Lock Inference in the Presence of Large Libraries

Khilan Gudka, Imperial College London*
Tim Harris, Microsoft Research Cambridge
Susan Eisenbach, Imperial College London

ECOOP 2012

This work was generously funded by Microsoft Research Cambridge
* Now at University of Cambridge Computer Laboratory

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++;
}
```


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

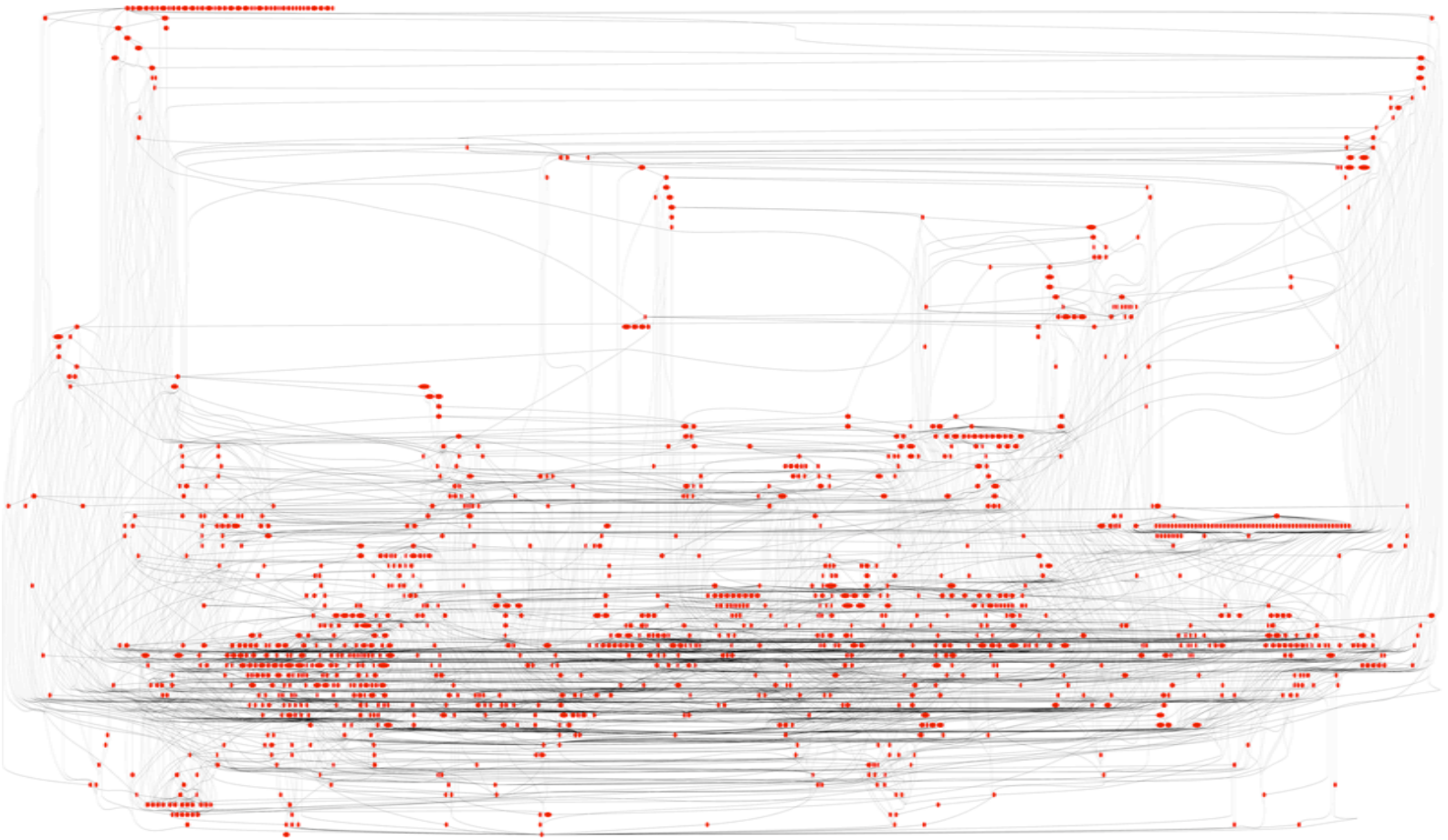
- Acquire locks in two-phased order for atomicity
- **Can handle irreversible operations!**

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!

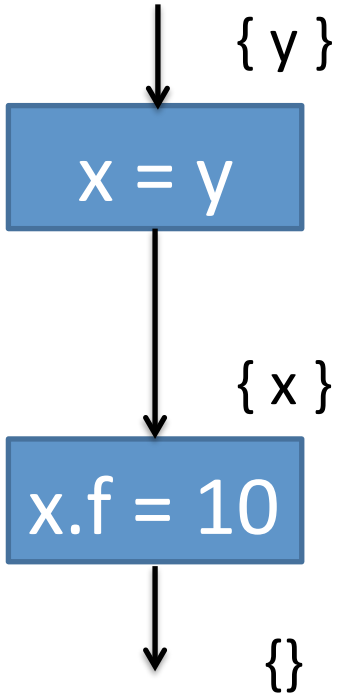
- General goals/challenges of lock inference
  - Maximise concurrency
  - Minimise locking overhead
  - Avoid deadlock

- 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)

Our lock inference analysis: Infer fine-grained locks

- Infer path expressions at each program point:

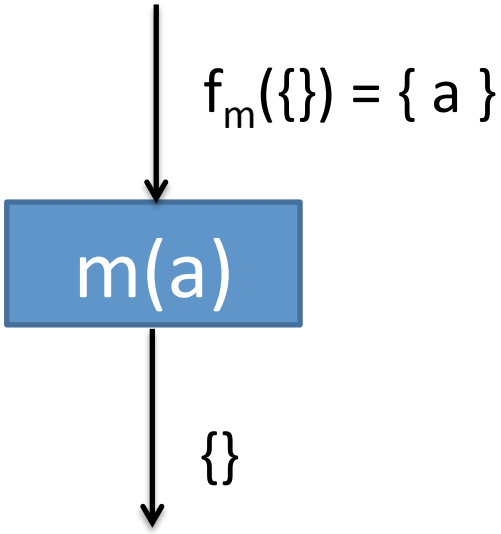
```
Obj x = …;
Obj y = …;
atomic {
   x = y;
   x.f++;
}
```




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

Scaling by computing summaries

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

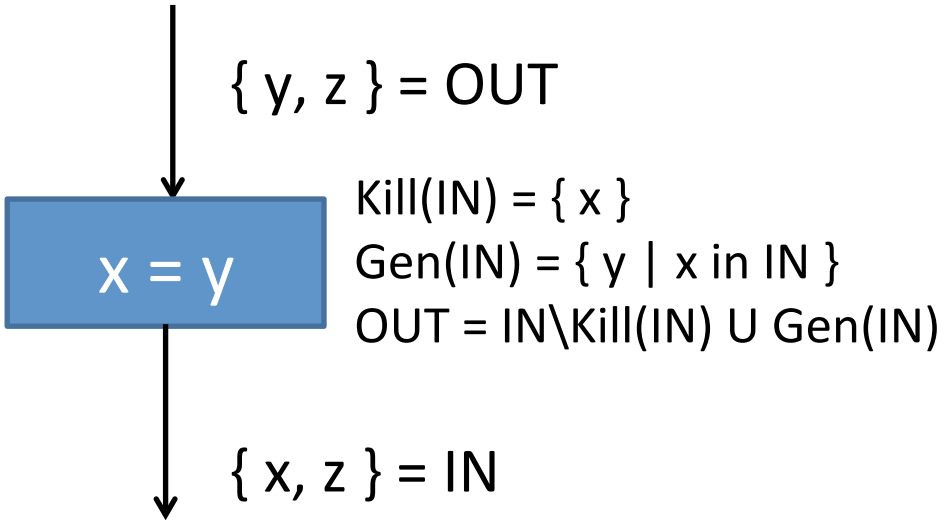


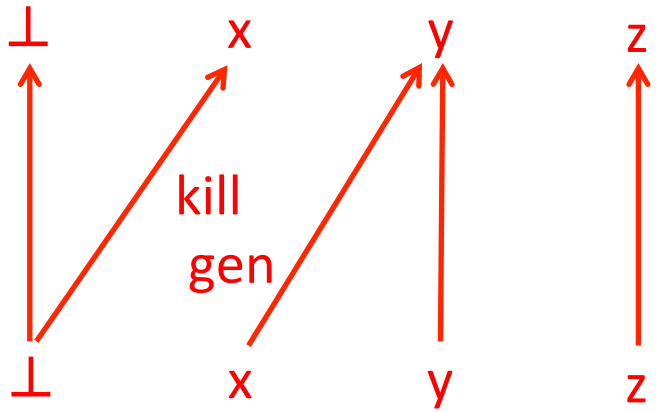
f_m is m's summary function

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

IDE Analyses

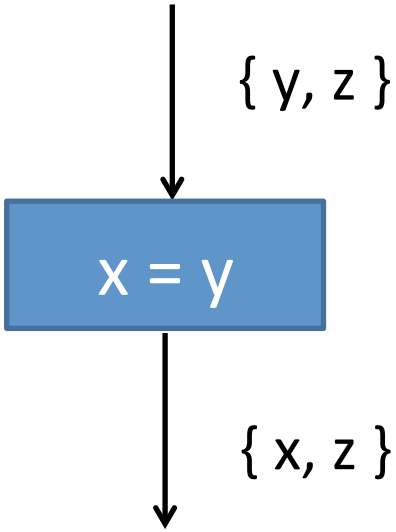
- Use Sagiv et al's Interprocedural Distributive Environment framework
- **Advantage**: efficient graph representation of transfer functions that allows fast composition and meet

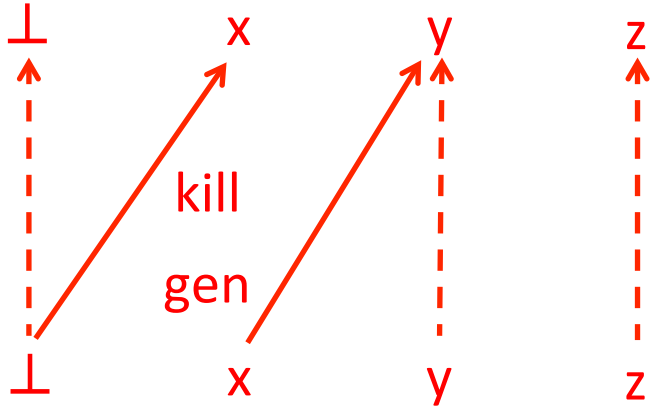




Transfer functions as graphs

- Graphs are kept sparse by not explicitly representing trivial edges





- Transformer composition is simply transitive closure

Transfer functions as graphs

- Implicit edges should not have to be made explicit as that would be expensive
- For our analysis, most transformer functions perform rewrites, thus determining whether an implicit edge exists is costly using Sagiv et al's graphs

Transfer functions as graphs (Ours)

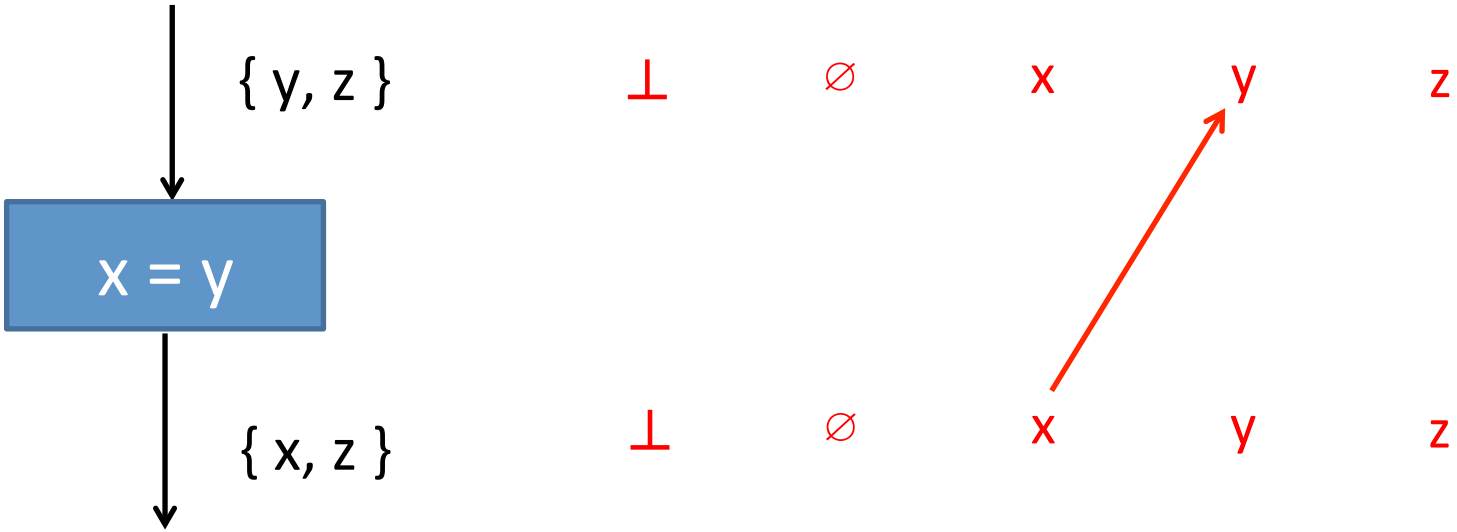
- We represent kills in transformers as:

- Transformer edges also implicitly kill:

- Result: implicit edge very easy to determine. This leads to fast transitive closure

Transfer functions as graphs (Ours)

- Example:



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 (a modern desktop):

  8-core i7 3.4Ghz, 8GB RAM, Ubuntu 11.04, Oracle Java 6

- Java options:

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

Name	Paths	Locks	Total
sync	0.05s	0.01s	2m 7s
pcmab	0.15s	0.02s	2m 7s
bank	0.15s	0.02s	2m 7s
traffic	0.37s	0.06s	2m 10s
mtrt	33.9s	1.89s	2m 49s
hsqldb	?	?	?

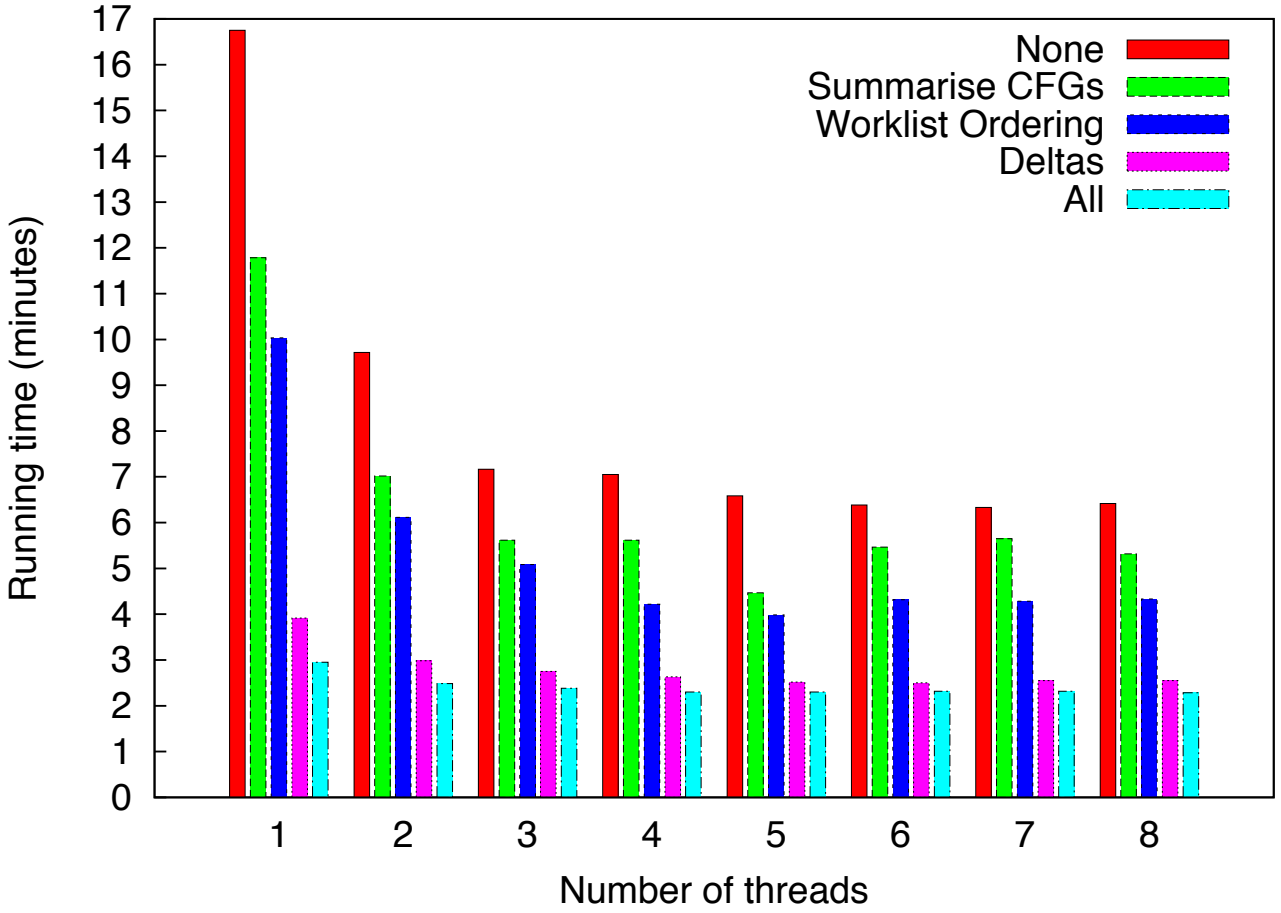
Simple analysis not enough

- 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

Evaluation of analysis optimisations: Analysis running time

- On the Hello World program…



Evaluation of analysis optimisations: Analysis memory usage

- On the Hello World program…

Optimisation	Average MB	Peak MB
None	4923.92	8183.18
Summarise CFGs	2094.68	3470.65
Worklist Ordering	4804.73	8037.14
Deltas	3848.98	6538.27
All	1741.39	3122.84

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	0.05s	0.01s	2m 7s
pcmab	0.15s	0.02s	2m 7s
bank	0.15s	0.02s	2m 7s
traffic	0.37s	0.06s	2m 10s
mtrt	33.9s	1.89s	2m 49s
hsqldb	6h 6m	22m	6h 38m

What about deadlock?

- Lock inference inserts locks automatically, so it must ensure that deadlock doesn't happen
- Static analysis is too conservative
- Deadlock happens very infrequently
- All locks are taken at the start of the atomic, so can just **rollback** the locks if deadlock occurs and try again!

What about runtime performance?

Benchmark	Manual	Global	Us	Us vs Manual
sync	69.14s	71.22	81.59s	**1.18x**
pcmab	2.28s	3.15	54.61s	**23.95x**
bank	20.89s	19.50	76.88s	**3.68x**
traffic	2.56s	4.22	20.77s	**8.11x**
mtrt	0.80s	0.82	0.91s	**1.14x**
hsqldb	3.25s	3.12	419s	**129.03x**

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	81.59s	56.61s	**0.82x**	**0.79x**
pcmab	2.28s	3.15s	54.61s	2.47s	**1.08x**	**0.78x**
bank	20.89s	19.50s	76.88s	3.88s	**0.19x**	**0.20x**
traffic	2.56s	4.22s	20.77s	4.42s	**1.73x**	**1.05x**
mtrt	0.80s	0.82s	0.91s	0.85s	**1.06x**	**1.04x**
hsqldb	3.25s	3.12s	419s	11.39s	**3.50x**	**3.65x**

What about Hello World?

- Concurrent Hello World benchmark with 8 threads.
- Each thread prints “Hello World! from thread X” 1000 times
- Analysis results

Analysis time	No. of locks (no lock opts)	No. of locks (all lock opts)
2m 30s	495	25

- Runtime performance

Manual	Global	Us (no opt.)	Us (all opt.)
0.32s	0.27s	2.21s	0.8s

Conclusion

- Existing lock inference approaches are **unsound** because they do not analyse library code
  - i.e. due to scalability, imprecision, etc.
- Our approach does, thus correct by construction
- With an enormous number of optimisations, we manage to get worst-case execution time of only **3.50x** and <**2x** in general case vs *perfect and well-tested* manual locking as well as some speed-ups!
- So, programmers get the simplicity of atomic sections with almost the speed of manual locks

Questions?

*"Out of this nettle, **danger**, we pluck this flower, **safety**"*

William Shakespeare

If he was a programmer today…

*"Out of this nettle, **concurrency**, we pluck this flower, **atomicity**"*