Distributed Java applications: dynamic instrumentation and automatic optimisation

Kwok Cheung Yeung
Paul H J Kelly
With contributions from Thomas Petrou, Tim Wiffen, Doug Brear, Sarah Bennett

Software Performance Optimisation group
Imperial College, London
Background…

✔️ I lead the Software Performance Optimisation group at Imperial College, London

✔️ Stuff I’d love to talk about another time:
  - Cross-component optimisation of scientific applications at run-time
  - Is Morton-order layout for 2D arrays competitive?
  - Bounds-checking for C, links with unchecked code
  - Dynamic instrumentation for the Linux kernel
  - Run-time specialisation in C++
  - Proxying in CC-NUMA cache-coherence protocols – adaptive randomisation and combining
Mission statement

- Extend optimising compiler technology to challenging contexts beyond scope of conventional compilers
- Component-based software: cross-component optimisation
- Distributed systems:
  - Across network boundaries
  - Between different security domains
  - Maintaining proper semantics in the event of failures
This work…

- Virtual JVM, virtual JIT
  - Framework allows run-time manipulation of Java application’s binary
  - Running on top of a standard JVM

- Two applications:
  - Message aggregation and related optimisations for RMI and EJB applications
    - Optimising Java applications across network boundaries
  - Dynamic instrumentation
    - Run-time binary patching
Dynamic instrumentation:
- Run-time binary patching
- Insert code into running application on the fly
- Paradyn, DynInst for Sparc/Solaris, GILK for x86/Linux ([www.doc.ic.ac.uk/~djp1/gilk](http://www.doc.ic.ac.uk/~djp1/gilk))

Dynamic instrumentation for Java:
- Could be done inside JVM...
- Could be done via debugging interface...
- We did it by building a virtual JVM
- Runs on standard JVMs, with full JIT optimisation
A virtual virtual machine

- A VJVM is just a JVM written in Java, running on a Java JVM
- Our VJVM is carefully constructed...
  - To run fast
  - By running most of the application code directly – jump to corresponding bytecode (in some sense, a virtual JIT)
  - But the VJVM maintains control over execution by intercepting control flow
  - We can choose where to intercept control flow – method entry, basic blocks, back edges
Method fragmentation

- Control flow is intercepted by fragmenting each method
- Fragmentation policy depends on application
- Example: basic block fragmentation

```java
int f() {
    while (p<N)
        p += x.g(p);
    return p;
}
```

- Method body split into blocks
- Method entry replaced by “executor” loop: walks control-flow graph invoking each block in turn
- Fragmented method’s control-flow graph can be updated on the fly
- We built a prototype dynamic instrumentation tool JUDI:
  - Client GUI connects to set of remote (virtual) JVMs running fragmented code
  - Browse remote system’s classes, methods
  - Upload “instrument” to remote system and patch it into running code
**JUDI: deploying instrumentation**

- Construct “experiment” by applying instruments to methods
- Execute experiment: add requested instruments to each JVM
- Each instrument runs for given period or till trip count reached
- Client collects data logged from instruments for analysis
- Instruments removed from all JVMs when experiment is over

- Select class and method to be instrumented
- Show fragmented control-flow graph to see where instrumentation could be applied
- Select instrument (an instrument is just a Java class file implementing the Instrument interface)
- Apply instruments to methods
- Select instrumentation strategy
- Select methods to which instrument is to be applied
- Execute experiment
JUDI: not just for instrumentation

- Instruments are simple Java objects, which can be compiled and uploaded on the fly

- Instruments can:
  - Count, measure time
  - Access locals and parameters
  - Histogram values
  - Verify assertions
  - Modify values...
  - Impose/audit security policy
  - Trigger insertion/removal of further instrumentation
**DESORMI: delayed-evaluation, self-optimising RMI**

- Optimising Remote Method Invocation
- Another application for the VJVM

- Goal is to reduce amount of communication
- Thus, complementary to other work on faster RMI implementation

- RMI is a “heavyweight” operation: cost of an RMI call is large, so run-time optimisation can pay off even if slow(ish)
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);
}

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

**Another example: Server Forwarding:**

- the result from a call on one remote server is passed as a parameter to a call on a second remote server
- Avoids deserialisation/re-serialisation by client
- Uses server-to-server network
int m() {
    while (p<N) {
        q += x₁.m₁(p);
        B1 // non-remote code
        p += x₂.m₂(p);
        B2 // non-remote code
    }
    return p;
}

- Fragment at potential RMI call sites
- Potential RMI call sites are interface invocations with java.rmi.RemoteException on the throw list
- Whether a call is actually remote depends on the identity of the object – determined at run-time
Remote calls are delayed if possible
Executor inspects local fragment following remote call
Fragment carries def-use metadata
If no data dependence, execute local fragment:
  \( \text{Defs}(X) \cap \text{Uses}(B1) \)
If antidependence on RMI argument, copy first:
  \( \text{Uses}(X) \cap \text{Defs}(B1) \)

\[ q = x_1.m_1(p); \]
\[ p = x_2.m_2(p); \]

Chain of delayed remote calls accumulates
Until forced by dependence
int m() {
    while (p<N) {
        q += x1.m1(p);
        p = 0;
        p += x2.m2(p);
        System.out.println(p);
    }
    return p;
}

- Each fragment carries use/def and liveness info:
  
  Defs:  {q}  {p}  {p}  {}
  
  Uses:  {x1,p,q}  {}  {x2,p}  {p}

- Y can be executed before X, but p must be copied
- Z cannot be delayed because p is printed
At this point, executor has collected a sequence of delayed remote calls (fragments X and Z)

But execution is now forced by need to print

Now, we can inspect delayed fragments and construct optimised execution plan

- If $x_1$ and $x_2$ are on same server, send aggregate call
- If $x_1$ and $x_2$ are on different servers, send execution plan to $x_2$’s server, telling it to invoke $x_1.m_1(p_{\text{orig}})$ on $x_1$’s server
Maintaining semantics

Objective:
- Optimised RMI/EJB application behaves in exactly the same way as original, but is more responsive and uses less resources

Not that easy...what if...
- the remote call overwrites its parameter?
- aggregated call raises an exception?
- the client is malicious?
- a third JVM makes RMI calls on both client and server to observe sequence of actions?
- aggregated call involves a call back to the client, which changes one of the parameters?

All these problems can be solved, though in some cases at considerable cost
Microbenchmark results… aggregation

- How much performance can be gained by aggregation?
- Server adds vectors of doubles
  
  \[ \text{Result} = r.\text{add}(r.\text{add}(...r.\text{add}(v_1,v_2),v_3),...),v_n); \]

- Substantial overheads
- Due to fragmentation
- Also due to transfer of execution plan
- Alleviated by caching

![Graph showing speedup relative to unoptimised version vs number of aggregated RMI calls.](image-url)
Microbenchmark results... forwarding

- How much performance can be gained by server forwarding?
- Client code: `Result = r1.add(r2.add(v1,v2),v3);`

Servers holding remote objects r1 and r2 connected by fast ethernet
- In optimised code, vector result is passed directly from r2 to r1
- If client has slow connection, speedup even for short messages
- If client also has fast connection, no speedup yet possible
- Caching of execution plans is essential
- It would be really good not to have to duplicate parameters
Real-world benchmarks...

- Simple example: Multi-user Dungeon (from Flanagan’s Java Examples in a Nutshell)
- “Look” method:
  ```java
  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 aggregated calls:
  - Time taken to execute “look”:
    - Without call aggregation: 6.38ms
    - With call aggregation: 5.45ms
  - Speedup: 1.164
- Prototype implementation, more results soon!
Conclusions and future directions

- Virtual JVM/JIT provides extremely powerful, flexible tool
  - Dynamic instrumentation, automatic bottleneck search, dynamic “aspect weaver”
  - Research vehicle to study how to combine static analysis with run-time information
  - Currently rather slow... faster soon!

- RMI optimisation
  - Extremely challenging, due to complex dependences
  - Prototype works for simple examples
  - Currently being extended:
    - More sophisticated dependence analysis
    - EJB
  - Many more optimisations:
    - Object replication and caching
    - Cross-network code motion (“code motion for mobile code”)