Lock Inference for Java

Khilan Gudka
Imperial College London

Supervised by
Professor Susan Eisenbach, Imperial College London
Professor Sophia Drossopoulou, Imperial College London

This work was generously funded by Microsoft Research Cambridge

Concurrency control
Status quo: we use locks

- But there are problems with them
  - Not composable
  - Break modularity
  - Deadlock
  - Priority inversion
  - Convoying
  - Starvation
  - Hard to change granularity (and maintain in general)
- We want to eliminate the lock abstraction but is there a better alternative?

Atomic sections

- What programmers probably can do is tell which parts of their program should not involve interferences
- **Atomic sections**
  - Declarative concurrency control
  - Move responsibility for figuring out what to do to the compiler/runtime

```
atomic {
    x.f++;
    y.f++;
}
```

Atomic sections

- Simple semantics (no interference allowed)
- Naïve implementation: one global lock
- But we still want to allow parallelism without:
  - Interference
  - Deadlock
- Optimistic vs. Pessimistic implementations

Implementing Atomic Sections: Optimistic = transactional memory

- Advantages
  - None of the problems associated with locks
  - More concurrency
- Disadvantages
  - Irreversible operations (IO, System calls)
  - Runtime overhead
- Much interest

Implementing Atomic Sections: Pessimistic = lock inference

- Statically infer and instrument the locks that are needed to protect shared accesses

```
atomic {
   x.f++;
   y.f++;
}
```

compiled to →

```
lock(x);
lock(y);
   x.f++;
   y.f++;
unlock(y);
unlock(x);
```

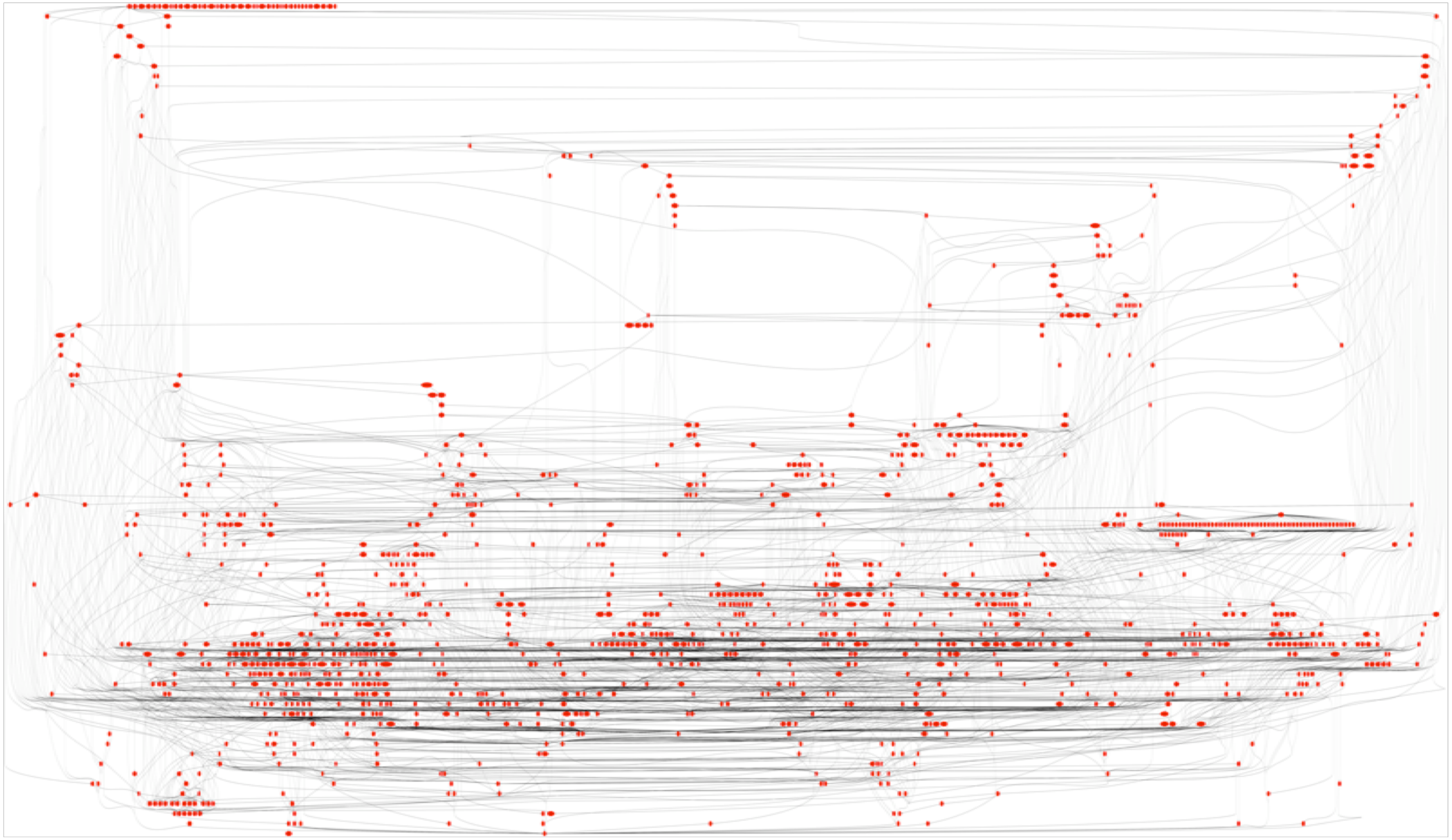
- Acquire locks in two-phased order for atomicity
- **Can handle irreversible operations!**

Motivation:
A “Simple” I/O Example

```
atomic {
  System.out.println(“Hello World!”);
}
```

Motivation: A “Simple” I/O Example

- Callgraph:



Motivation:
A "Simple" I/O Example

- Cannot find in the literature any lock inference analysis which can handle this!
  - Ignore it due to the imprecision and resulting performance

- General goals/challenges of lock inference
  - Maximise concurrency
  - Minimise locking overhead
  - Avoid deadlock

Prior work

- Achieve all of the above **in the presence of libraries**. Challenges that libraries introduce:
  - Scalability (many and long call chains)
  - Imprecision (have to consider all library execution paths)

This work

Thesis

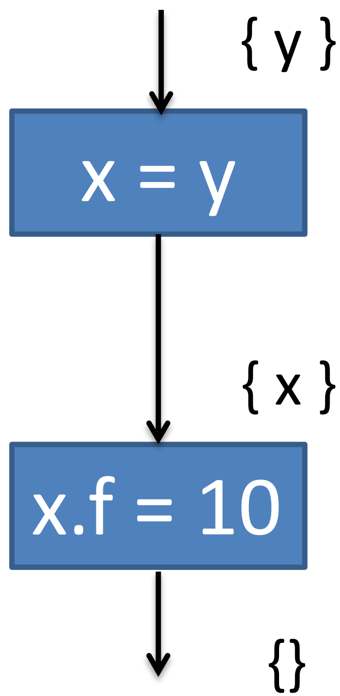
We argue:

“It is possible to develop lock inference techniques that scale to real-world Java programs that make use of the library and still obtain performance comparable to hand-crafted locking.”

Our lock inference analysis: Infer fine-grained locks

- Infer path expressions at each program point:

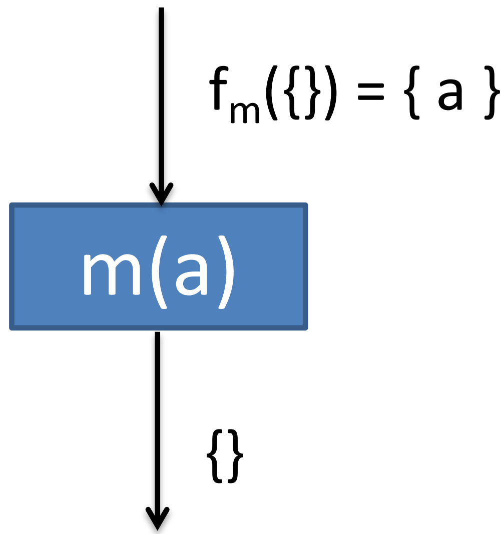
```
Obj x = …;
Obj y = …;
atomic {
   x = y;
   x.f++;
}
```




```
Obj x = …;
Obj y = …;
lock(y);
   x = y;
   x.f++;
unlock(y);
```

Scaling by computing summaries

```
void m(Obj p) {
   p.f = 1;
}
```



f_m is m’s summary function

Summaries can get large: challenge is to find a representation of transfer functions that allows fast composition and meet operations

Implementation

- We implemented our approach in the SOOT framework
- Evaluated using standard benchmarks for atomicity (that do not perform system calls).

Name	#Threads	#Atomics	#client methods	#lib methods	LOC (client)
sync	8	2	0	0	1177
pcmab	50	2	2	15	457
bank	8	8	6	7	269
traffic	2	24	4	63	2128
mtrt	2	6	67	1324	11312
hsqldb	20	240	2107	2955	301971

Analysis times

- Experimental machine:

  256-core Xeon E7-8837 2.67Ghz, 3TB RAM, SUSE Linux Enterprise Server, Oracle Java 6

- Java options:

  Min & Max heap: 70GB, Stack: 128MB

Name	Paths	Locks	Total
sync	0.122s	0.14s	5m 31s
pcmab	0.246s	0.092s	5m 15s
bank	0.247s	0.129s	5m 27s
traffic	1.695s	0.2s	5m 40s
mtrt	1h 30m	8.579s	1h 36m
hsqldb	?	?	?

Simple analysis not enough

- Our analysis still wasn't efficient enough to analyse hsqldb.
- We performed further optimisations to reduce space-time:
  - **Primitives for state**
    Encode analysis state as sets of longs for efficiency. All subsequent optimisations assume this
  - **Parallel propagation**
    - Perform intra-procedural propagation in parallel for different methods
    - Perform inter-procedural propagation in parallel for different call-sites
  - **Summarising CFGs**
    - Merging CFG nodes to reduce the amount of storage space and propagation
  - **Worklist Ordering**
    - Ordering the worklist so that successor nodes are processed before predecessor nodes. This helps reduce redundant propagation
  - **Deltas**
    - Only propagate new dataflow information
    - Reduces the amount of redundant work

Analysis times

- Experimental machine for hsqldb:

  256-core Xeon E7-8837 2.67Ghz, 3TB RAM, SUSE Linux Enterprise Server, Oracle Java 6

- Java options:

  Min & Max heap: 70GB, Stack: 128MB, 8 threads

Name	Paths	Locks	Total
sync	0.122s	0.14s	5m 31s
pcmab	0.246s	0.092s	5m 15s
bank	0.247s	0.129s	5m 27s
traffic	1.695s	0.2s	5m 40s
mtrt	1h 30m	8.579s	1h 36m
hsqldb	6h 6m	22m	6h 38m

What about runtime performance?

- Experimental machine (a modern desktop):

8-core i7 3.4Ghz, 8GB RAM, Ubuntu 11.04, Jikes RVM

Benchmark	Manual	Global	Us	Us vs Manual
sync	69.14s	71.22	74.61s	**1.08x**
pcmab	2.28s	3.15	12.47s	**5.47x**
bank	20.89s	19.50	30.88s	**1.47x**
traffic	2.56s	4.22	91.42s	**35.71x**
mtrt	0.80s	0.82	0.95s	**1.19x**
hsqldb	3.25s	3.12	500s	**153.85x**

Improve run-time performance: Avoid unnecessary locking

- We avoid unnecessary locking to improve the performance of the resulting instrumented programs.

Lock optimisation	Type of analysis	Runtime slowdown vs. manual locking
Single-threaded lock elision	Dynamic	1.10x – 16.13x
Thread-local	Static	1.09x – 14.84x
Instance-local	Static	1.13x – 13.16x
Class-local	Static	1.14x – 15.32x
Method-local	Static	1.14x – 15.05x
Dominated	Static	1.14x – 15.47x
Read-only	Static	1.14x – 13.26x

Removing locks: All optimisations

Benchmark	Manual	Global	Us (no opt.)	Us (all opt.)	Us vs Manual	Us vs Global
sync	69.14s	71.22s	74.61s	56.61s	**0.82x**	**0.79x**
pcmab	2.28s	3.15s	12.47s	2.47s	**1.08x**	**0.78x**
bank	20.89s	19.50s	30.88s	3.88s	**0.19x**	**0.20x**
traffic	2.56s	4.22s	91.42s	4.42s	**1.73x**	**1.05x**
mtrt	0.80s	0.82s	0.95s	0.85s	**1.06x**	**1.04x**
hsqldb	3.25s	3.12s	500s	11.39s	**3.50x**	**3.65x**

Achievements

- We present a **scalable** set of analyses and optimisations that are able to **fully** analyse library code in reasonable space and time
- Ours is thus the **first sound approach**
- With a large number of optimisations, we manage to get worst-case execution times of only **3.50x** and <**2x** in the general case vs *perfect and well-tested* manual locking
- We also achieve some **speed ups**