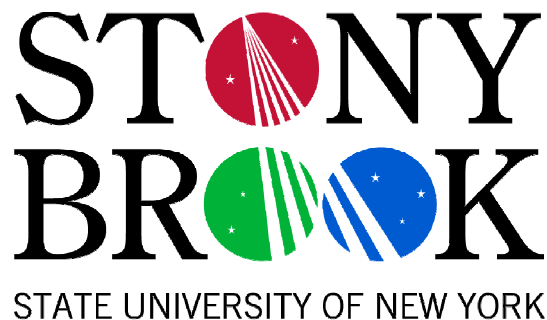
Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring

Rahul Agarwal, Liqiang Wang,
Scott D. Stoller



Common Concurrent Programming Errors

- **Deadlock** occurs when each thread is blocked trying to acquire a lock held by another thread.
- A **data race** occurs when two threads concurrently access a shared variable and at least one of the accesses is a write.
- **Atomicity violation**. A code block is **atomic** if every execution of the program is equivalent to an execution in which the code block is executed without interruption by other threads.

Example: Data Race

```
class Account {
   int balance =0;
  void deposit(int amt) {
     this.balance = this.balance + amt;
    }
}
```

```
Account a = new Account();
```

Suppose 2 threads invoke the deposit method without any synchronization.

Example: Data Race

```
class Account {
  int balance =0;
  void deposit(int amt) {
    this.balance = this.balance + amt;
  }
}

Account a = new Account();
```

Race condition can lead to lost update if both threads read balance and then both threads write it.

Example: Atomicity violation [Flanagan+, 2003]

```
void deposit(int amt) {
    int bal;
    synchronized (this) {
        bal = this.balance;
    }
    bal = bal + amt;
    synchronized (this) {
     this.balance = bal;
    } }
```

deposit(amt) is race-free but not atomic.

Atomicity violation can lead to lost update.

Example: Deadlock [Boyapati+, 2002]

```
class CombinedAccount {
  final Account saving = new Account();
  final Account checking = new Account();
    void transfer(int amt) {
      synchronized (checking) {
        synchronized (saving) {
          saving.bal -= amt;
          checking.bal += amt;
        }}}
}
```

```
int balance() {
  synchronized (saving) {
    synchronized (checking) {
      return saving.bal +
             checking.bal
    }}}
```

Deadlock Checking

- Run-time deadlock checking
  - GoodLock algorithm [Havelund 2000]
    - + automatic
    - - run-time overhead
    - - no guarantees about other executions
- Type systems for deadlock prevention
  - SafeJava [Boyapati et al., 2002]
    - + typable programs are deadlock free.
    - - writing the types is a burden
    - - more false alarms than run-time checking.

Contributions

- Generalized GoodLock algorithm for run-time detection of potential deadlocks involving more than 2 threads.
- Added deadlock types to our atomicity type system, Extended Parameterized Atomic Java (EPAJ), to give stronger atomicity guarantees.
- First type inference algorithm for deadlock types.
- Focused run-time deadlock checking: Run-Time checking is omitted for parts of the program guaranteed by the deadlock type system not to contribute to deadlocks.
  - This can reduce the overhead of run-time deadlock checking.

Outline

- Run-Time Deadlock Checking
- Race Types and Deadlock Types
- Inference of Deadlock Types
- Focused Run-Time Checking for Potential Deadlocks
- Experience
- Related Work

Outline

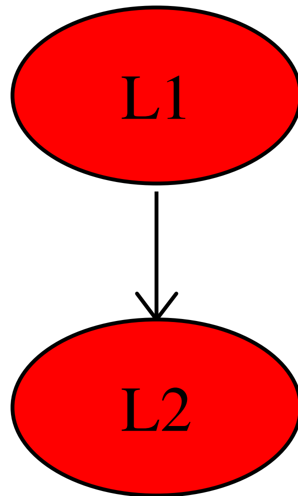
- Run-Time Deadlock Checking
- Race Types and Deadlock Types
- Inference of Deadlock Types
- Focused Run-Time Checking for Potential Deadlocks
- Experience
- Related Work

GoodLock Algorithm

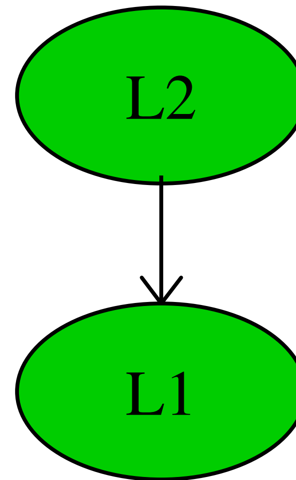
- Record lock tree for each thread.
  - Captures nested pattern in which locks were acquired by each thread.
- At the end of the execution of the program, if there exist threads *t1* and *t2* and locks *l1* and *l2* such that *t1* acquires *l2* while holding *l1*, and *t2* acquires *l1* while holding *l2*, then issue a warning of potential deadlock.
- Suppress warning when a gate lock is present.
  - Gate lock: a lock *l3* held by *t1* and *t2* that prevents interleaving of their acquires of *l1* and *l2*.
- Only detects potential deadlocks caused by interleaving of lock acquires in two threads.

Example

synchronized (L1) {
  synchronized (L2) {
   … }}



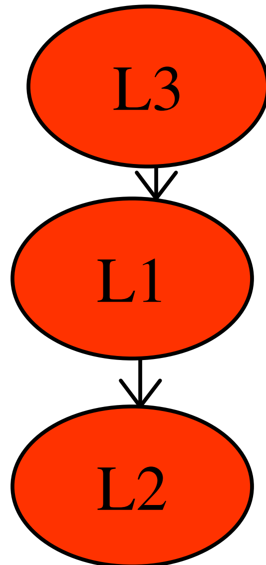
synchronized (L2) {
  synchronized (L1) {
   … }}



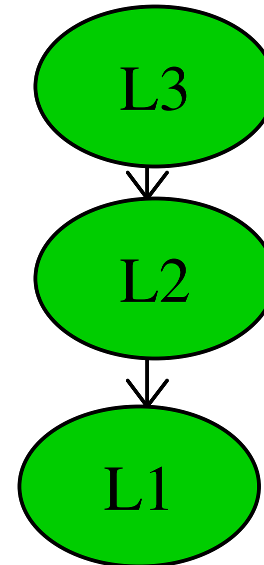
Warning: Potential Deadlock

Example

```
synchronized (L3) {
 synchronized (L1) {
   synchronized (L2)  {
   … }}}
```




```
synchronized (L3) {
  synchronized (L2) {
    synchronized (L2)  {
      … }}}
```



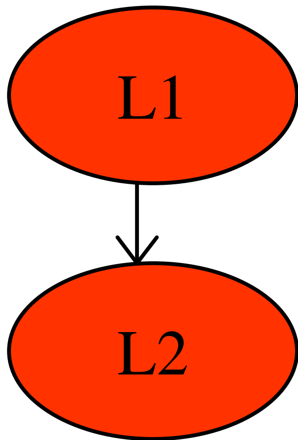
Gate lock L3 is present. No warning.

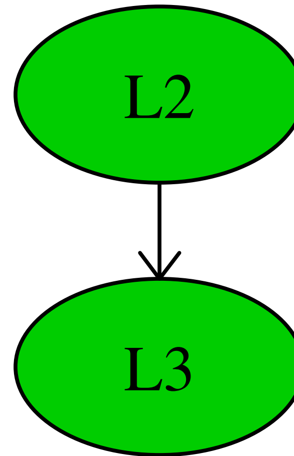
Example

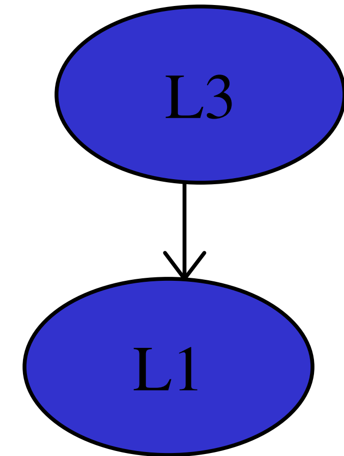
synchronized (L1) {
  synchronized (L2) {
    … }}

synchronized (L2) {
  synchronized (L3) {
    … }}

synchronized (L3) {
  synchronized (L1) {
    … }}







No warning from GoodLock algorithm, but there is potential for deadlock

Generalized GoodLock Algorithm Ignoring Gate Locks

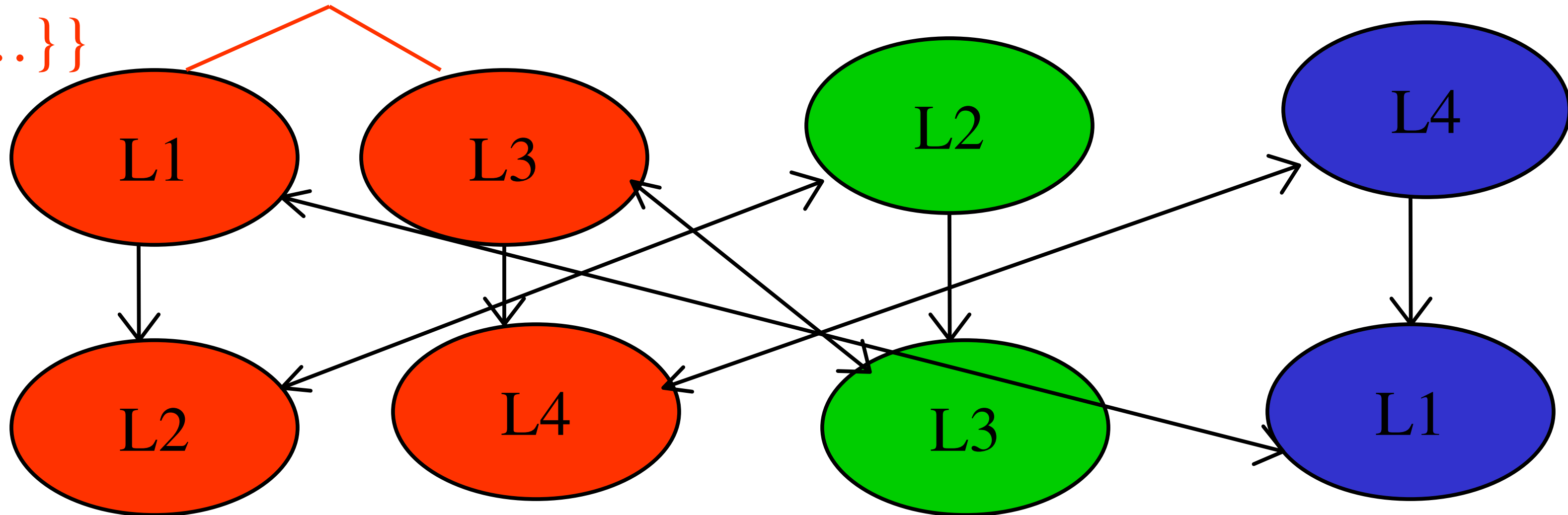
- Construct a run-time lock graph that contains
  - tree edges: the directed (from parent to child) edges in each run-time lock tree, and
  - inter edges: bidirectional edges between nodes that are labeled with the same lock and that are in different run-time lock trees.
- Issue a warning iff the lock graph contains a valid cycle.
- Valid cycle: a cycle that
  - does not contain consecutive inter edges, and
  - for each thread t, nodes from t appear as at most one consecutive subsequence in the cycle.

Example

```
synchronized (L1) {
  synchronized (L2)  {
   … }}
synchronized (L3) {
  synchronized (L4) {
   …}}
```

```
synchronized (L2) {
  synchronized (L3) {
   … }}
```

```
synchronized (L4) {
  synchronized (L1)  {
   … }}
```



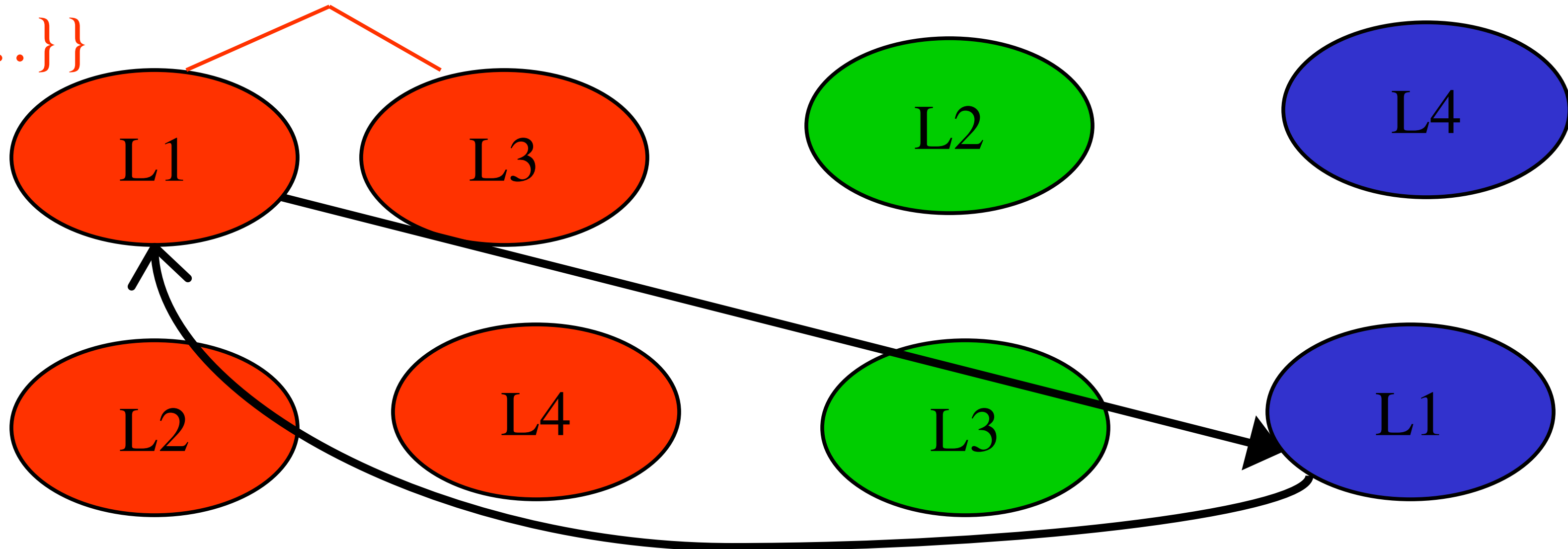
No valid cycles. No deadlock.

Example

synchronized (L1) {
  synchronized (L2) {
    … }}
synchronized (L3) {
  synchronized (L4) {
    …}}

synchronized (L2) {
  synchronized (L3) {
    … }}

synchronized (L4) {
  synchronized (L1) {
    … }}



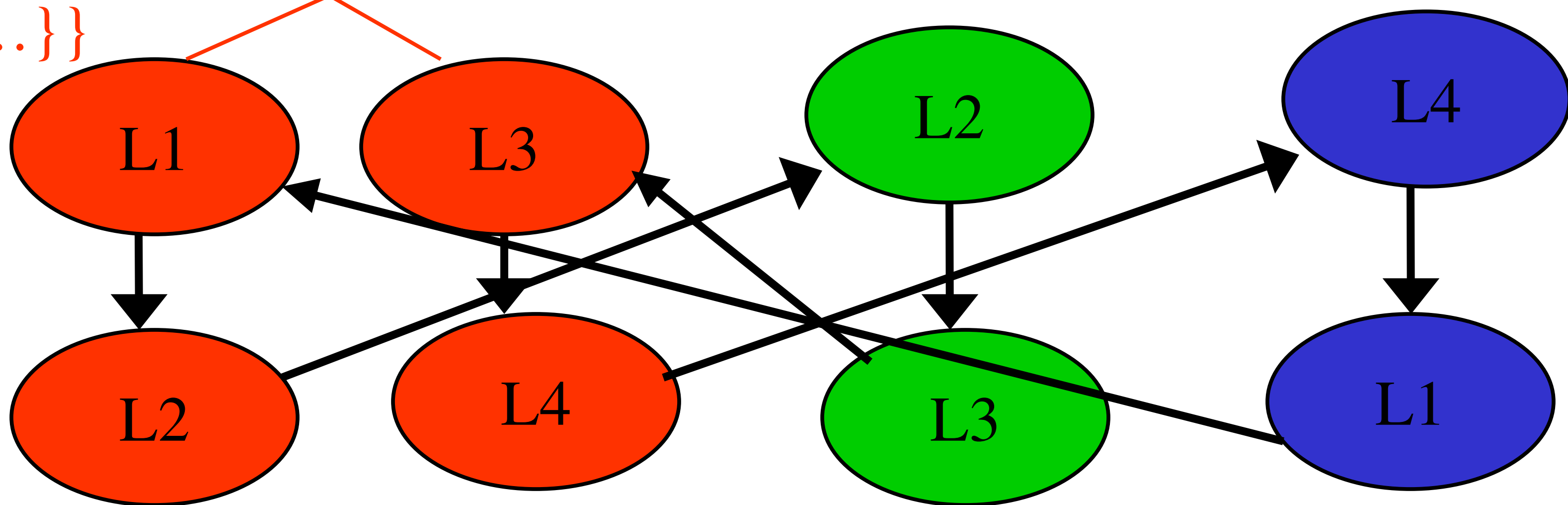
No valid cycles (consecutive inter edges). No deadlock.

Example

```
synchronized (L1) {
  synchronized (L2)  {
   … }}
synchronized (L3) {
  synchronized (L4) {
   …}}
```

```
synchronized (L2) {
  synchronized (L3) {
   … }}
```

```
synchronized (L4) {
  synchronized (L1)  {
   … }}
```



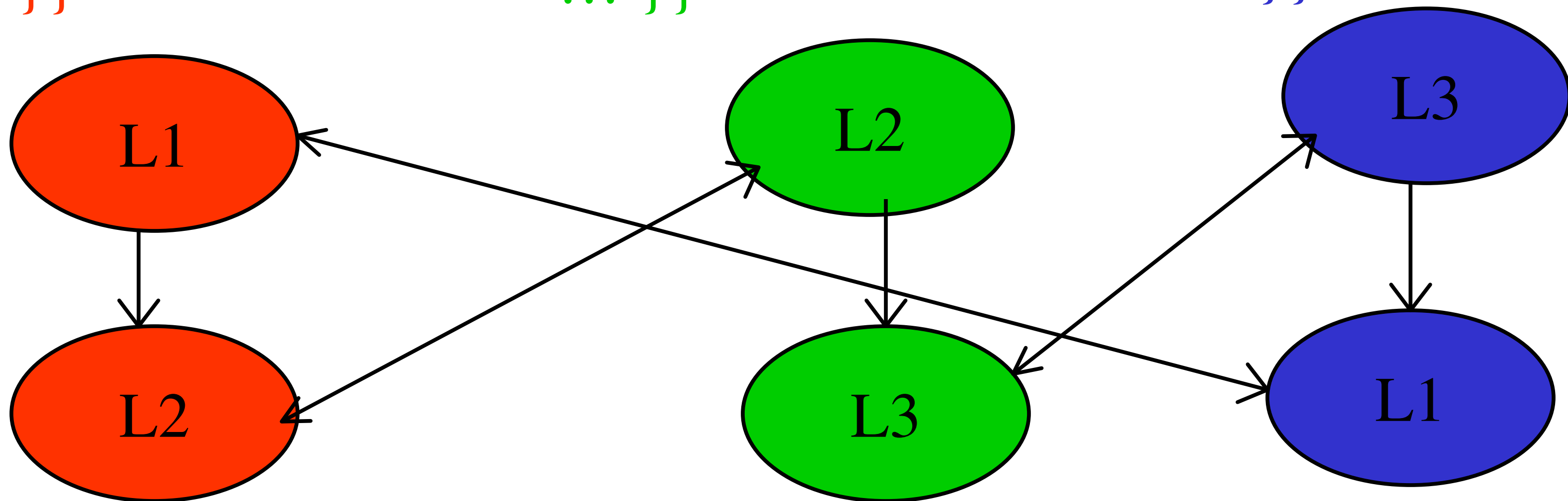
No valid cycles (red thread visited twice). No deadlock.

Example

synchronized (L1) {
  synchronized (L2) {
    … }}

synchronized (L2) {
  synchronized (L3) {
    … }}

synchronized (L3) {
  synchronized (L1) {
    … }}



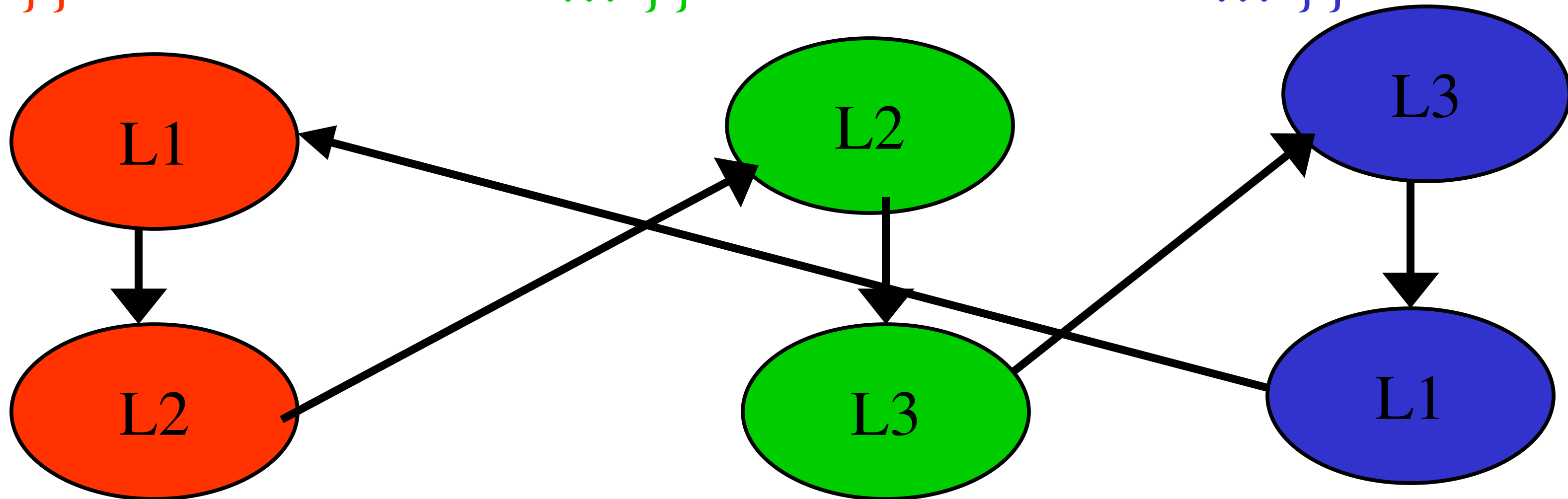
Potential Deadlock

Example

synchronized (L1) {
synchronized (L2) {
… }}

synchronized (L2) {
synchronized (L3) {
… }}

synchronized (L3) {
synchronized (L1) {
… }}



Valid cycle. Warning: Potential deadlock.

Detecting Valid Cycles

- Perform a modified depth-first search, starting from the root of each lock tree.
- Extend current path to traverse only valid paths (defined just like valid cycles).
- Do not keep track of which nodes were visited. This allows a node to be explored multiple times. This is necessary because the set of threads with a lock-tree node on the stack might be different on different visits, allowing different valid extensions to the path.
- Worst-case time complexity: $O(|V|^{|Thread|}\ |Thread|!)$

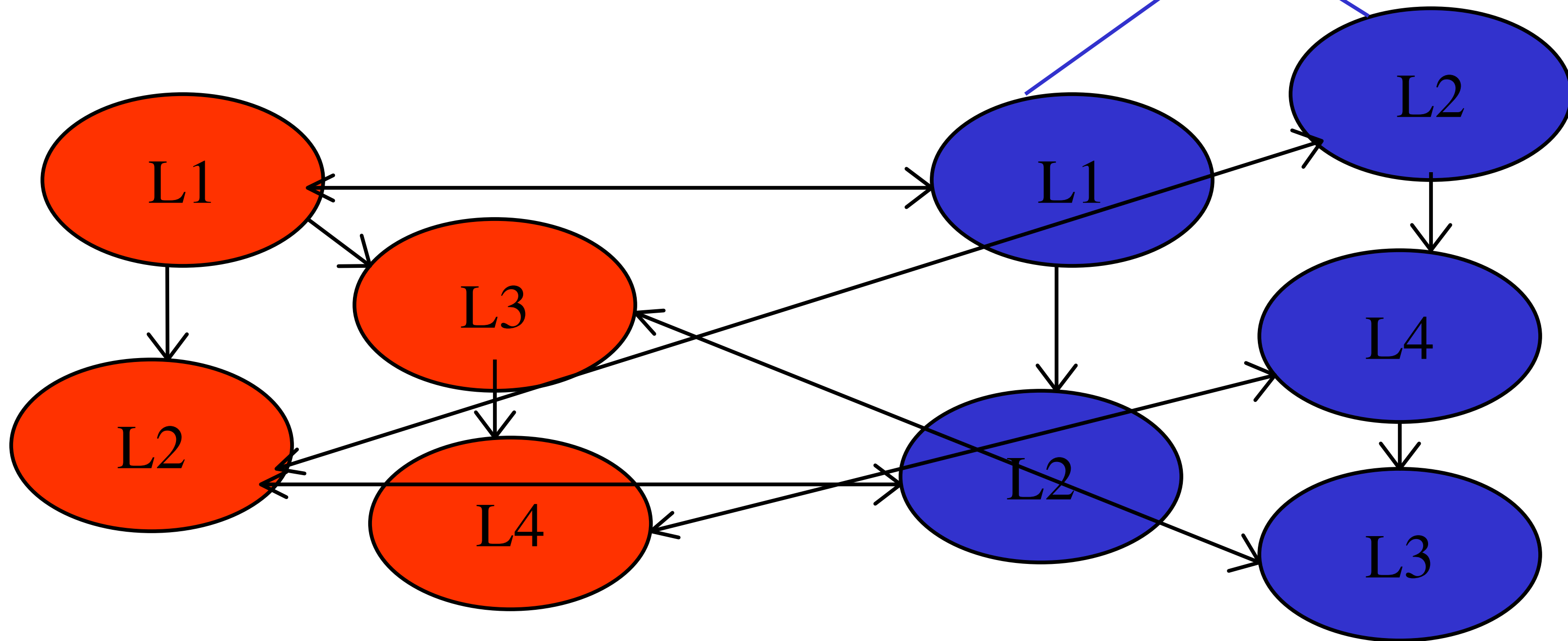
Detecting Valid Cycles: Optimized Algorithm

- When a node is visited, record the set of threads with a lock-tree node on the stack.
- Perform a modified DFS starting at each node, looking for cycles containing that node.
- If the search started at node *n* reaches a node *n1* again with the same set of threads with a lock-tree node on the stack, *n1* is not visited again.
- Justification: There is no valid path from *n1* back to *n* that avoids the lock trees on the stack, because if there were, the search would have detected the cycle (and terminated) during the earlier visit.
- Worst-case time complexity: $O(2^{|Thread|} \ |V|^3)$

Starting Optimized Alg from Roots Only is Unsound

```
synchronized (L1) {
  synchronized (L2)  {… }}
synchronized(L1)  {
  synchronized (L3) {
    synchronized (L4) {…}}
```

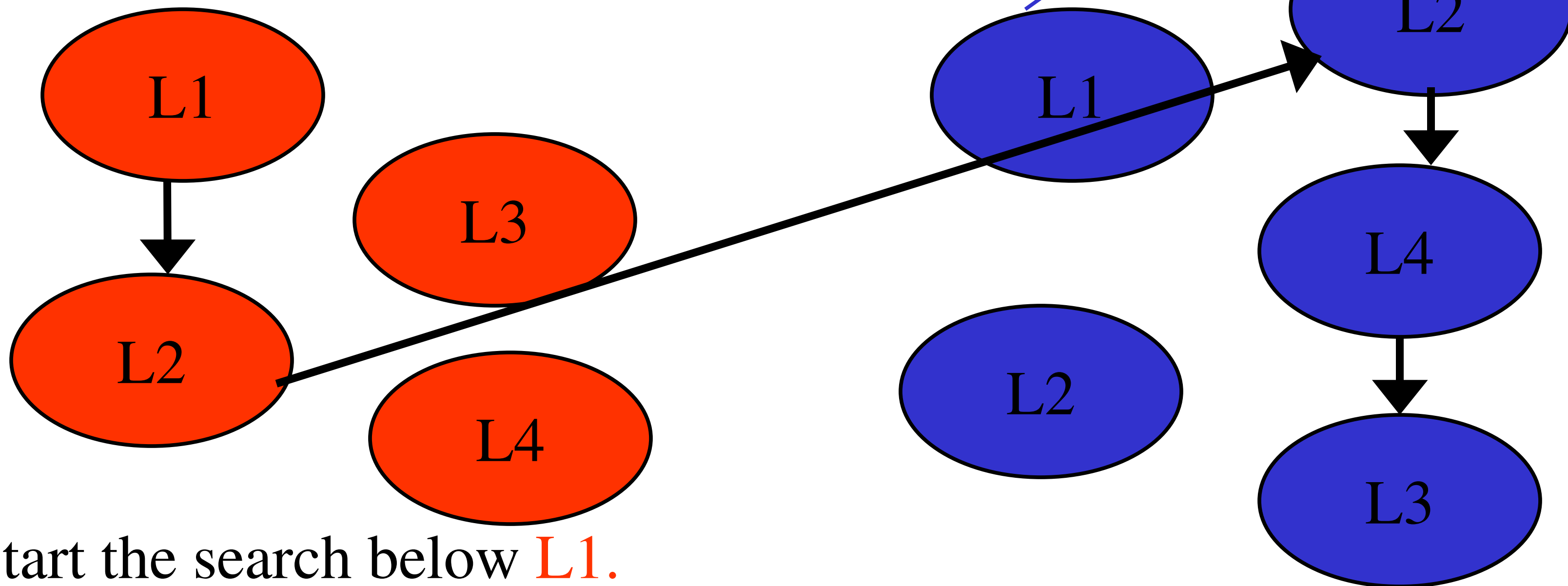
```
synchronized (L1) {
  synchronized (L2)  {… }}
synchronized (L2) {
  synchronized (L4)  {
    synchronized(L3)  {… }}
```



Starting Optimized Alg from Roots Only is Unsound

```
synchronized (L1) {
  synchronized (L2)  {… }}
synchronized(L1)  {
  synchronized (L3) {
    synchronized (L4) {…}}
```

```
synchronized (L1) {
  synchronized (L2)  {… }}
synchronized (L2) {
  synchronized (L4)  {
    synchronized(L3)  {… }}
```

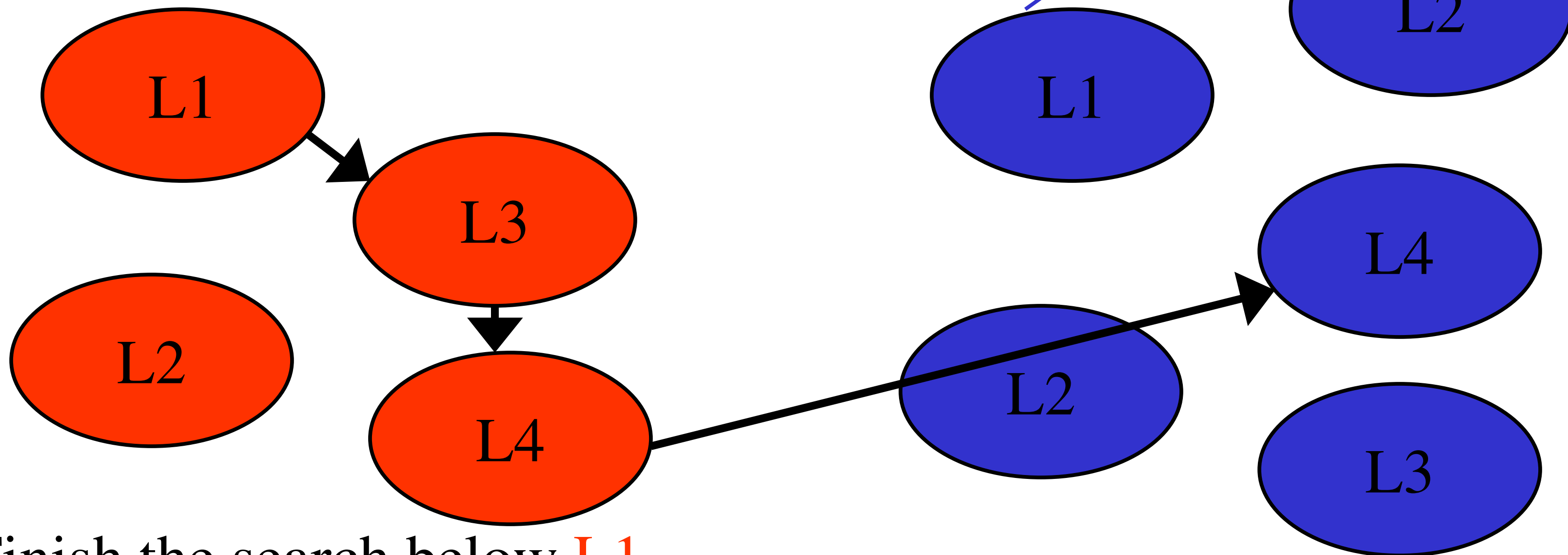


Start the search below L1.

Starting Optimized Alg from Roots Only is Unsound

synchronized (L1) {
  synchronized (L2) {… }}
synchronized(L1) {
  synchronized (L3) {
    synchronized (L4) {…}}

synchronized (L1) {
  synchronized (L2) {… }}
synchronized (L2) {
  synchronized (L4) {
    synchronized(L3) {… }}

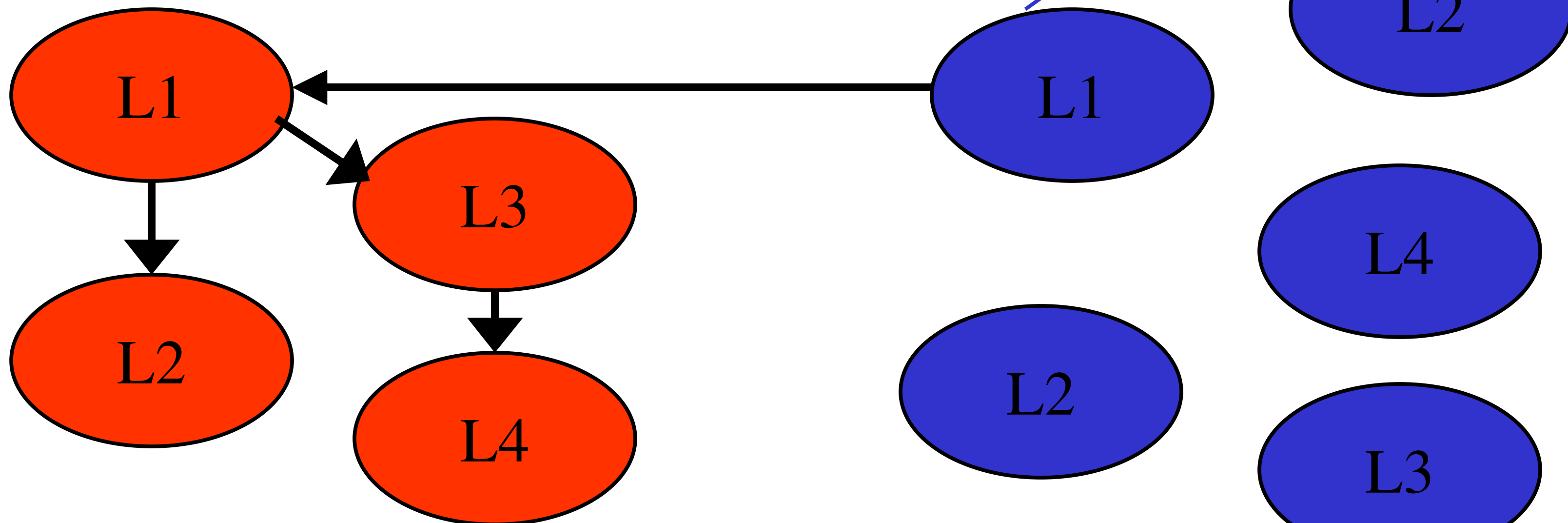


Finish the search below L1.

Starting Optimized Alg from Roots Only is Unsound

```
synchronized (L1) {
  synchronized (L2)  {… }}
synchronized(L1)  {
  synchronized (L3) {
    synchronized (L4) {…}}
```

```
synchronized (L1) {
  synchronized (L2)  {… }}
synchronized (L2) {
  synchronized (L4)  {
    synchronized(L3)  {… }}
```

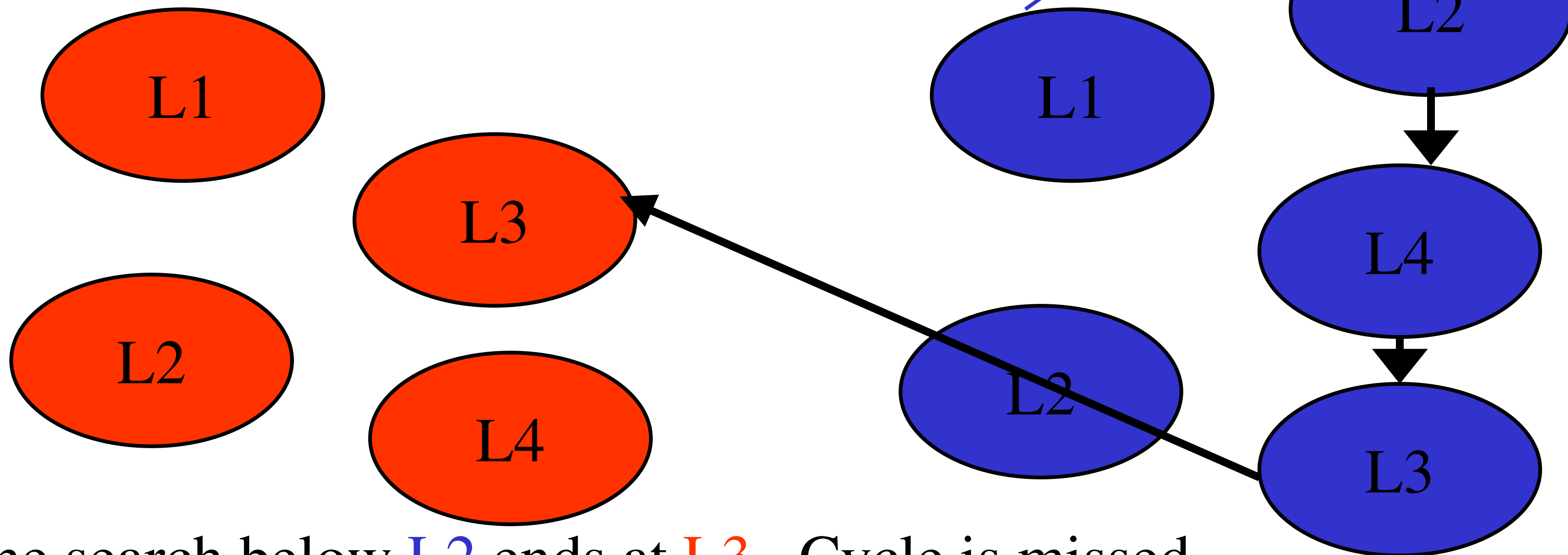


Perform the search below L1.

Starting Optimized Alg from Roots Only is Unsound

synchronized (L1) {
  synchronized (L2) {… }}
synchronized(L1) {
  synchronized (L3) {
    synchronized (L4) {…}}}

synchronized (L1) {
  synchronized (L2) {… }}
synchronized (L2) {
  synchronized (L4) {
    synchronized(L3) {… }}}



The search below L2 ends at L3. Cycle is missed.

Generalized GoodLock Algorithm with Gate Locks

- *l* is a gate lock for a cycle if two or more nodes that are on the cycle and are in different run-time lock trees have ancestors labeled with *l*.
- Issue a warning of potential deadlock iff the run-time lock graph contains a valid cycle with no gate lock.

Outline

- Run-Time Deadlock Detection
- Race Types and Deadlock Types
- Inference of Deadlock Types
- Focused Run-Time Checking for Potential Deadlocks
- Experience
- Related Work

Parameterized Race Free Java (PRFJ) [Boyapati & Rinard, 2001]

- Each object is associated with an owner and a rootowner
- Owner is normally an object indicated by a final expression or self.
- Lock on rootowner must be held when object is accessed.
- Special owners indicate cases where no lock is needed:
  - thisThread: unshared object
  - unique: unique reference to that object (so race condition is impossible)
  - readonly: object's fields cannot be updated
- A method *m* may have an accesses *e1*,*e2*,... clause; rootowners of *e1*,*e2*,... must be held when *m* is invoked.

Example PRFJ program

```
class Account<thisOwner> {
  int balance;
  public Account(int balance) { this.balance = balance; }

  void deposit(int amt) accesses this {
     this.balance = this.balance + amt;
 }
}
Account<thisThread> a1 = new Account<thisThread>(0);
a1.deposit(10);
Account<self> a2 = new Account<self>(0);
fork { synchronized (a2) { a2.deposit(10); }}
fork { synchronized (a2) { a2.deposit(10); }}
```

Example PRFJ Program

```
class CombinedAccount <readonly>{
  final Account<self> saving = new Account<self>();
  final Account <self> checking = new Account <self> ();
    void transfer(int amt) {
      synchronized (saving) {
            synchronized (checking) {
              saving.bal -= amt;
              checking.bal += amt;
            }}}
}
```

```
int balance() {
  synchronized (saving) {
    synchronized (checking) {
      return saving.bal +
                checking.bal
    }}}
```

Deadlock Types [Boyapati+, 2002]

- Associate a lock level with each lock.
  - In PRFJ, only locks on objects with owner self can be acquired. (Only those objects can be rootowners.)
- Lock levels are partially ordered.
- Typing rules ensure that if a thread acquires a lock *l2* (which the thread does not already hold) while holding a lock *l1*, then *l2*'s level is less than *l1*'s level.
- This ensures there are no deadlocks due to locking.
- We consider the basic deadlock type system where all instances of a class have the same lock level.

Example

```
class CombinedAccount <readonly>{
  Locklevel savL = new;
  Locklevel chkgL < savL;
  final Account<self:savL> saving = new Account;
  final Account <self:chkgL> checking = new Account;
    void transfer(int amt) {
      synchronized (saving) {
            synchronized (checking) {
              saving.bal -= amt;
              checking.bal +=amt;
}           }}}

    int balance() {
      synchronized (saving) {
        synchronized (checking) {
          return saving.bal +
                    checking.bal
    }}}
```

Deadlock Types: locks Clause

- Each method is annotated with a locks *lvl1*, *lvl2*, … clause.
- The outermost synchronized expressions of the method may acquire locks with these levels.
- The typing rule for method calls ensures that the levels of locks held by the caller are higher than the levels in the called method's locks clause.

Example

```
class CombinedAccount <readonly>{
 LockLevel savL = new;
 LockLevel chkgL < savL;
 final Account<self:savL> saving = new Account;
 final Account <self:chkgL> checking = new Account;
   void transfer(int amt) locks (savL){
    synchronized (saving) {
        synchronized (checking) {
         saving.bal -= amt;
         checking.bal += amt;
}       }}}

   int balance() locks (savL){
    synchronized (saving) {
     synchronized (checking) {
       return saving.bal +
            checking.bal
     }}}
```

Deadlock Types: locks Clause

- A locks clause may also contain a lock *l*, which indicates that the caller may hold a lock on object *l* and the called method may reacquire *l*.
  - Normally a method can only acquire locks whose level is less than the levels of locks held by the caller.
- Example:

```
class C<self:cL> {
   LockLevel cL = new;
   synchronized void foo() { …; this.bar(); … }
   synchronized void bar() locks (this) { … }
}
```

Deadlock-Free Extended Parameterized Atomic Java (DEPAJ)

- DEPAJ = Extended Parameterized Atomic Java (EPAJ) extended with deadlock types.
- EPAJ is an extension of PRFJ (described earlier) with atomicity types [Flanagan & Qadeer].
- DEPAJ provides stronger atomicity guarantees by ensuring absence of deadlocks due to locks.
  - A code segment that can deadlock in the middle is not atomic.
  - Earlier atomicity type systems (Atomic Java, EPAJ) ignore the possibility of deadlocks.

Outline

- Run-Time Deadlock Detection
- Race Types and Deadlock Types
- Inference of Deadlock Types
- Focused Run-Time Checking for Potential Deadlocks
- Experience
- Related Work

Inference of Deadlock Types: Step 0

- Deadlock type inference algorithm produces the best types it can. It does not check typability. The type checker does.
- Deadlock type inference assumes race-free types are known. Inference of race-free types is NP-hard.
- We obtain race-free types using type discovery [Agarwal, Sasturkar, & Stoller 2004].
  - Alternative approach: translate to SAT, and use a SAT solver [Flanagan & Freund 2004]. SAT solvers are impressive, but it's still NP-hard.

Type Discovery for Race-Free Types

- Novel combination of static and run-time analysis
  - Run the program, guess race-free types based on the observed behavior, check them with the type checker.
- Inexpensive
- Necessarily incomplete (fails for some typable programs)
- Effective: gets 98% of annotations in our experiments
- Test suite with low branch coverage is fine!
  - Static intra-procedural type inference efficiently propagates discovered types into unexecuted branches.
- Does not discover types for unexecuted methods.
  - Can still show that parts of the program are race-free.

Type Discovery for Race-Free Types

1. Identify unique references using static analysis
2. Instrument the program to record, for each declaration d of a field, parameter, or return type, a set S(d) of objects stored there, and for each field f of each o in S(d),
   - lockset for o.f
   - whether o.f is readonly (not written after initialization)
   - whether o.f is shared (accessed by multiple threads)
   - values val(e) of final expressions e in scope at d
     - If val(e) is in lockset(o.f), then e is a candidate owner (protecting lock) of o.f.
3. Analyze log to get race-free types for these declarations.
4. Infer remaining types using intra-procedural static anal.

Inference of Deadlock Types: Step 1

- Assign a new distinct lock level to each field, method parameter and local variable with owner self .
  - Initially these lock levels are unordered.
- Merge lock levels based on the constraints:
  - Left and right sides of an assignment must have the same lock level.
  - An argument to a method and the corresponding parameter must have the same lock level.

Example

```
class C {
 final Object<self> lock1 = new Object();
 final Object<self> lock2 = new Object();
 final Object<self> lock3  = new Object();
```

lock level l1

lock level l2

lock level l3

```
m1()  {
  synchronized (lock1) {
      synchronized (lock2) {
         m2(lock2);
      }}}
}
```

```
m2(Object<self>  lock4) {
```

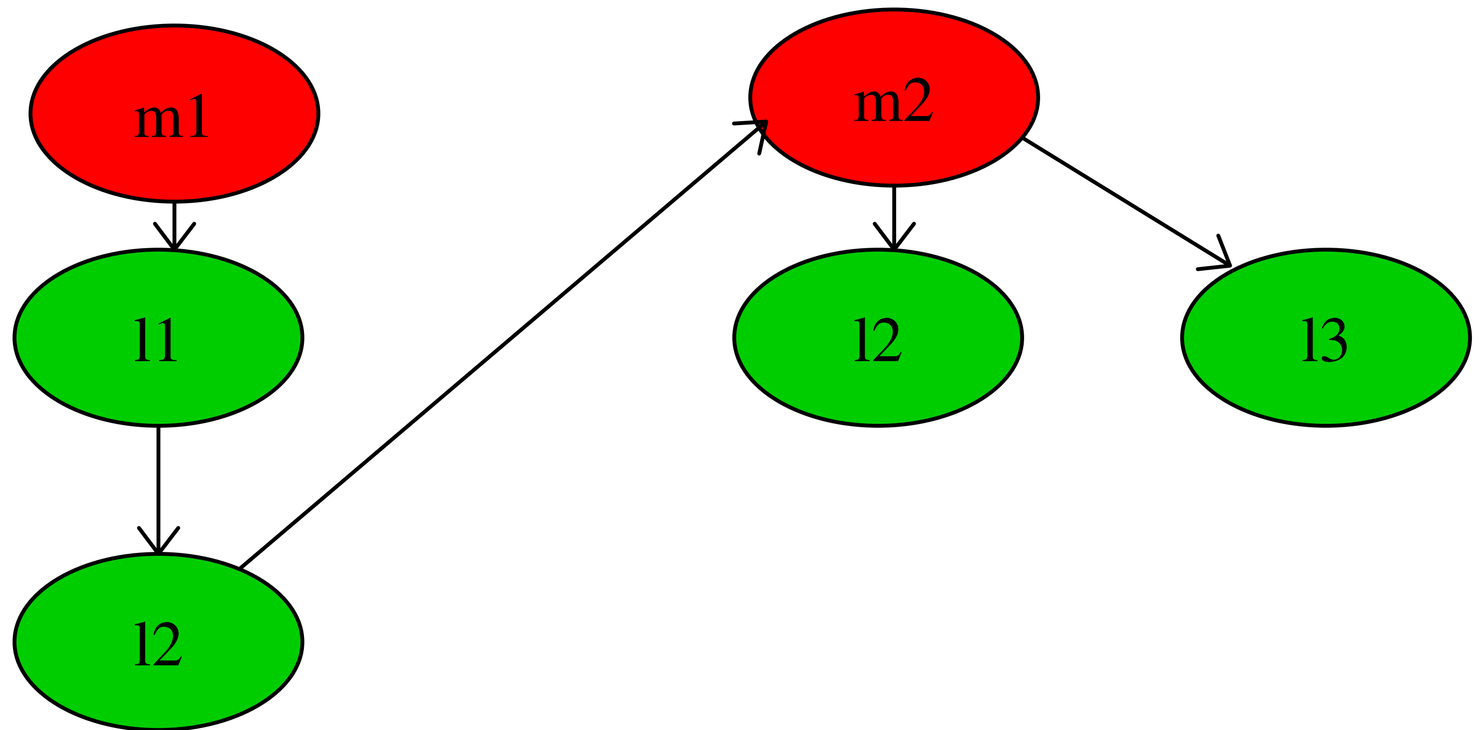
lock level l4 = l2

```
   synchronized (lock4) {
    }
   synchronized (lock3) {
    }}
```

Inference of Deadlock Types: Step 2

- Construct the static lock graph.
  - A lock node for each synchronized statement
  - A method node for each method.
  - An edge between nested synchronized statements.
  - An edge from method node to lock node of each outermost synchronized statement in the method.
  - For each method call within the scope of a synchronized statement, an edge from the lock node corresponding to the inner-most synchronized statement enclosing the method call to the method node for the called method.

Example



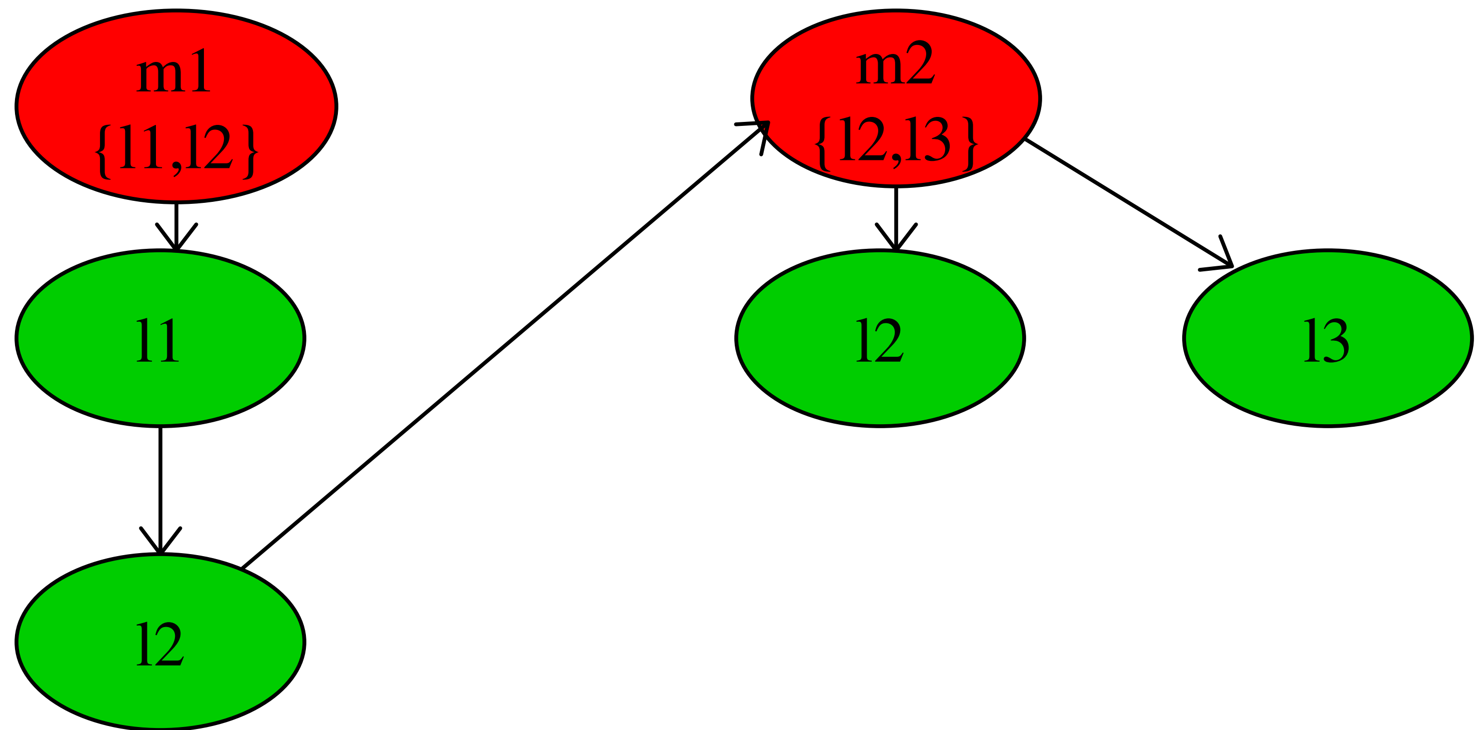
Static lock graph

Inference of Deadlock Types: Step 3

- Infer lock levels in the locks clause of each method.
  - Associate a set L_m of lock levels with each method m.
  - Let L_m^{init} denote the lock levels of lock nodes that are children of the method node for m.
  - For each method m, L_m is the least solution to

    $L_m = L_m^{init} \cup \cup_{m1 \in called(m)} L_{m1}$
  - called(m): set of methods directly called by method m outside of synchronized blocks.
  - Add L_m to m's locks clause.

Example (contd..)

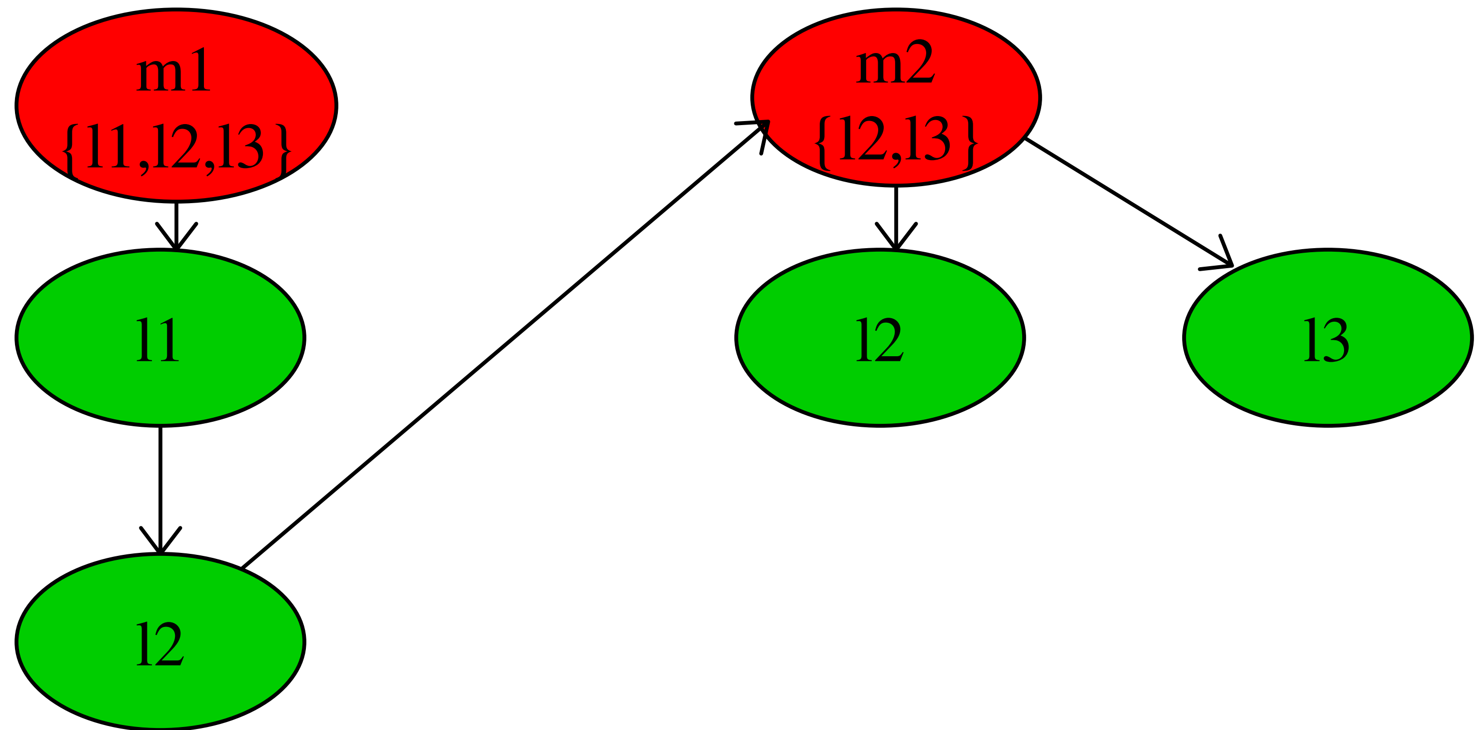


m1 locks l1,l2 m2 locks l2,l3

Inference of Deadlock Types: Step 4

- Infer orderings between lock levels.
  - For each edge from a lock node with level l1 to a child lock node with level l2, add the declaration $l1 > l2$.
  - For each edge from a lock node with level l1 to a method node for method m, for each lock level l2 in L_m, if $l1 \neq l2$, add the declaration $l1 > l2$.

Example (contd..)



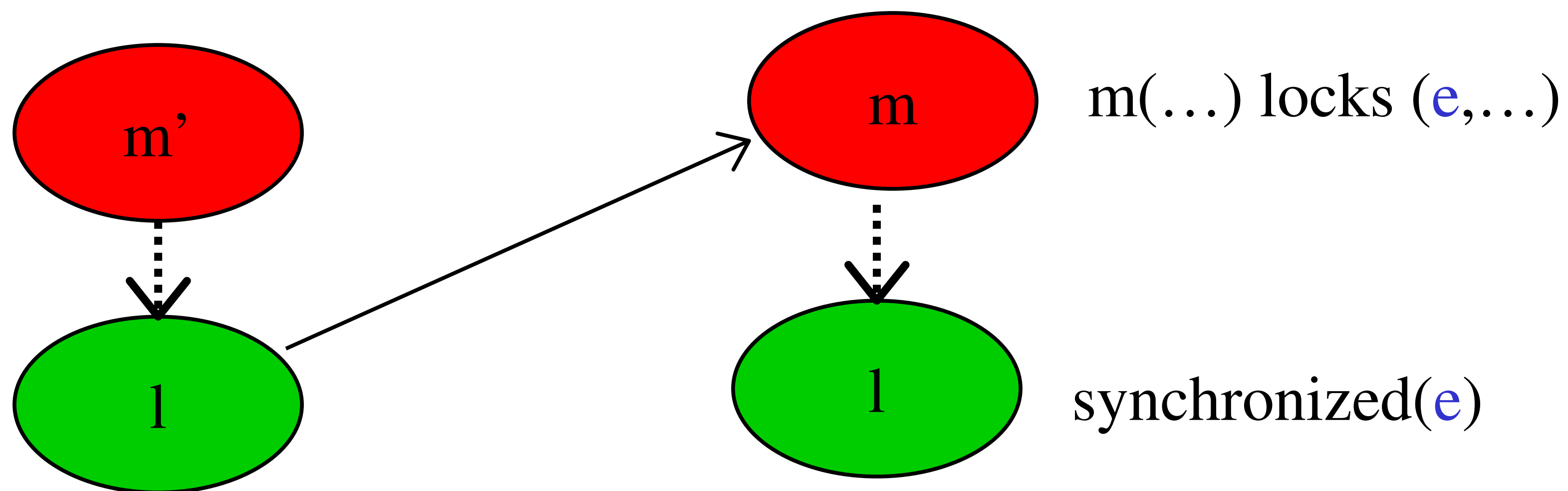
l1> l2   l2 > l3

Inference of Deadlock Types: Step 5

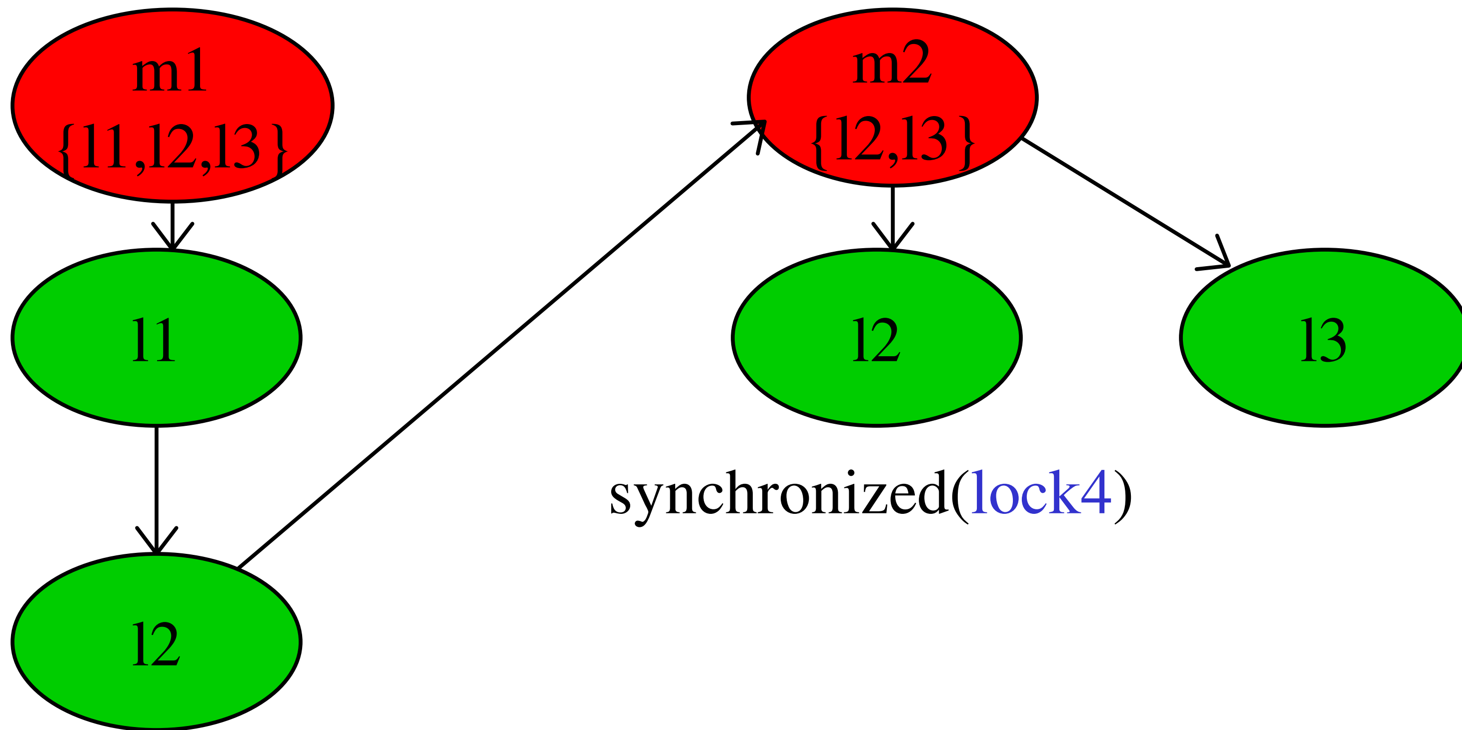
- Infer the lock (if any) in the locks clause of each method.
  - For each lock node n with lock level l in method m, if n is reachable in the lock graph from a lock node ancestor of n with the same lock level and in a different method, then add to the locks clause of m the final expression denoting the lock acquired by the synchronized statement corresponding to n.

Inference of Deadlock Types: Step 5

- Infer the lock (if any) in the locks clause of each method.
  - For each lock node n representing acquire of a lock e in method m, if n is reachable from a lock node with the same lock level and in a different method, then add e to the locks clause of m.



Example



Add lock4 to m2's locks clause.

Example: Inferred Typing

```
Class C {
 Locklevel l1>l2>l3;
  final Object<self:l1> lock1 = new Object;
 final Object <self:l2> lock2 = new Object;
 final Object <self:l3> lock3  = new Object;

  m1()  locks l1,l2,l3 {
     synchronized (lock1) {
         synchronized (lock2) {
            m2(lock2);
         }}}
```

```
m2(Object <self:l2> lock4 ) locks l2,l3, lock4{
   synchronized (lock4) {
    }
   synchronized (lock3) {
    }}
```

```
}
```

Outline

- Run-Time Deadlock Detection
- Race Types and Deadlock Types
- Inference of Deadlock Types
- Focused Run-Time Checking for Potential Deadlocks
- Experience
- Related Work

Focused Run-Time Checking for Potential Deadlocks

- Deadlock types are used to focus run-time checking for partially typable programs.
- Run-time checking is omitted for parts of the program guaranteed not to contribute to deadlocks.
- Find all cycles of the form l1>l2>...>l1 among orderings inferred by the deadlock type inference algorithm.
- Instrument and analyze only acquires and releases of locks whose lock level is part of a cycle.
  - Other synchronized statements do not need to be instrumented.

Example of Focused Run-Time Checking

```
m1() {
 synchronized (l1) {
  synchronized (l2) {
    synchronized (l3) {
    }
  }
 }
}
```

```
m2() {
 synchronized (l4) {
  synchronized (l3) {
    synchronized (l2) {
    }
  }
 }
}
```

Instrument only l2 and l3

Partially Typable Deadlock-Free Programs

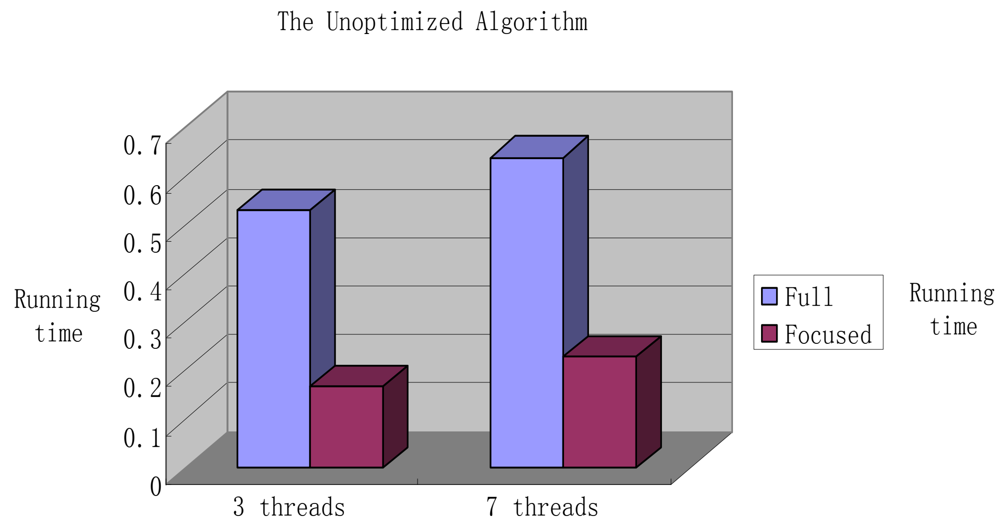
- Programs which have cycles in the static lock level ordering may be deadlock-free because
  - the nested locks acquired by different threads are different, even though they have the same lock level, or
  - other synchronization mechanisms, e.g., fork-join, prevent deadlock.
- Programs which put objects with different lock levels in the same Collection are untypable.

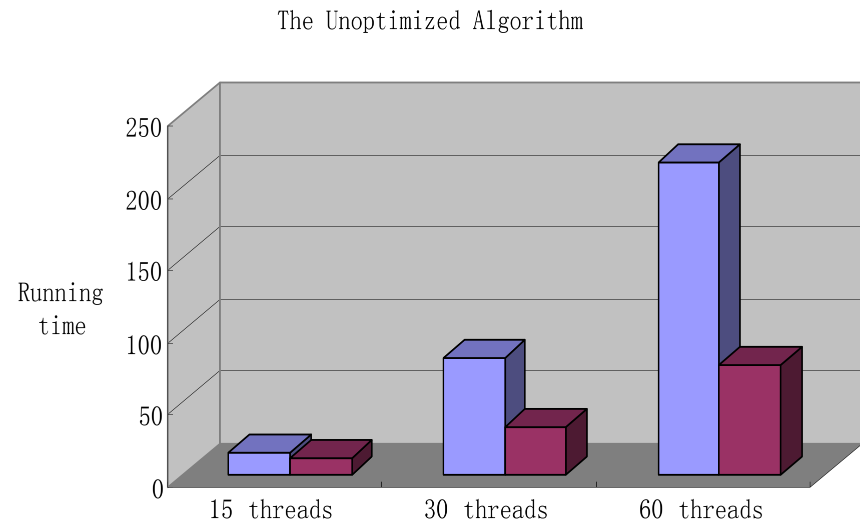
Outline

- Run-Time Deadlock Detection
- Race Types and Deadlock Types
- Inference of Deadlock Types
- Focused Run-Time Checking for Potential Deadlocks
- Experience
- Related Work

Experience

- Implement the unoptimized and optimized generalized Goodlock algorithms without gate locks.
- Compare performance of full and focused checking.
- Perform experiment on the modified elevator program, which is deadlock-free but not typable.





Experience

- For both algorithms (unoptimized and optimized), focusing provides average speedup of about 57% and average space reduction of 41%.
- Optimized algorithm is slower for elevator! It uses more complicated data structures, and the graph is simple.

		3 threads	7 threads	15 threads	30 threads	60 threads
Base time		0.23s	0.30s	0.52s	2.60s	6.60s
Full	Size	621	1037	1848	3359	5734
	Unopt	0.76s	0.94s	14.93s	1m23.08s	3m42.9s
	Opt	1.10s	1.66s	17.32s	1m28.05s	4m3.0s
Focused	Size	433	646	1063	1824	2947
	Unopt	0.40s	0.53s	11.71s	34.91s	1m22.66s
	Opt	0.51s	0.72s	12.40s	36.35s	1m28.28s

Outline

- Run-Time Deadlock Detection
- Race Types and Deadlock Types
- Inference of Deadlock Types
- Focused Run-Time Checking for Potential Deadlocks
- Experience
- Related Work

Related Work on Deadlock

- Run-time checking
  - GoodLock Algorithm [Havelund, SPIN 2000]
  - Compaq's Visual Threads [Harrow, SPIN 2000]
  - ConTest [Edelstein et al., 2003]
  - Generalized GoodLock [Bensalem & Havelund, PADTAD 2005]
- Static analysis
  - Boyapati et al. [OOPSLA 2002], Engler et al. [SOSP 2003], von Praun [2004],Williams et al. [ECOOP 2005]
  - Other static analyses also produce numerous false alarms and could be used to focus run-time checking.

Related Work: Static Analysis To Optimize Run-Time Checking

- Detecting Races
  - von Praun and Gross [OOPSLA 2001]
  - Choi, Lee, Loginov, O'Callahan, Sarkar, and Sridharan [PLDI 2002]
  - Agarwal, Sasturkar, Wang, Stoller [ASE 2005]
- Detecting Atomicity Violations
  - Sasturkar, Agarwal, Wang, Stoller [PPoPP 2005]
  - Agarwal, Sasturkar, Wang, Stoller [ASE 2005]

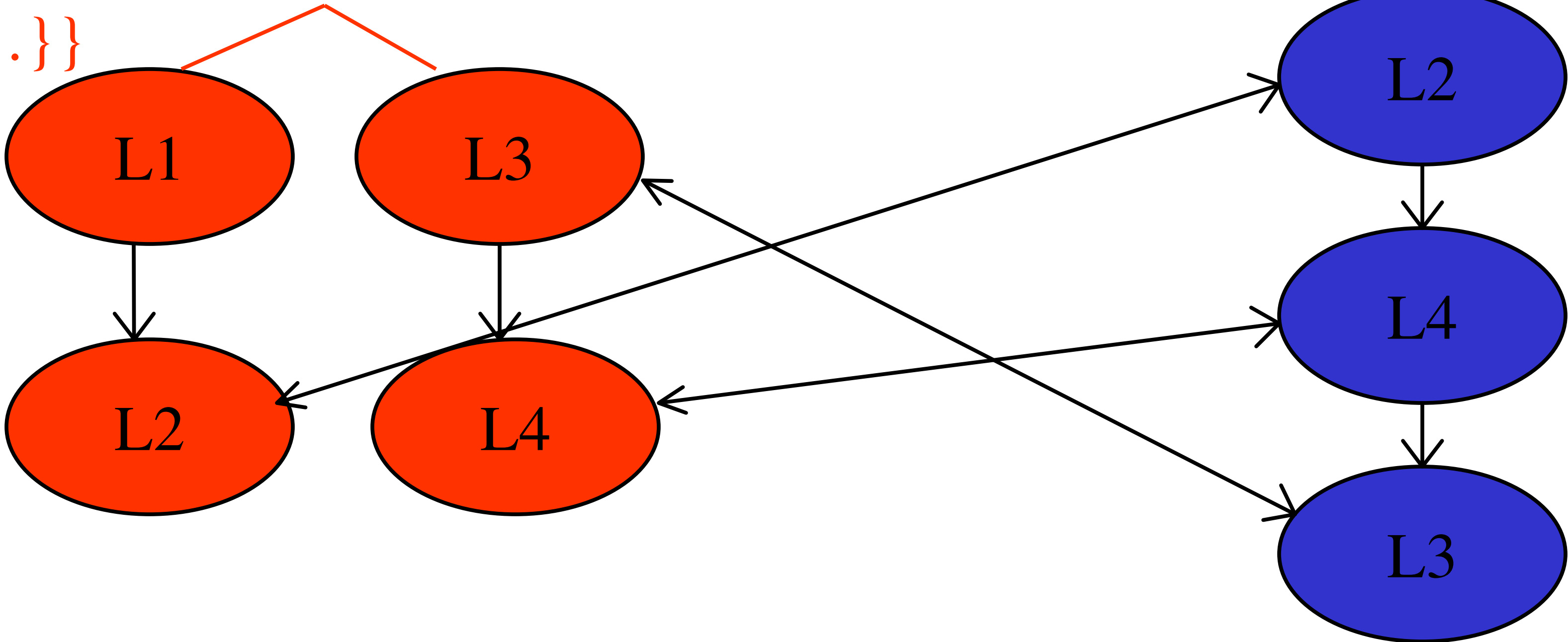
Contributions

- Generalized GoodLock algorithm for run-time detection of potential deadlocks involving more than 2 threads.
- Added deadlock types to our atomicity type system, Extended Parameterized Atomic Java (EPAJ), to give stronger atomicity guarantees.
- First type inference algorithm for deadlock types.
- Focused run-time deadlock checking: Run-Time checking is omitted for parts of the program guaranteed by the deadlock type system not to contribute to deadlocks.
  - This can reduce the overhead of run-time deadlock checking.

Optimized Alg Starts Search From Each Node

```
synchronized (L1) {
  synchronized (L2)  {
   … }}
synchronized (L3) {
  synchronized (L4) {
   …}}
```

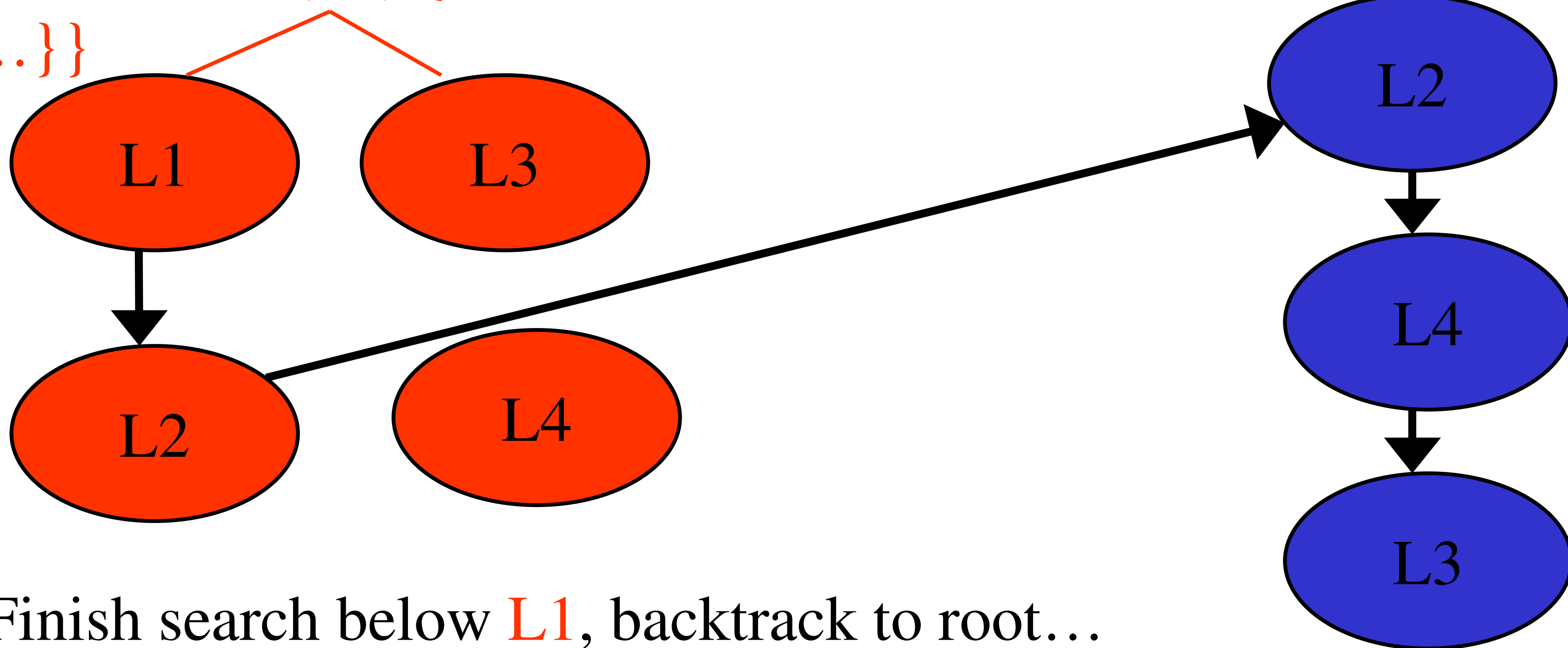
```
synchronized (L2) {
  synchronized (L4)  {
    synchronized(L3)  {
   … }}}
```



Optimized Alg Starts Search From Each Node

```
synchronized (L1) {
  synchronized (L2)  {
   … }}
synchronized (L3) {
  synchronized (L4) {
   …}}
```

```
synchronized (L2) {
  synchronized (L4)  {
    synchronized(L3)  {
   … }}}
```

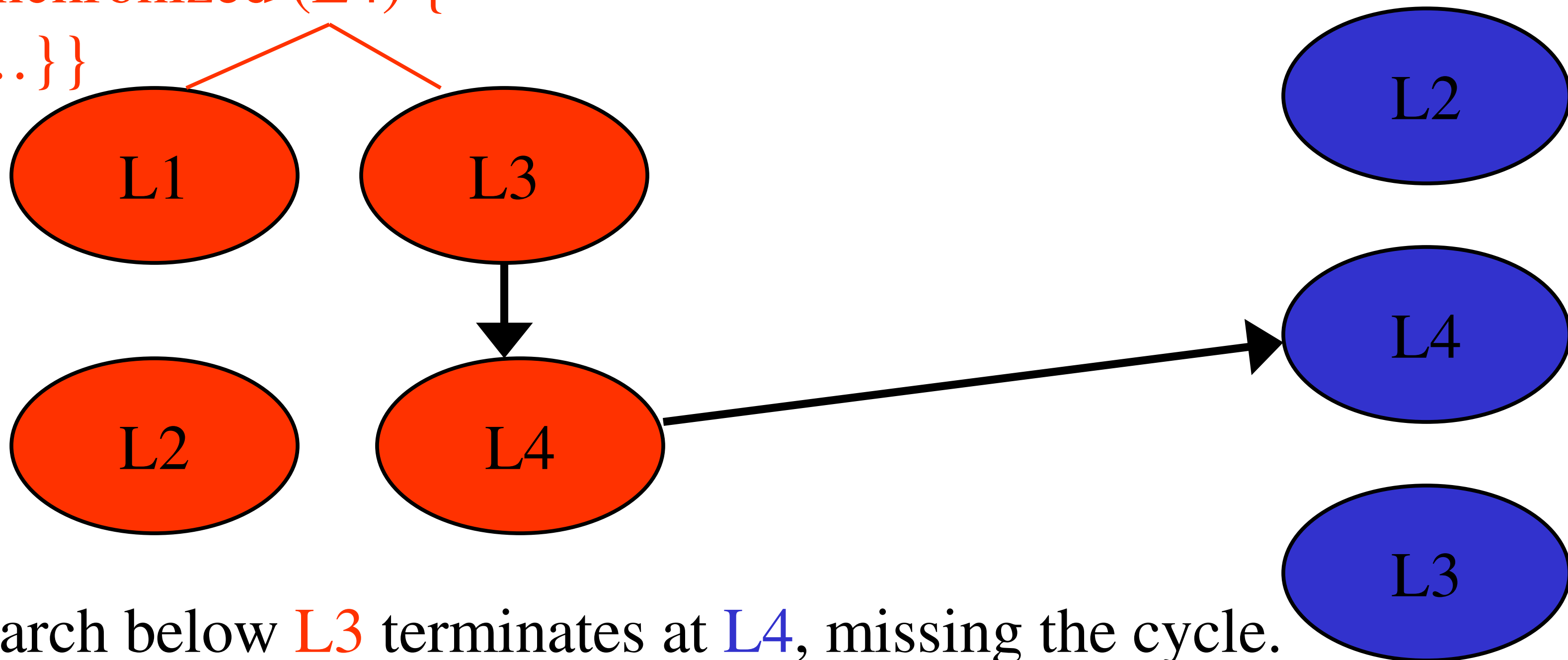


Finish search below L1, backtrack to root…

Optimized Alg Starts Search From Each Node

```
synchronized (L1) {
  synchronized (L2)  {
   … }}
synchronized (L3) {
  synchronized (L4) {
   …}}
```

```
synchronized (L2) {
  synchronized (L4)  {
    synchronized(L3)  {
   … }}}
```



Search below L3 terminates at L4, missing the cycle.