Shared Memory Concurrency

Khilan Gudka
Susan Eisenbach

Where we are: we use locks

- But there are problems with them
  - Not composable
  - Introduce deadlock
  - Break modularity
  - Priority inversion
  - Convoying
  - Starvation
  - ...

A better solution: atomic sections

- What programmers probably can do is tell which parts of their program should not involve interferences
- Atomic sections [Lomet77]
  - Declarative concurrency control
  - Move responsibility for figuring out what to do to the compiler/runtime

```
atomic {
   … access shared state …
}
```

A better solution: atomic sections

- Simple semantics (no interference allowed)
- Naïve implementation: one global lock
- But we want to allow parallelism without:
  - Interference
  - Deadlock

Implementing Atomic Sections: transactional memory

- Much interest [For review of work up until 2010, see Harris10]
- Advantages
  - No problems associated with locks
  - More concurrency
- Disadvantages
  - Irreversible operations (IO, System calls)
  - Runtime overhead

Implementing Atomic Sections: lock inference

- Statically infer the locks that are needed to protect shared accesses
- Insert lock()/unlock() statements for them into the program to ensure atomic execution

```
atomic {                            synchronized(x) {
  x.f = 1;    -- compiled to -->      x.f = 1;
}                                   }
```

Implementing Atomic Sections: lock inference

- Challenges
  - Maximise concurrency
  - Minimise locking overhead
  - Avoid deadlock

Caveat: locking must be two-phased for atomicity

- Cannot acquire a lock once a release occurs

```
atomic {
  …
  lock(x);
  …
  lock(y);
  …
  unlock(x);
  …
  unlock(y);
  …
}
```

Correct

```
atomic {
  …
  lock(x);
  …
  unlock(x);
  …
  lock(y);
  …
  unlock(y);
  …
}
```

Wrong

What about deadlock?

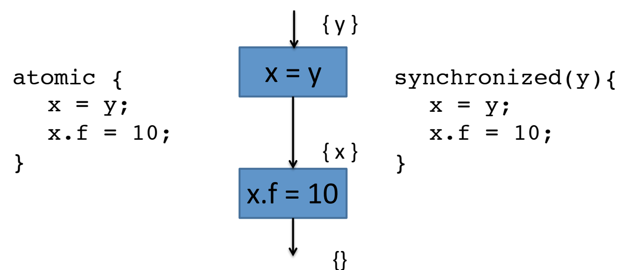
- Lock inference inserts locks automatically, so it must ensure that deadlock doesn't happen
- Static analysis is too conservative.
- Deadlock happens very infrequently
- All locks are taken at the start of the atomic, so can just **rollback** the locks if deadlock occurs and try again!

Importance of locking granularity

- To maximise parallelism, locks should be as fine-grained as possible
- The granularity of locks depends on the compile-time representation of objects
- Paths (e.g. x.f) allow per-instance locks when each object has it's own lock (e.g. Java)
- We developed an analysis to infer paths

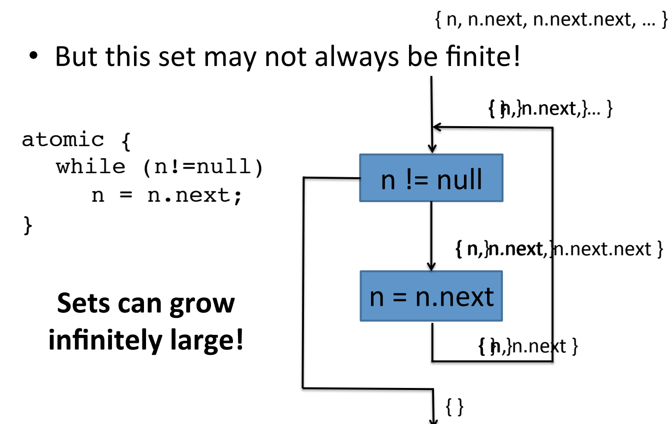
Inferring fine-grained locks

- Infer sets of paths at each program point



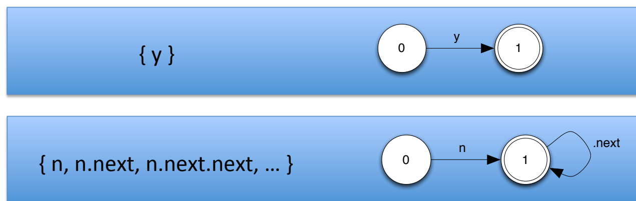
Problem of infinite sets of locks

- But this set may not always be finite!



Automata

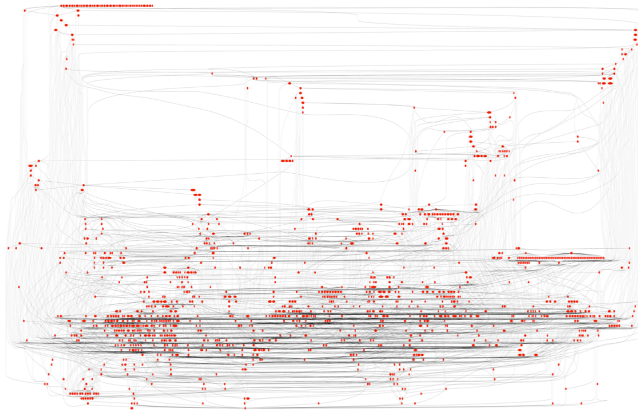
- Can represent a possibly infinite set of paths
- A compact compile-time representation
- Our analysis flows automata around the CFG



Scaling to Java: "Hello World!"

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

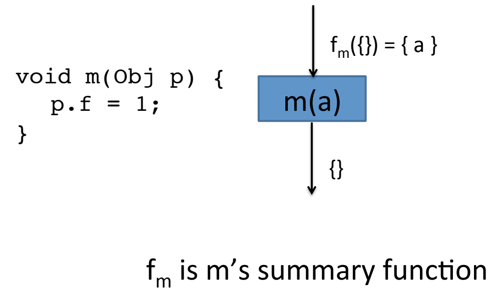
Callgraph for "Hello World!"



Scaling by computing summaries

- Our previous analysis didn't scale
- We therefore switched to computing summaries
- A summary is a function that describes how a method as a whole translates dataflow information
- Summaries scale better – they can be computed once for a method and re-used
- Uses Sagiv's interprocedural dataflow analysis

Scaling by computing summaries



f_m is m's summary function

Benchmarks

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:

  8-core i7 3.4Ghz, 8GB RAM, Ubuntu 11.04
- Java options:

  Min & Max heap: 4GB, Stack: 128MB, 8 threads

Name	Paths	Locks	Total
sync	0.05s	0.01s	2m10s
pcmab	0.15s	0.02s	2m10s
bank	0.15s	0.02s	2m11s
traffic	0.37s	0.06s	2m13s
mtrt	33.9s	1.89s	2m54s
hsqldb	?	?	?

Simple analysis not enough

- Our analysis still wasn't efficient enough to analyse hsqldb.
- We performed further optimisations to reduce space-time requirements:
  - Delta propagation
    - Only propagate new dataflow information
    - Reduces the amount of redundant work
  - Compressing CFGs
    - Merging CFG nodes to reduce the amount of storage space and propagation carried out
  - Primitives for state

    Encode analysis state as sets of longs for efficiency
  - Parallel propagation

Analysis times

- Experimental machine for hsqldb:
  80-core Xeon E7-8870 2.4Ghz, 1TB RAM, Ubuntu 10.04
- Java options:
  Min & Max heap: 70GB, Stack: 128MB, 8 threads

Name	Paths	Locks	Total
sync	0.05s	0.01s	2m 10s
pcmab	0.15s	0.02s	2m 10s
bank	0.15s	0.02s	2m 11s
traffic	0.37s	0.06s	2m 13s
mtrt	33.9s	1.89s	2m 54s
hsqldb	14h 47m	38m	15h 40m

Runtime Performance

Benchmark	Manual	Us
sync	47.5s	56.1s
pcmab	1.9s	45.5s
bank	2.8s	10.3s
traffic	1.9s	15.4s
mtrt	0.7s	0.8s
hsqldb	3.1s	400s

Improving runtime performance

- We remove locks to improve the performance of the resulting programs:
  - Single-threaded execution
  - Thread-local
  - Instance-local
  - Class-local
  - Method-local
  - Dominated
  - Read-only
  - Implicit locks

Single-threaded execution

- Do not acquire any locks unless multiple threads are executing.
- All optimisations that follow will assume that this one is enabled

Removing locks: Thread-local

- Remove locks on objects that are not shared and thus do not need to be locked

Benchmark	Manual	Before	After
sync	47.5s	56.1s	59.9s
pcmab	1.9s	45.5s	3.9s
bank	2.8s	10.3s	10.6s
traffic	1.9s	15.4s	13s
mtrt	0.7s	0.8s	0.76s

Removing locks: Instance-local

- Look for implementation-only objects (e.g. Node instances in LinkedList) that do not escape their enclosing object
- Protect them by locking the owning instance (e.g. LinkedList)

Benchmark	Manual	Before	After
sync	47.5s	56.1s	53.8s
pcmab	1.9s	45.5s	2.4s
bank	2.8s	10.3s	9.4s
traffic	1.9s	15.4s	11.7s
mtrt	0.7s	0.8s	0.8s

Removing locks: Class-local

- Similar as instance-local but for static variables
- Class-local objects are not accessed from classes except the creating one

Benchmark	Manual	Before	After
sync	47.5s	56.1s	61.3s
pcmab	1.9s	45.5s	4.3s
bank	2.8s	10.3s	10.4s
traffic	1.9s	15.4s	13.9s
mtrt	0.7s	0.8s	0.8s

Removing locks: Method-local

- Identify objects that do not escape the method they are created in and thus also do not need to be locked
- Purpose is to find at the start of an atomic section, which local variables point to new objects that have not escaped the current method
- These objects don't need to be locked

Benchmark	Manual	Before	After
sync	47.5s	56.1s	55.8s
pcmab	1.9s	45.5s	4.3s
bank	2.8s	10.3s	11.1s
traffic	1.9s	15.4s	13.4s
mtrt	0.7s	0.8s	0.8s

Removing locks: Dominated

- If a lock A is always acquired when lock B is, then it is sufficient to only lock A and not B.
- We say that lock A dominates lock B

Benchmark	Manual	Before	After
sync	47.5s	56.1s	55.1s
pcmab	1.9s	45.5s	3.8s
bank	2.8s	10.3s	9.3s
traffic	1.9s	15.4s	14.2s
mtrt	0.7s	0.8s	0.8s

Removing locks: Read-only

- If an object is only locked in read mode, then we don't need to acquire it at all
- It is never acquired in a conflicting mode so locking it is superfluous

Benchmark	Manual	Us
sync	47.5s	56.3s
pcmab	1.9s	4s
bank	2.8s	9.5s
traffic	1.9s	10s
mtrt	0.7s	0.8s

Removing locks: Implicit-locks

- Don't need to acquire the type lock in intention mode if the type itself is never locked!

Benchmark	Manual	Before	After
sync	47.5s	56.1s	54.5s
pcmab	1.9s	45.5s	3.9s
bank	2.8s	10.3s	7.6s
traffic	1.9s	15.4s	9.5s
mtrt	0.7s	0.8s	0.8s

Removing locks: All optimisations

Benchmark	Manual	Before	After
sync	47.5s	56.1s	54.5s
pcmab	1.9s	45.5s	2.2s
bank	2.8s	10.3s	5.3s
traffic	1.9s	15.4s	4s
mtrt	0.7s	0.8s	0.8s
hsqldb	3.1s	400s	15s

Conclusion

- What kind of evaluation do we need to do to see whether
  - The hypothesis that atomicity is a better model than explicitly locking is validated or rejected
  - The analysis is fast enough
  - The code is fast enough