Statically Computing Upper Bounds on Object Calls for Atomic RMI

EC^2 2010

Konrad Siek, Paweł Wojciechowski

20 July 2010

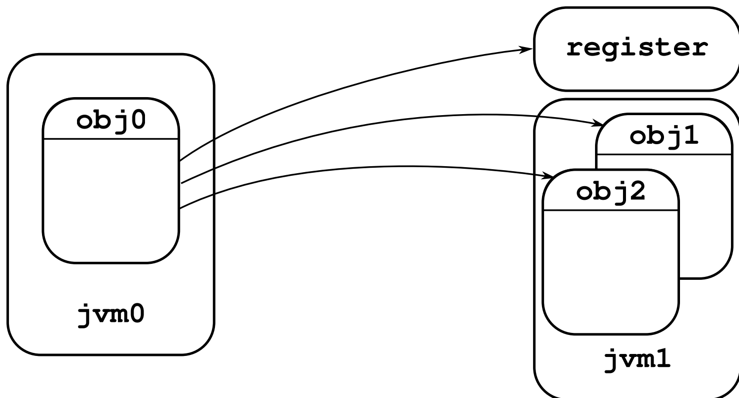
1. Introduction
2. Concurrency control
3. Finding upper bounds
4. Code generation
5. Conclusions

Introduction

- The emergence of multi-core CPUs
- A need for concurrency and distribution in programming
- Low-level mechanisms are notoriously difficult
- Some new approaches: transactions

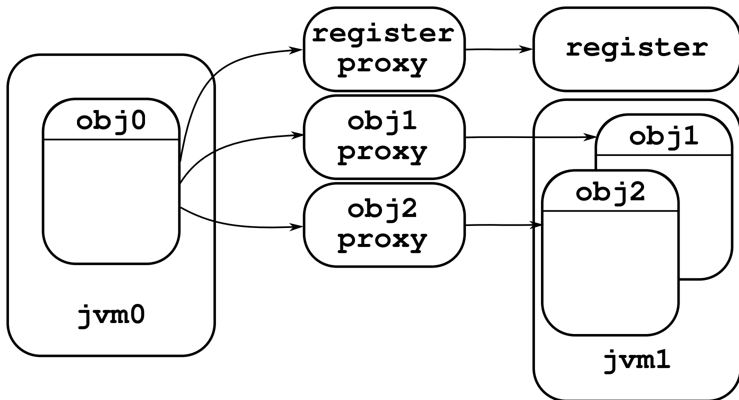
Java RMI

Allows objects in one JVM to use methods of objects in another JVM (remote objects).



Atomic RMI

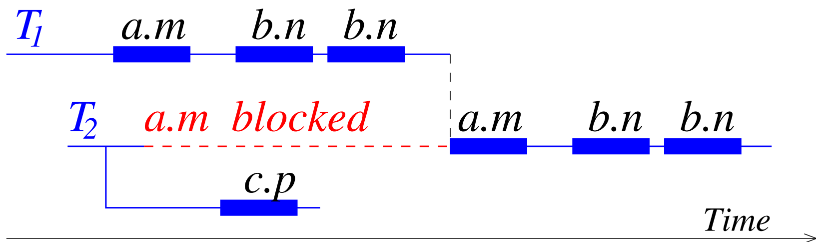
A pessimistic transaction-based concurrency control library built on top of Java RMI.



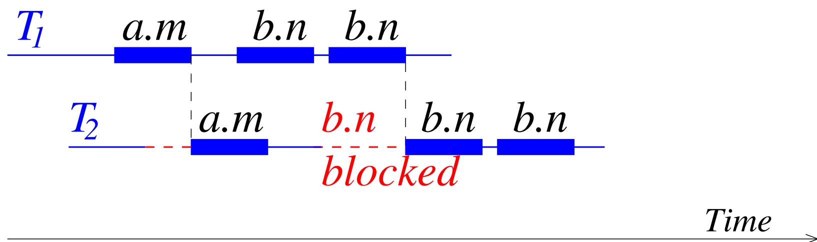
Versioning algorithms

Critical operations of concurrent tasks with respect to versions (global and local counters).

Basic Versioning Algorithm



Supremum Versioning Algorithm



Drawback

Versioning algorithms need information about which objects are used and how many times:

- BVA—identification of used objects
- SVA—quantitive object use information

This needs to be done by hand.

```java
AtomicTaskManager taskManager = AtomicTaskManagerService.lookupTaskManager(managerHost);
Registry registry = VersionedLocateRegistry.getRegistry(taskManager);

BankAccount accountA = (BankAccount) registry.lookup("accountA");
BankAccount accountB = (BankAccount) registry.lookup("accountB");

TaskDescription taskDescription = new TaskDescription();
taskDescription.addObjectAccess(accountA, 2);
taskDescription.addObjectAccess(accountB, 2);

RmiAtomicTask task = new RmiAtomicTask(taskManager);
task.startTask(taskDescription);

int valueA = accountA.getValue();
int valueB = accountB.getValue();

accountA.setValue(valueA - 200);
accountB.setValue(valueB + 200);

System.err.println("Finished transfer from A to B.");

task.endTask();
```

Thus. . .

A tool to extract that information and include it in the source code.

Properties

What we expect:

- All object identified and all calls counted
- Call count not lower than through any execution
- The more accurate the better
- Analysis terminates

Object Call Count Analysis

- Analyze the source code
- Analyze each instruction in a method
- Register expected effects for all simple instructions
- Recursively enter all complex instructions...

Example

```
task.startTask(taskDescription);

Random random = new Random();

bankAccountA.withdraw(200);     //A: 1

if (random.nextBoolean()) {
    bankAccountA.deposit(200); //A: 1 or 2
} else {
    bankAccountB.deposit(200); //B: maybe 1
}

Account bankAccountC = bankAcountB;
while (random.nextBoolean()) {
    bankAccountC.withdraw(20); //B: unknown
}

task.endTask();
```

Limitations

- Support for only a small subset of Java
- Lost precision with loops and recursion

Code generation

Once the upper bounds are known, the preambles need to be generated. . .

Where to put the preambles

Transactions and remote objects are identified in the source code—objects implementing specific interfaces, particular method calls.

```
AtomicTaskManager taskManager = AtomicTaskManagerService.lookupTaskManager(managerHost);
Registry registry = VersionedLocateRegistry.getRegistry(taskManager);

BankAccount accountA = (BankAccount) registry.lookup("accountA");
BankAccount accountB = (BankAccount) registry.lookup("accountB");

TaskDescription taskDescription = new TaskDescription();
taskDescription.addObjectAccess(accountA, 2);
taskDescription.addObjectAccess(accountB, 2);

RmiAtomicTask task = new RmiAtomicTask(taskManager);
task.startTask(taskDescription);
```

Injecting tokens

The source code is read on the lexical lexel: appropriate tokens are generated and inserted into the stream.

Conclusions

Results:

- Atomic RMI: transactions
- code analyzer: easier use of versioning algorithms

Questions

?