Detecting Deadlock in Programs with Data-Centric Synchronization

By Daniel Marino, Christian Hammery, Julian Dolbyz, Mandana Vaziriz, Frank Tipx, Jan Vitek

Teruki Tauchi

9th April, 2014

Introduction

- Previous works
  - Java extension called AJ
    - A data-centric approach to concurrency control
    - Programmers can specify synchronization constraints declaratively
- What they presented in the paper
  - Detecting deadlocks in AJ program

AJ

```
class Tree {
  atomicset(t);
  private atomic(t) Node root|n=this.t|;
  private atomic(t) int    size = 1;
  Tree(int v)           { root=new Node|n=this.t|(v); }
  int size()            { return size; }
```

- Atomic sets associated with units of work
- atomic(t) field belongs to atomicset(t)
  - Fields with the same atomic set will have the same lock

AJ

Aliasing Annotation

```
class Node implements INode {
  atomicset(n);
  private atomic(n) Node left|n=this.n| ;
  private atomic(n) Node right|n=this.n|;
  private atomic(n) int    value, weight = 1;

  Node(int v)      { value = v; }
  int getValue() { return value; }
  void insert(int v) {
    if (value==v) weight++;
    else if (v < value) {
      if ( left==null)   left  = new Node|n=this.n|(v);
      else               left . insert(v);
    } else {
      if ( right==null) right  = new Node|n=this.n|(v);
      else              right . insert(v);
    }
  }
```

AJ Example

```
class T extends Thread {
  T(Tree t0, int v)   { tree=t0; value=v; }
  public void run() { tree.insert(value);  }
  Tree tree; int value;
}          void insert(int v)    { root.insert(v); size++; }

public static void main(String[] args) throws ... {
  Tree tree   = new Tree(10);
  Thread T1 = new T(tree, 12);
  Thread T2 = new T(tree, 5);
  T1.start (); T2.start (); T1.join (); T2.join ();
}
```

(a)

- Client code does not refer to atomic sets

AJ Example

```
class U extends Thread {
  U(Tree t1, Tree t2) { tree1=t1; tree2=t2; }
  public void run()    { tree1.copyRoot(tree2); }
  Tree tree1, tree2;
}
        void copyRoot(Tree tree) { tree.insert(root.getValue()); }

public static void main(String[] args) throws ... {
  Tree tree1 = new Tree(1), tree2 = new Tree(2);
  Thread T3 = new U(tree1, tree2);
  Thread T4 = new U(tree2, tree1);
  T3.start(); T4.start(); T3.join(); T4.join();
}
```

(b)

- Deadlock Happens!

AJ's unitfor

```
void copyRoot(unitfor(t) Tree tree){
  tree.insert(root.getValue());
}
```

- unitfor annotation make the method unit of work
- By writing this, the deadlock in the previous case is now disappeared

Deadlock Prevention

- a < b
  - threads never attempt to acquire a lock on “a” while holding a lock on “b”
- a = b
  - The same atomic-set instance
- If < is not a partial order after generating ordering constraint:
  - Possible Deadlock is reported

Desired Mutual Exclusive

- In the previous example tree, all nodes shared single lock
- However, there are methods that should be able to access concurrently on different nodes
- E.g.

```
public void incWeight(int n){ weight += n; }
```

Desired Mutual Exclusion

- Exclude Aliasing Annotations!
- But there is a problem…

```
class Tree {
  atomicset(t);
  private atomic(t) Node root;
  Tree(int v){ root = new Node(v); }
   ...
}
class Node implements INode {
  atomicset(n);
  private atomic(n) Node left;
  private atomic(n) Node right;
   ...
  void insert(int v){
     ...  left = new Node(v); ...
     ...  right = new Node(v); ...
  }
}
```

Fixing issue

```
class Node implements INode {
  atomicset(n);
  private atomic(n) Node left|this.n<n|;
  private atomic(n) Node right|this.n<n|;
   ...
  void insert(int v){
     ...  left  = new Node|this.n<n|(v); ...
     ...  right = new Node|this.n<n|(v); ...
  }
}
```

- Now it is ordered from top to bottom

Algorithms

$$\begin{array}{lll} \mathcal{M} & := & \textit{set of methods in program} \\ \mathcal{V} & := & \textit{set of final method params plus a special ? symbol} \\ \mathcal{A} & := & \textit{set of atomic sets} \end{array}$$

$$\begin{array}{lll} \mathcal{N} & := & \{=,<\} \times \mathcal{V} \times \mathcal{A} \ \textit{set of lock identifiers} \\ \mathcal{L} & := & 2^{\mathcal{N}} \ \textit{set of atomic-set instances (i.e., locks)} \\ \mathcal{D} & := & 2^{\mathcal{L}} \ \textit{set of locksets} \end{array}$$

$$\begin{array}{lll} \mathtt{uow} : \mathcal{M} \to \mathcal{D} & := & \textit{returns the set of locks that a method grabs} \\ \mathtt{padaptName} : (\mathcal{M} \times \mathcal{V} \times \mathcal{M}) \to \mathcal{V} & := & \textit{renames a variable from the perspective of caller to callee} \\ \mathtt{padaptLock} : (\mathcal{M} \times \mathcal{L} \times \mathcal{M}) \to \mathcal{L} & := & \textit{adapts all names identifying a lock from the perspective of caller to callee} \\ \mathtt{addNames} : (\mathcal{M} \times \mathcal{L}) \to \mathcal{L} & := & \textit{consults annotations in scope to add other names for a lock to its representation.} \end{array}$$

$$\mathtt{uow}(m) = \{ \{ v.A \} | \ m \text{ is a unit-of-work for } v.A \}$$

$$\mathtt{addNames}(m, l) = l \cup \{ v.A \mid w.B \in l \text{ and } v.A \text{ is annotated to be an alias for } w.B \text{ in } m\text{'s scope} \}$$

$$\mathtt{padaptName}(m_s, v, m_t) = \begin{cases} \text{this} & \text{if } m_s \text{ contains the call } v.m_t(\ldots) \\ w & \text{if } m_s \text{ passes } v \text{ as the actual argument for the formal parameter } w \text{ of } m_t \\ ? & \text{otherwise} \end{cases}$$

$$\mathtt{padaptLock}(m_s, l, m_t) = \{ *v.A | \ *w.A \in \mathtt{addNames}(m_s, l) \wedge \mathtt{padaptName}(m_s, w, m_t) = v \}$$

$$\frac{m \text{ is an entry point}}{\emptyset \in \mathit{LBE}(m)} \ (\text{LBE-ENTRY}) \qquad \frac{n \to m \qquad d \in \mathit{LBE}(n)}{\{ \mathtt{padaptLock}(n, l, m) | \ l \in (d \cup \mathtt{uow}(n)) \} \in \mathit{LBE}(m)} \ (\text{LBE-CALL})$$

Implementation

- Implemented deadlock analysis as an extension of existing AJ-to-Java compiler
- Constructing on call graph relies on WALA framework

Result

	Ordering annotations	|locksets|	Time [s]
elevator	0	39	1.0
tsp	0	33	1.4
weblech	0	39	4.6
jcurzez1	0	409	10.3
jcurzez2	4	541	9.4
tuplesoup	0	785	8.8
cewolf	0	25	19.7
mailpuccino	0	205	48.2
jphonelite	0	34	7.2
specjbb	0	414	75.1

Result

- Out of 10 programmes
  - All were shown to be deadlock free
  - One needed 4 ordering annotations
  - Two needed minor refactorings
  - Remaining 7 programmes needed no programmer intervention of any kind
- At most 75 seconds running time
  - MacBook Air Core i5 1.8GHz, 4GB RAM