Component-Based Lock Allocation

Richard Halpert   Chris Pickett   **Clark Verbrugge**

School of Computer Science, McGill University
{rhalpe,cpicke,clump}@sable.mcgill.ca

6th Workshop on Compiler-Driven Performance
CASCON
October 22nd, 2007

Lock Allocation

- *Critical section*: piece of code that accesses shared state exclusively
- *Lock*: object that guards access to a critical section

- *Lock allocation*: mapping locks to critical sections

Sounds straightforward, but manual approaches are tricky!

Race Conditions

```
class T1 extends Thread
{
  public static Object a;

  run ()
  {
    synchronized (T1.a)
    {
      Main.i++;
    }
  }
}
```

race condition!

```
class T2 extends Thread
{
  public static Object b;

  run ()
  {
    synchronized (T2.b)
    {
      Main.i++;
    }
  }
}
```

Deadlock

```
class T1 extends Thread
{
  public static Object a;

  run ()
  {
    synchronized (T1.a)
      {
        synchronized (T2.b)
          {
            Main.i++;
          }
      }
  }
}
```

deadlock!

```
class T2 extends Thread
{
  public static Object b;

  run ()
  {
    synchronized (T2.b)
      {
        synchronized (T1.a)
          {
            Main.i++;
          }
      }
  }
}
```

Performance Degradation

```
class T1 extends Thread
{
  public static Object a;

  run ()
  {
    synchronized (T1.a)
      {
        synchronized (T2.b)
          {
            t1Work();
          }
      }
  }
}
```

```
class T2 extends Thread
{
  public static Object b;

  run ()
  {
    synchronized (T1.a)
      {
        synchronized (T2.b)
          {
            t2Work();
          }
      }
  }
}
```

performance degradation!

Goal

Our approach: *automatic* lock allocation

Goal: simplify concurrent programming

- Remove burden of manual allocation from programmer
- Aim to be *strictly* simpler: no extra language constructs
- Ideal result: automatic allocation performance matches or exceeds manual allocation performance

Contributions

Our contributions:

- We investigate *component-based* lock allocation:
  - Coarse locking granularity
  - Construct a critical section interference graph
  - One lock per graph component
- Experiment with many static compiler analyses
- Show results for small and large Java benchmarks

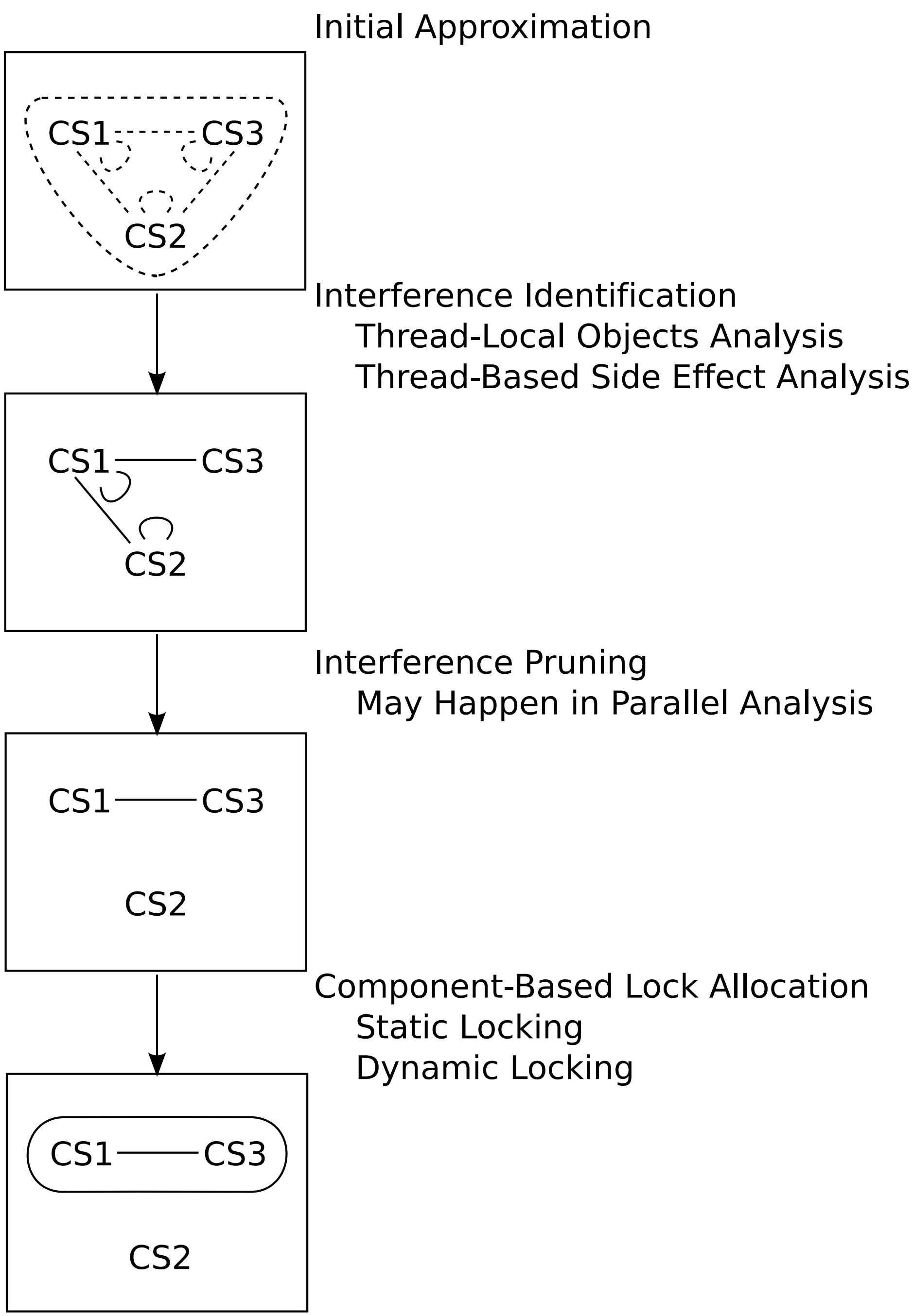
The technique often performs well:

- Matches manual allocation performance on 2, 4, 8-way hardware for mtrt (SPEC JVM98), lusearch and xalan (DaCapo), and SPEC JBB2005.

Outline

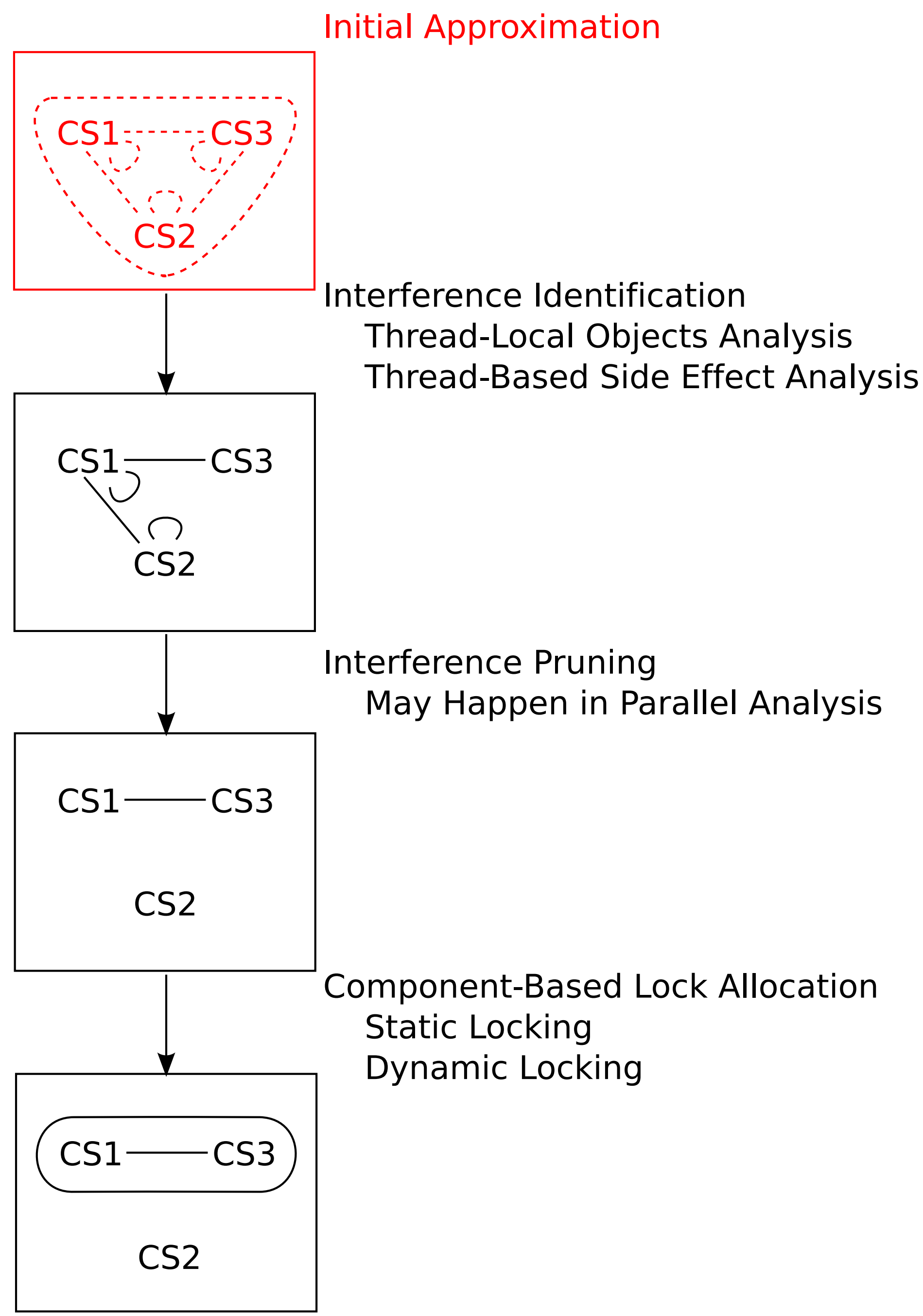
Analysis Pipeline

```
class G {
   public static int X, Y;
}

class T1 extends Runnable {
   run() {
      synchronized(...) { // CS1
         G.Y = G.X;
      }
      synchronized(...) { // CS2
         G.X = G.X + 1;
      }
   }
}

class T2 extends Runnable {
   run() {
      synchronized(...) { // CS3
         int a = G.Y;
      }
   }
}
```



Initial Approximation

```
class G {
   public static int X, Y;
}

class T1 extends Runnable {
   run() {
      synchronized(...singletonObject...) { // CS1
         G.Y = G.X;
      }
      synchronized(...singletonObject...) { // CS2
         G.X = G.X + 1;
      }
   }
}

class T2 extends Runnable {
   run() {
      synchronized(...singletonObject...) { // CS3
         int a = G.Y;
      }
   }
}
```



Thread-Based Side Effect Analysis

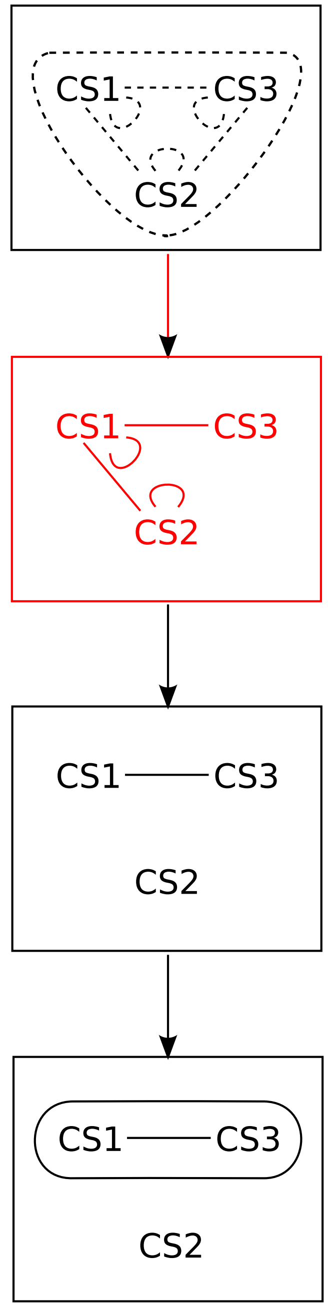
```
class G {
   public static int X, Y;
}

class T1 extends Runnable {
   run() {
      synchronized(...) { // CS1
         G.Y = G.X;
      }
      synchronized(...) { // CS2
         G.X = G.X + 1;
      }
   }
}

class T2 extends Runnable {
   run() {
      synchronized(...) { // CS3
         int a = G.Y;
      }
   }
}
```

G.Y = G.X; ← Read from X, thread-shared; Write to Y, thread-shared

G.X = G.X + 1; ← Read from X, thread-shared; Write to X, thread-shared

int a = G.Y; ← Read from Y, thread-shared

Initial Approximation



Interference Identification
- Thread-Local Objects Analysis
- Thread-Based Side Effect Analysis

Interference Pruning
- May Happen in Parallel Analysis

Component-Based Lock Allocation
- Static Locking
- Dynamic Locking

May Happen in Parallel Analysis

Find and apply MHP information

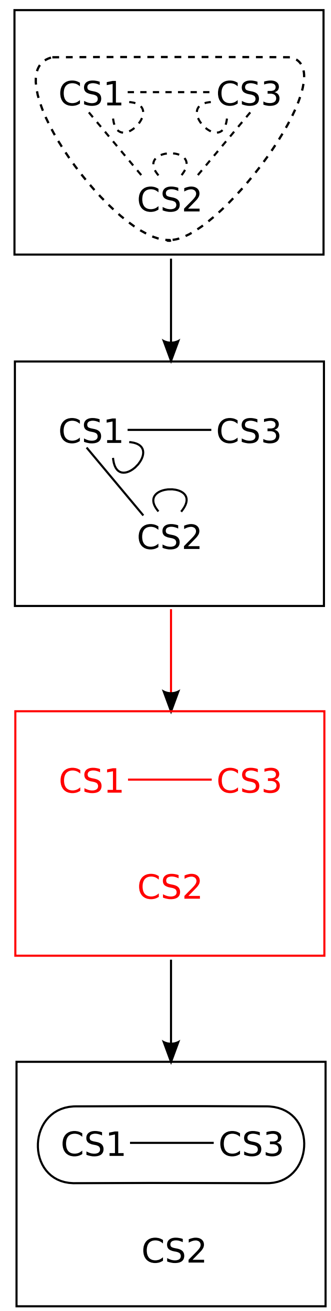
```
class G {
   public static int X, Y;
}

class T1 extends Runnable {
   run() {
      synchronized(...) { // CS1
         G.Y = G.X;
      }
      synchronized(...) { // CS2
         G.X = G.X + 1;
      }
   }
}

class T2 extends Runnable {
   run() {
      synchronized(...) { // CS3
         int a = G.Y;
      }
   }
}
```

Initial Approximation

Interference Identification
- Thread-Local Objects Analysis
- Thread-Based Side Effect Analysis

Interference Pruning
- May Happen in Parallel Analysis

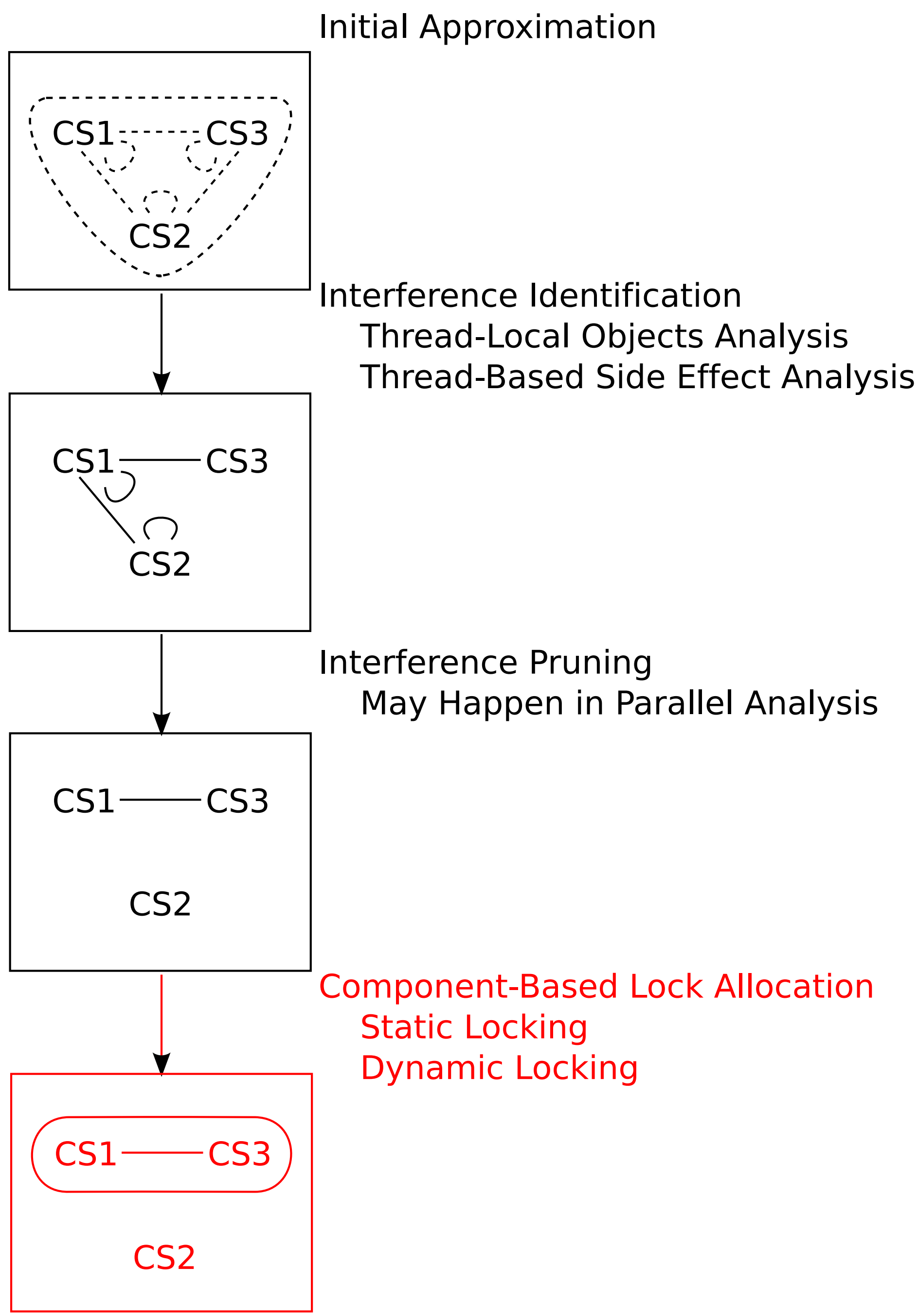
Component-Based Lock Allocation
- Static Locking
- Dynamic Locking



Component-Based Lock Allocation

Static Lock Allocation:
(Dynamic is the same in this case)

```
class G {
    public static int X, Y;
}

class T1 extends Runnable {
    run() {
        synchronized(LockObject1) { // CS1
            G.Y = G.X;
        }
        synchronized(...) { // CS2
            G.X = G.X + 1;
        }
    }
}

class T2 extends Runnable {
    run() {
        synchronized(LockObject1) { // CS3
            int a = G.Y;
        }
    }
    public static Object LockObject1 =
        new Object();
}
```



Finding Thread-Based Side Effects

Build on an existing side-effect analysis

- Identify fields that are read & written
- Each field has a points-to set of possible base objects

Extend it to be thread-sensitive

- Approximate the thread-visible effects of library calls
- Exclude thread-local side effects

Use it to construct a critical section *interference graph*

Constructing an Interference Graph

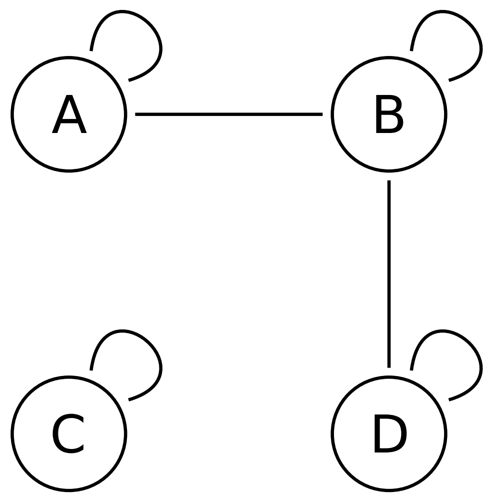
```
class A {
  public static int f;

  synchronized void a() {
    A.f = B.f + 1;
  }
}
```

```
class B {
  public static int f;

  synchronized void b() {
    B.f = B.f + D.f;
  }
}
```

```
class C {
  public static int f;

  synchronized void c() {
    C.f = C.f + 1;
  }
}
```

```
class D {
  public static int f;

  synchronized void d() {
    D.f = D.f + 1;
  }
}
```

Constructing an Interference Graph



Interference Graph

	A	B	C	D
A	1	1	0	0
B	1	1	0	1
C	0	0	1	0
D	0	1	0	1

Finding Thread-Local Objects

Thread-local object: object only read & written by a single thread

Similar to escape analysis

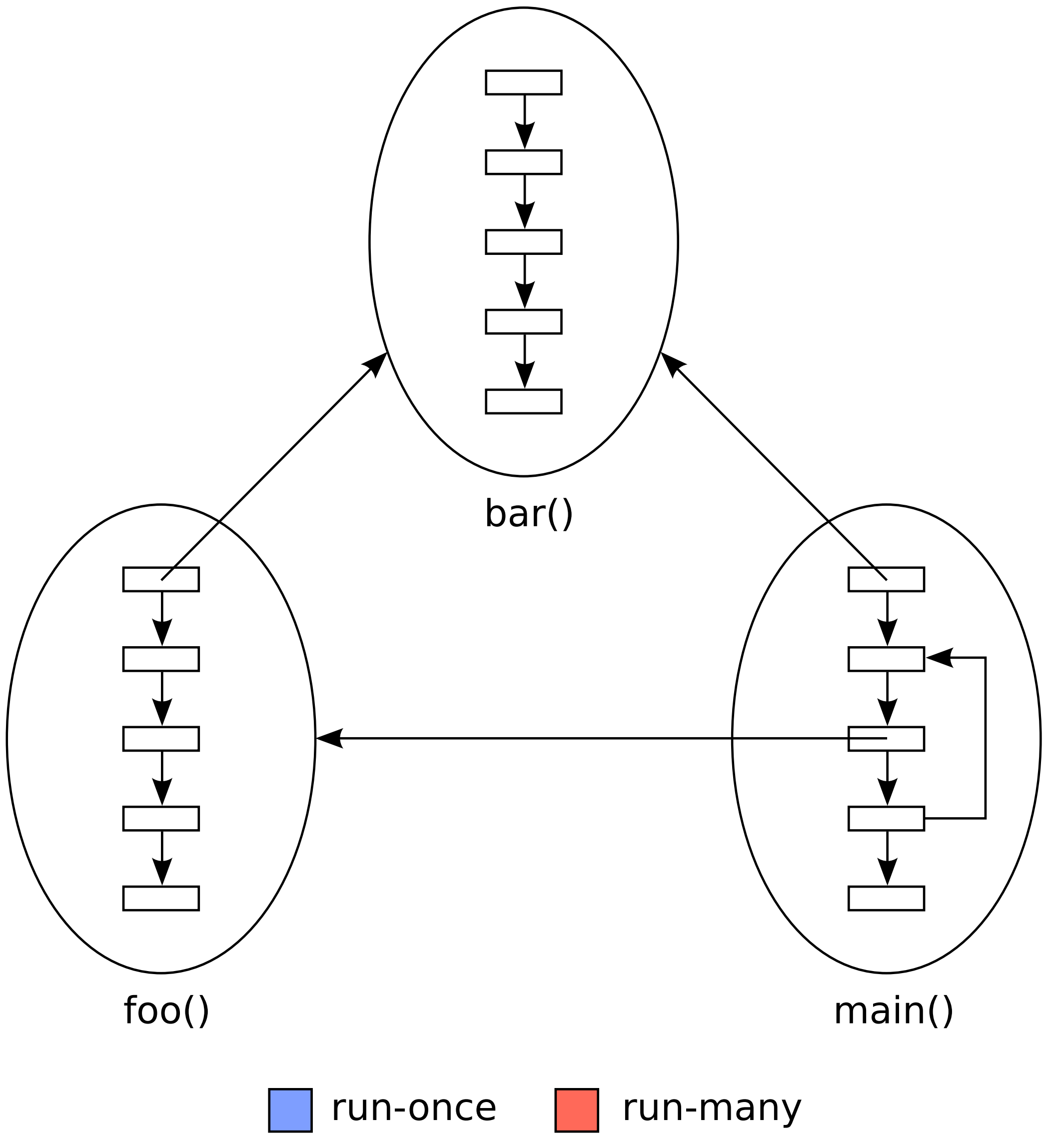
- Partition the heap into thread-shared and thread-local data
- Use information flow analysis to propagate thread-shared status

Values identified as thread-local do not require synchronized access

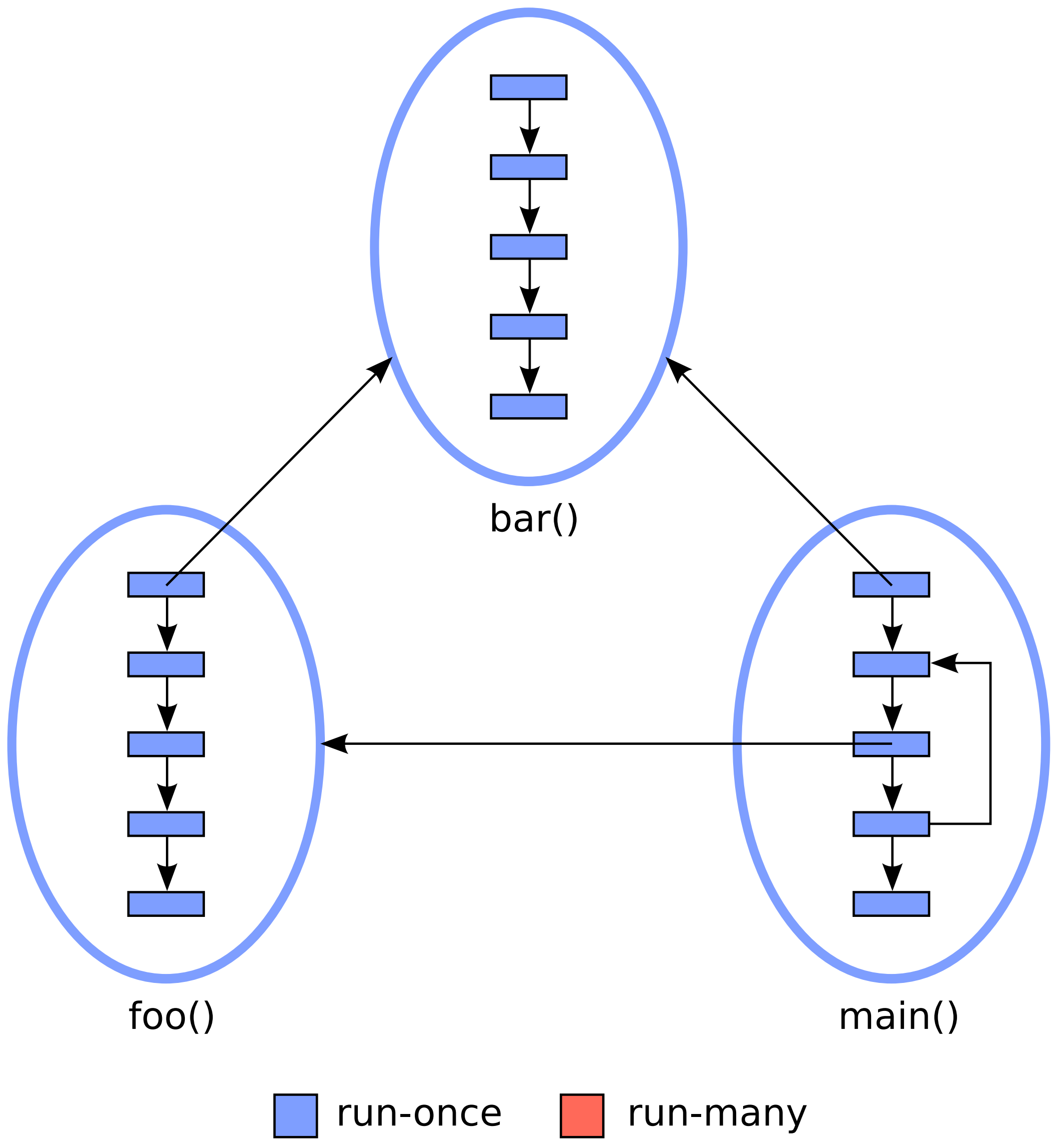
May Happen in Parallel Analysis

- MHP analysis finds methods that execute concurrently
- Several distinct steps:
  1. Identify run-once and run-many statements
  2. Identify run-once and run-many threads
  3. Categorize run-many threads as run-one-at-a-time or run-many-at-time
  4. Find methods that may happen in parallel based on thread reachability
- Critical sections that may not happen in parallel cannot interfere!

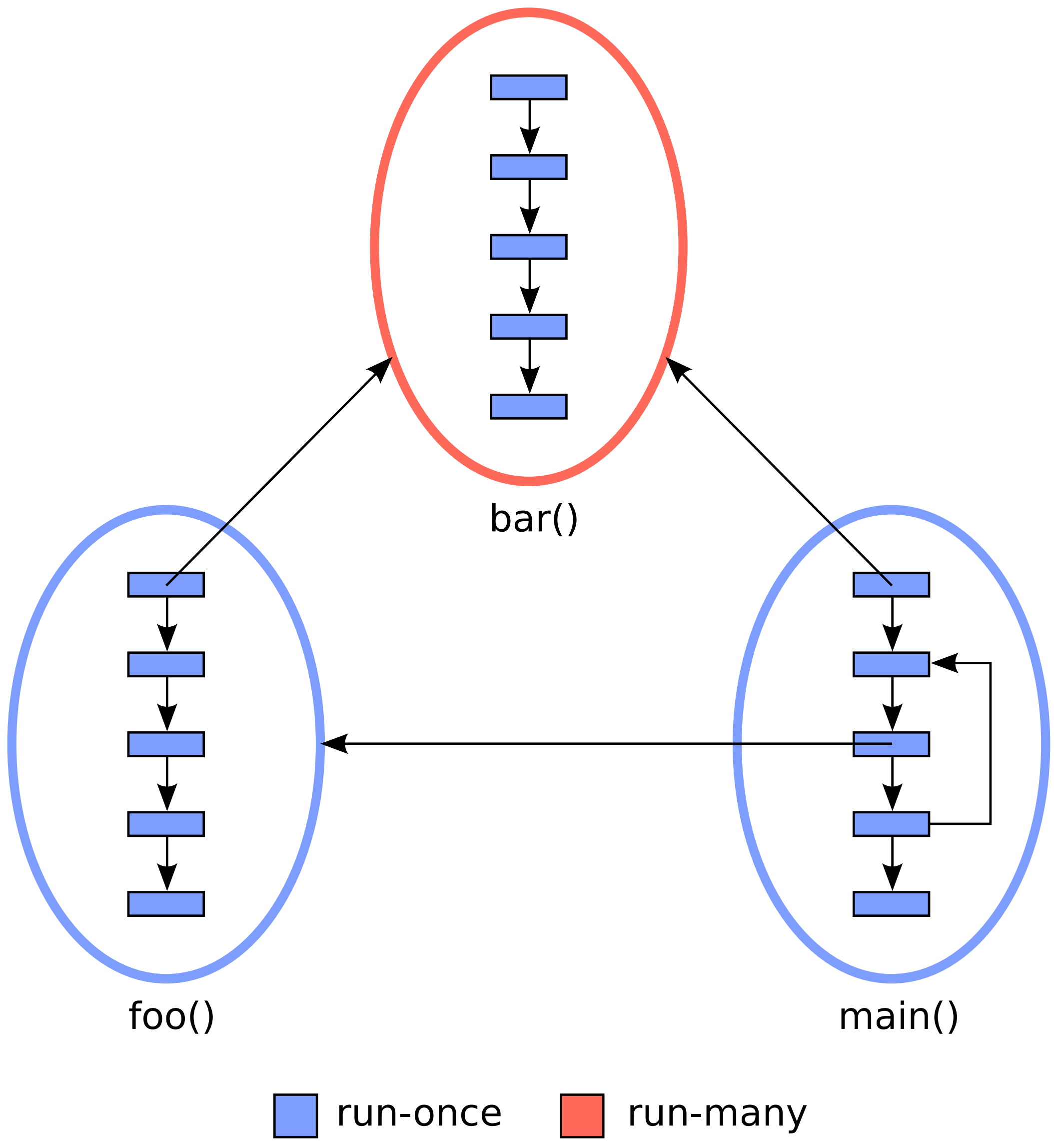
Run-Once Run-Many Analysis



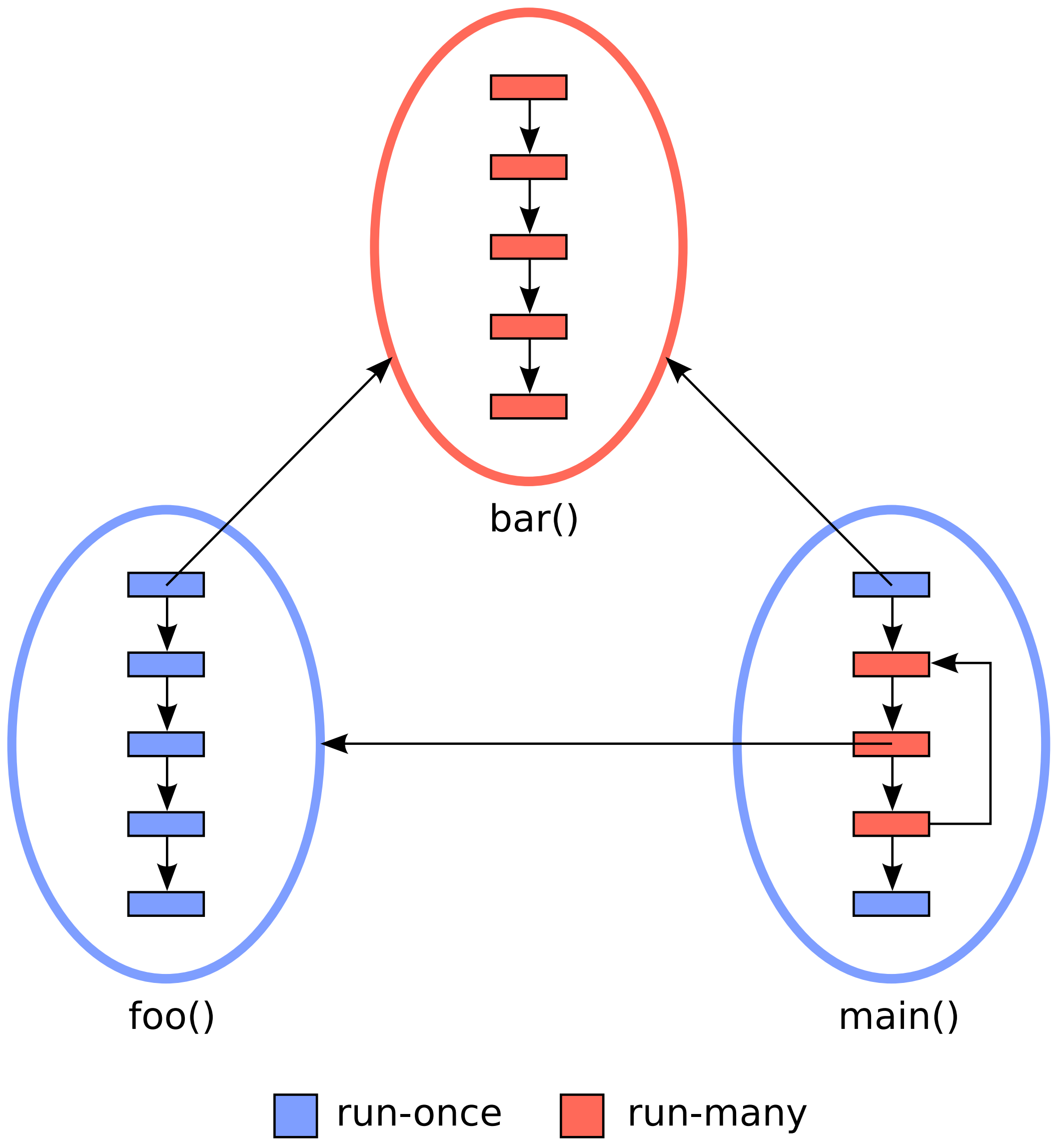
Run-Once Run-Many Analysis



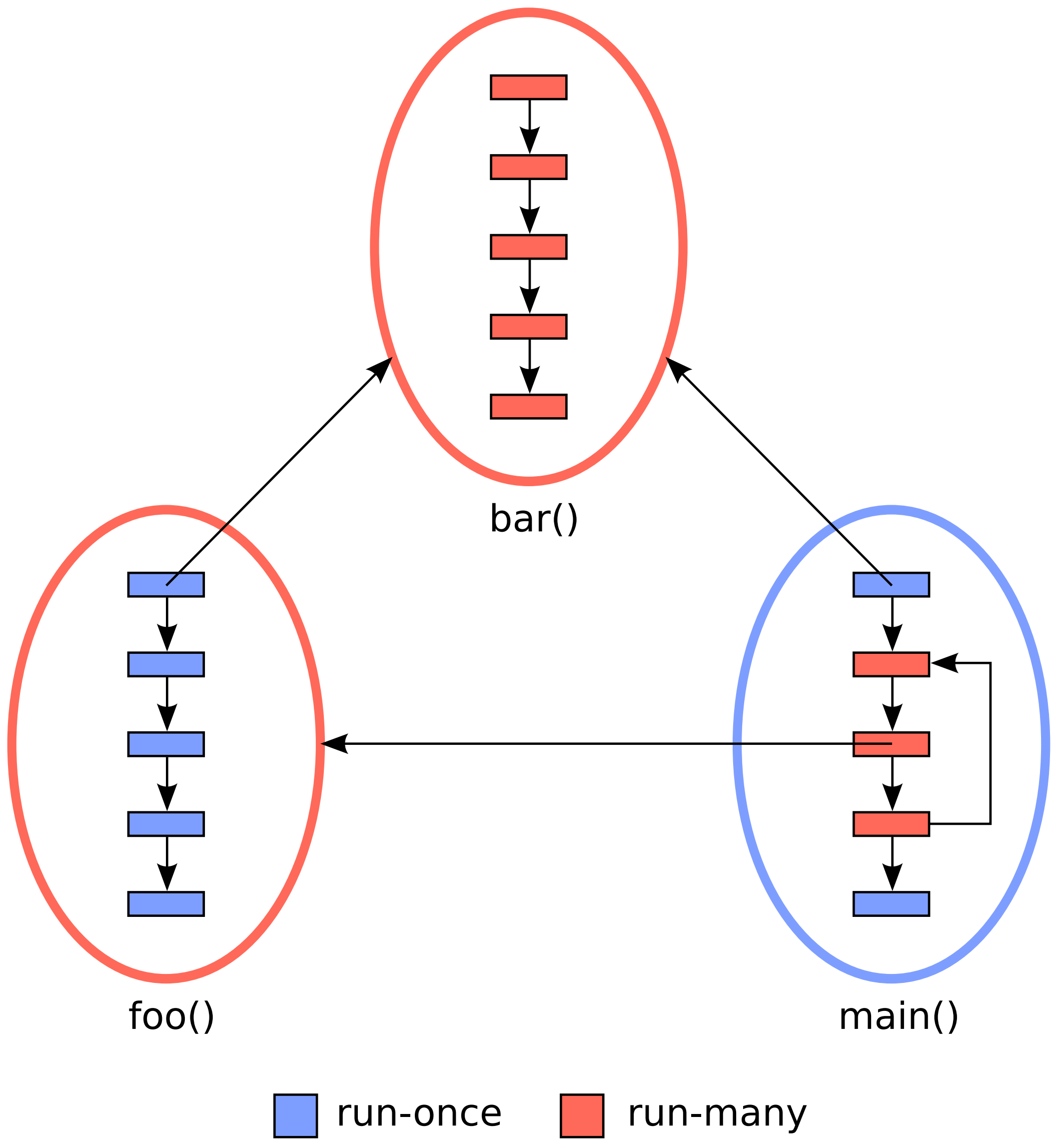
Run-Once Run-Many Analysis



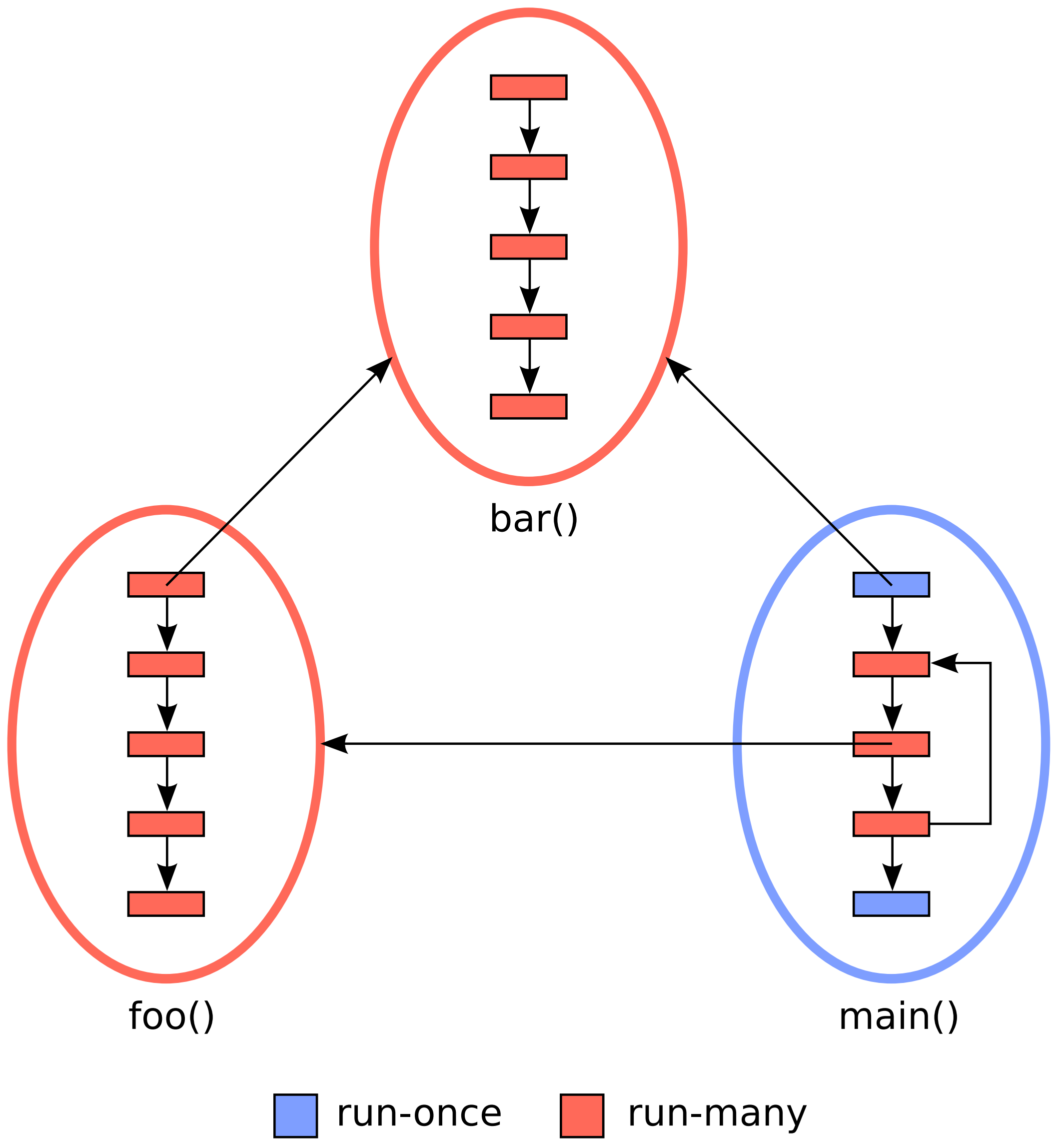
Run-Once Run-Many Analysis



Run-Once Run-Many Analysis



Run-Once Run-Many Analysis



Thread Categorization

Thread t1, t2, t3

int i

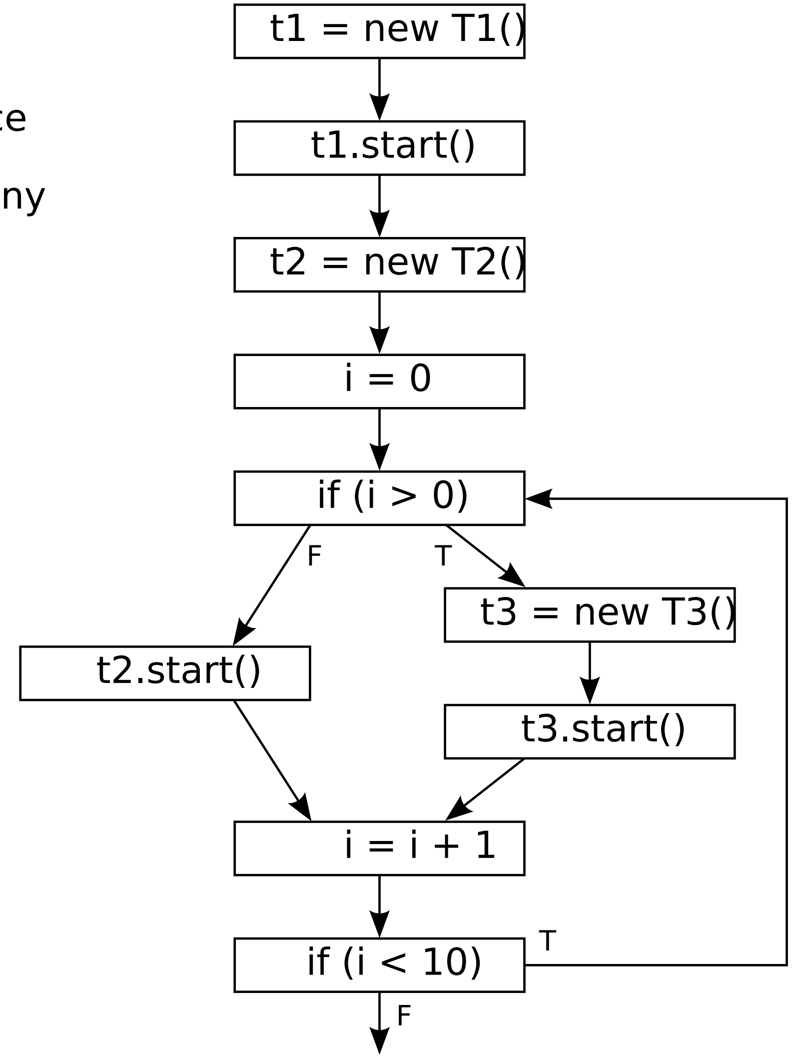
run-once

run-many

T1: ?

T2: ?

T3: ?



Thread Categorization

Thread t1, t2, t3

int i

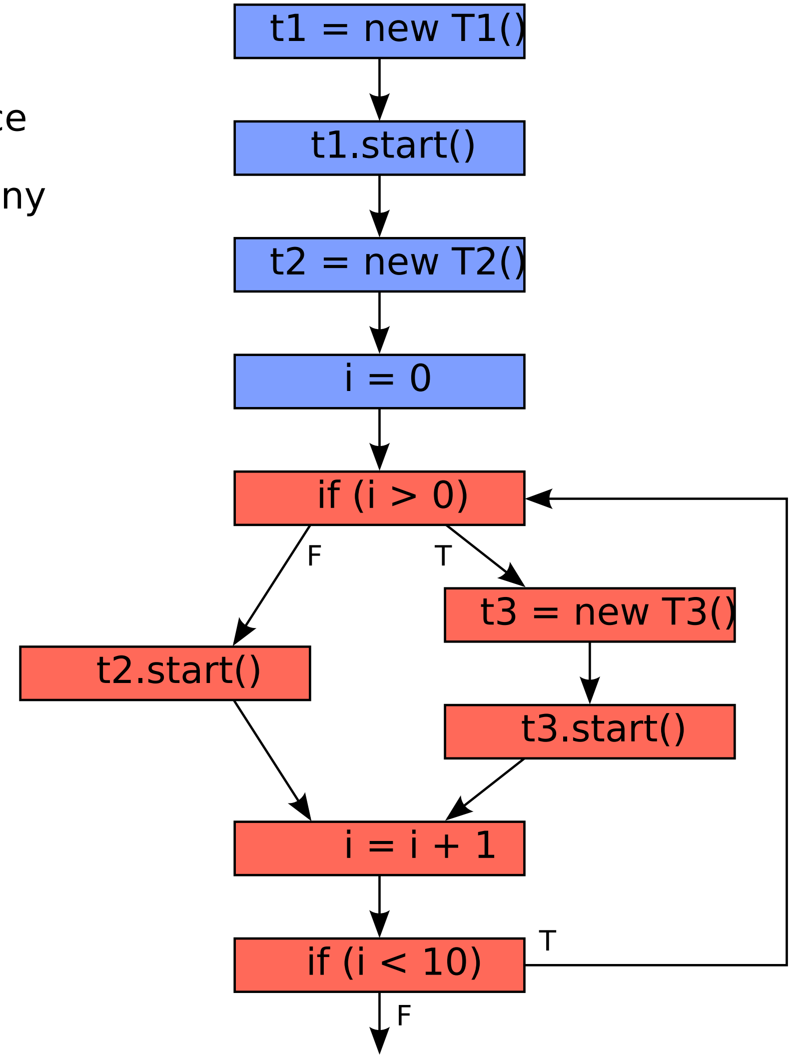
run-once

run-many

T1: ?

T2: ?

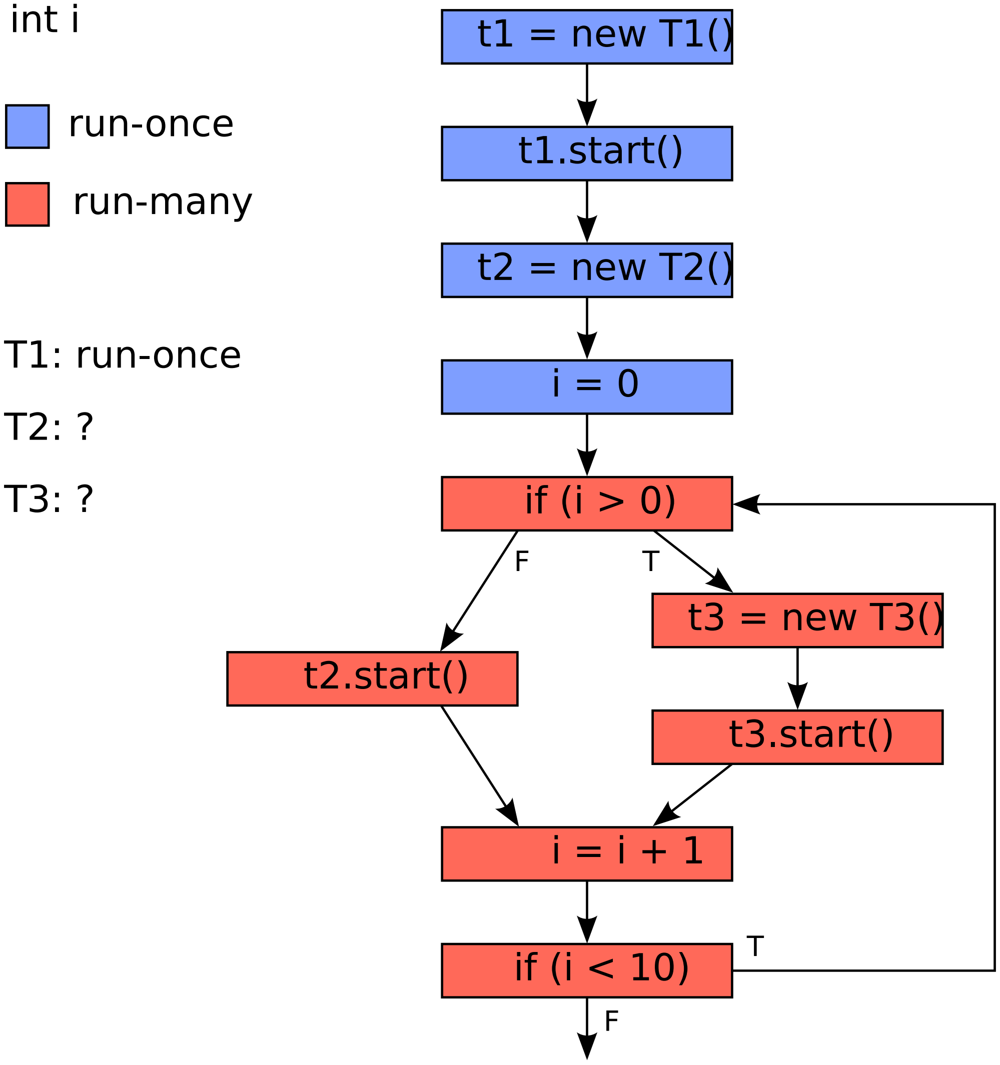
T3: ?



Thread Categorization

Thread t1, t2, t3

int i



T1: run-once

T2: ?

T3: ?

Thread Categorization

Thread t1, t2, t3

int i

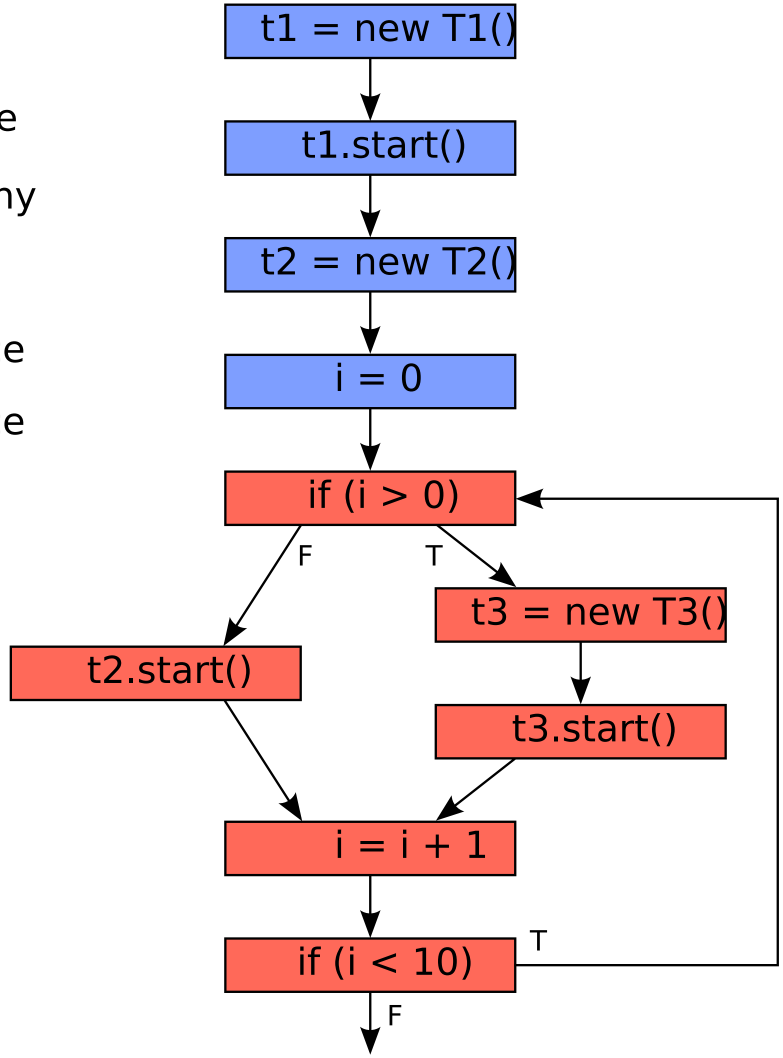
run-once

run-many

T1: run-once

T2: run-once

T3: ?



Thread Categorization

Thread t1, t2, t3

int i

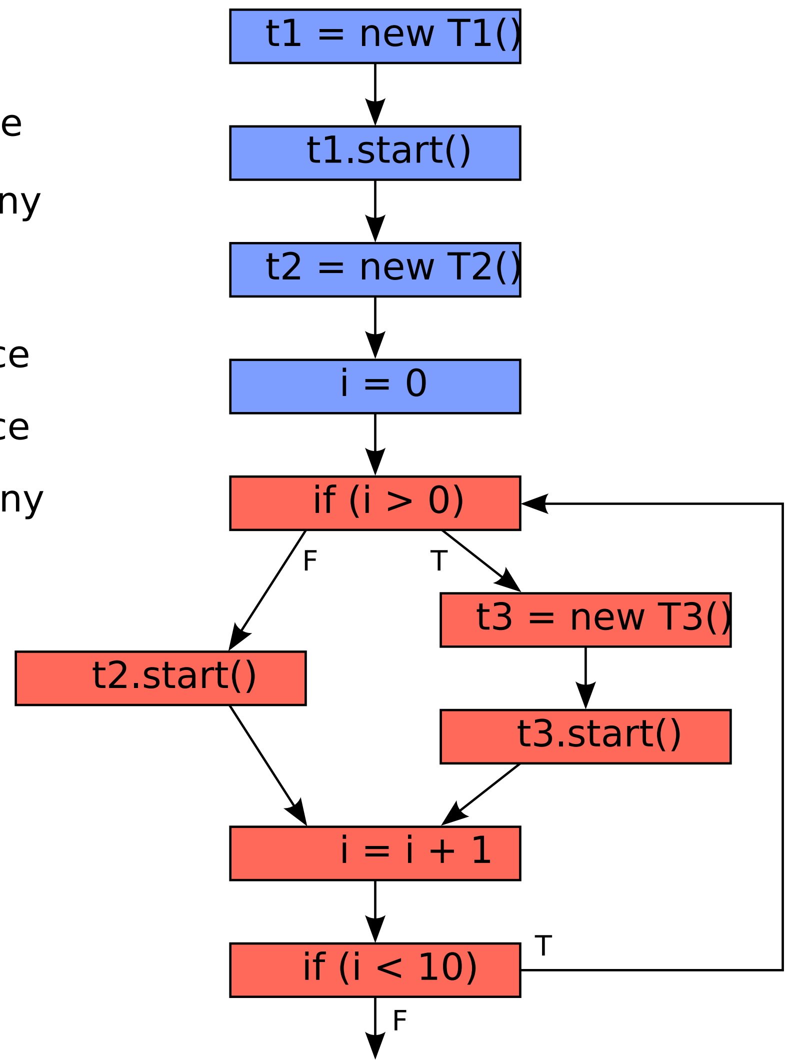
run-once

run-many

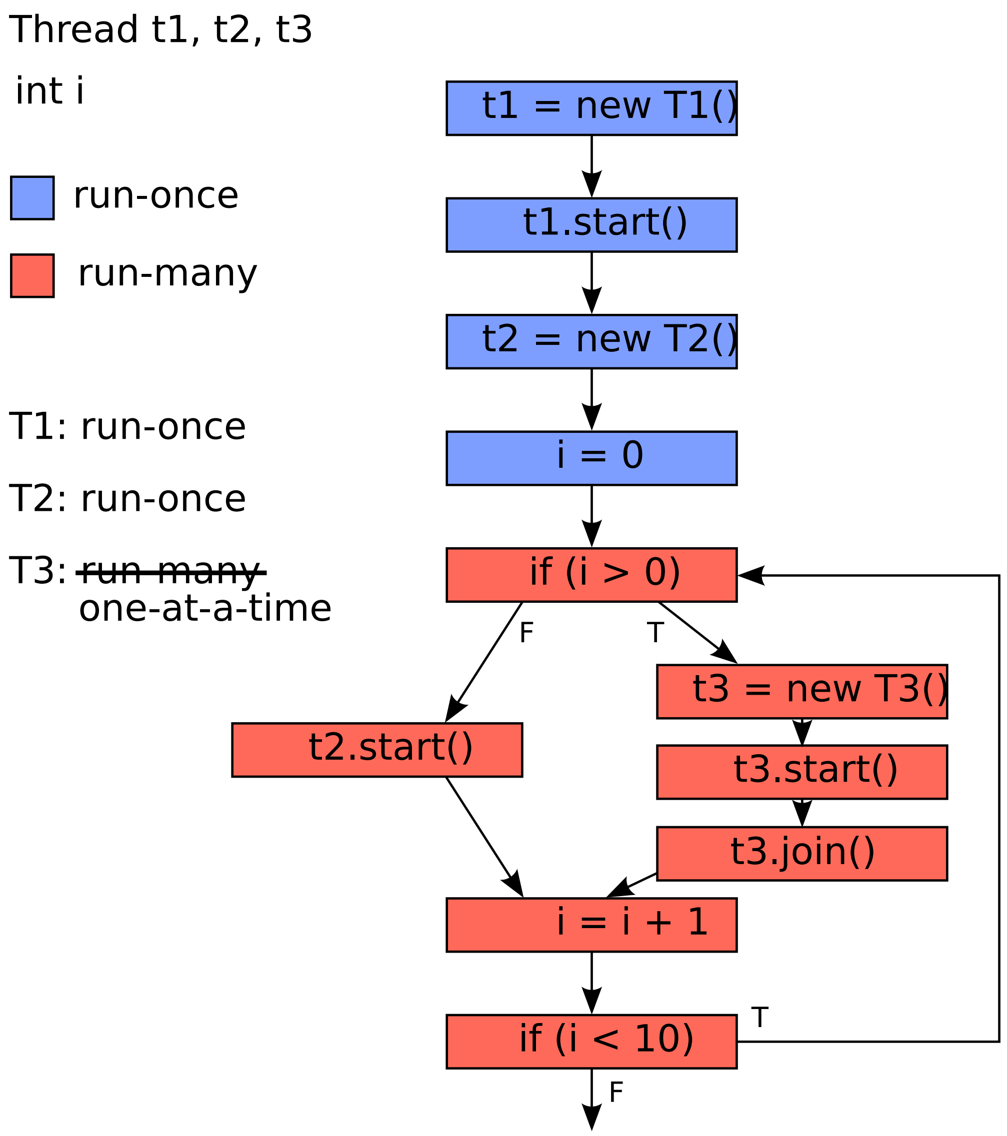
T1: run-once

T2: run-once

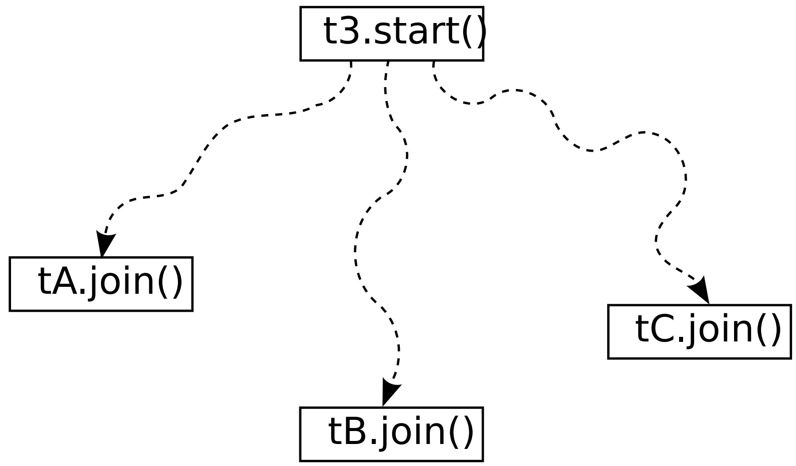
T3: run-many



Finding run-one-at-a-time threads

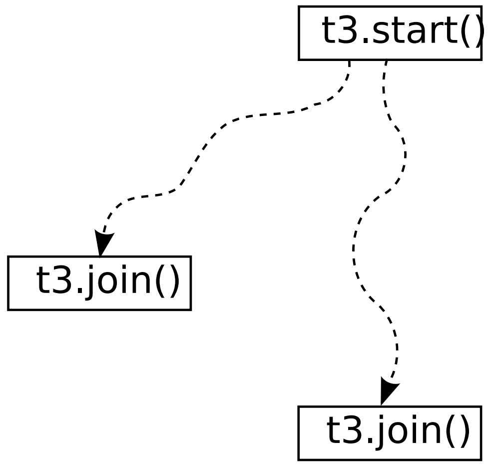


Finding run-one-at-a-time threads



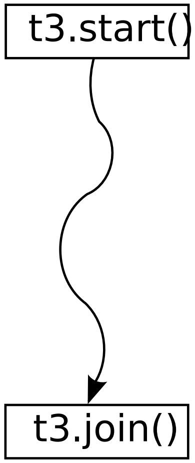
- For each start, consider all joins:

Finding run-one-at-a-time threads



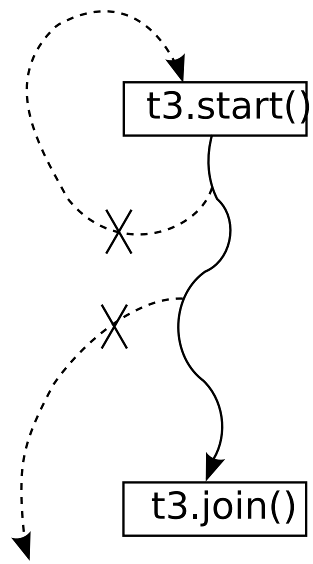
- For each start, consider all joins:
    - Any valid join receiver must alias start receiver

Finding run-one-at-a-time threads



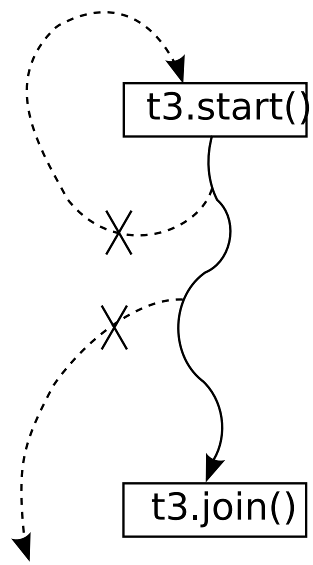
- For each start, consider all joins:
  - Any valid join receiver must alias start receiver
  - Any valid join must post-dominate start

Finding run-one-at-a-time threads



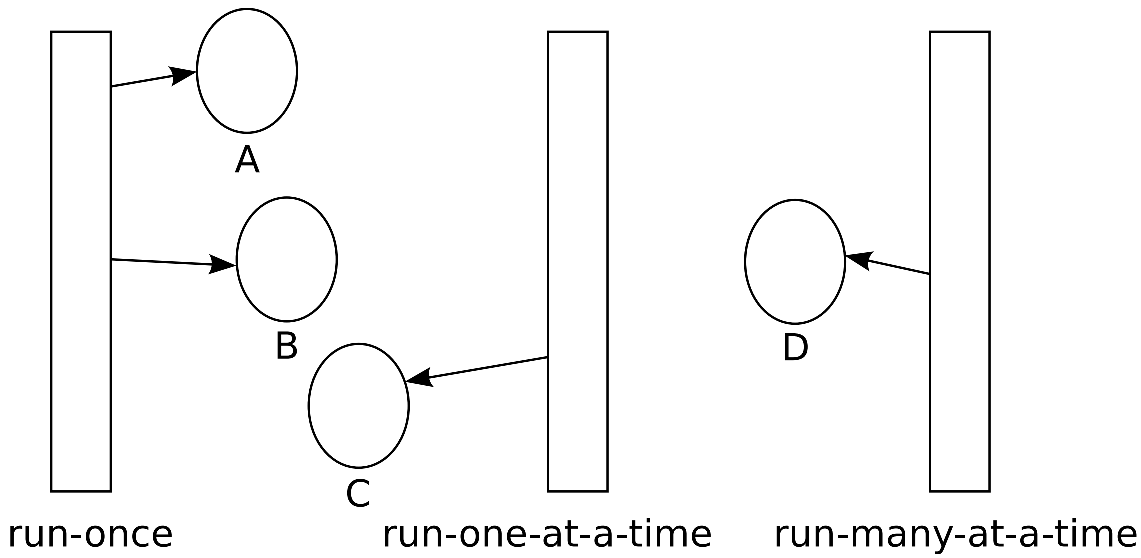
- For each start, consider all joins:
  - Any valid join receiver must alias start receiver
  - Any valid join must post-dominate start
  - And not have loops to start between the start and join...

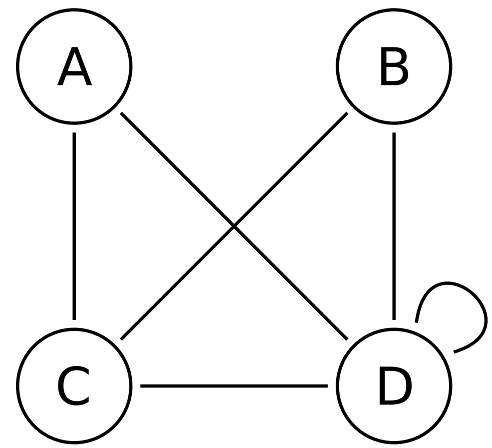
Finding run-one-at-a-time threads



- For each start, consider all joins:
  - Any valid join receiver must alias start receiver
  - Any valid join must post-dominate start
  - And not have loops to start between the start and join...
- If join is valid, check method validity:
  - Method must not be called recursively
  - Method must not happen in parallel with itself

Finding MHP Information

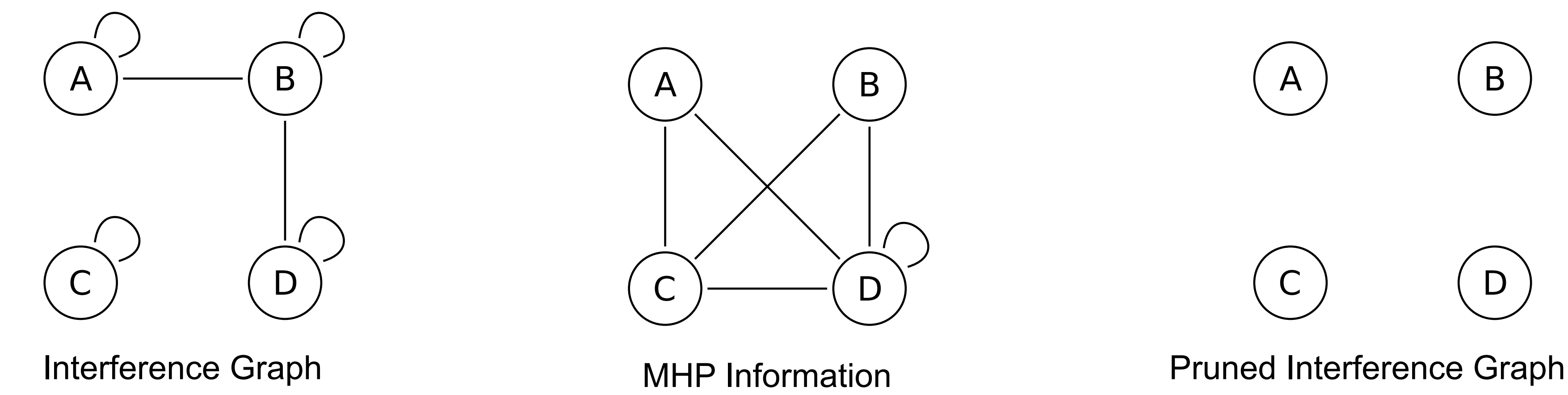




MHP Information

	A	B	C	D
A	0	0	1	1
B	0	0	1	1
C	1	1	0	1
D	1	1	1	1

Applying MHP Information



	A	B	C	D
A	1	1	0	0
B	1	1	0	1
C	0	0	1	0
D	0	1	0	1

•

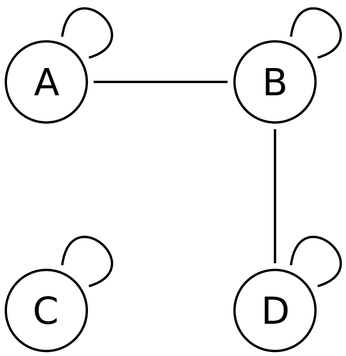
	A	B	C	D
A	0	0	1	1
B	0	0	1	1
C	1	1	0	1
D	1	1	1	1

=

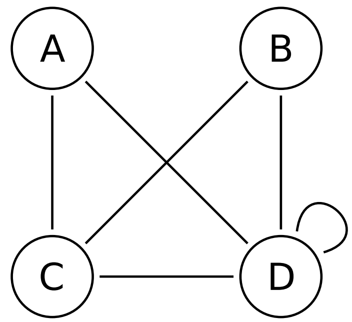
	A	B	C	D
A				
B				
C				
D				

A simple Hadamard product

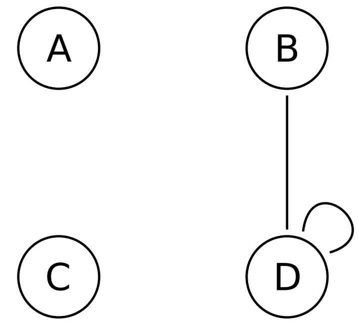
Applying MHP Information



Interference Graph



MHP Information



Pruned Interference Graph

	A	B	C	D
A	1	1	0	0
B	1	1	0	1
C	0	0	1	0
D	0	1	0	1

$\bullet$

	A	B	C	D
A	0	0	1	1
B	0	0	1	1
C	1	1	0	1
D	1	1	1	1

$=$

	A	B	C	D
A	0	0	0	0
B	0	0	0	1
C	0	0	0	0
D	0	1	0	1

A simple Hadamard product

Component-Based Lock Allocation

Three kinds of component-based lock allocation:

1. Singleton: a single static lock protects all components
2. Static: one static lock per component
3. Dynamic: attempt to use per-data structure locks for each component, otherwise static

Finally, isolated vertices with no self loops are *unlocked*

Outline

Experimental Setup

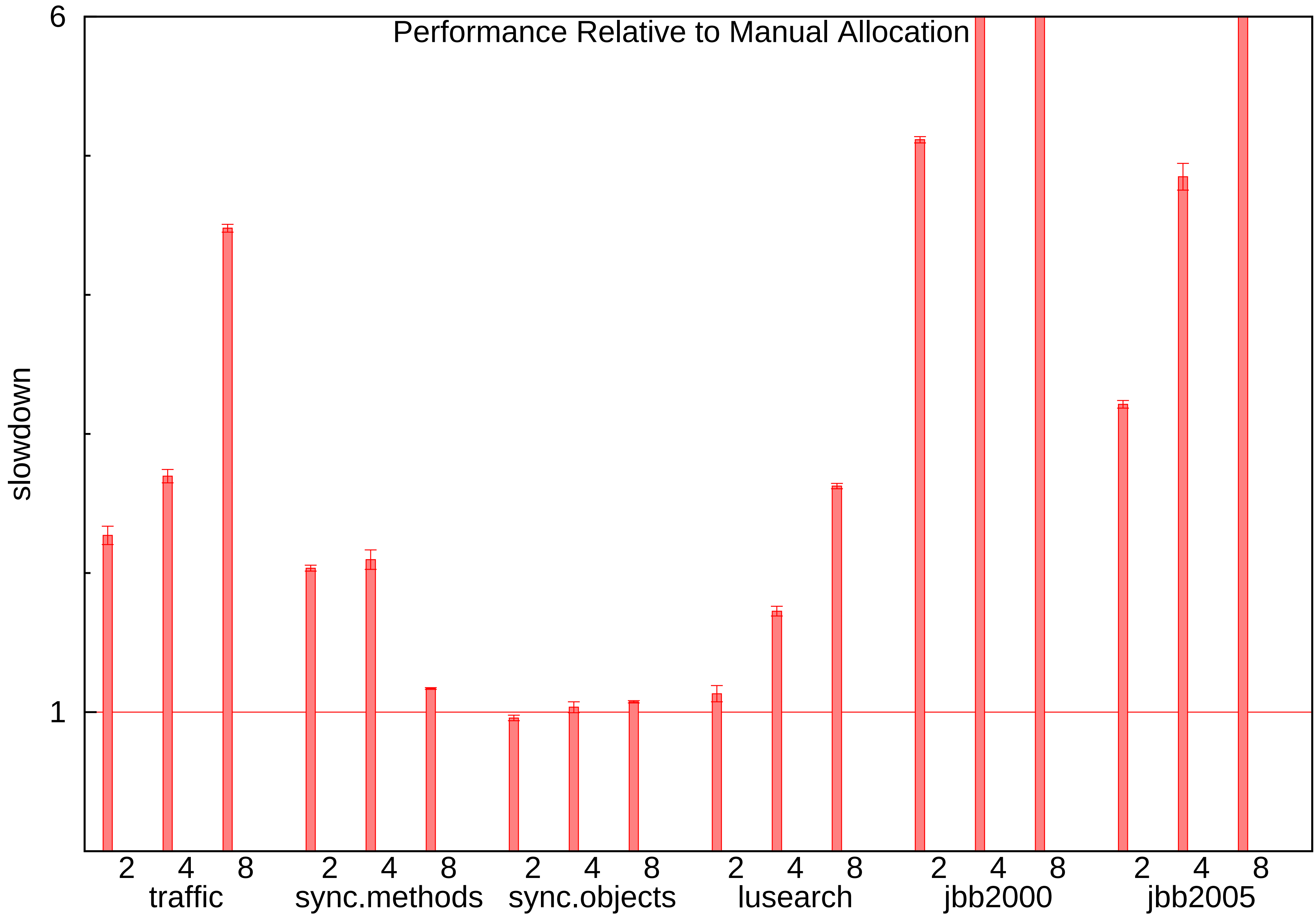
For each benchmark, we do 13 experiments:

- control: original benchmark program
- singleton: single static lock for all critical sections
- 5 static locking allocations:
  1. CHA: class hierarchy analysis points-to and side effects
  2. Spark: context-insensitive points-to and side effects
  3. Spark-MHP: Spark with may happen in parallel [MHP] analysis
  4. Spark-TLO-MHP: Spark with both TLO and MHP
- 5 analogous dynamic locking allocations

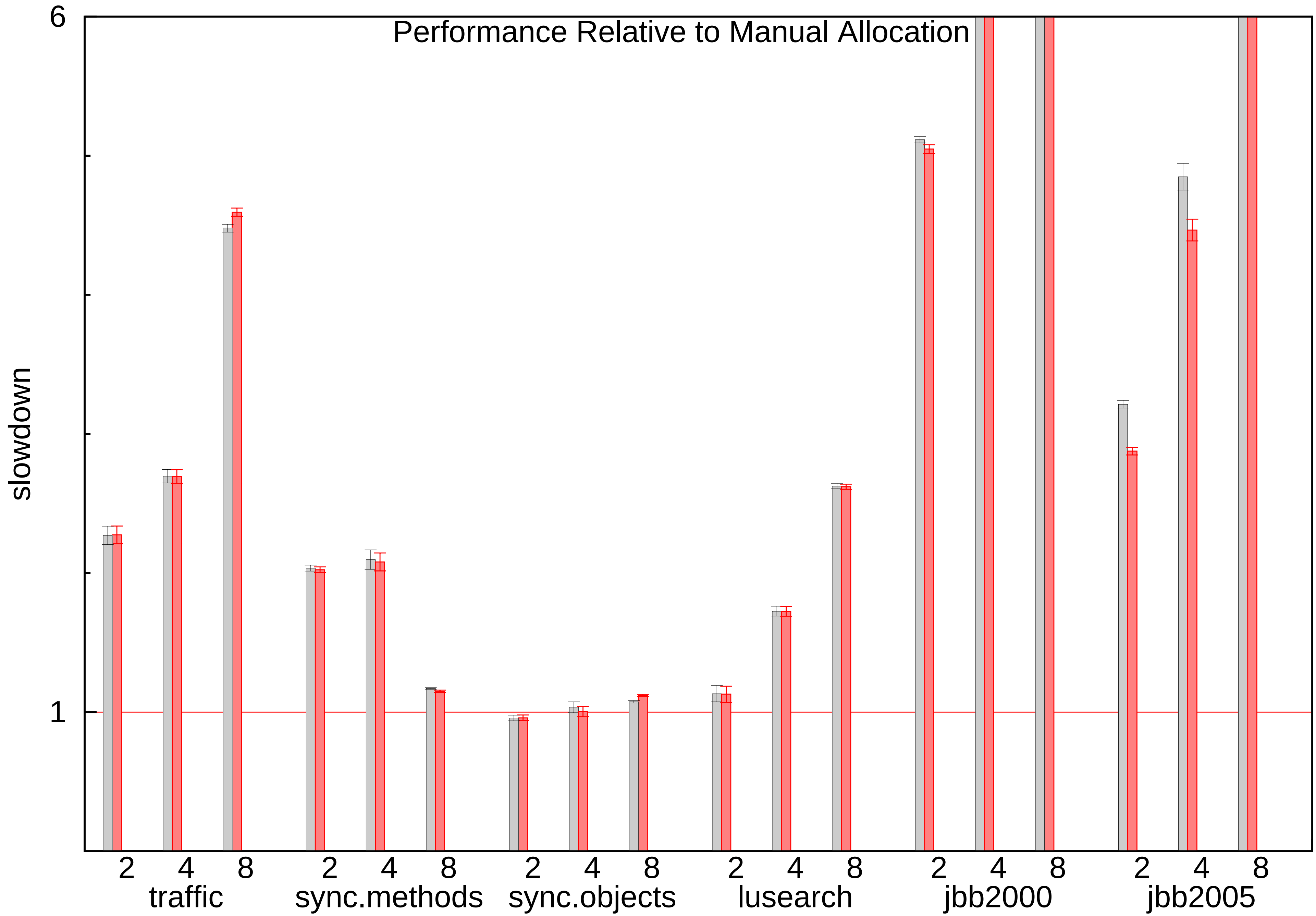
11 benchmarks: 5 micro, 6 standard

64-bit AMD Machines (dual, 4-way, 4-way dual), Sun JDK1.5

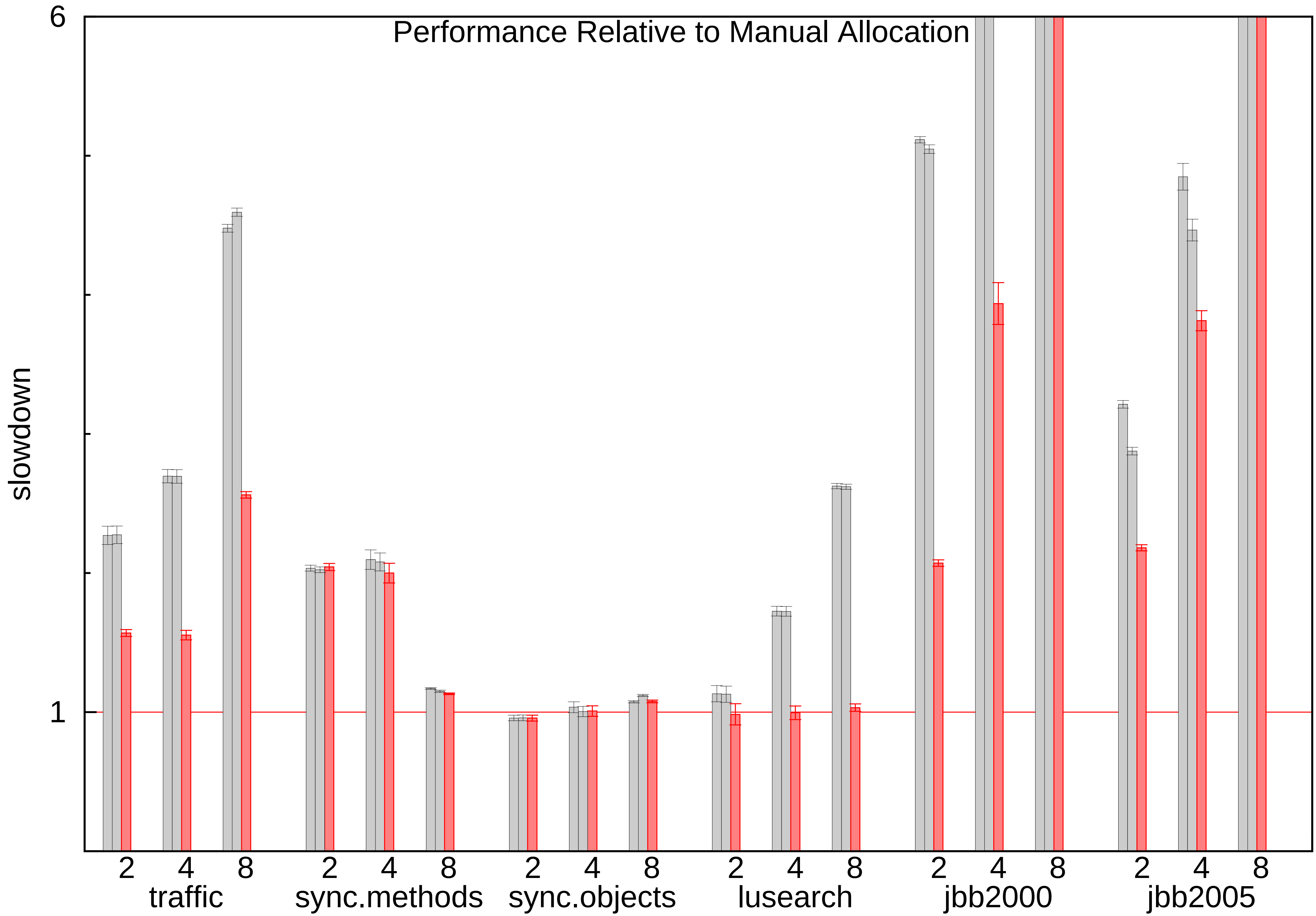
Singleton Lock Slowdown



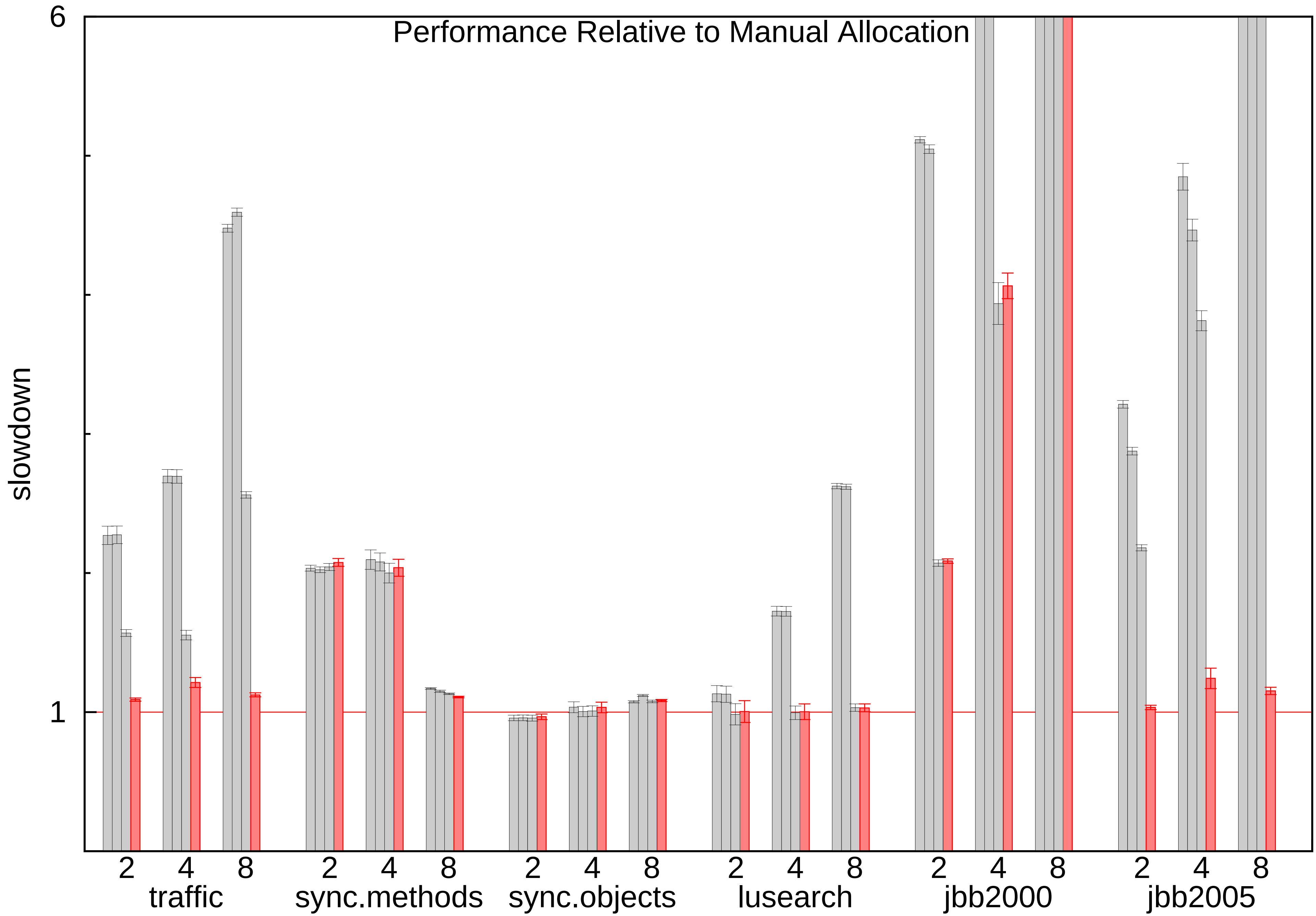
Relative Speedup of Using CHA



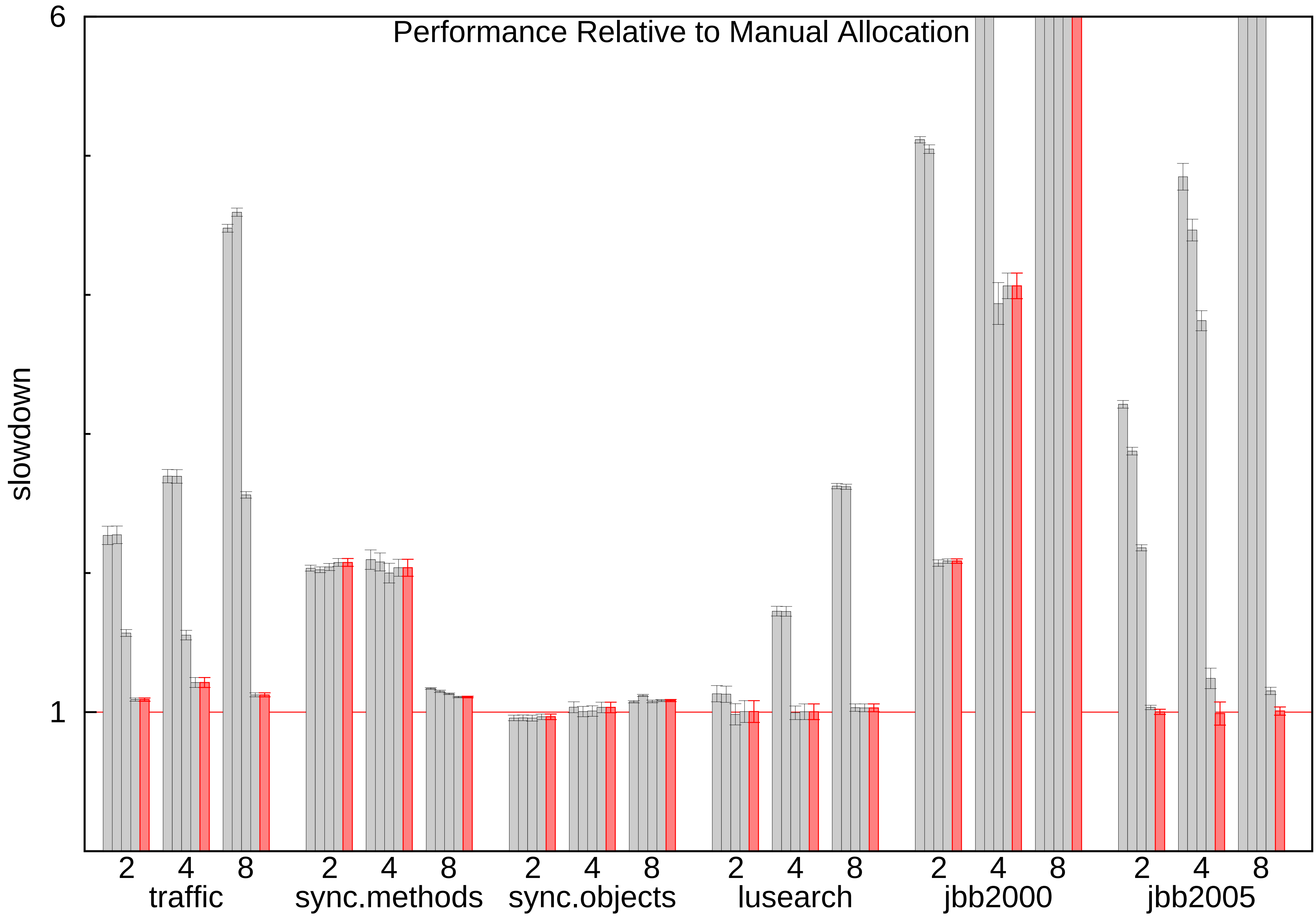
Relative Speedup of Using Spark



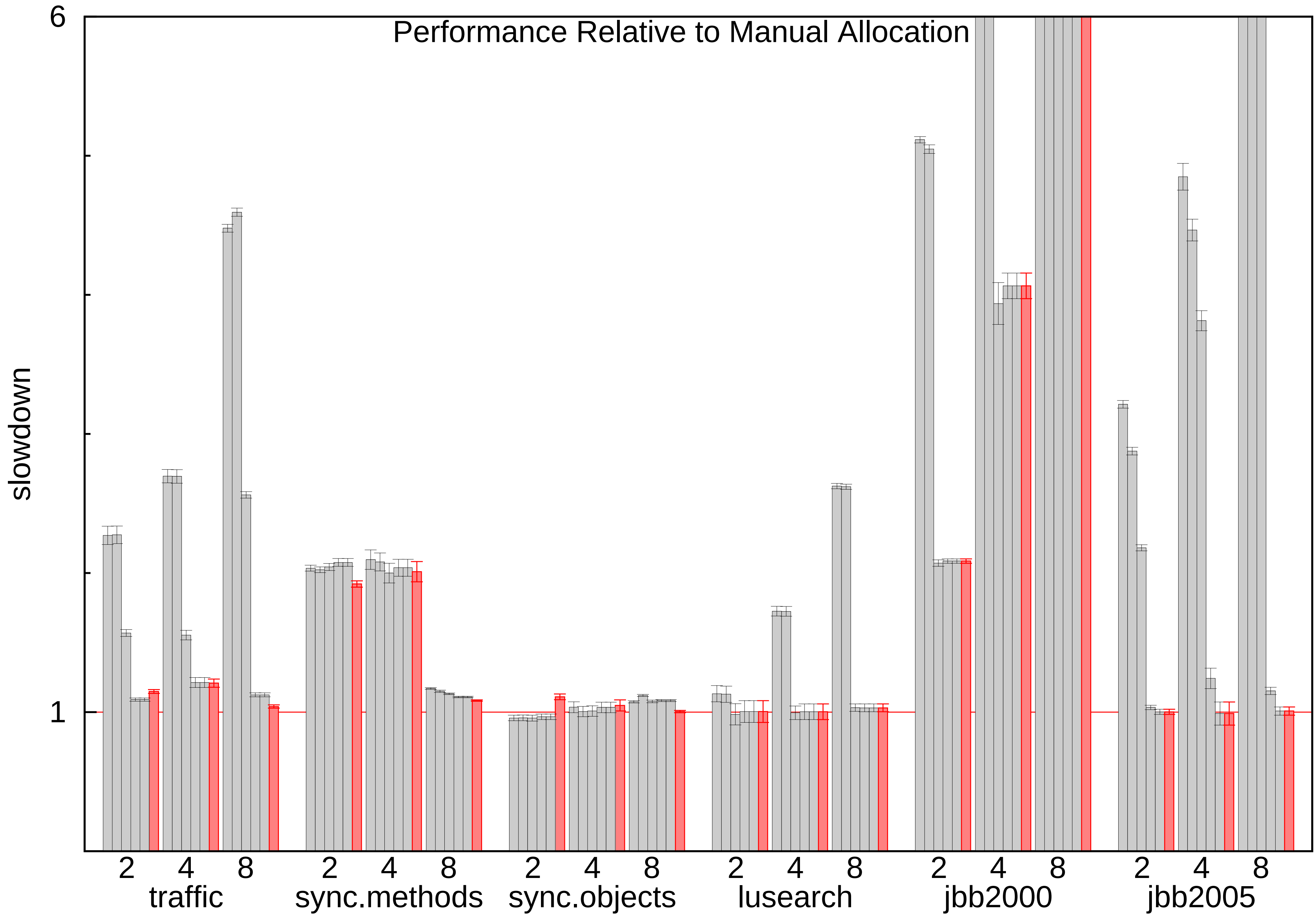
Relative Speedup of Adding MHP Analysis



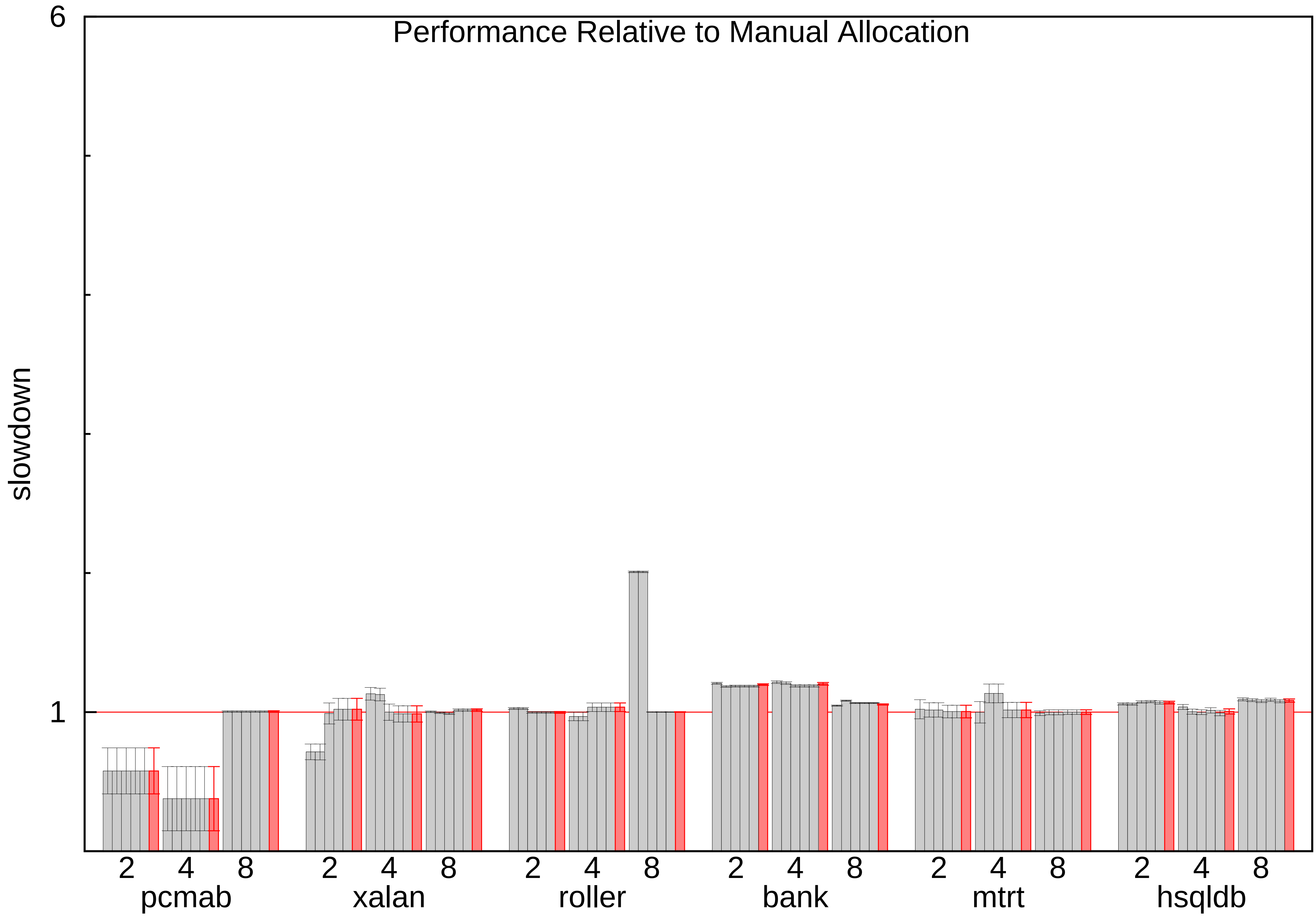
Relative Speedup of Adding TLO Analysis



Relative Speedup of Using Dynamic Locking



Relative Speedup of Using Dynamic Locking



Outline

Conclusions

- Singleton allocation is not generally viable
- Points-to analysis precision is important
- MHP analysis helps if it can split a larger component
- TLO analysis usually has a negligible effect
- Dynamic locking has a small impact; may degrade or improve performance
- Component-based allocation works surprisingly well for many benchmarks

Future Work

- More precise compiler analyses
- Finer locking granularities
- Method synchronization
- Critical section inference
- Speculative locking and transactional memory

Questions?

Thank you for your attention.

Related Work

- May Happen in Parallel analysis for Java (Naumovich *et al.* '99, Li '04).
- Thread-sensitive points-to and escape analysis (Chang and Choi '04, Sălcianu and Rinard '01).
- Thread-local objects analysis for synchronization elimination (Ruf '00).
- Pessimistic atomic sections/transactions (McCloskey *et al.* '06, Hicks *et al.* '06).
- Lock allocation
  - Concurrency graph (Sreedhar, Zhang, *et al.* '05).
  - ILP-based optimal allocations (Sreedhar, Zhang, *et al.* '05, Emmi *et al.* '07).
- Static race detection (Naik *et al.*'06, and *many others*).
- Optimistic concurrency, transactional memory (*see Larus & Rajwar '06*).