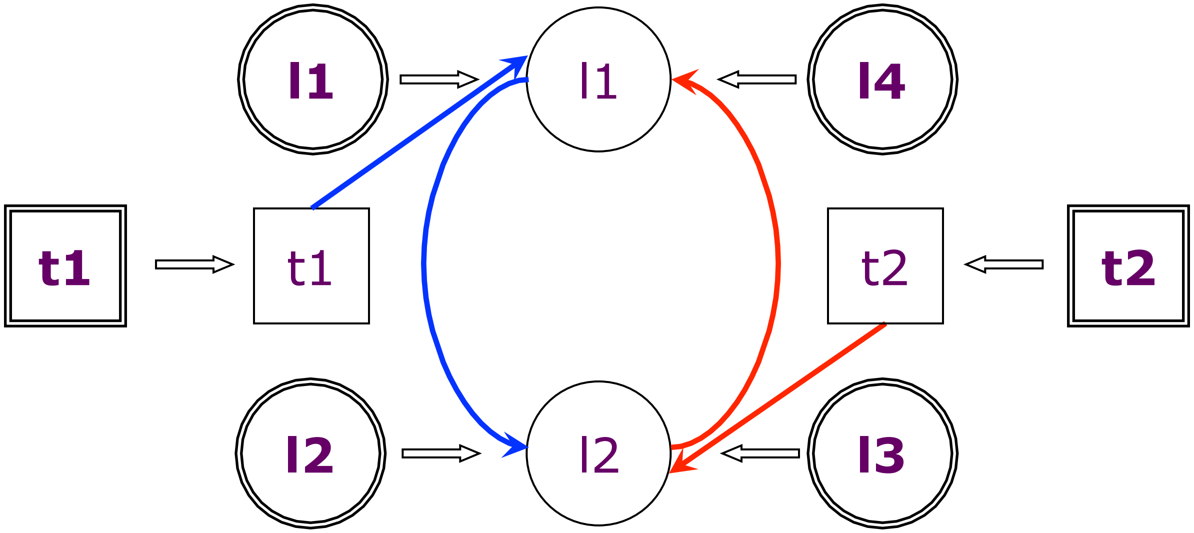
Motivation

- Today's concurrent programs are rife with deadlocks
  - 6,500 (~3%) of bugs reported in Sun's bug database are deadlocks
- Difficult to detect
  - Triggered non deterministically on specific thread schedules
  - Fail-stop behavior not guaranteed
- Fixing other concurrency bugs like races can introduce new deadlocks

Previous Work

- Dynamic approaches
  - Unsound
  - Inapplicable to open programs
  - Rely on input data
- Static approaches(Soundness)
  - Type systems
  - Model checking
  - Dataflow analysis

Challenges



- 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 …

Example: jdk1.4 java.util.logging

l4 l1

l2

l3

```
class LogManager {
    static LogManager manager =
        new↑h↓3 LogManager();
155: Hashtable loggers = new Hashtable();
280: sync↑m↓2 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;
    }
420: sync↑m↓3 Logger getLogger(...) {
        return (Logger) loggers.get(name);
    }
}
```

```
class Logger {
226: static sync↑m↓1 Logger getLogger(...) {
        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↑h↓1 Thread() { void run↑m↓4 () {
13:         Logger.getLogger(...);
        }}.start();
16:     new↑h↓2 Thread() { void run↑m↓5 () {
18:         LogManager.manager.addLogger(...);
        }}.start();
    }
```

Solution Outline

- List set of conditions for a deadlock
- Check each condition separately
- Utilize existing static analyses for each condition
- Report candidates that satisfy all conditions
- Only find deadlocks involving 2 locks(threads)

Solution

Java deadlock detection algorithm

Take II:
Effective Static Deadlock Detection

Mayur Naik, Chang-Seo Park, Koushik Sen and David Gay
ICSE'09

Presented by Alex Kogan
11.05.2014

Context

- Context in the seminar
  - Concurrency present challenges
  - We have seen mostly C analyses
    - OO?

Object-Sensitivity

- Invented by A. Milanova et al. [ISSTA 2002]
- Produces points-to and call graph

Related Work

- Anderson's Algorithm
  - Flow & context insensitive
  - Base for speed & accuracy comparison
- Flow Sensitivity
- (k) CFA – Control Flow Analysis
  - Context sensitive

Idea

- OO paradigm presents special needs
  - Objects play a major role
  - Call graph defined by inheritance
- Fast (practical)
- Adjustable

Object Sensitivity

- Comparable to CFA
  - Uses the receiver object instead of the caller
- Saves up to k receiving (this) objects
- Uses them to calculate the context of each pointer

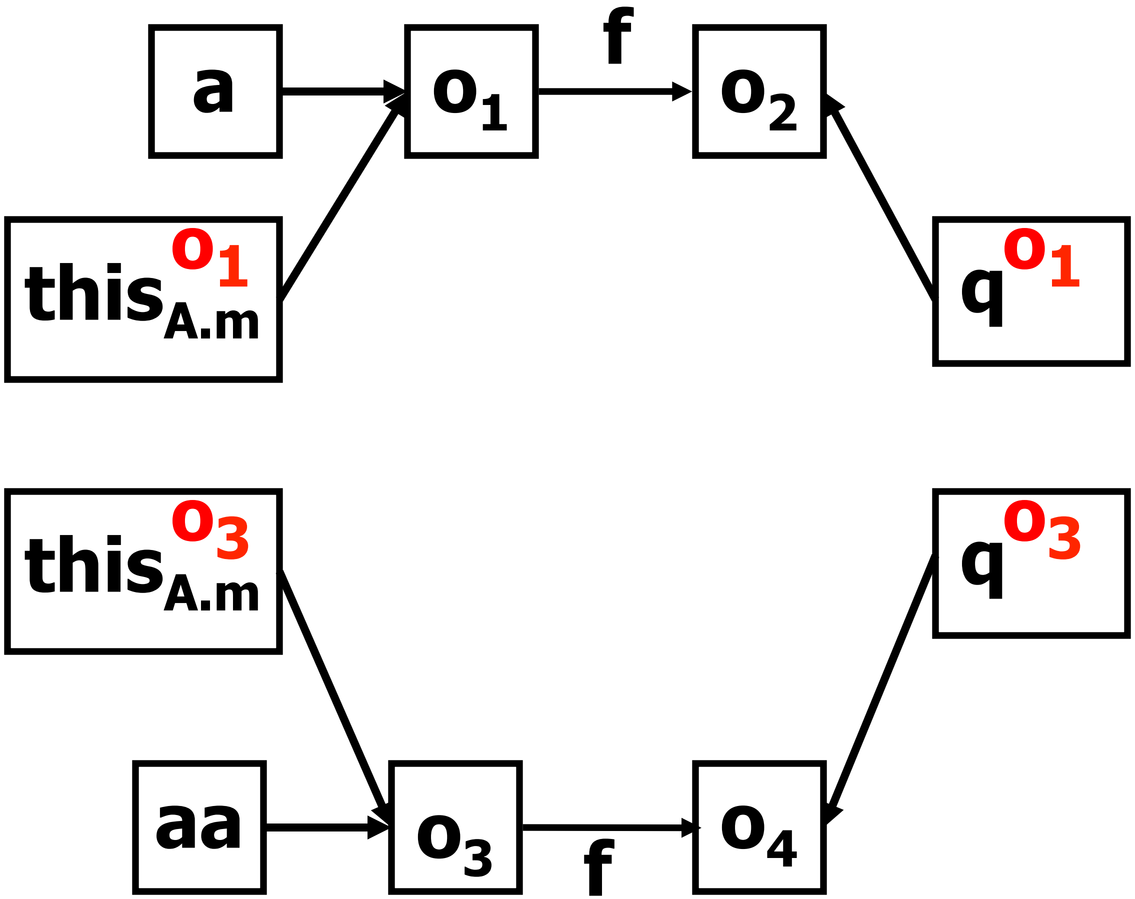
$$\text{this.f=q} \xrightarrow{o_1} \boxed{\text{this}^{o_1}_{A.m}\text{.f=q}^{o_1}}$$

Example: Object-Sensitive Analysis

```
class Y extends X {}

class A {
  X f;
  void m(X q) {
this_A.m^o3.f=q^o3 ;
  }}

A a = new A() ;
a.m(new X()) ;
A aa = new A() ;
aa.m(new Y()) ;
```

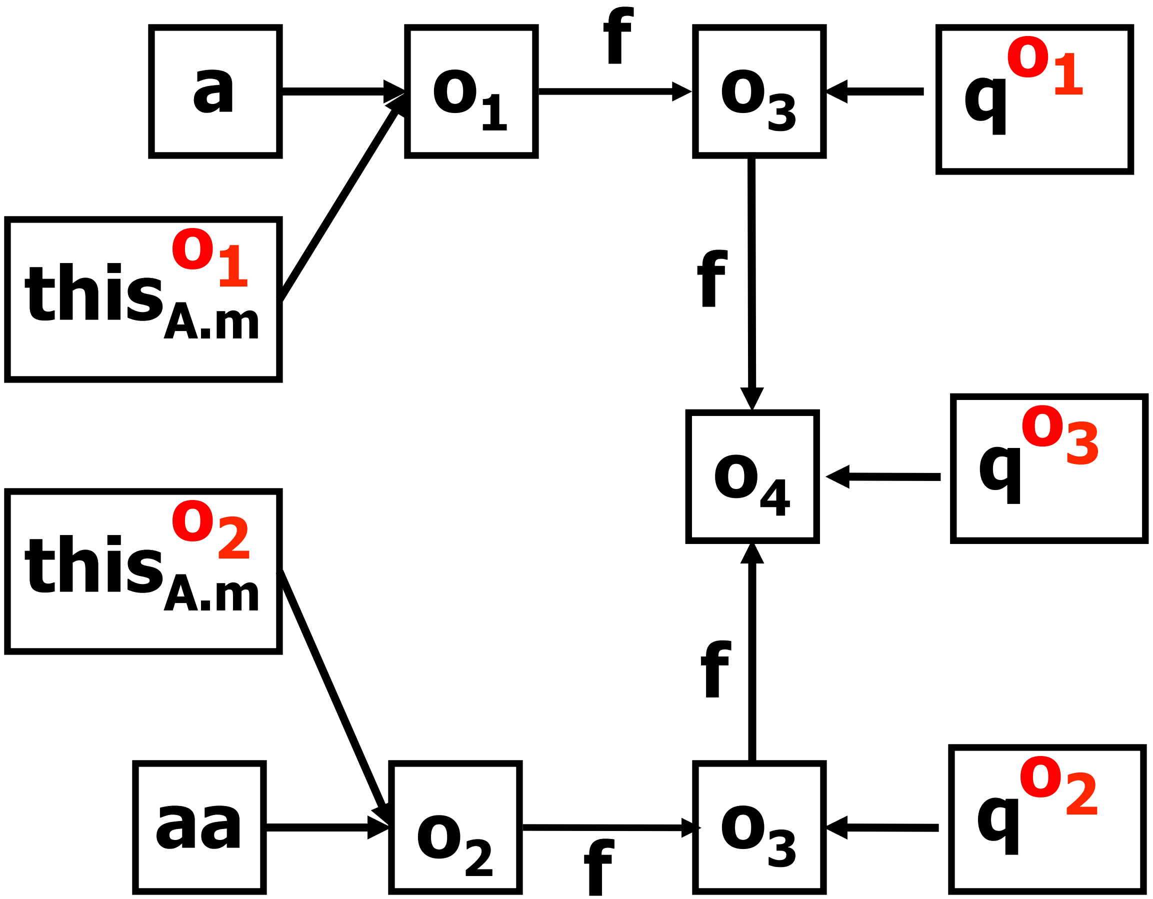


Example: Object-Sensitive Analysis Loss Of Information for k=1

```
class A {
  X f;
  void m(A q) {
       this.f=q ;
   }}
A[] a= new A[2];
a[0] = new A() ;
a[1] = new A() ;

for(int i=0; i<2; i++)
    a[i].m(new A()) ;


 a[0].f.m(new A()) ;
```

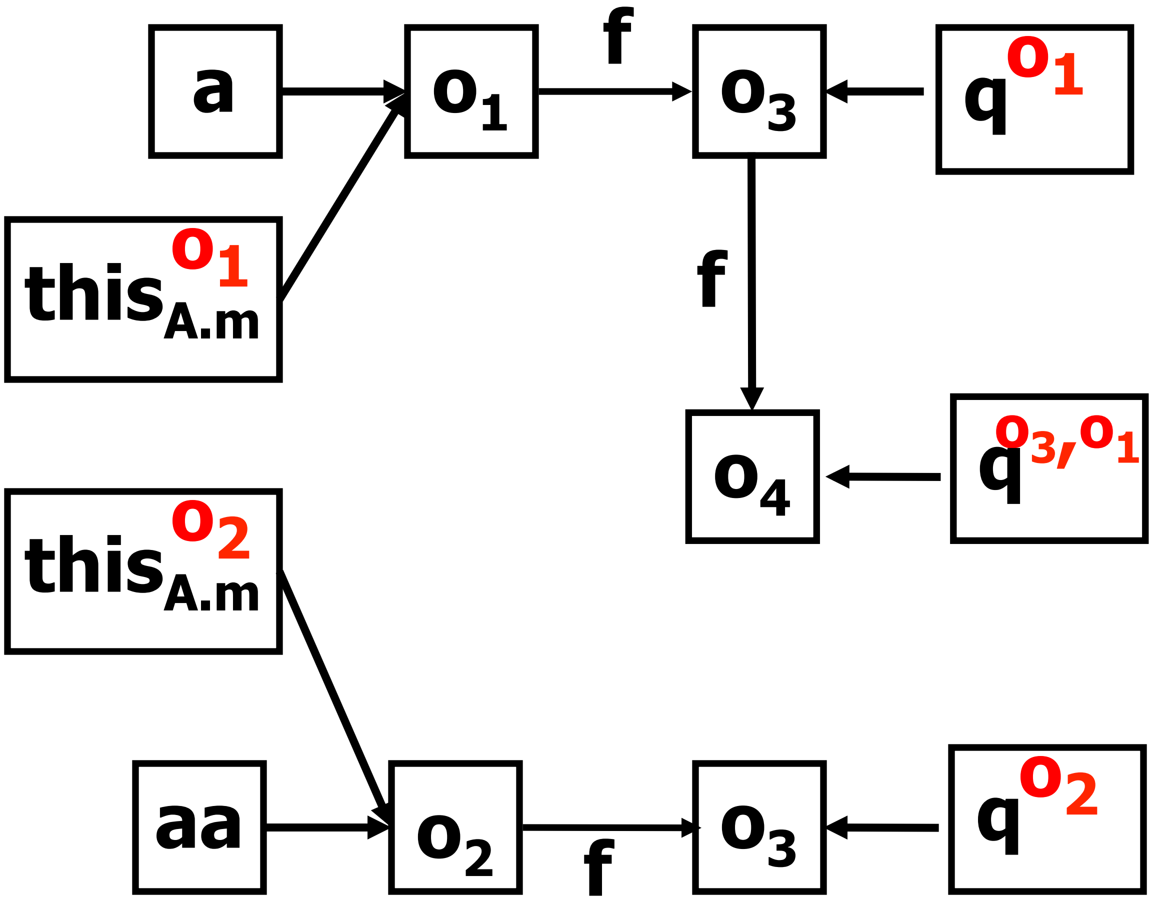


Example: Object-Sensitive Analysis Success For k=2

```
class A {
  X f;
  void m(A q) {
       this.f=q ;
   }}
A[] a= new A[2];
a[0] = new A() ;
a[1] = new A() ;

for(int i=0; i<2; i++)
    a[i].m(new A()) ;


a[0].f.m(new A()) ;
```



k-Object-Sensitivity Computation

- Starting point
  - consider main method and class initializers reachable.
- Add points-to for every new object (at local var v)
  - objects created in context c = (o = [$h_{1,\ldots,}h_k$], m)
    - add ((o,m), v, [this, h_1, …, h_{k-1}])
- Add to call graph for every method call
  - for a instance method call v.n(...) add ([$h_{1,\ldots,}h_k$], n’)
    - for context in which v is reachable
    - if n’= m_{start} then also deem reachable n’’ = m_{run}
    - for a static method call n(..) add ([], n)

k-Object-Sensitivity Cont.

- Produce the following relations
  - cg $\subseteq$(C x C)
    - pairs of (o_1 , m_1) x (o_2 , m_2)
      - method m_2 (in context o_2) is reachable from m_1 (with context o_1)
  - pt $\subseteq$ (C x V x O)
    - C – context of the call to create
    - V – pointer to objects(local variable)
    - O – abstract object
      - finite sequence of object allocation sites h_1 …h_k
      - h_1 - object allocation site
      - The rest are "this" objects of methods where o was allocated

Building Call Graph & Points To

```
class Y extends X {}

class A {
   X f;
   void m(X q) {
this^{o3}_{A.m}.f=q^{o3} ;
   }

A a = new A() ;
a.m(new X()) ;
A aa = new A() ;
aa.m(new Y()) ;
```

Points-To	Call Graph
$(([], m_{main}), v_1, [h_1])$	$(([], m_{main}), ([h_1], m))$
$(([h_1], m), v_2, [h_2])$	$(([], m_{main}), ([h_3], m))$
$(([], m_{main}), v_3, [h_3])$	
$(([h_3], m), v_2, [h_4])$	

Example: jdk1.4 java.util.logging

l4 l1

```
class LogManager {
    static LogManager manager =
        new↑h↓3 LogManager();
155: Hashtable loggers = new Hashtable();
280: sync↑m↓2 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;
    }
420: sync↑m↓3 Logger getLogger(...) {
        return (Logger) loggers.get(name);
    }
}
```

l2

l3

t1

t2

```
class Logger {
226: static sync↑m↓1 Logger getLogger(...) {
        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) {
11:     new↑h↓1 Thread() { void run↑m↓4 () {
13:         Logger.getLogger(...);
        }}.start();
16:     new↑h↓2 Thread() { void run↑m↓5 () {
18:         LogManager.manager.addLogger(...);
        }}.start();
    }
```

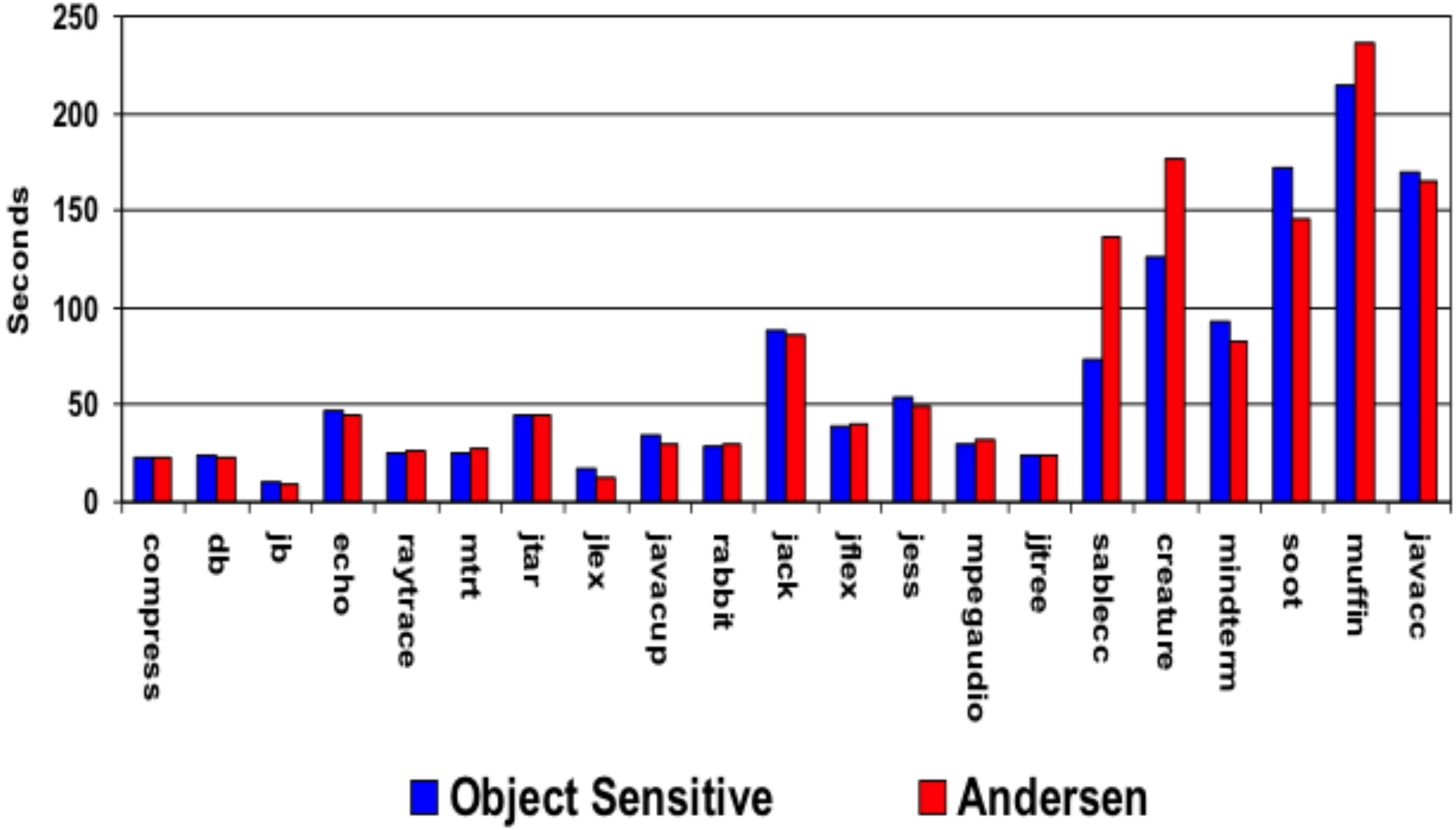
Example: Step 1

- pt = $\{((([], m_{main}), v_1, [h_1]),$
  $(([], m_{main}), v_2, [h_2])\}$
- cg = $\{((([], m_{main}), ([h_1], m_{start})),$
  $(([], m_{main}), ([h_2], m_{start}))\}$
- Deem reachable
  - $([h_1], m_4)$
  - $([h_2], m_5)$
  - m_4 , m_5 are the run methods of the threads

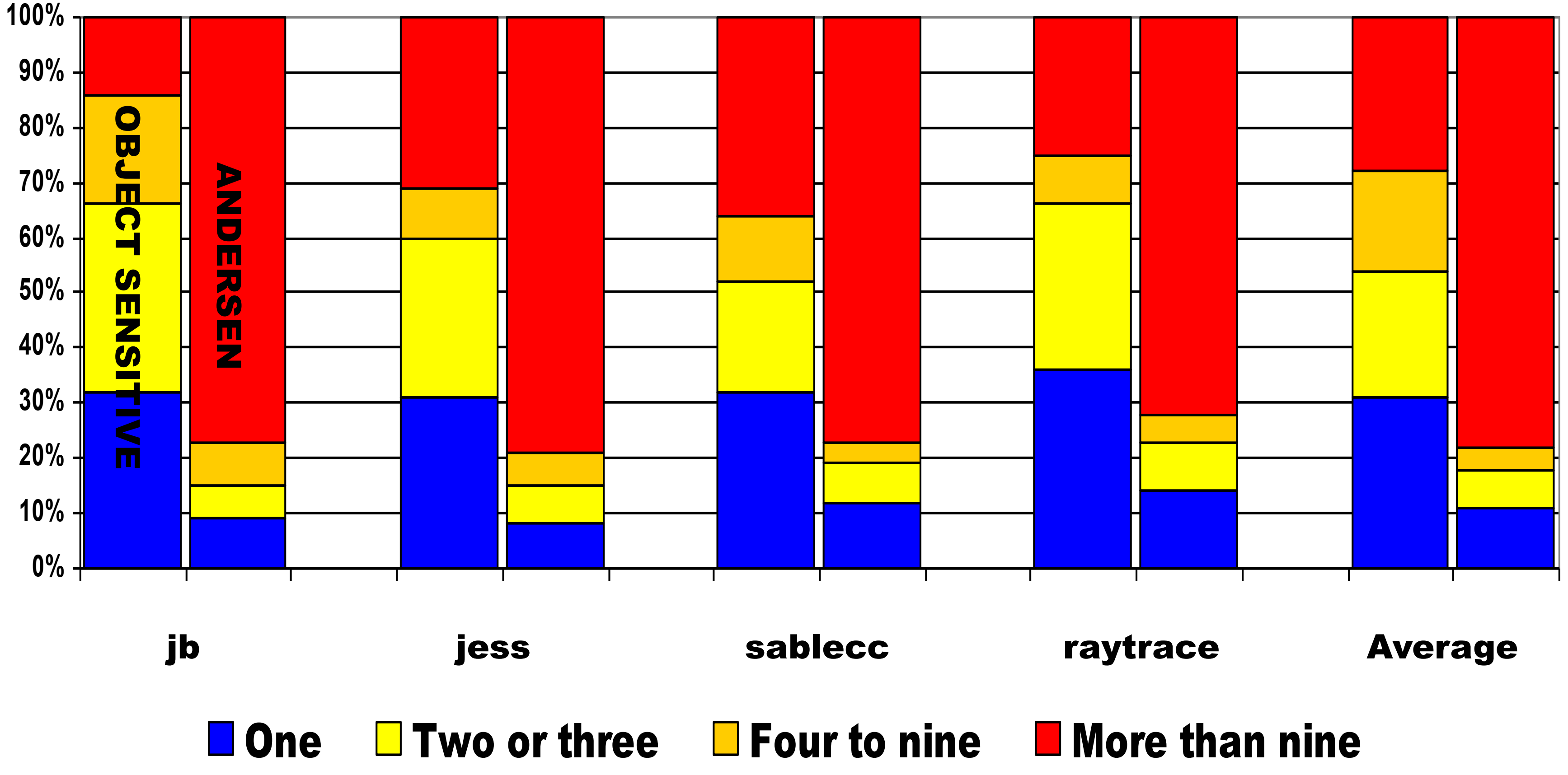
Example: Step 2*

- What is added to the call graph?
  - $(([h_1], m_4) \times ([], m_1))$ – Logger.getLogger()
  - $(([h_2], m_5) \times ([h_3], m_2))$ – LogManager.lm.addLogger()
- How about point-to?
  - Nothing new to be added in this step
  - Next step will add all local variables of functions getLogger() and addLogger()

Object Sensitivity: Analysis Time (k=2)



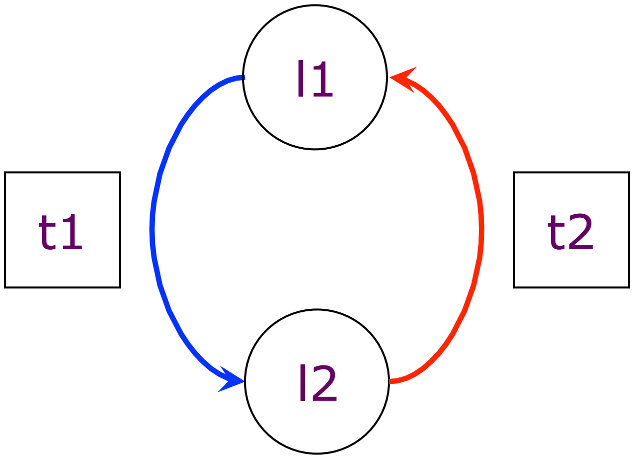
A Word About Precision: Modified Objects Per Statement



For The Main Event

Deadlock Detection

A Deadlock



Deadlock Computation (Java)

- Lock aliasing:
  - mayAlias(l_1,l_2) $\Longleftrightarrow$
    $\exists o : (l_1, sync(l_1), o) \in pt \land (l_2, sync(l_2), o) \in pt$
- Lock-set aliasing
  - mayAlias(L_1,L_2) $\Longleftrightarrow$
    $\exists l_1 \in L_1, l_2 \in L_2 :$ mayAlias(l_1,l_2)
- Reachability: $c_1 \rightarrow c_2 \triangleright L \;\Leftrightarrow\; \exists n :\; c_1 \rightarrow^n c_2 \triangleright L$ where:

$$(1)\; c \rightarrow^0 c \triangleright \emptyset$$

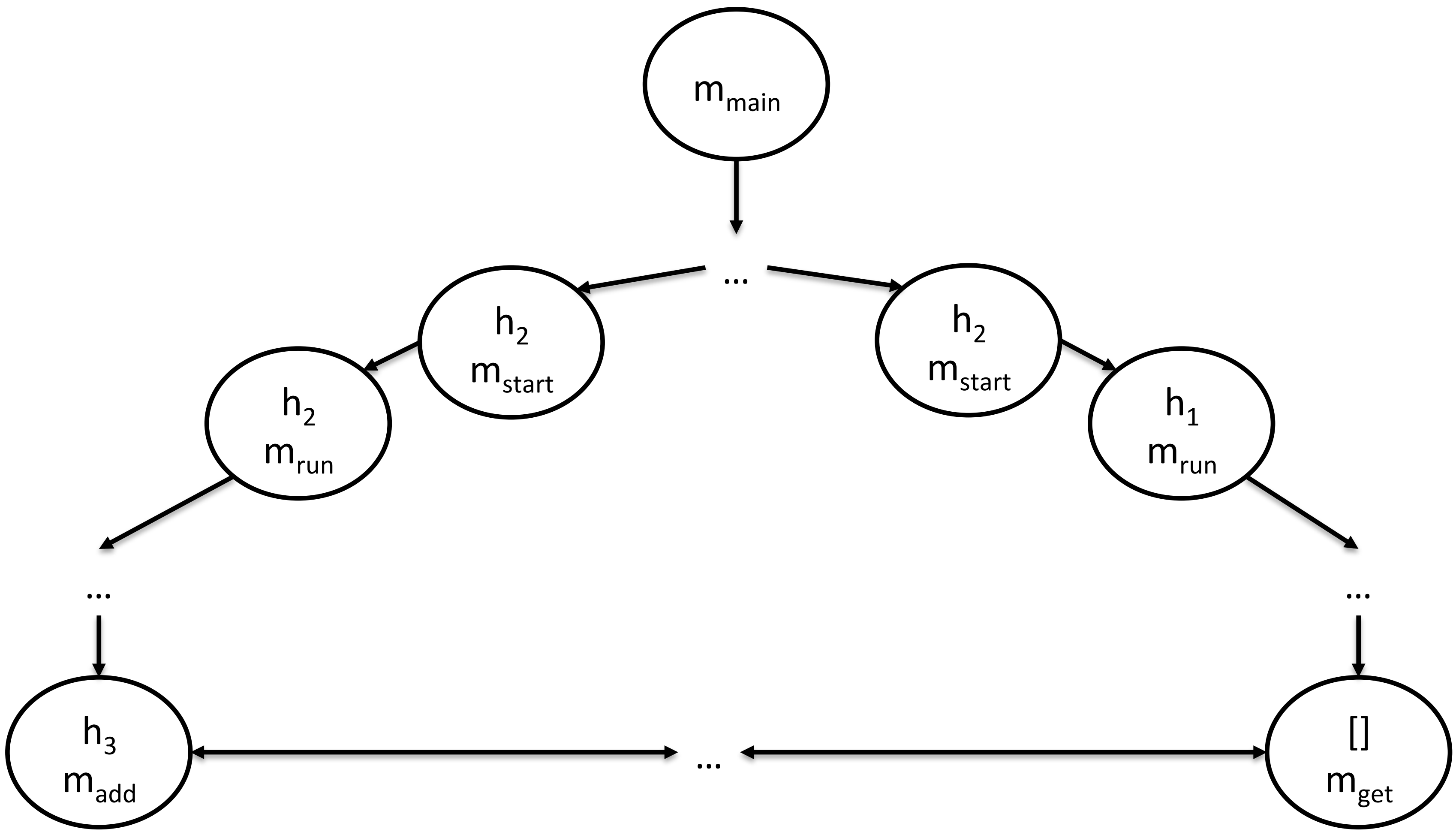
$$(2)\; c_1 \rightarrow^{n+1} c_2 \triangleright L'$$

$$\exists c, L :\; c_1 \rightarrow^n c \triangleright L \;\land\; (c, c_2) \in \mathsf{cg} \;\land\; L' = \begin{cases} L \cup \{c\} & \text{if } \mathsf{sync}(c) \text{ defined} \\ L & \text{otherwise} \end{cases}$$

Deadlock Computation (Java)

- Threads = $\{x \mid \exists n : x \in threads_n\}$
  - $threads_0 = \{([], m_{main})\}$
  - $threads_{n+1} = threads_n \cup \{(o, run)\}$
    - $c \in threads_n$
    - $c \rightarrow (o, m_{start})$
- Locks =
  - $\{c \mid c' \in threads \wedge c' \rightarrow c \wedge sync(c)\}$
- FinalDeadlocks = $\{ d \mid d = (t^a, l_1^a, l_2^a, t^b, l_1^b, l_2^b) \}$
  - $t^a, t^b \in threads$
  - $l_1^a, l_2^a, l_1^b, l_2^b \in locks$
  - *all six conditions pass for d*

Call Graph vs Threads & Locks



Example: jdk1.4 java.util.logging

l4 l1 l2 l3 t1 t2

```
class LogManager {
    static LogManager manager =
        new↑h↓3  LogManager();
155: Hashtable loggers = new Hashtable();
280: sync↑m↓2  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;
    }
420: sync↑m↓3  Logger getLogger(...) {
        return (Logger) loggers.get(name);
    }
}
```

```
class Logger {
226: static sync↑m↓1  Logger getLogger(...) {
        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) {
11:     new↑h↓1  Thread() { void run↑m↓4  () {
13:         Logger.getLogger(...);
        }}.start();
16:     new↑h↓2  Thread() { void run↑m↓5  () {
18:         LogManager.manager.addLogger(...);
        }}.start();
    }
```

Deadlock computation example

- Threads = $\{ ([], mmain), t_1, t_2 \}$
  - $t_1 = ([h_1], m_4)$
  - $t_2 = ([h_2], m_5)$
- Locks = $\{ l1, l2, l3 \}$
  - $l1 = ([], m_1)$
  - $l2 = ([h_3], m_2)$
  - $l3 = ([h_3], m_3)$

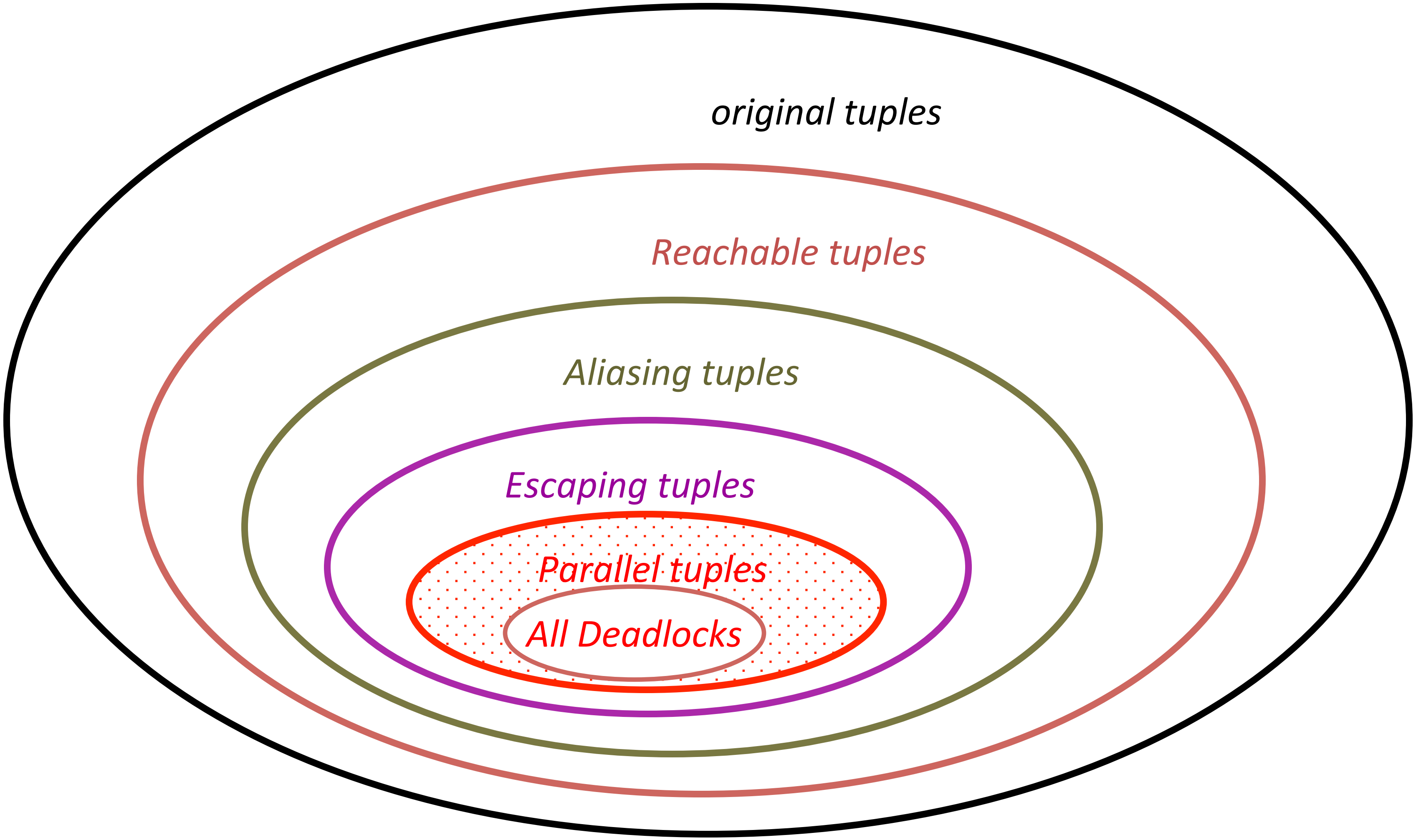
Deadlock Computation - Tuples

$(t_1, l_2, l_1, t_2, l_2, l_1)$ $(t_1, l_1, l_1, t_2, l_1, l_1)$ $(t_1, l_1, l_2, t_1, l_2, l_1)$

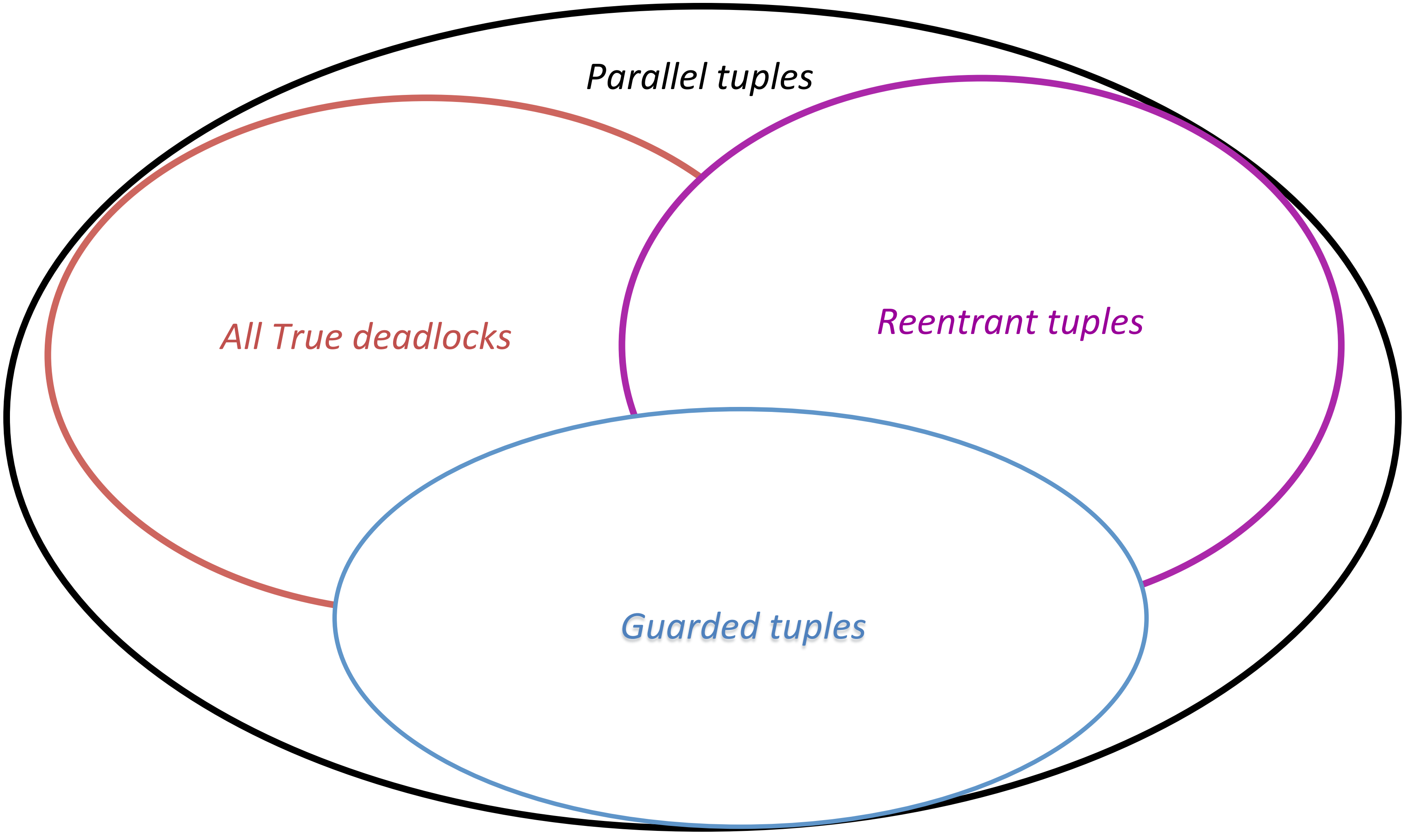
$(t_1, l_1, l_2, t_2, l_2, l_1)$

$(t_1, l_1, l_3, t_2, l_2, l_1)$

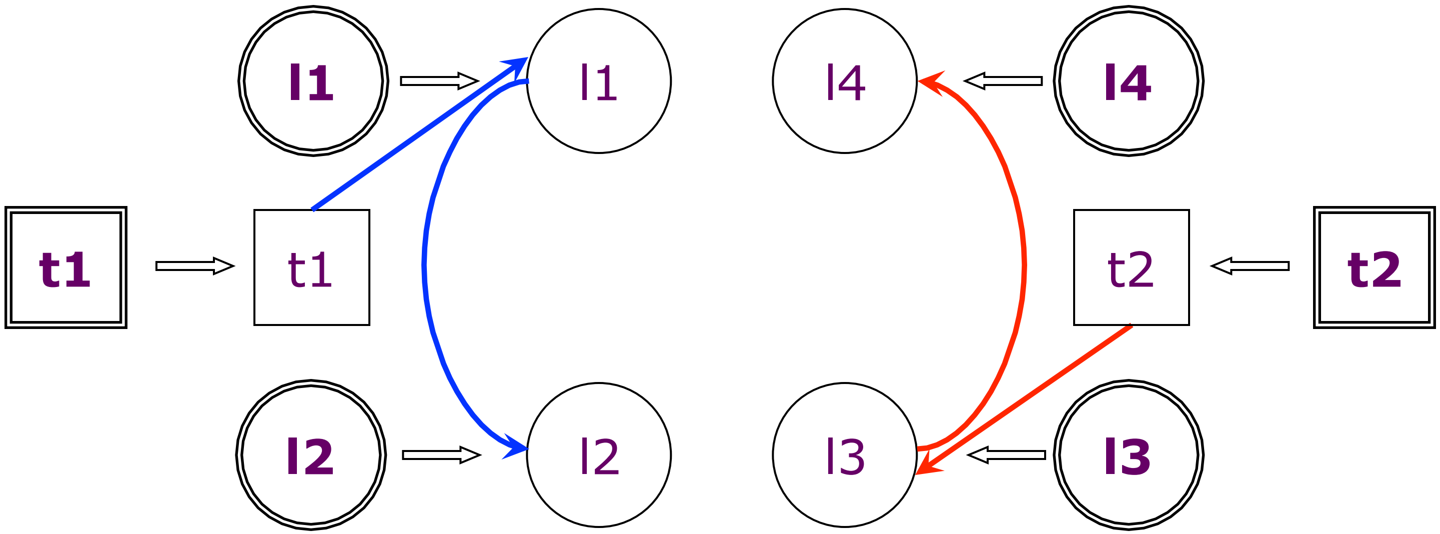
Sound conditions for deadlock



Unsound conditions for deadlock



Condition 1: Reachable



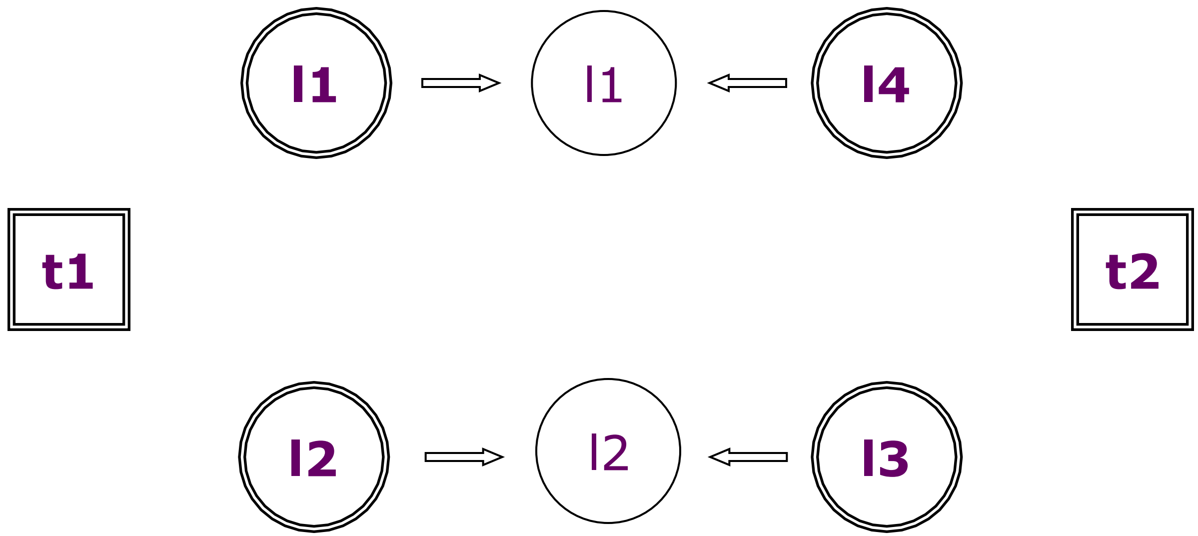
- In some execution:
  - can a thread abstracted by **t1** reach **l1**
  - and after acquiring lock at **l1**, proceed to reach **l2** while holding that lock?
  - and similarly for **t2**, **l3**, **l4**
- Solution: Use call-graph analysis

Reachable Cont.

- $reachableDeadlock(t{\upharpoonleft}a, l{\downarrow}1{\upharpoonleft}a, l{\downarrow}2{\upharpoonleft}a, t{\upharpoonleft}b, l{\downarrow}1{\upharpoonleft}b, l{\downarrow}2{\upharpoonleft}b)$
  - $t{\upharpoonleft}a \rightarrow l{\downarrow}1{\upharpoonleft}a \wedge l{\downarrow}1{\upharpoonleft}a \rightarrow l{\downarrow}2{\upharpoonleft}a \wedge t{\upharpoonleft}b \rightarrow l{\downarrow}1{\upharpoonleft}b \wedge l{\downarrow}1{\upharpoonleft}b \rightarrow l{\downarrow}2{\upharpoonleft}b$

- Running example:
  - easy to see both pass
  - $t{\downarrow}1 \rightarrow l{\downarrow}1 \rightarrow l{\downarrow}2$
    - $t{\downarrow}1 \rightarrow l{\downarrow}1 \rightarrow l{\downarrow}3$
  - $t{\downarrow}2 \rightarrow l{\downarrow}2 \rightarrow l{\downarrow}1$

Condition 2: Aliasing



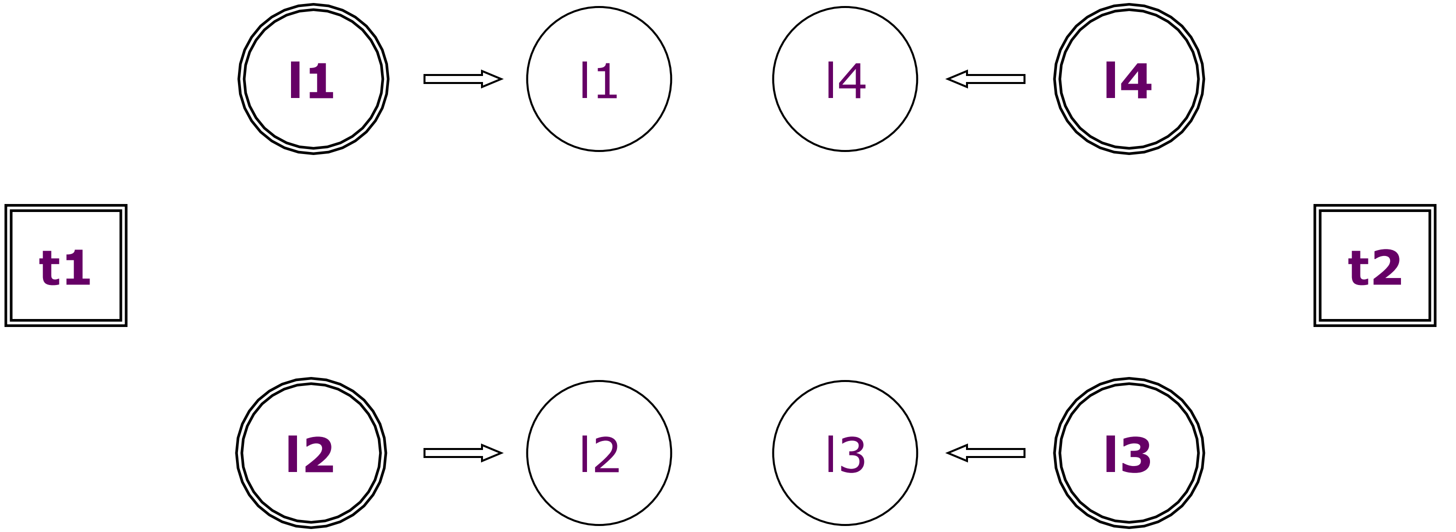
- In some execution:
  - can a lock acquired at **l1** be the same as a lock acquired at **l4**?
  - and similarly for **l2**, **l3**
- Solution: Use may-alias analysis

Aliasing Cont.

- $aliasingDeadlock(t^a, l_1^a, l_2^a, t^b, l_1^b, l_2^b)$
  - $mayAlias(l_1^a, l_2^b) \wedge mayAlias(l_2^a, l_1^b)$

- Running example:
  - $mayAlias(l_1, l_1)$ *and* $mayAlias(l_2, l_2)$
  - $mayAlias(l_2, l_3)$? – $(t_1, l_1, l_3, t_2, l_2, l_1)$
    - Pass because $[h_3]$ is the context of both, which satisfies the may-alias condition

Condition 3: Escaping



- 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

Escaping Analysis Gist

- A method for determining the scope of pointers
- Thread-escape
  - has a pointer from a "thread spawning" site.
  - may be referenced by another thread-shared object
  - static objects
- Produce relation *esc*
  - **(c, v)**
  - argument **v** of object **c** can be access by more than 1 thread

❖ their thread escape is fairly imprecise

Escaping Cont.

- $escapingDeadlock(t^{a}, l_{1}^{a}, l_{2}^{a}, t^{b}, l_{1}^{b}, l_{2}^{b})$
  - $(l_{1}^{a}, sync(l_{1}^{a})) \in esc \wedge (l_{2}^{a}, sync(l_{2}^{a})) \in esc \wedge$
  - $(l_{1}^{b}, sync(l_{1}^{b})) \in esc \wedge (l_{2}^{b}, sync(l_{2}^{b})) \in esc$

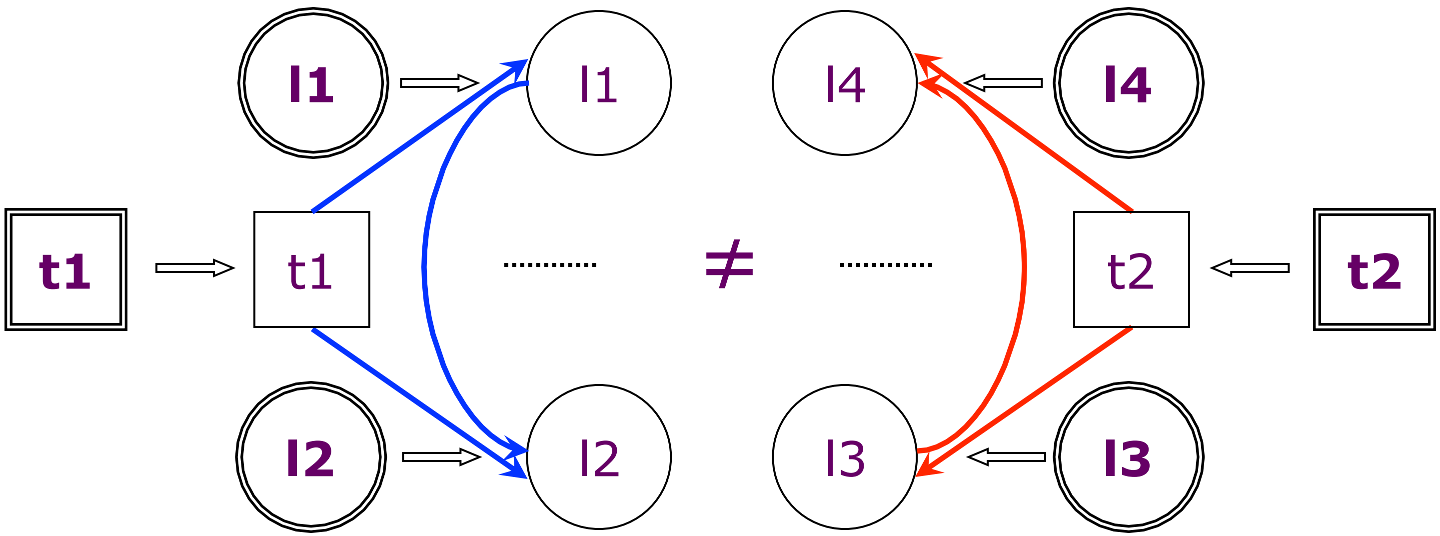
- Running Example:
  - LogManager.manager & Logger.class are static
    - So they escape everywhere and both tuples pass

Example: jdk1.4 java.util.logging

```
class LogManager {
    static LogManager manager =
        new↑h↓3 LogManager();                                   l4  l1
l2
155: Hashtable loggers = new Hashtable();
280: sync↑m↓2 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;                                             t1
    }
l3
420: sync↑m↓3 Logger getLogger(...) {                            t2
        return (Logger) loggers.get(name);
    }
}

class Logger {
226: static sync↑m↓1 Logger getLogger(...) {
        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) {
11:     new↑h↓1 Thread() { void run↑m↓4 () {
13:         Logger.getLogger(...);
        }}.start();
16:     new↑h↓2 Thread() { void run↑m↓5 () {
18:         LogManager.manager.addLogger(...);
        }}.start();
    }
```

Condition 4: Parallel



- In some execution:
  - can different threads abstracted by **t1** and **t2**
  - simultaneously reach **l2** and **l4**?
- Solution: Use may-happen-in-parallel analysis

May-happen-in-parallel Gist

- Filters:
  - Exact same threads
    - may-alias threads
  - Threads in child/parent relationship
    - by annotations

- Produces relation mhp
  - $(t_1\ (o, m), t_2\ )$
  - $\boldsymbol{t_2}$ may be running in parallel when $\boldsymbol{t_1}$ reaches method $\boldsymbol{m}$ in context $\boldsymbol{o}$

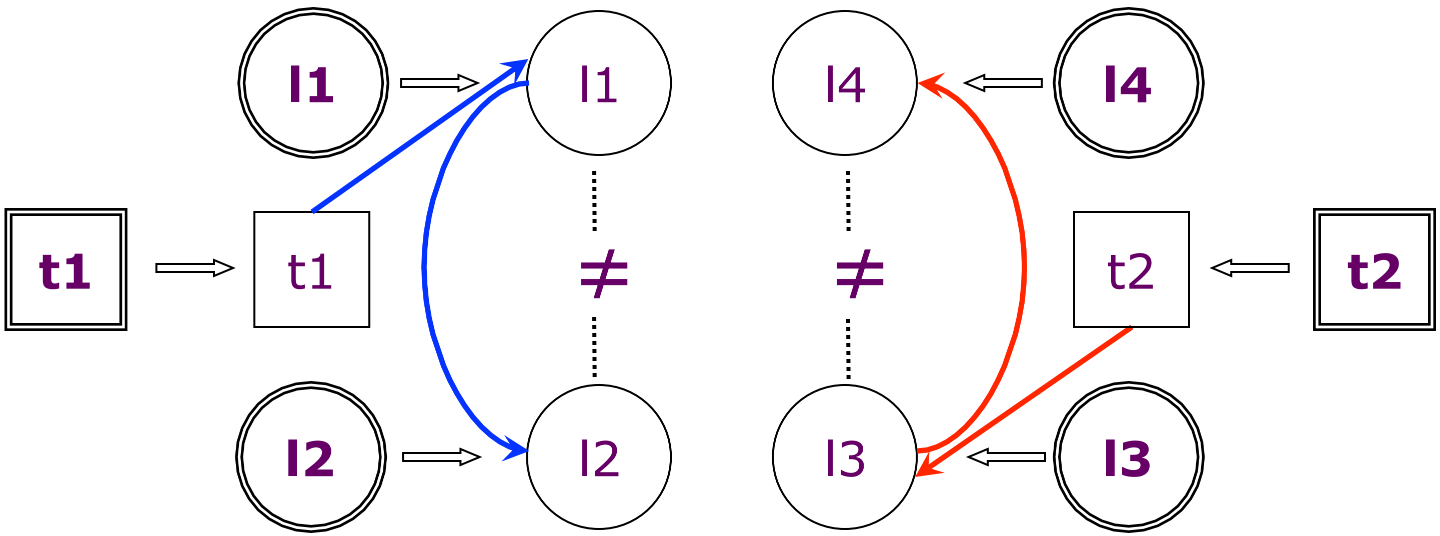
May-happen-in-parallel Cont.

- $parallelDeadlock(t_a, l_{1a}, l_{2a}, t_b, l_{1b}, l_{2b})$
  - $(t_a, l_{2a}, t_b) \in mhp \wedge (t_b, l_{2b}, t_a) \in mhp$

- Running example:
  - Nothing prevents t_1 and t_2 from running in parallel
    - both tuples pass

- What will it remove?
  - $i \in \{1,2\}$: $(t_i, *, *, t_i, *, *)$

Condition 5: Non-reentrant



- Property: In some execution:
  - can a thread abstracted by **t1** acquire a non-reentrant lock at **l1**
  - and, while holding that lock, proceed to acquire a non-reentrant lock at **l2**?
  - and similarly for **t2**, **l3**, **l4**
- Solution: Use call-graph + may-alias analysis
  - Unsound: Negation of over-approximation

Non-reentrant Cont.

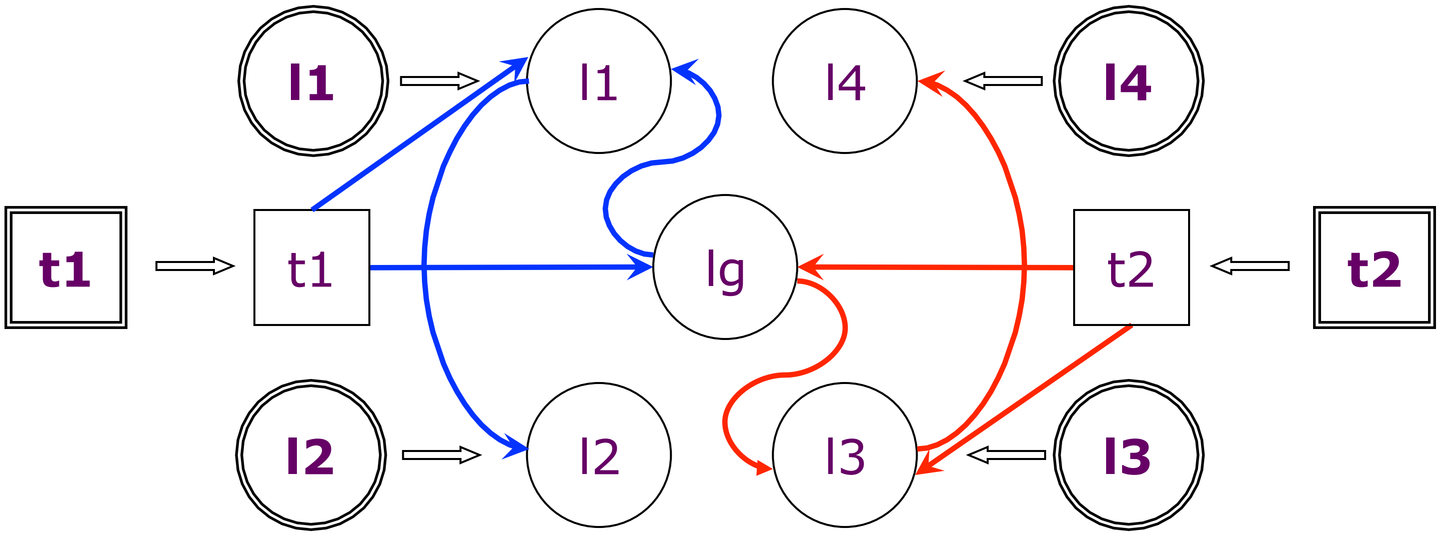
- $reentrant(t_1, l_1, l_2)$
  - $l_1 = l_2$
  - $\forall L_1 : (t_1 \rightarrow l_1 \triangleright L_1 \Rightarrow mayAlias(\{l_1, l_2\}, L_1))$
  - $\forall L_2 : (l_1 \rightarrow l_2 \triangleright L_2 \Rightarrow mayAlias(\{l_2\}, L_2))$

- $nonReentDeadlock(t^a, l_1^a, l_2^a, t^b, l_1^b, l_2^b)$
  - $\neg reentrant(t^a, l_1^a, l_2^a) \wedge \neg reentrant(t^b, l_1^b, l_2^b)$

Reentrant examples

- Exact same lock
  - $(t_1, l_1, l_1, t_2, l_1, l_1)$
- Already got that lock before
  - $(t_1, l_2, l_1, t_2, l_1, l_2)$
- Running Example:
  - Locks don’t alias and no lock acquired between them

Condition 6: Non-guarded



- In some execution:
  - can different threads abstracted by **t1** and **t2** reach **l1** and **l3**, respectively, without holding a common lock?
- Solution: Use call-graph + may-alias analysis
  - Unsound: Same as for condition 5.

Non-guarded Cont.

- $guarded(t_1, l_1, t_2, l_2)$
  - $\forall L_1 \, \forall L_2 : (t_1 \to l_1 \rhd L_1 \wedge t_2 \to l_2 \rhd L_2) \Rightarrow mayAlias(L_1, L_2)$

- $nonGuardedDeadlock(t^a, l_1^a, l_2^a, t^b, l_1^b, l_2^b)$
  - $\neg guarded(t^a, l_1^a, t^b, l_1^b)$

- Running Example:
  - No locks are acquired before l_1

Example: jdk1.4 java.util.logging

l4 l1 l2 l3 t1 t2

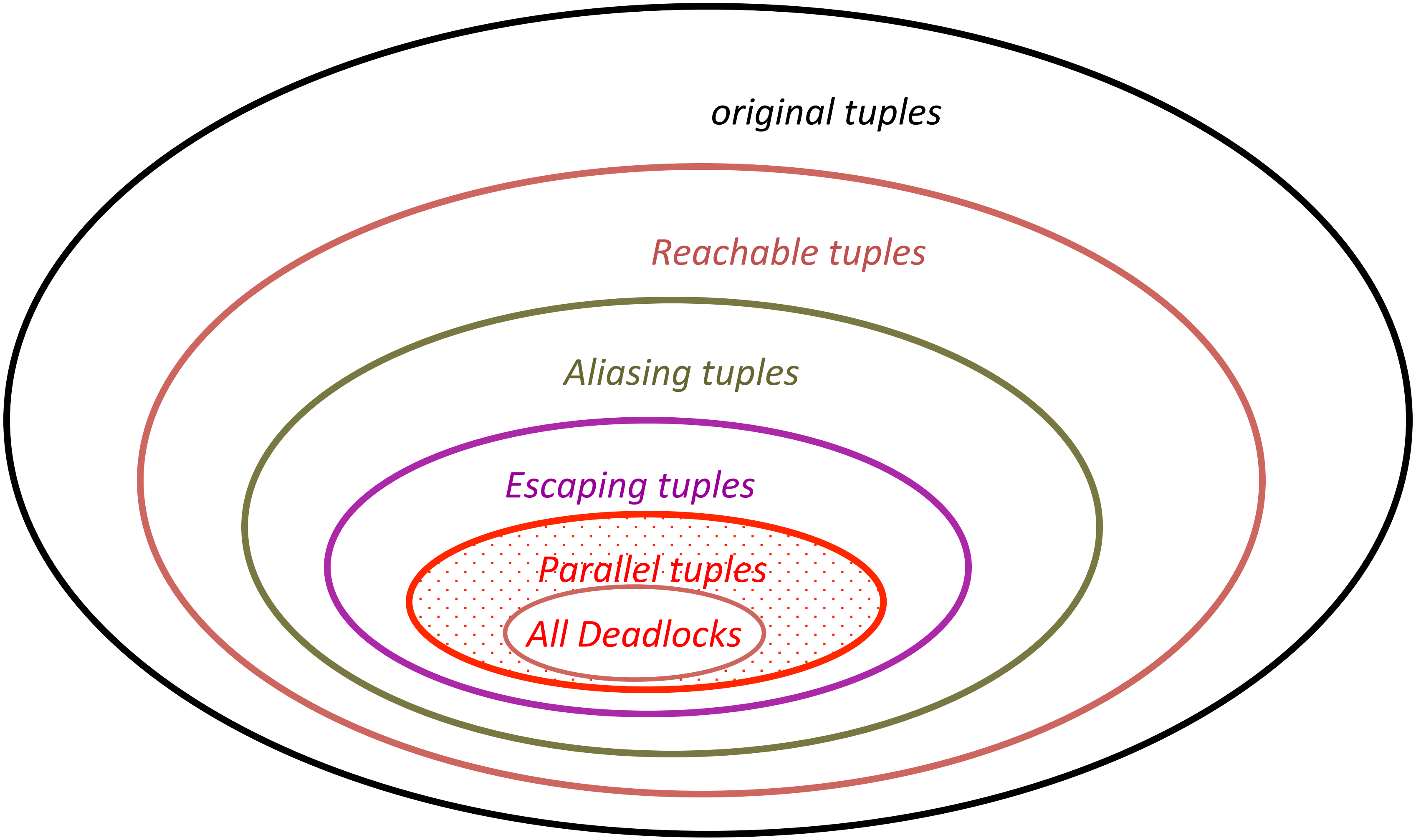
```
class LogManager {
    static LogManager manager =
        new↑h↓3 LogManager();
155: Hashtable loggers = new Hashtable();
280: sync↑m↓2 boolean addLogger(Logger l) {
       String name = l.getName();
       if (!loggers.put(name, l))
          return false;
       // ensure l's parents are instantiated
       for (...) {
          String pname = ...;
314:      Logger.getLogger(pname);
       }
       return true;
    }
420: sync↑m↓3 Logger getLogger(String name) {
       return (Logger) loggers.get(name);
    }
}
```

```
class Logger {
226: static sync↑m↓1 Logger getLogger(String na
       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) {
11:     new↑h↓1 Thread() { void run↑m↓4 () {
13:        Logger.getLogger(...);
        }}.start();
16:     new↑h↓2 Thread() { void run↑m↓5 () {
18:        LogManager.manager.addLogger(...);
        }}.start();
    }
}
```

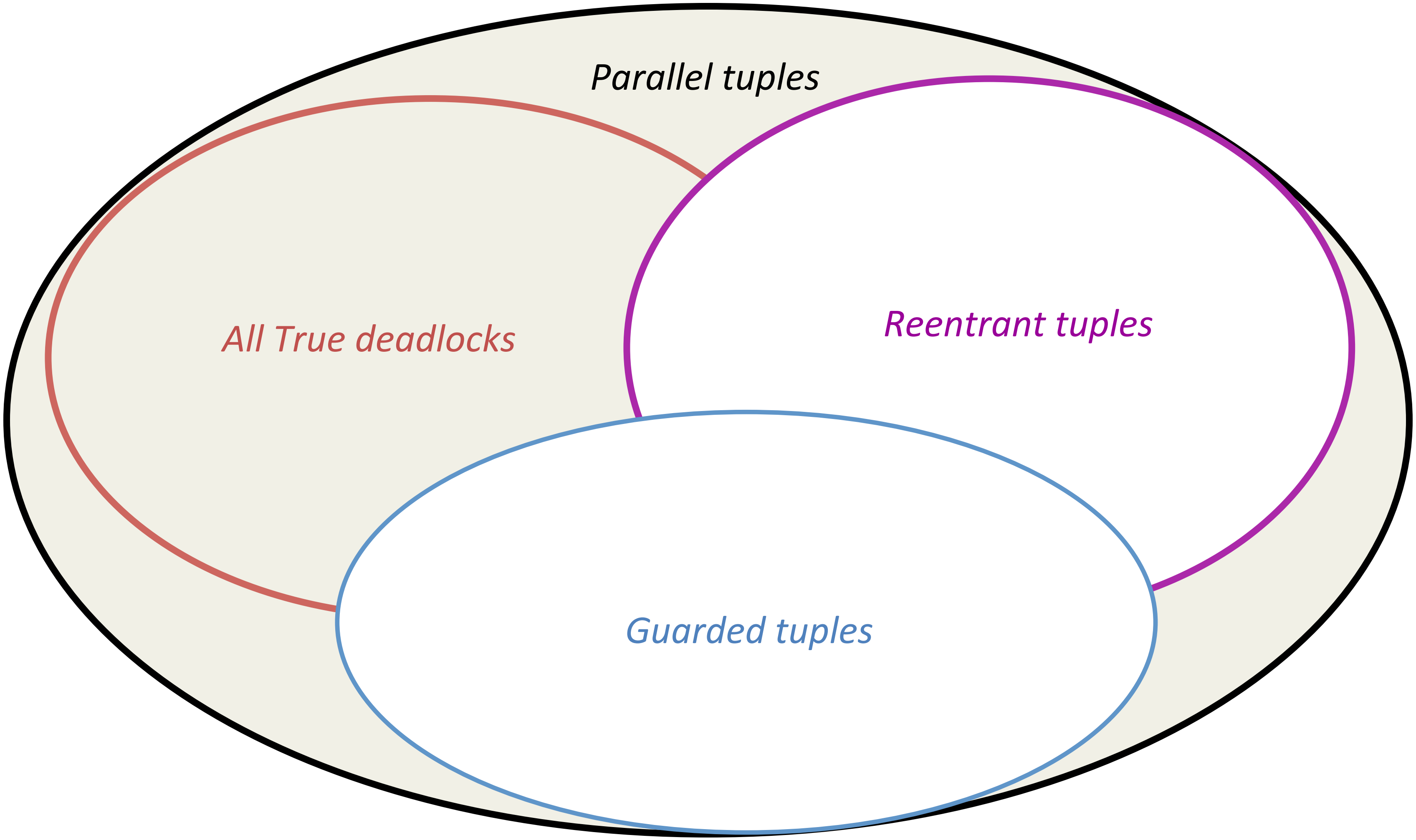
Recap

- k-object-sensitivity
  - call graph
  - points-to (may-alias)
- Deadlock conditions
- Some other analyses
  - thread-escape
  - may-happen-in-parallel
- Correctness
  - unsound(last two)
  - incomplete

Sound conditions for deadlock



Unsound conditions for deadlock



Post Processing

- Providing a counter example(path) for each deadlock
  - look like a stack trace
  - Shortest path between each lock acquisition
  - May be infeasible

- Grouping of counterexamples
  - Running Example:
    - Same lock types
    - Group both tuples together

Output example

Deadlock Reports	
Group 1	
Report 1	
Thread spawned by method java.lang.Thread.start() in context [main]	Thread spawned by method java.lang.Thread.start() in context [main]
Lock held at T.java:24 in context [main] allocated at {[java. lang. Object]}	Lock held at T.java:34 in context [main] allocated at {[java. lang. Object]}
Lock held at T.java:29 in context [main] allocated at {[java. lang. Object]}	Lock held at T.java:39 in context [main] allocated at {[java. lang. Object]}
[Shortest path from thread root to first lock] [Shortest path from first lock to second lock]	[Shortest path from thread root to first lock] [Shortest path from first lock to second lock]

Source: http://pag-www.gtisc.gatech.edu/conchord/examples/deadlock_test/deadlocks.html

Implementation

- JADE
  - Soot framework to make initial 0-CFA
  - Rewrite synchronized blocks to methods
  - Convert to SSA for percision
- k-object-sensitivity
  - Iteration and refinement
  - Stop conditions

Evaluation

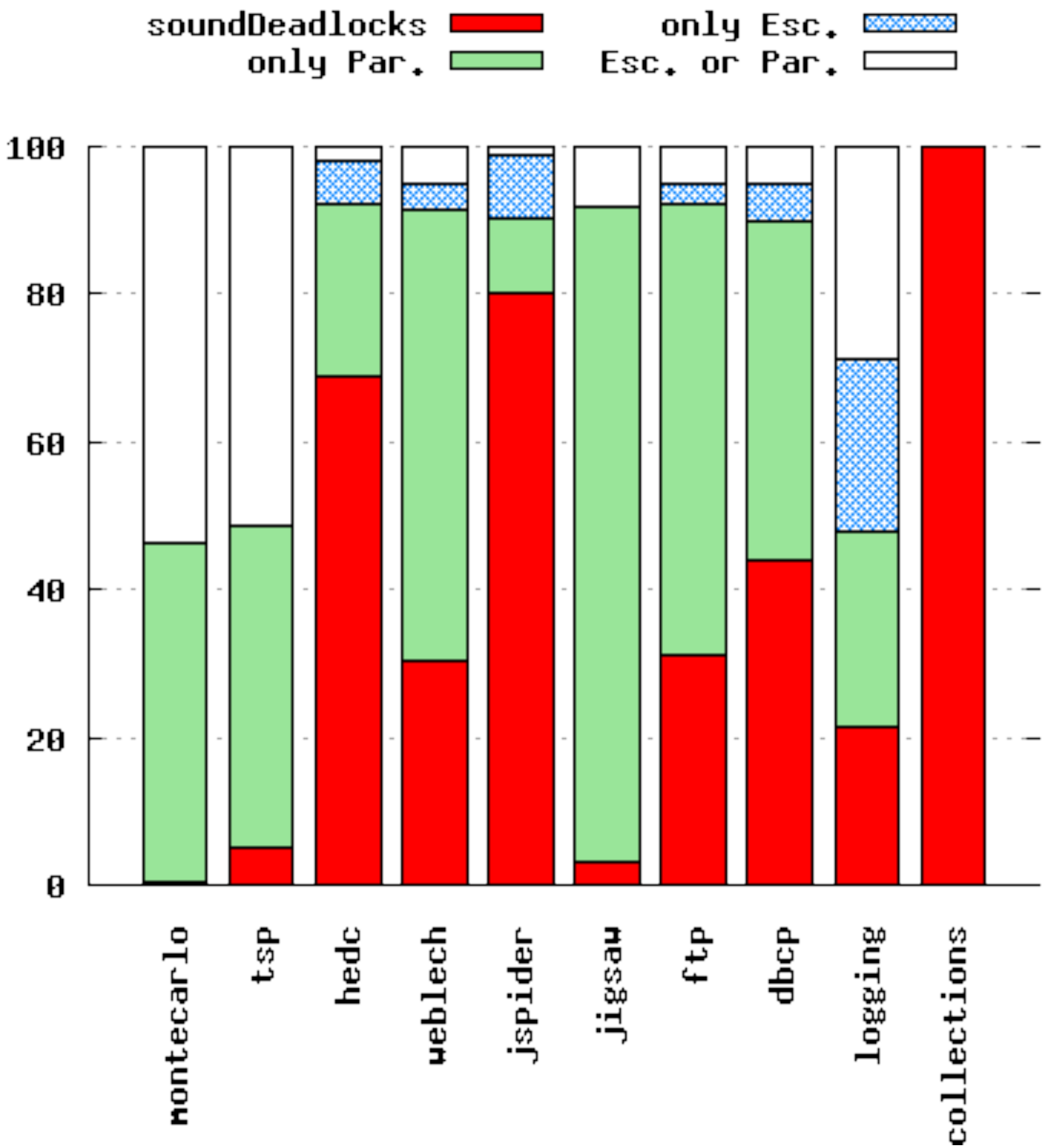
Benchmarks

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

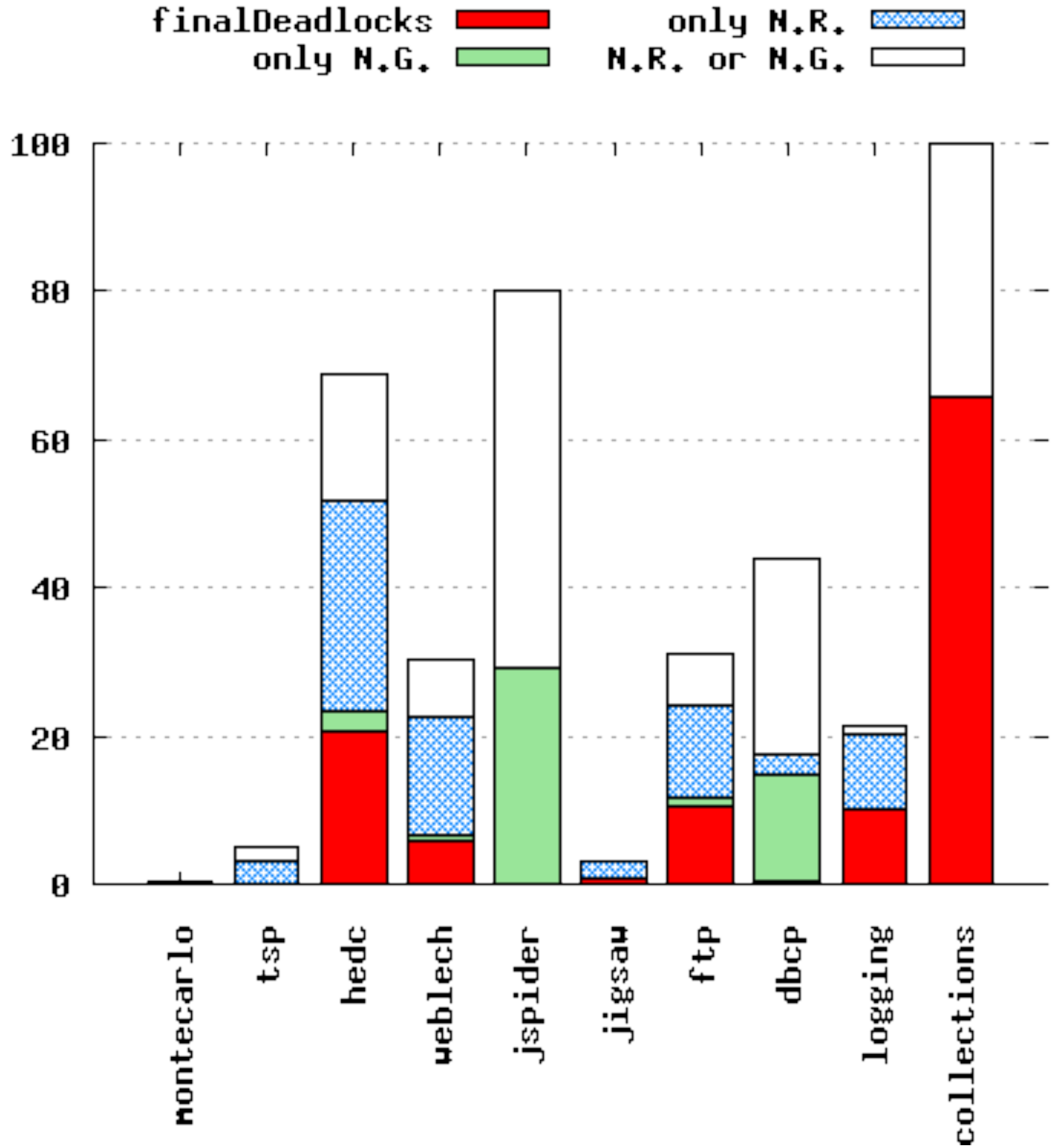
Experimental Results (k=1)

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 Analyses Contributions



Individual Analyses Contributions



Evaluation Conclusions

- Java.logging is(was) a mess
- Conditions are useful for reducing false positives
- Found all known deadlocks and some new ones
  - "Effective in practice"

Related Work

Static Analyses

- Type Systems
  - SafeJava
  - Annotation burden
- Model Checking
  - Promela/SPIN
  - finate-state

Static Analyses Cont.

- Dataflow Analysis
  - LockLint/Jlint augmentation
  - 0-CFA without parallel, escaping, guarded [Von Praum, 2008]
  - Lock-order graph + heuristics [A. Williams, 2005]
  - RacerX – flow sensitive, imprecise, hueristics
  - Sound, Ada [S. Masticola, 1993]

Dynamic Analysis

- Approaches
  - Visual Threads
  - Goodlock algorithm

- Inherently unsound
- Require test input data

Conclusions

- Object sensitivity for object sensitive idioms
  - Scalable, Precise
- Novel approach to static deadlock detection for Java
  - Use off-the-shelf static analyses
- Effective static analysis is possible
  - If we specialize in a given setting
  - And sacrifice soundness and/or completeness

Running Chord

- Chord is the open source tool that implements this algorithm (among others)
- Running on my code
  - Found 3 “possible”, out of them 2 were deadlocks
- Running on latest versions of JDK and some others
  - No deadlocks found
- Searching for deadlock reports
  - Going back to the versions that had bugs (reported)
  - Found 6/7 deadlocks that were reported and accepted

My* Idea

- Hybrid approach
- Compute possible paths
  - Use static analysis to compute a sound approximation
- Remove false positives using dynamic approaches
  - Try to run the trace output
  - Schedule the threads to create a deadlock.
- Remove most false-positive (noise), but keep soundness

My* Idea: Related Work

- An effective dynamic analysis for detecting generalized deadlocks. [P. Joshi, M. Naik, K. Sen, and D. Gay. FSE 2010]

- Jnuke [C. Artho, A. Biere. 2005]
- ConLock [Y. Cai, S. Wu, W. Chan. 2014]

Questions?

Discussion

- Will you use it?
- Sacrificing soundness, is it a big deal?
- How about JavaScript?
  - Field sensitivity?