Improving the performance of Atomic Sections

Khilan Gudka

Supervised by:
Prof. Susan Eisenbach
The multi-core revolution has made concurrency a **hot topic**

Programmers are now forced to think about it for **performance**

But shared memory concurrency is **hard!**
Where we are: we use locks

Problems

- Not composable
- Introduce deadlock
- Break modularity
- Other problems: priority inversion, convoying, starvation...
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

```c
atomic {
    ... access shared state ...
}
```
Atomic sections

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

Very hot research area - lots of papers!
[For review of work up until 2006, see Larus06]

Advantages

- No problems associated with locks
- More concurrency

Disadvantages

- Irreversible operations (IO, System calls)
- Run-time overhead
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
Lock inference

Challenges

- Maximise concurrency
- Minimise locking overhead
- Avoid deadlock
Restriction for atomicity: Two-phase locking

**Correct**

```java
atomic {
    ...
    lock(A);
    ...
    lock(B);
    ...
    unlock(B);
    ...
    unlock(A);
    ...
}
```

**Wrong**

```java
atomic {
    ...
    lock(A);
    ...
    unlock(A);
    ...
    lock(B);
    ...
    unlock(B);
    ...
}
```
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.

Lvalues (e.g. x.f) allow per-instance locks when each object has its own lock (e.g. Java).

During my masters, we developed an analysis to infer lvalues and it was published in CC’08 [Cunningham08].
Finite State Automata

- A compact compile-time object representation
- Represents a possibly infinite set of lvalues
- Our analysis flows automata around the CFG

\[
\{ y \} = \{ n, n.next, n.next.next, \ldots \} = \]

```
Finite State Automata
```

```text
\[
\{ y \} = 0 \xrightarrow{y} 1
\{ n, n.next, n.next.next, \ldots \} = 0 \xrightarrow{n} 2 \xrightarrow{.next} 0
\]
```
Scaling to Java: “Hello world”

```java
atomic {
    System.out.println("Hello World");
}
```
Scaling to Java: “Hello world”

Call graph:
Scaling to Java: “Hello world”

- This work doesn’t scale
- We switch to computing summaries
- A summary is a function that describes how a method as a whole translates dataflow information
- Summaries are also context-sensitive but can scale better
Method summaries

m()

{x}

x.f = 1
Method summaries

\[ f_m(\{x\}) \]

\[ m() \]

\[ \{x\} \]

\[ x.f = 1 \]

\[ f_m \text{ is } m\text{'s summary} \]

Tuesday, 16 June 2009
Computing summaries

- Define, for each statement, transfer functions describing how they translate dataflow information.
- Compose them into one large transfer function for the entire method by flowing them through the CFG using a normal dataflow analysis.
- Summaries can get large: challenge is to find a representation of transfer functions that allows fast composition and meet operations.
IDE Analyses

- Interprocedural Distributive Environment [Sagiv96]
- Dataflow facts are functions of type $D \rightarrow L$, called environments
- Transfer functions are called environment transformers
- Advantage: efficient graph representation of environment transformers exists that allows fast composition and meet [Reps95, Sagiv96, Rountev08]
Reformulate our lvalue analysis

Step 1: Express automata as environments (functions of type $D \rightarrow L$)

We represent automata as functions from transition labels to sets of pairs of states (of the transitions for those labels)

$$[x \rightarrow \{ (0,1) \}, \ y \rightarrow \{ (0,2) \}]$$
Environment transformers

- Step 2: Define environment transformers (i.e. the transfer functions)
- They describe how the ‘outgoing’ environment is computed from the ‘incoming’ environment
Environment transformers

Step 2: Define environment transformers (i.e. the transfer functions)

They describe how the ‘outgoing’ environment is computed from the ‘incoming’ environment

\[ x = y \]

\[ e_{env_{out}} = t_{[x=y]}(e_{env_{in}}) \]
Environment transformers

\[ x = y \]

\[
[x \rightarrow \{ (0,1) \},
  y \rightarrow \{ (0,2) \}]\]
Environment transformers

\[
x = y
\]

\[
[x \rightarrow \emptyset,
y \rightarrow \{(0,2), (0,1)\}]
\]

\[
[x \rightarrow \{(0,1)\},
y \rightarrow \{(0,2)\}]
\]
Environment transformers

\[ x = y \]

\[ [x \rightarrow \emptyset \quad y \rightarrow \{(0,2), (0,1)\}] \]

\[ [x \rightarrow \{(0,1)\}, \quad y \rightarrow \{(0,2)\}] \]

\[ \dagger_{[x=y]} = \lambda e. e[y \rightarrow e(y) \cup e(x)][x \rightarrow \emptyset] \]
Environment transformers (as in [Sagiv96])

These transformers can be represented as graphs

\[ \hat{t}_{[x=y]} = \lambda e. e[y \rightarrow e(y) U e(x)][x \rightarrow \emptyset] \]
Environment transformers (as in [Sagiv96])

Graphs are kept sparse by not explicitly representing obvious edges

\[
\begin{align*}
\{x & \rightarrow \{ (0,1) \} \\
y & \rightarrow \emptyset \}
\end{align*}
\]

\[
\{x & \rightarrow \{ (0,1) \} \\
y & \rightarrow \{ (0,2) \} \}
\]
Environment transformers (as in [Sagiv96])

- Transformer composition is simply the transitive closure
- Implicit edges should not have to be made explicit as that would be expensive
- But determining whether an implicit edge exists is costly in [Sagiv96] for our analysis

\[
\lambda x. \varnothing
\]
Environment transformers

(Ours)

- We represent kills in transformers as:

\[ x \rightarrow \emptyset \]

\[ \lambda e. e[x->\emptyset] \]

- Our lvalues analysis mostly rewrites lvalues, hence we change the meaning of transformer edges to pass on but also implicitly kill:

\[ x \rightarrow y \]

\[ \lambda e. e[y->\text{env}(y)\cup\text{env}(x)][x->\emptyset] \]

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

(Ours)

We represent kills in transformers as:

\[ x \rightarrow \emptyset \]

\[ \lambda e.e[x->\emptyset] \]

Our lvalues analysis mostly rewrites lvalues, hence we change the meaning of transformer edges to pass on but also implicitly kill:

\[ x \rightarrow y \]

\[ \lambda e.e[y->\text{env}(y) U \text{env}(x)][x->\emptyset] \]

Result: implicit edge very easy to determine. This leads to fast transitive closure.
Environment transformers
(Ours)

\[ y = \text{null} \]

\[ x = y \]
Environment transformers
(Ours)
Implementation

- Implemented in the Soot bytecode analysis framework and am experimenting with small programs at present
- Implementation identifies strongly connected components (SCC) and propagates summaries up the SCC-DAG
Future Work: Area 1

Maximise concurrency between atomic sections that only partially conflict

Existing work either:
- Serialises whole atomics
  - [Halpert07, Zhang07, Cherem08, Hicks06]
- Serialises up to a conflict
  - [Cunningham08]
- Serialises after a conflict
  - [McCloskey06, Emmi07]

Two-phase locking can be too restrictive and thus hamper concurrency unnecessarily
Future Work: Area 1

A1

listA.add(o)

A2

o1 = listA.get(0)

o2 = process(o1)

listB.add(0, o2)

A3

listB.size()
Future Work: Area 1

A1

listA.add(o)

A2

o1 = listA.get(0)

o2 = process(o1)

A3

listB.size()

listB.add(0, o2)
Future Work: Area 1

Basic locking:

A1

\[ \text{listA.add(o)} \]

\[ \text{L(listA)} \]

\[ \text{U(listA)} \]

A2

\[ o1 = \text{listA.get(0)} \]

\[ \text{L(listA)} \]

\[ \text{L(listB)} \]

A3

\[ \text{listB.size()} \]

\[ \text{L(listB)} \]

\[ \text{U(listB)} \]

Tuesday, 16 June 2009
Future Work: Area 1

Late locking:

A1

listA.add(o)

L(listA)

U(listA)

A2

o1 = listA.get(0)

L(listA)

A3

listB.size()

L(listB)

U(listB)

listB.add(0,o2)

L(listB)

U(listB)

U(listB)

U(listA)

Tuesday, 16 June 2009
Early unlocking:

A1
- listA.add(o)
- L(listA)
- U(listA)

A2
- o1 = listA.get(0)
- L(listA)
- U(listA)
- o2 = process(o1)
- L(listA)
- U(listA)
- listB.add(0,o2)
- listB.size()
- U(listB)

A3
- L(listB)
- U(listB)
- listB.size()
- U(listB)

Future Work: Area 1

Tuesday, 16 June 2009
Future Work: Area 1

A1

listA.add(o)

L(listA)

U(listA)

A2

o1 = listA.get(0)

L(listA)

U(listA)

A3

listB.size()

o2 = process(o1)

listB.add(0, o2)
Future Work: Area 1

A1

listA.add(o)

A2

o1 = listA.get(0)

L(listA)

U(listA)

L(listA)

U(listA)

o2 = process(o1)

A3

listB.size()

L(listB)

U(listB)

L(listB)

U(listB)

listB.add(0,o2)

L(listB)

U(listB)

U(listB)
Future Work: Area 1

\[ A_1 \]
\[ \text{i++} \]
\[ \text{m() \quad i++} \]
\[ \text{i++} \]
\[ \text{A2} \]
\[ \text{i++} \]
\[ \text{n() \quad i++} \]
\[ \text{i++} \]

\text{m and n disjoint}
Future Work: Area 1

A1
\[ \text{i}++ \]
\[ \text{m()} \]
\[ \text{i}++ \]
\[ \text{L(i)} \]
\[ \text{U(i)} \]

m and n disjoint but serialised!

A2
\[ \text{i}++ \]
\[ \text{n()} \]
\[ \text{i}++ \]
\[ \text{L(i)} \]
\[ \text{U(i)} \]

Tuesday, 16 June 2009
Future Work: Area 1

one solution: re-order
Future Work: Area 2

Area 2: concurrent accesses to arrays:
e.g. parallel map function:

```cpp
for (int i=0; i<numChunks; i++) {
    spawn {
        int start = i*chunkSize;
        int end = start+chunkSize;
        for (int j=start; j<end; j++) {
            atomic {
                a[j] = f(a[j]);
            }
        }
    }
}
```
Future Work: Area 3

- Area 3: allow the use of multi-threaded code within atomic sections
- Amdahl’s law, composability
- Support a spawn construct inside atomic {}
- Could also use to automatically improve the performance of atomic sections
Future Work: Area 3

m() and n() disjoint but serialised!
Future Work: Area 3

A1

\[ i++ \]

\[ L(i) \]

\[ \text{spawn m()} \]

\[ \text{m and n not serialised!} \]

\[ i++ \]

\[ U(i) \]

A2

\[ i++ \]

\[ L(i) \]

\[ \text{spawn n()} \]

\[ i++ \]

\[ U(i) \]
Future Work: Area 4

- Area 4: consider a hybrid implementation with transactional memory
- Benefit of transactional memory’s high concurrency
- Reduce run-time overhead and allow irreversible operations using locks
Related work

Philosophy of approach

Top down [Zhang07, Halpert07]
Bottom up [McCloskey06, Hicks06, Emmi07, Cunningham08, Cherem08]

Compile-time representation of objects:

Abstract objects [Hicks06, Halpert07]
Lvalues [McCloskey06, Hicks06, Emmi07, Cunningham08, Cherem08]

Granularity of locks:

Fine [McCloskey06, Emmi07, Halpert07]
Coarse [Hicks06, Halpert07, Zhang07]
Related work

The specific two-phase locking policy:

- **Basic** [Hicks06, Zhang07, Halpert07, Cherem08]
- **Late locking** [McCloskey06, Emmi07]
- **Early unlocking** [Cunningham08]

Deadlock avoidance:

- **Static** [McCloskey06, Hicks06, Emmi07, Zhang07, Halpert07]
- **Dynamic** [Cunningham08, Cherem08]
Conclusion

My thesis:

- Implement atomic using locks
- Maximise concurrency between atomics
- Be able to handle a real language
"The most likely way for the world to be destroyed, most experts agree, is by accident. That's where we come in; we're computer professionals. We cause accidents."

Nathaniel Borenstein (co-creator of MIME)

We need better abstractions!
The problem: shared memory

- Reading from disk
- Updating display
- Solving equation

Memory
Bank account example

A
£10

B
£10

T1
A £10 B

T2
A £10
Method that transfers money between accounts, if sufficient funds are available:

```java
void transfer(Acct A, Acct B, int amt) {
    int bal = A.getBalance();
    if (amt <= bal) {
        A.withdraw(amt);
        B.deposit(amt);
    }
}
```
Bank account (locks)

\[
\text{transfer}(A, B, 10) \parallel a.\text{withdraw}(10)
\]

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Check A`s balance</td>
<td></td>
<td>£10</td>
<td>£10</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Withdraw £10 from A</td>
<td>£0</td>
<td>£10</td>
</tr>
<tr>
<td>3</td>
<td>Withdraw £10 from A</td>
<td></td>
<td>-£10</td>
<td>£10</td>
</tr>
<tr>
<td>4</td>
<td>Deposit £10 into B</td>
<td></td>
<td>-£10</td>
<td>£20</td>
</tr>
</tbody>
</table>

Tuesday, 16 June 2009
## Bank account (locks)

```
\text{transfer}(A, B, 10) \parallel a.\text{withdraw}(10)
```

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Check A`s balance</td>
<td></td>
<td>£10</td>
<td>£10</td>
</tr>
<tr>
<td>2</td>
<td>Withdraw £10 from A</td>
<td>Withdraw £10 from A</td>
<td>£0</td>
<td>£10</td>
</tr>
<tr>
<td>3</td>
<td>Withdraw £10 from A</td>
<td></td>
<td>£0</td>
<td>£10</td>
</tr>
<tr>
<td>4</td>
<td>Deposit £10 into B</td>
<td></td>
<td>£0</td>
<td>£20</td>
</tr>
</tbody>
</table>

Tuesday, 16 June 2009
Bank account (locks)

\[
\text{transfer}(A, B, 10) \parallel a.\text{withdraw}(10)
\]

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Check A`s balance</td>
<td></td>
<td>£10</td>
<td>£10</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Withdraw £10 from A</td>
<td>£0</td>
<td>£10</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Withdraw £10 from A</td>
<td>-£10</td>
<td>£10</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Deposit £10 into B</td>
<td>-£10</td>
<td>£20</td>
</tr>
</tbody>
</table>

Tuesday, 16 June 2009
## Bank account (locks)

\[
\text{transfer}(\text{A, B, 10}) \parallel \text{a.withdraw(10)}
\]

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Check A`s balance</td>
<td></td>
<td>£10</td>
<td>£10</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Withdraw £10 from A</td>
<td>£0</td>
<td>£10</td>
</tr>
<tr>
<td>3</td>
<td>Withdraw £10 from A</td>
<td></td>
<td>-£10</td>
<td>£10</td>
</tr>
<tr>
<td>4</td>
<td>Deposit £10 into B</td>
<td></td>
<td>-£10</td>
<td>£20</td>
</tr>
</tbody>
</table>
void transfer(Acct A, Acct B, int amt) {
    synchronized(A) {
        synchronized(B) {
            int bal = A.getBalance();
            if (amt <= bal) {
                A.withdraw(amt);
                B.deposit(amt);
            }
        }
    }
}
Bank account (locks)

The new implementation has introduced the possibility of deadlock:

\[ \text{transfer}(A, B, 10) \parallel \text{transfer}(B, A, 20) \]

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>lock A</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>lock B</td>
</tr>
<tr>
<td>3</td>
<td>lock B</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>waiting</td>
<td>lock A</td>
</tr>
<tr>
<td>5</td>
<td>waiting</td>
<td>waiting</td>
</tr>
</tbody>
</table>
Inferring lvalues

```c
atomic {
    x = y;
    x.f = 10;
}
```
Inferring lvalues

atomic {
  x = y;
  x.f = 10;
}

x = y

x.f = 10

{}
Inferring lvalues

atomic {
    x = y;
    x.f = 10;
}

{x}

x.f = 10

{x}

{}
Inferring lvalues

atomic {
    x = y;
    x.f = 10;
}

x = y

{ y }

x.f = 10

{ x }

{}
Inferring lvalues

atomic {
    x = y;
    x.f = 10;
}

synchronized(y) {
    x = y;
    x.f = 10;
}

x.f = 10

Tuesday, 16 June 2009
Problems with iteration

How many objects accessed?

```java
atomic {
    while (n != null) {
        n = n.next;
    }
}
```
Problems with iteration

How many objects accessed?

atomic {
    while (n != null) {
        n = n.next;
    }
}

{ n, n.next, n.next.next, ... }

Sets can grow infinitely large!