

DeepWeaver: a Declarative Framework for Analysis and Optimization

Paul Kelly

Software Performance Optimisation Group

Imperial College London

<http://www.doc.ic.ac.uk/~phjk>

Joint work with

Michael Mellor, Henry Falconer, David Ingram, Olav Beckmann and
Tony Field

See our paper at Compiler Construction, next week in Braga

London, March 2007

Where I'm coming from...



- I lead the Software Performance Optimisation group in the Department of Computing at Imperial College London
- Work I'd love to talk about another time:
 - “Active”, self-optimising libraries
 - GPU acceleration of image processing libraries for film and video
 - Efficient algorithms for scalable pointer alias analysis (now in GCC)
 - Bounds-checking for C, links with unchecked code (patch for GCC)
 - Runtime code generation tools for high-performance computing
 - Morton storage layout for 2D arrays

Mission statement

3

- Extensible optimising compiler technology
- Extensible performance profiling tools
- To tackle challenging contexts:
 - Dynamically-configured software
 - Cross-component optimisation
 - Across network boundaries
 - Between different security domains
 - Domain-specific optimisations

Mission - delivery

4

To deliver novel, aggressive tools, we need to be able to market and deploy them as **independent pluggable components**

- Selectively, to tackle particular domains – eg particular libraries or frameworks
- Electively: pick and choose the tools that help

This talk presents some of our experience in doing this:

- Profiling using AspectJ
- A more powerful aspect-like language
- Example optimisation components:
 - ◆ JDBC – selecting only columns actually used
 - ◆ Java RMI – aggregating calls

■ Google “Java performance tips”

- “Use StringBuffer not string concat operator (+)”
- “Inserting and deleting elements to Vectors and ArrayLists always require an array copy”
- “Avoid retrieving unnecessary columns: don't use ‘SELECT *’ ”.
- “Use JDBC’s prepared statements”
- “Cache data when reuse is likely”
- “To add many items to a JComboBox, add them in one go. This generates only one changed event”
- Never let a DBMS transaction span user input
-

■ Lots of advice – much specific to particular APIs and application frameworks

■ How can a tool/technology tell you what you need to know?

Example – JDBC prepared statements

6

- Without prepared statement:

```
for (int i=0; i<10;i++)  
    stmt.execute("UPDATE Order "+  
                "SET price = price * 1.10"+  
                "WHERE quantity = "+i);
```

Database sees A.length different SQL statements

- With prepared statement:

```
PreparedStatement pstmt = conn.prepareStatement(  
    "UPDATE Order "+  
    "SET price = price * 1.10"+  
    "WHERE quantity = ?");
```

```
for (int i=0; i<10;i++){  
    pstmt.setInt(1, i);  
    pstmt.execute();  
}
```

- Database sees one statement invoked with A.length different parameter values

Using aspects...

- How can we find opportunities to use prepared statements automatically?
- Use AspectJ:

```
public aspect SQLExecuteLogging {  
    pointcut SQLExecute( String sql ) :  
        call ( public * Statement.execute*(String) &&  
            args(sql);  
  
    after(String sql) : SQLExecute(sql){  
        System.out.println(StackAnalyser.  
            sourceFileLocation(2)+":"+sql);  
    }  
}
```

- AspectJ compiler “weaves” code in during build
- Inserts Java fragment after each program point that matches the “pointcut”

Using aspects...

- How can we find opportunities to use prepared statements automatically?
- Use AspectJ:

```
public aspect SQLExecuteLogging {
    pointcut SQLExecute( String sql ) :
        call ( public * Statement.execute*(String) &&
            args(sql);

    after(String sql) : SQLExecute(sql){
        System.out.println(StackAnalyser.
            sourceFileLocation(2)+":"+sql);
    }
}
```

■ Match any Statement method with name beginning with "execute" and a String parameter

Using aspects...

- How can we find opportunities to use prepared statements automatically?
- Use AspectJ:

```
public aspect SQLExecuteLogging {  
    pointcut SQLExecute( String sql ) :  
        call ( public * Statement.execute*(String) &&  
            args(sql);  
  
    after(String sql) : SQLExecute(sql){  
        System.out.println(StackAnalyser.  
            sourceFileLocation(2)+":"+sql);  
    }  
}
```

Call its parameter "sql"

After the Statement.execute*(sql) call has taken place, ...

Print a log of sql string and source code location that issued it

Example – JDBC prepared statements

10

■ Without prepared statement:

```
for (int i=0; i<10;i++)  
    stmt.execute("UPDATE Order "+  
        "SET price = price * 1.10"+  
        "WHERE quantity = "+i);
```

- Database sees 10 different SQL statements

■ Output from logging aspect:

```
Main.java\main\90:UPDATE Order SET price = price * 1.10 WHERE quantity = 0  
Main.java\main\90:UPDATE Order SET price = price * 1.10 WHERE quantity = 1  
Main.java\main\90:UPDATE Order SET price = price * 1.10 WHERE quantity = 2  
Main.java\main\90:UPDATE Order SET price = price * 1.10 WHERE quantity = 3
```

- With just a little more work we can diff the sql from each line and highlight where parameterisation would be effective

- Using AspectJ encourages experimental profiling ideas:
 - Simple profiling tasks are a page of AspectJ
 - No need for JVM support (JVMPI, JVMTI)
 - No bytecode-level hacking – it's hard to produce broken bytecode in AspectJ
- Not just profiling
 - Some optimisations, such as caching, are easy
- AspectJ has some shortcomings:
 - Can't weave system libraries at load-time
 - Only inserts code before, after or instead of method calls
 - No concept of intra-method control flow, synchronizations, loops
- Next example introduces our work on fixing some of these problems...

```
aspect preparedStatements {
  weave executeStatementInLoop(CodeBlock Loop, CodeBlock ExecCall):

  // Query part – in Prolog

  method("* Statement.execute*(String)", ExecMethod),
  call(ExecMethod, _, ExecCall, _),
  loop(Loop),
  encloses(Loop, ExecCall).

  // “Action” part – in Java
  {
    System.out.println("JDBC execute occurs in loop at "+ExecCall.location())
    System.out.println("consider preparedStatement at "+Loop.location());
  }
}
```

- At weave-time, Action part is executed for each Query result

```
aspect preparedStatements {  
  weave executeStatementInLoop(CodeBlock Loop, CodeBlock ExecCall):  
  // Query part – in Prolog  
  method("* Statement.execute*(String)", ExecMethod),  
  call(ExecMethod, _, ExecCall, _),  
  loop(Loop),  
  encloses(Loop, ExecCall).  
  // "Action" part – in Java  
  {  
    System.out.println("JDBC execute occurs in loop at "+ExecCall.location());  
    System.out.println("consider preparedStatement at "+Loop.location());  
  }  
}
```

Results from Query, passed to Action

Find methods with this signature

Find a call to one

Find a loop

Succeed if the call is in the loop

At weave-time, weaver prints advisory message:

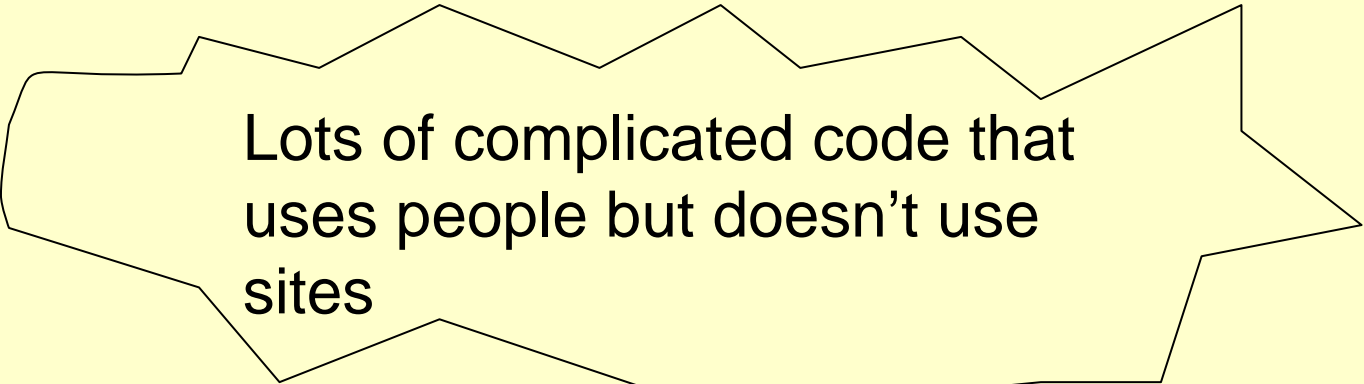
JDBC execute*() occurs in loop at 12 - 14

consider preparedStatement at 12 - 14

Another example: unused results

14

```
public void example()
{
    ResultSet people = stmt.executeQuery("SELECT * FROM Personnel");
    ResultSet sites = stmt.executeQuery("SELECT * FROM Sites");
```



Lots of complicated code that
uses people but doesn't use
sites

- Find JDBC query results which are requested but never actually used

Another example: unused results

15

```
weave removeRedundantSelect(CodeBlock ExecCall):  
  method("ResultSet Statement.executeQuery(String)",  
        ExecMethod),  
  call(ExecMethod, _, ExecCall, _),  
  assignment(Result, ExecCall, _),  
  \+ reaching_def(Result, Use, _).  
{  
  System.out.println("Removing unused call");  
  ExecCall.remove();  
}
```

- This weave matches when no use of the result returned from “executeQuery” can be found (“\+” is the Prolog “not” operator)
- In this case the Java action issues a warning and also removes the redundant call automatically
 - (Strictly, we should also check that the query could not result in any side-effects – eg due to stored procedures)

The “select *” performance anti-pattern

16

```
ResultSet r = s.execute(“select * from test”);
while ( r.next() ) {
    String col1 = r.getString(1); // Get column 1
}
```

JDBC driver is being asked to fetch too much:

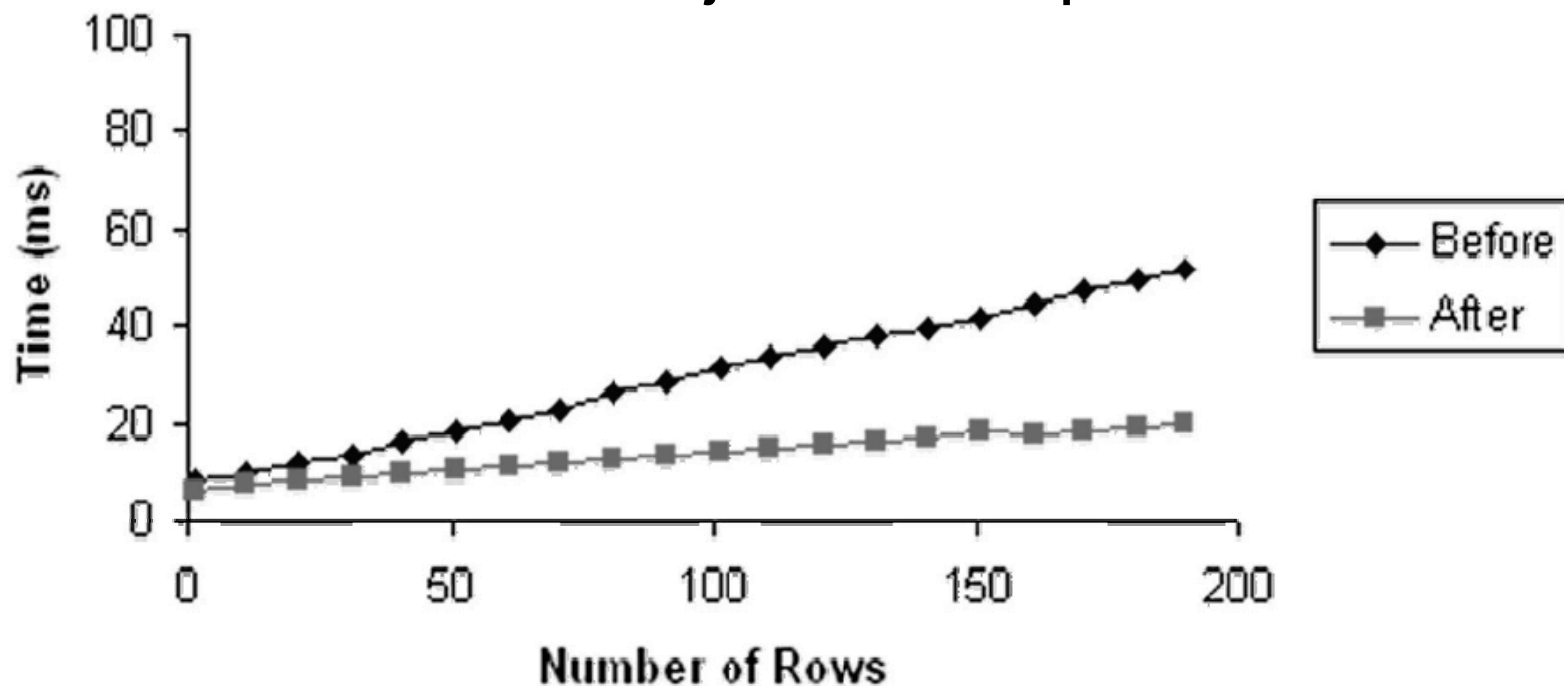
- Eg: 10-column table
- “select *” retrieves all ten columns
- Loop only actually uses one column

- Widely advocated in introductory JDBC tutorials
- Avoids unnecessary coupling between query and use

The “select *” performance anti-pattern

```
ResultSet r = s.execute(“select col1 from test”);  
while ( r.next() ) {  
    String col1 = r.getString(1); // Get column 1  
}
```

- Rewrite it to select just the required column:

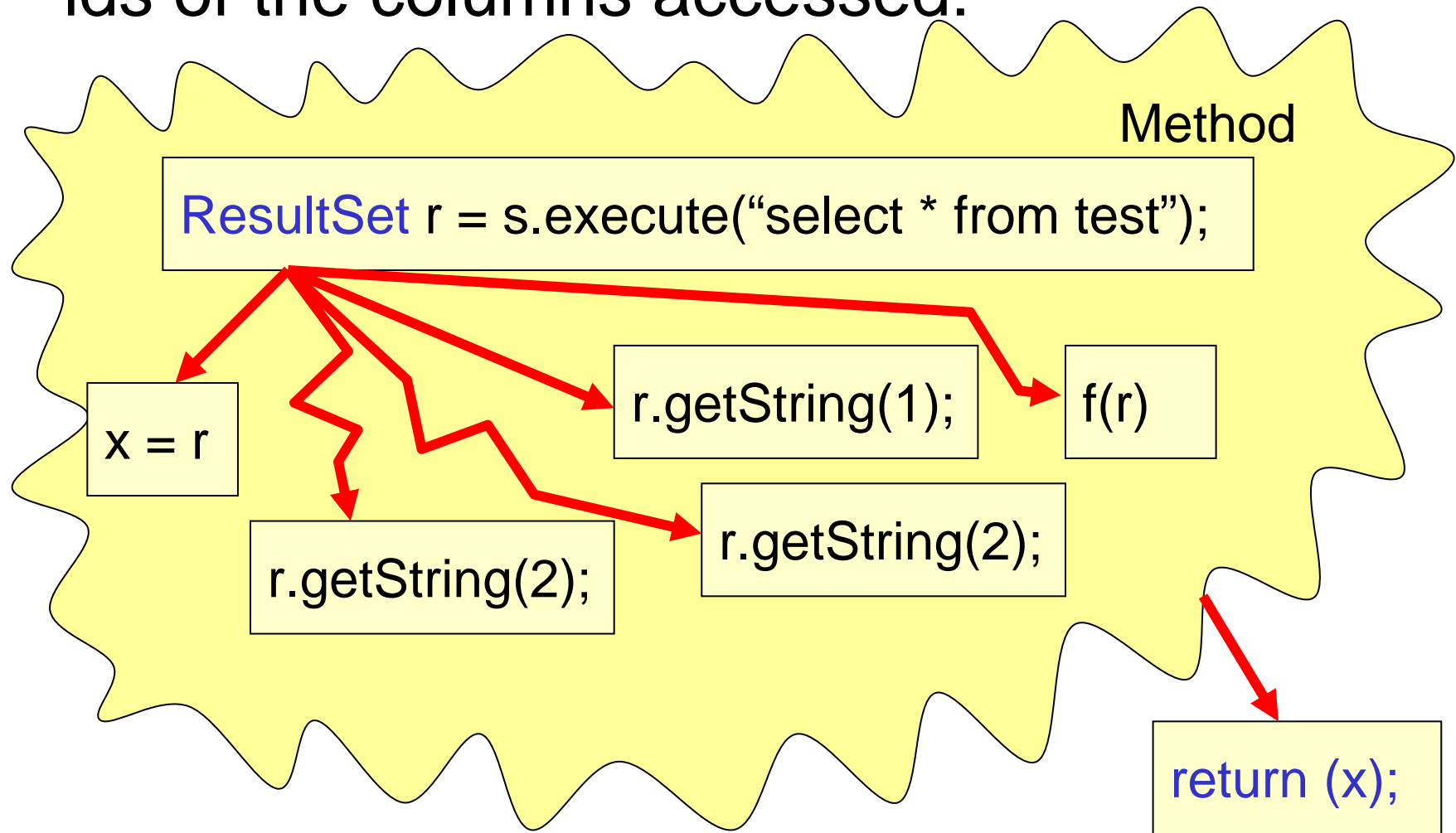


- More than twice as fast (MS SQL Server)

Fixing “Select *” using DeepWeaver

18

- For each call to executeQuery, find where the result is used, and collect the ids of the columns accessed:



Fixing “Select *” using DeepWeaver

19

- Prolog predicate "columnUsed" finds definitions of the Strings passed to get methods used to retrieve values from the ResultSet “Target”:

```
predicate columnUsed(CodeBlock Target, CodeBlock Column):
```

```
// Find the program points where Target is used
```

```
reaching_def(Target, Use, false),
```

```
// Check that the use is the subject of a “get” method
```

```
call("* ResultSet.get*(String)", Use, Location, ColumnArgs),
```


```
encloses(Location, Use),
```

```
// Get the parameter value, ie the column name
```

```
member(ColumnArg, ColumnArgs),
```

```
local_constant_def(ColumnArg, Column).
```

Fixing “Select *” using DeepWeaver

```
weave refineSelectQuery(CodeBlock QueryString, List Columns):  
  // Find JDBC execute() call site  
  method("ResultSet Statement.executeQuery(String)", ExecMeth),  
  call(ExecMeth, _, ExecCall, QueryStringLocal),  
  
  // Find the query string (from the method's constant pool)  
  member(QSLocal, QueryStringLocal),  
  local_constant_def(QSLocal, QueryString),  
  
  // Find variable to which result is assigned  
  assignment(Target, ExecCall, _),  
  
  // Collect set of Column strings used to access the ResultSet  
  findall( Column, columnUsed(Target, Column), Columns ).  
{  
  // (Assume for brevity that it is a SELECT * and Columns is not empty)  
  // Create new query String from list of Columns to select  
  String newQuery = makeNewQuery(QueryString, Columns);  
                       (defined elsewhere, in Java)  
  // Replace existing string constant with new query string  
  QueryString.replaceValue(StringConstant.v(newQuery));  
}
```

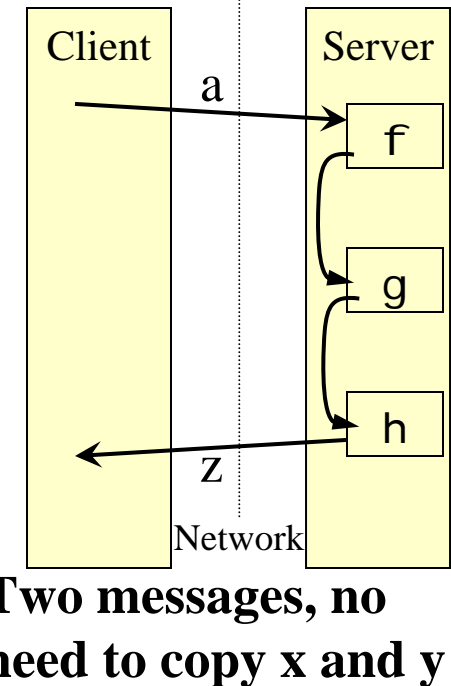
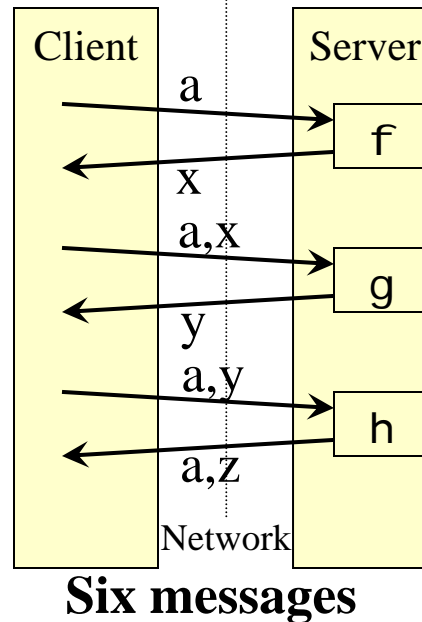
Select * example - summary

21

- Data-flow analysis, looking inside method
- Power of Prolog as query language
 - to collect set of parameters of get methods applied to result set returned by call to execute method
- Independent of syntactic structure of the method's source code

Aggregating Java RMI - Motivation

```
void m(RemoteObject r, int a)
{
    int x = r.f(a);
    int y = r.g(a,x);
    int z = r.h(a,y);
    System.out.println(z);
}
```



● Aggregation

- ✿ A sequence of calls to same server can be executed in a single message exchange
- ✿ Reduce number of messages
- ✿ Also reduce amount of data transferred
 - Common parameters
 - Results passed from one call to another

Aggregating Java RMI - Motivation

- Simple example: Multi-user Dungeon (from Flanagan's Java Examples in a Nutshell)
- "Look" method:

```
String mudname = p.getServer().getMudName();  
String placename = p.getPlaceName();  
String description = p.getDescription();  
Vector things = p.getThings();  
Vector names = p.getNames();  
Vector exits = p.getExits();
```

- Seven aggregatable calls

Time taken to execute "look":

Without call aggregation:

With call aggregation:

Speedup (earlier implementation):

Slow ADSL

759.6ms

164.9ms

4.61 (Yeung&Kelly Middleware 03)

Client: Athlon XP 1800+
Servers: Pentium III 500MHz, 650MHz and dual 700MHz
Linux, Sun JDK 1.4.1_01 (Hotspot)
Network: Ethernet: 10.03 MB/s, ping 0.1ms
DSL: 10.7KB/s, ping 98ms
Mean of 3 trials of 1000 iterations each

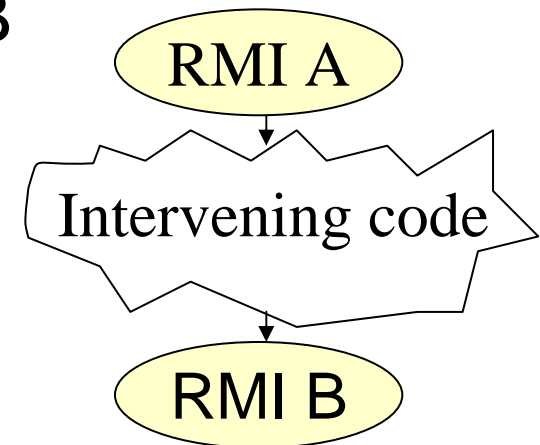
```
x = r.m1(a,b);           // first remote call: RMI A
q.f = 0;                // some other non-remote code
y = r.m2(a,c,x);        // second remote call: RMI B
System.out.println(r);
```

- Suppose “x1.m1(p)” throws an exception
- In aggregated implementation, exception will be delayed until after “q.f = 0”
- Does this matter?
 - Yes!
 - But only if the object q points to “escapes” this thread or is live in, or after, the catch clause
- Any externally-visible effect between the RMI calls blocks aggregation

- A pair of RMI calls A and B can be aggregated by delaying execution of call A until B

- This is valid when:

- All paths to B go through A:
 - ◆ $\text{dominates}(A,B)$
- All paths from A go through B:
 - ◆ $\text{post_dominates}(B,A)$
- If A or B is in a loop they're both in the same one
- A's result is not used before B
- Assignments on paths between A and B do not have externally-visible effects
- No method or constructor calls can occur between A and B (except calls known to be side-effect free)



- You can define a DeepWeaver predicate to test these conditions in about a page
 - (Though a good definition of “externally-visible” is more work)

predicate isAggregatableRMIPair(CodeBlock CallA, CodeBlock CallB,
List ParamA, List ParamB, CodeBlock ResultOfA):

// A and B are distinct RMI calls, and A precedes B:

```
call(RemoteMethodA, RemoteObjectA, CallA, ParamA),  
type(RemoteObjectA, "java.rmi.Remote"),  
precedes(CallA, CallB), CallA \= CallB,  
call(RemoteMethodB, RemoteObjectB, CallB, ParamB),  
type(RemoteObjectB, "java.rmi.Remote"),  
dominates(CallA, CallB), post_dominates(CallB, CallA),
```

// If A or B is in a loop they're both in the same one:

```
forall ( loop(Loop),  
  ( ( encloses(Loop, CallA), encloses(Loop, CallB)  
    ); // OR  
    (\+ encloses(Loop, CallA), \+ encloses(Loop, CallB)  
    ) ) ),
```

// A's result is not used before B

```
assignment(ResultOfA, CallA, _),  
\+ ( reaching_def(ResultOfA, UseOfResult, _),  
    precedes(UseOfResult, CallB)  
),
```

// Assignments between A and B do not have externally-visible effects:

```
\+ ( between(CallA, OnPath, CallB, false),  
    assignment(Lhs, Rhs, OnPath),  
    externally_visible(Lhs)  
),
```

// No method or constructor calls can occur between call A and call B:

```
\+ ( between(CallA, Location, CallB, false),  
    call(Method, Target, Location, Params),  
    \+ side_effect_free_method(Method) ).
```

Aggregating Java RMI:

Validity predicate

- Then the action is another couple of pages of code:

```
weave rmiResultForwarding(  
    CodeBlock CallA, CodeBlock CallB,  
    List ParamA, List ParamB,  
    CodeBlock ResultA, CodeBlock ResultB )  
isAggregatableRMIPair(CallA, CallB,  
    ParamA, ParamB, ResultA, ResultB).
```

{

Remove call A.

Create new server method, to implement the aggregated call.

Insert it into the callee class.

Replace call B with call to the new aggregated method.

}

- Interestingly, this weave modifies both the client method and the callee class

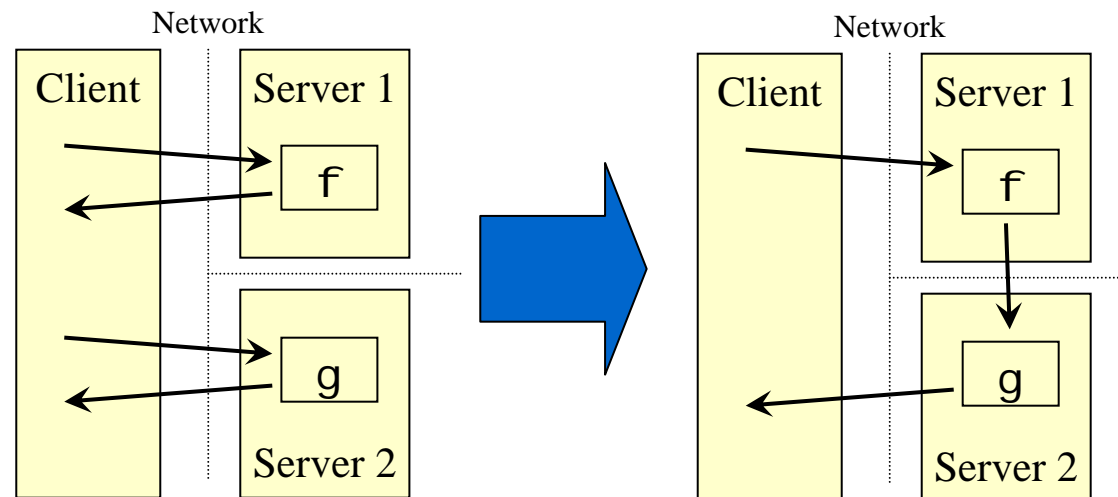
RMI aggregation - summary

- Simple example of optimisation *across* a distributed system
- There is one more validity issue: callbacks – see our Middleware paper (or Ahern&Yoshida’s OOPSLA06 analysis)
- Extensions:

- RMI forwarding:

```
void m(RemoteObject r1,
      RemoteObject r2)
{
  Object a = r1.f();
  r2.g(a);
}
```

=



(security issues?)

- Potential:
 - Whole-system analysis: caller and callee
 - Messages, event-driven architectures

- Vision:
 - Create open framework to support a marketplace of analysis and transformation components
 - Components for profilers
 - Components in the build chain
 - Components at load time, at run time...
- Aspect-oriented programming (AOP) is natural for this: “find all program points matching this pattern...”
 - Classical AOP tools like AspectJ can dramatically simplify instrumentation tasks
 - AspectJ makes simple instrumentation tasks quick and safe – encouraging developers to explore
- A more powerful weaver can deliver really interesting optimisations
 - In a lightweight form, with low barriers to adoption

Nearly last slide

■ Vision:

- Create open framework to support a marketplace of analysis and transformation components
- For diverse purposes:
 - ▶ Static debugging
 - Catch undesirable code patterns (“anti-patterns”)
 - ▶ Components and services with common sense
 - Software that checks it’s been plugged together right
 - ▶ “Active” libraries
 - That play an active role in optimising or validating the way they are called
 - ▶ Framework-specific reasoning
 - Analyses that use information about how software configuration
 - To track control flow through distributed systems
 - To track control flow through callback, observer and visitor patterns (to overcome the infamous “control inversion”)

■ Extend DeepWeaver

- Integrate ideas/tools from similar projects (Domo/WALA, bddb, PQL, JunGL, Codequest...)
- Custom program analyses
- Query optimisation
- More static safety for code transformations

■ Static and dynamic analyses

- You can use DeepWeaver queries to advise developers of potential problems at build time
- You can use DeepWeaver to deploy instrumentation to monitor for potential problems at run-time
 - ◆ During testing
 - ◆ In service, generating events for generic systems performance tools
- Test for optimisation validity at run-time
 - ◆ Just when the benefits will likely outweigh costs

■ Component metadata, service models

- Components packaged with associated DeepWeaver implementations of performance expertise
- Not just performance...

■ EPSRC:

- PhD studentships (Pearce, Yeung)
- Grants (GR/R21486, GR/R15566, EP/E002412)

■ IBM:

- Eclipse Innovation Grant
- Faculty Award
- Internship for PhD student David Pearce

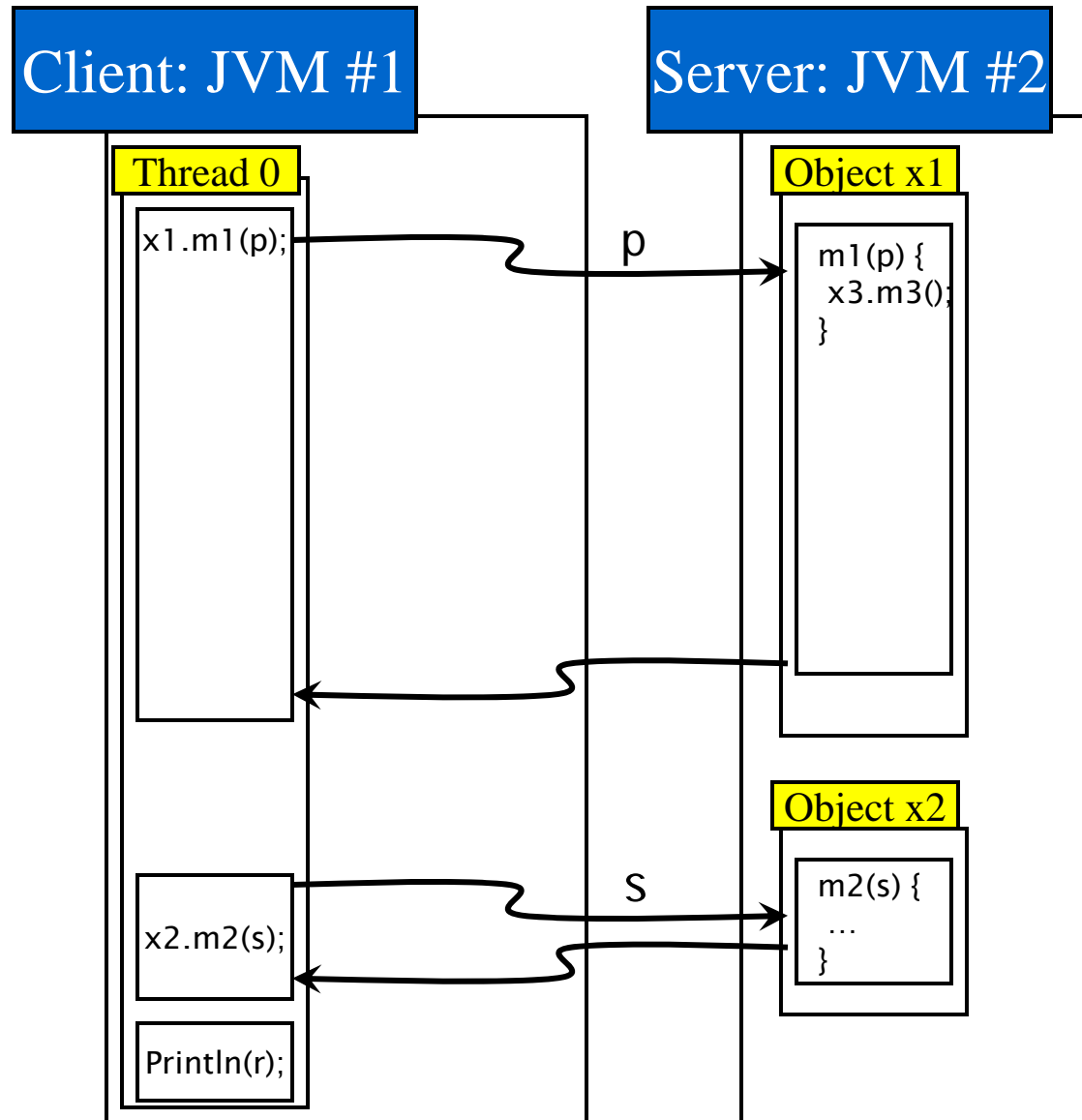
Publications

- **Profiling with AspectJ.** David J Pearce, Matthew Webster, Robert Berry, Paul H J Kelly. Software Practice and Experience, to appear (2006)
- **"Optimizing Java RMI programs by communication restructuring".** Kwok Cheung Yeung and Paul H J Kelly, Middleware 2003
- **"Formalising Java RMI with Explicit Code Mobility".** Alexander Ahern and Nobuko Yoshida, OOPSLA'05 (formal analysis of correctness of our RMI aggregation)
- **"DeepWeaver-1: an extensible declarative framework for program analysis and transformation".** Henry Falconer, Paul H J Kelly, David M Ingram, Michael R Mellor, Tony Field and Olav Beckmann. Compiler Construction 2007.

Extra slides for questions

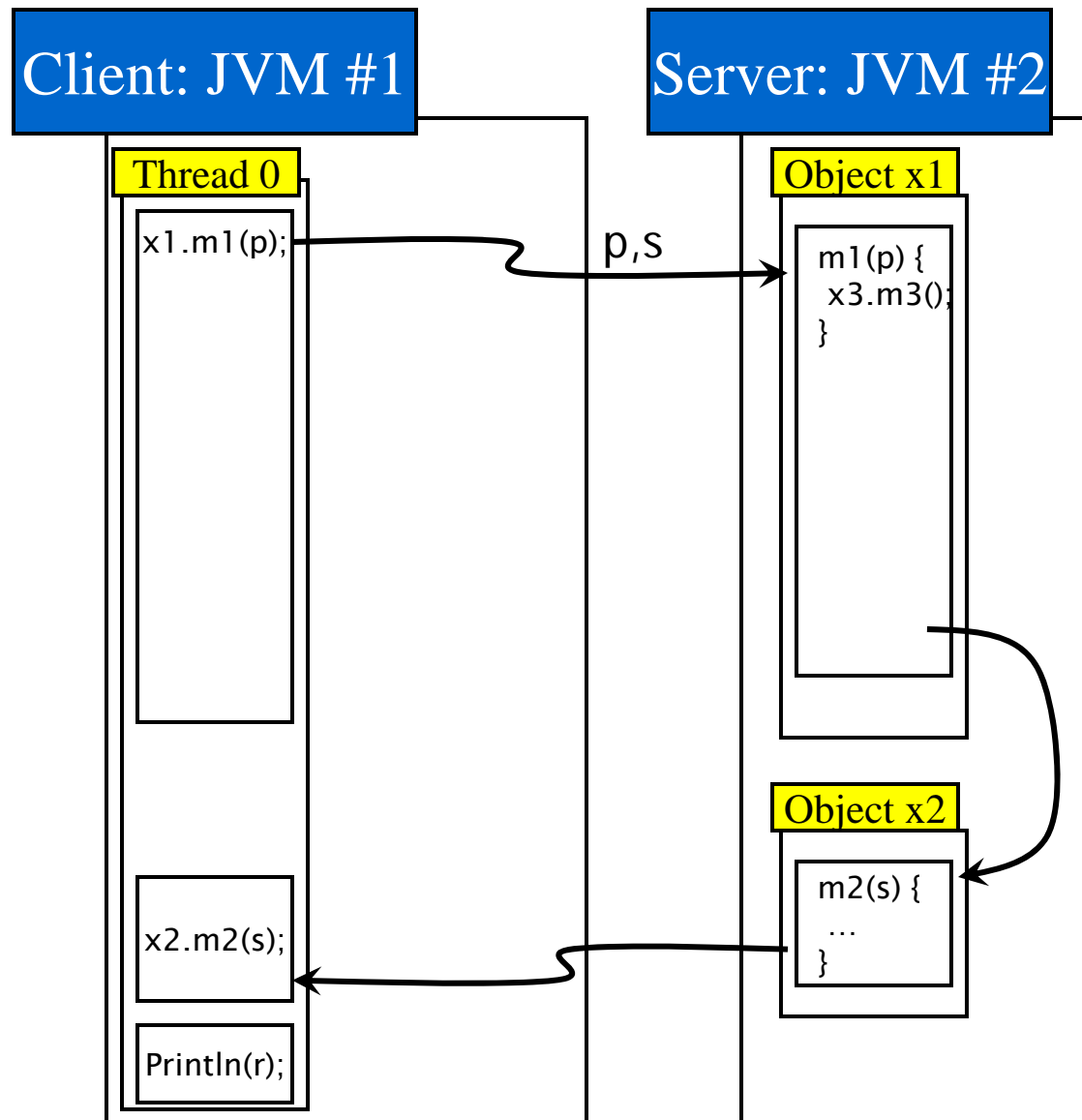
34

Validity of aggregation...



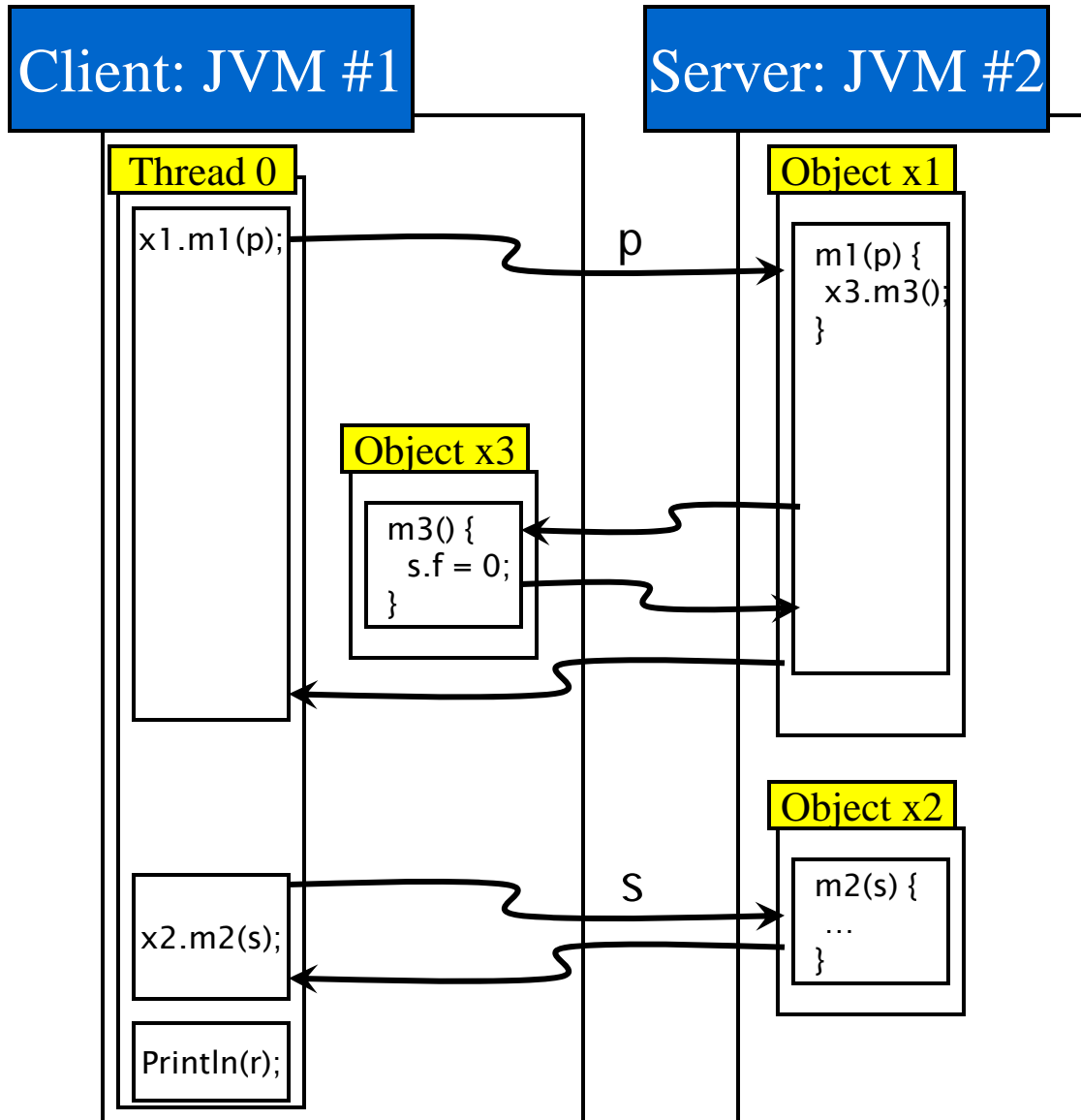
- Unoptimised code – sequence of events
- We have considered only the client code

Validity of aggregation...



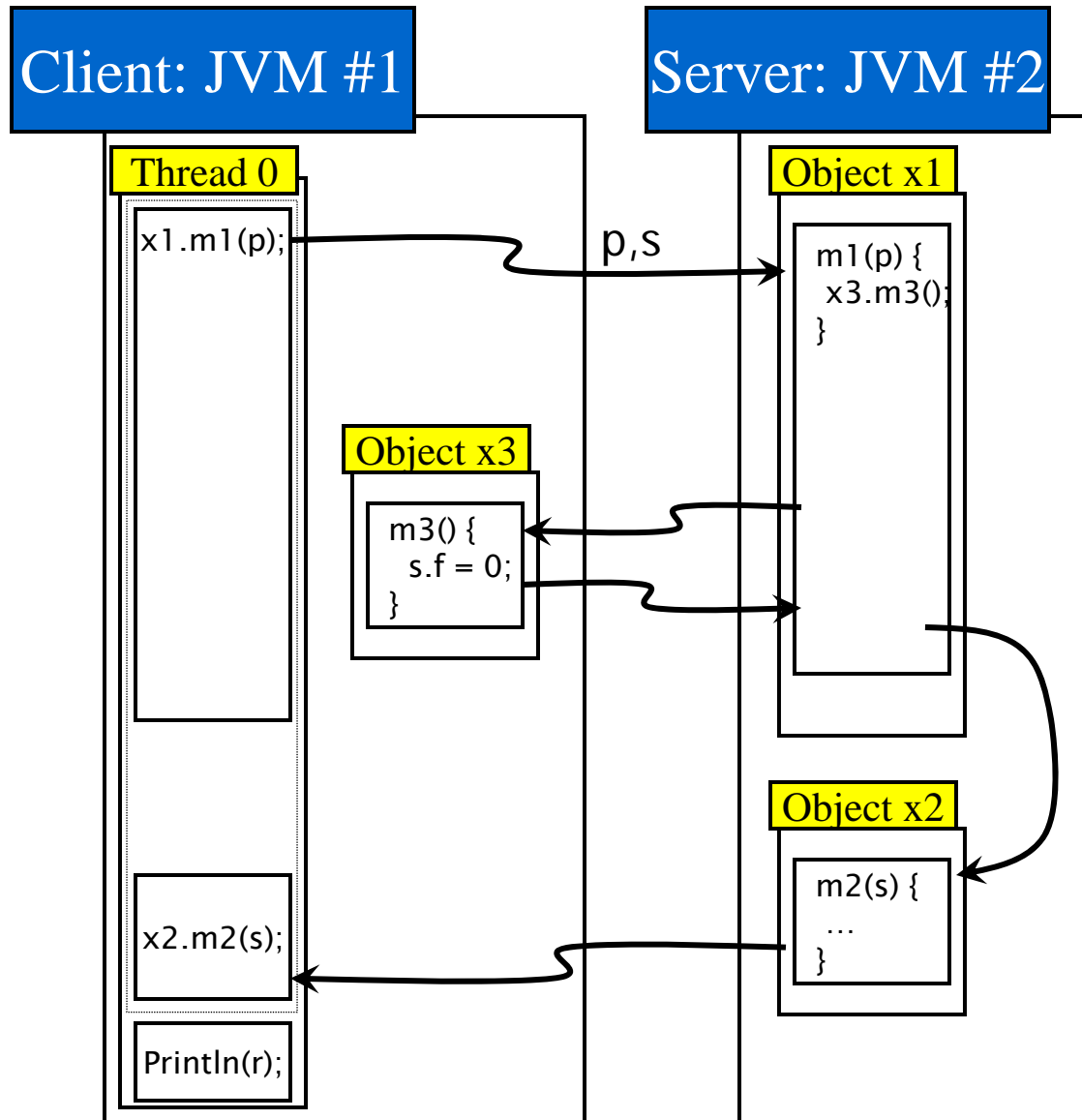
- Optimised code – sequence of events
- We have considered only the client code
- Does it make any difference what the server method does?

Aggregated call involves a call back to the client, which changes one of the parameters?



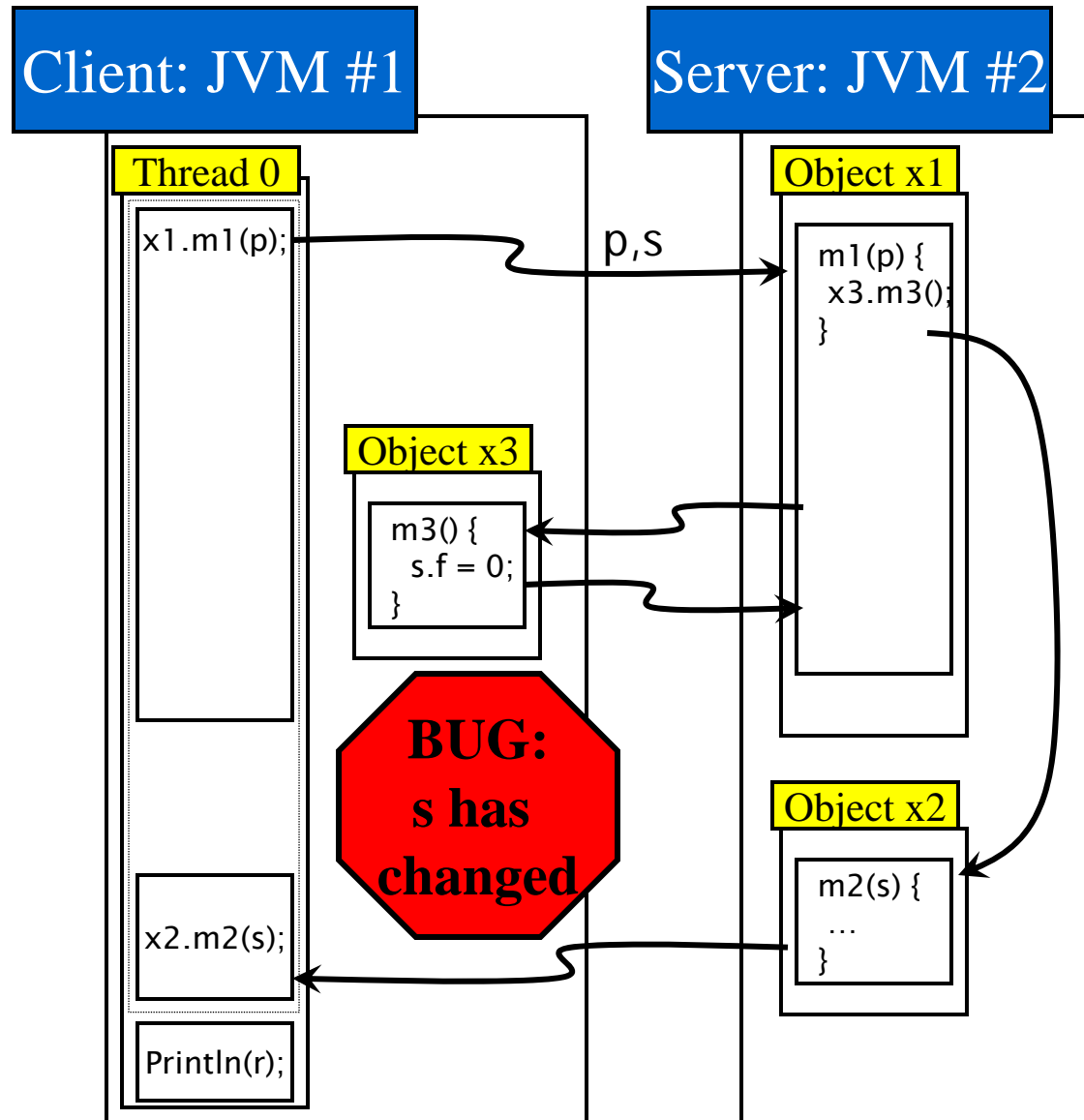
- Unoptimised code – sequence of events

Aggregated call involves a call back to the client, which changes one of the parameters?



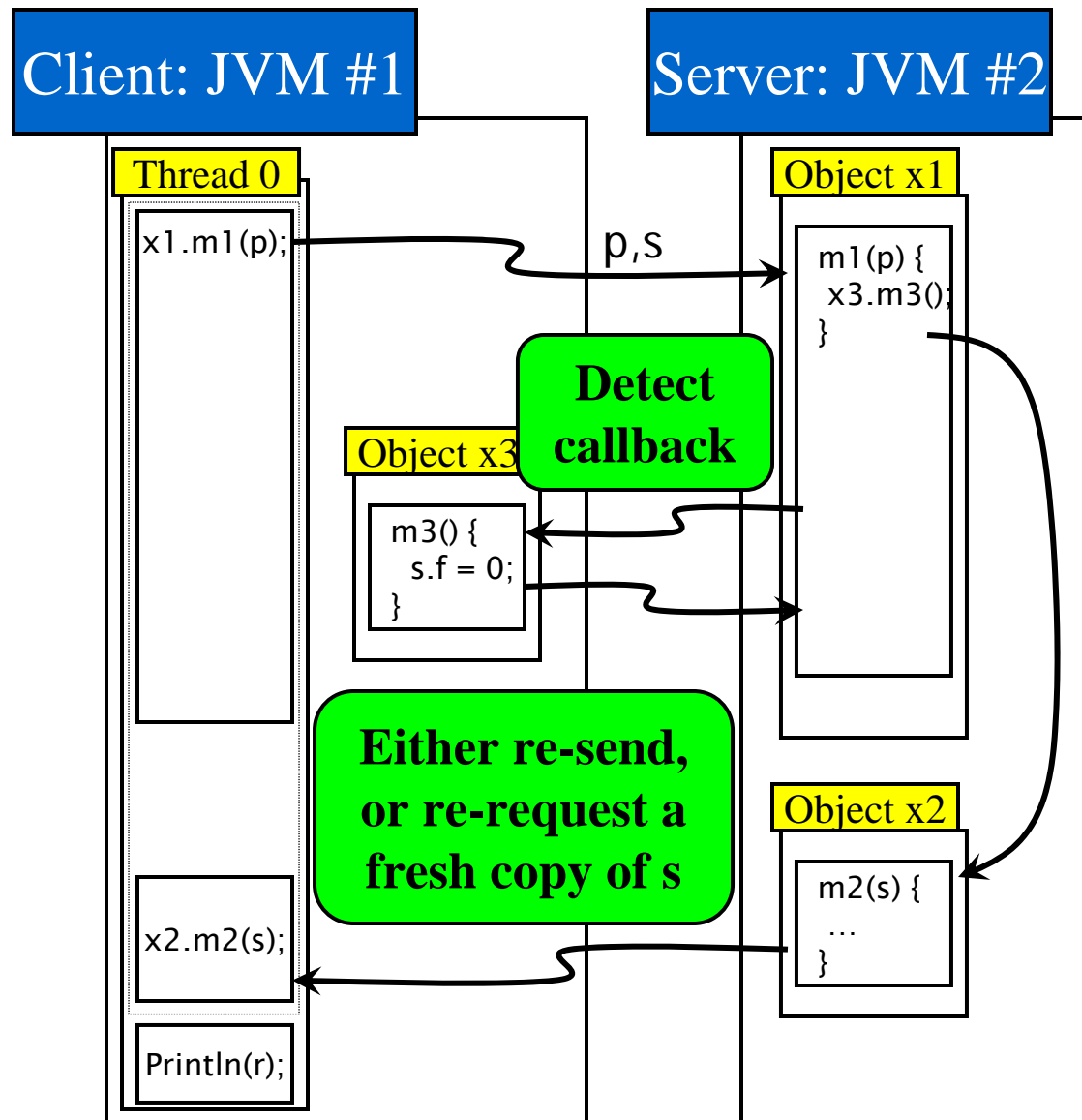
■ Optimised, aggregated implementation

Aggregated call involves a call back to the client, which changes one of the parameters?



- Optimised, aggregated implementation
- JVM#2 calls a method served by its client, JVM#1
- Which changes one of the parameters of a later call in the aggregate

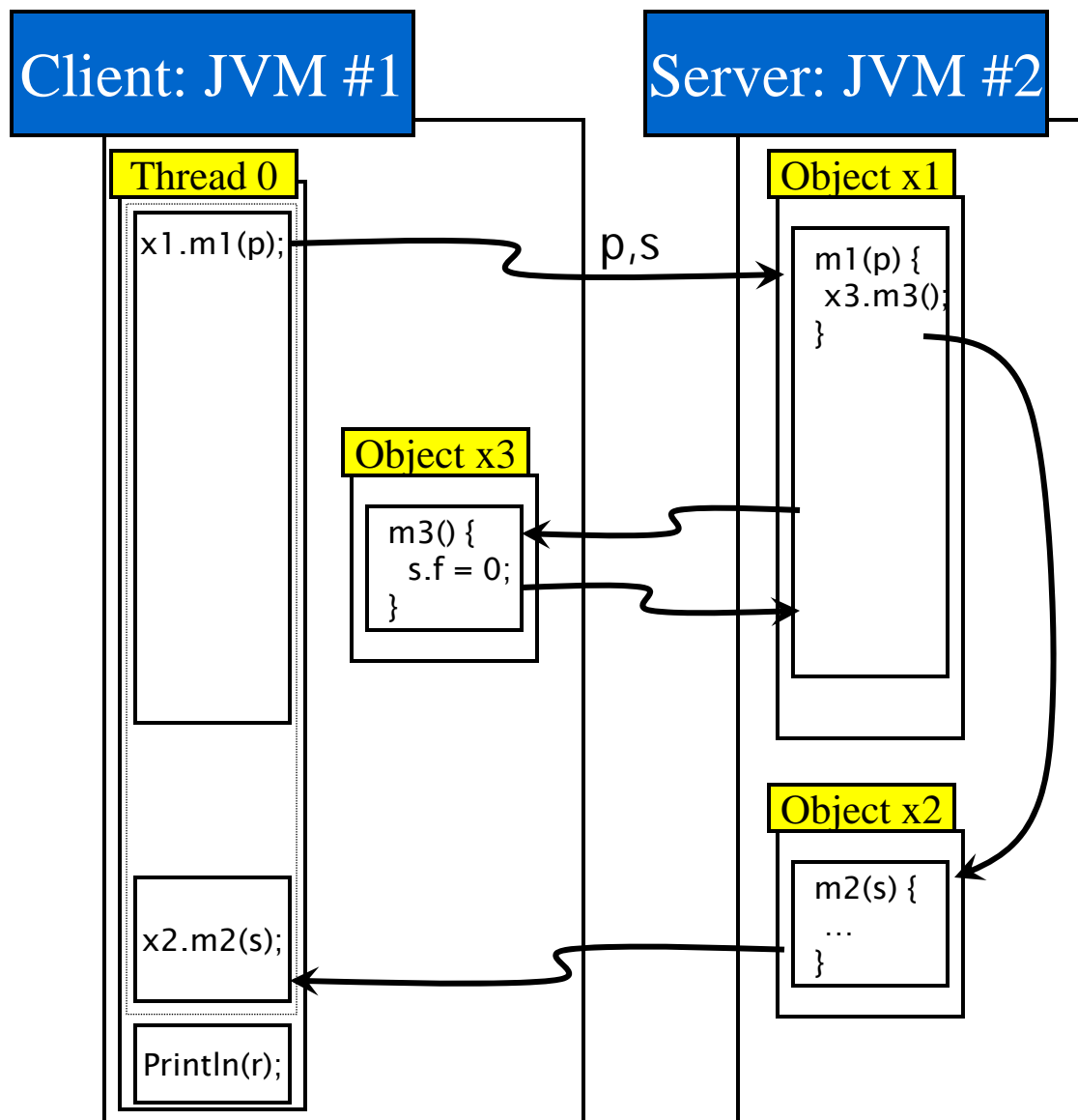
Dealing with the callback problem



Run-time check:

- If server makes an RMI call while executing an aggregated call, fetch s again
- Or:
- If client receives incoming RMI call, send an updated copy of s

Dealing with the callback problem



Static analysis:

- Add a rule to the validity predicate
- Check that
 - `m1` cannot result in a remote method call

Or:

- `s` is not visible to other threads

More aggressively:

- Transform code to avoid callback