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

```c
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

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

```plaintext
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

```java
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

• 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)

Prior work

This work
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++; 
}
```
Scaling by computing summaries

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

$f_m(\{\}) = \{a\}$

$f_m$ 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).

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Paths</th>
<th>Locks</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>sync</td>
<td>0.122s</td>
<td>0.14s</td>
<td>5m 31s</td>
</tr>
<tr>
<td>pcmab</td>
<td>0.246s</td>
<td>0.092s</td>
<td>5m 15s</td>
</tr>
<tr>
<td>bank</td>
<td>0.247s</td>
<td>0.129s</td>
<td>5m 27s</td>
</tr>
<tr>
<td>traffic</td>
<td>1.695s</td>
<td>0.2s</td>
<td>5m 40s</td>
</tr>
<tr>
<td>mtrt</td>
<td>1h 30m</td>
<td>8.579s</td>
<td>1h 36m</td>
</tr>
<tr>
<td>hsqldb</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Paths</th>
<th>Locks</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>sync</td>
<td>0.122s</td>
<td>0.14s</td>
<td>5m 31s</td>
</tr>
<tr>
<td>pcmab</td>
<td>0.246s</td>
<td>0.092s</td>
<td>5m 15s</td>
</tr>
<tr>
<td>bank</td>
<td>0.247s</td>
<td>0.129s</td>
<td>5m 27s</td>
</tr>
<tr>
<td>traffic</td>
<td>1.695s</td>
<td>0.2s</td>
<td>5m 40s</td>
</tr>
<tr>
<td>mtrt</td>
<td>1h 30m</td>
<td>8.579s</td>
<td>1h 36m</td>
</tr>
<tr>
<td>hsqldb</td>
<td>6h 6m</td>
<td>22m</td>
<td>6h 38m</td>
</tr>
</tbody>
</table>
What about runtime performance?

- Experimental machine (a modern desktop):
  
  8-core i7 3.4Ghz, 8GB RAM, Ubuntu 11.04, Jikes RVM

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Manual</th>
<th>Global</th>
<th>Us</th>
<th>Us vs Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>sync</td>
<td>69.14s</td>
<td>71.22</td>
<td>74.61s</td>
<td>1.08x</td>
</tr>
<tr>
<td>pcmab</td>
<td>2.28s</td>
<td>3.15</td>
<td>12.47s</td>
<td>5.47x</td>
</tr>
<tr>
<td>bank</td>
<td>20.89s</td>
<td>19.50</td>
<td>30.88s</td>
<td>1.47x</td>
</tr>
<tr>
<td>traffic</td>
<td>2.56s</td>
<td>4.22</td>
<td>91.42s</td>
<td>35.71x</td>
</tr>
<tr>
<td>mtrt</td>
<td>0.80s</td>
<td>0.82</td>
<td>0.95s</td>
<td>1.19x</td>
</tr>
<tr>
<td>hsqldb</td>
<td>3.25s</td>
<td>3.12</td>
<td>500s</td>
<td>153.85x</td>
</tr>
</tbody>
</table>
Improve run-time performance: Avoid unnecessary locking

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

<table>
<thead>
<tr>
<th>Lock optimisation</th>
<th>Type of analysis</th>
<th>Runtime slowdown vs. manual locking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-threaded lock elision</td>
<td>Dynamic</td>
<td>1.10x – 16.13x</td>
</tr>
<tr>
<td>Thread-local</td>
<td>Static</td>
<td>1.09x – 14.84x</td>
</tr>
<tr>
<td>Instance-local</td>
<td>Static</td>
<td>1.13x – 13.16x</td>
</tr>
<tr>
<td>Class-local</td>
<td>Static</td>
<td>1.14x – 15.32x</td>
</tr>
<tr>
<td>Method-local</td>
<td>Static</td>
<td>1.14x – 15.05x</td>
</tr>
<tr>
<td>Dominated</td>
<td>Static</td>
<td>1.14x – 15.47x</td>
</tr>
<tr>
<td>Read-only</td>
<td>Static</td>
<td>1.14x – 13.26x</td>
</tr>
</tbody>
</table>
Removing locks: All optimisations

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