Concurrent Programming with Futures

Presented by
Michael Hicks
Oregon Summer School 2006
on Concurrent and Distributed Software

This Lecture

- Introducing Futures
  - Programming model
  - Implementation in Multilisp
- Futures in Java
  - Java.util.concurrent
  - Transparency with static typing
    (Prathakakis et al, 2004)
  - Safety
    (Welc et al, 2005)

Thanks

- Adam Welc at Purdue for most of the Safe Futures slides

Scheme Merge Sort

(define (split x) ...)
(define (merge x y) ... (car x) ...)
(define (mergesort x)
  (let ((y,z) (split x))
    (merge (mergesort y) (mergesort z))))

How to make parallel?

Explicit Approach

- Threads, Message Passing
- Problems
  - Message passing requires partitioning the data among different address spaces
  - Must write code to exploit resources of underlying platform
  - Significant code changes

Implicit Approach

- Rely on the compiler to figure out opportunities for parallelism
- Problems
  - Really hard!
  - Instruction-level and loop-level parallelism can be inferred, but
  - Inferring larger "subroutine"-level parallelism has had less success.
Middle Ground: Futures

- Use future annotation [Halstead 85]
  - (future e) indicates e may run concurrently with parent

- Benefits
  - Notationally lightweight
    - Sequential algorithm still manifest
  - Implement to let concurrency be determined by
    the run-time system, based on system resources
  - Coordination between concurrent computations is transparent

Where to annotate?

(define (split x) ...)
(define (merge x y) ... (car x) ...)
(define (mergesort x)
  (let ((y,z) (split x))
    (merge (mergesort y) (mergesort z))))

No - result is used immediately in following call

Where to annotate?

(define (split x) ...)
(define (merge x y) ... (car x) ...)
(define (mergesort x)
  (let ((y,z) (split x))
    (merge (mergesort y) (mergesort z))))

Yes - recursive calls can operate in parallel

Multilisp Merge Sort

(define (split x) ...)
(define (merge x y) ... (car x) ...)
(define (mergesort x)
  (let ((y,z) (split x))
    (merge (future (mergesort y)
      (future (mergesort z))))))

Implementing Touches

(define (merge x y) ... (car x) ...)

- Futurized implementation of (car x)
  (if (pair? (touch x))
    (get first elem of x)
    (error))

- Where (touch x) is
  (if (future? x) (get x) x)

Could be a future...

Blocks until result has been computed
Optimization I

- Forking a thread per future could be expensive and without advantage
  - Particularly if not many CPUs
- Idea: only use as many threads as there are processors [Mohr et al 91]
  - At a future call, use idle thread, if any
  - Otherwise, continue using current thread
    - Save continuation on a separate queue
  - When a thread would block, save the current continuation and grab one from the queue

Optimization II

- Once a future computation completes, its result is immutable
  - Proxy and further touches redundant
- Thus
  - Use garbage collector to throw away the proxy and replace with the result [Halstead 85]
  - Avoid touching at all if static analysis can prove it's unnecessary [Flanagan & Felleisen 95]

What about side effects?

(let ((x 1)
  (let ((x 2)) x))
  (let ((x 1)
  (let ((future (set! x 2)) x)
  - Sequential version: 2
  - Parallel version: either 1 or 2

Safety and Concurrency

- Most Multilisp code is functional
  - No worry about inconsistencies
- Non-functional code
  - Encapsulate abstractions that are mutable
  - Synchronize all accesses
    - Like "fully synchronized" Vector class in Java
- What if the programmer makes a mistake?
  - Will look at this later in the talk

Futures in Java

- Java is not Lisp/Scheme
  - Static typing
  - Side-effects are far more prevalent
- Approach
  - Static analysis and transformation [Pratikakis et al 2004]
  - Detect safety problems at run-time [Welc et al 2005]

Example: HTTP handler

```java
procRequest(Socket sock) {
    Buffer in = readBuf(sock);
    Request req = translate(in);
    Buffer out = process(req);
    writeBuf(sock, out);
}

Request translate(Buffer in) {
    Request result;
    in.foo() --
    return result;
}
```

...
Sample execution (original)

```java
procRequest(Socket sock)
    Buffer in = readBuf(sock)
    Request req
    Buffer out
```

Read the buffer

```java
procRequest(Socket sock)
    Buffer in = readBuf(sock)
    Request req
    Buffer out
```

```java
readBuf(sock)
    result = ___
    return result;
```

Read the buffer

```java
procRequest(Socket sock)
    Buffer in = readBuf(sock)
    Request req
    Buffer out
```

```java
readBuf(sock)
    result = ___
    return result;
```

Return it

```java
procRequest(Socket sock)
    Buffer in = readBuf(sock)
    Request req
    Buffer out
```

```java
readBuf(sock)
    result = ___
    return result;
```

Return it

```java
procRequest(Socket sock)
    Buffer in = readBuf(sock)
    Request req
    Buffer out
```

Next call...

```java
procRequest(Socket sock)
    Buffer in = ___
    Request req = ___
    Buffer out
```

```java
translate(in)
    Request result;
    _ in.foo() _
    return result;
```
Suppose we had future

```java
procRequest(Socket sock) {
    Buffer in = future readBuf(sock);
    Request req = future translate(in);
    Buffer out = future process(req);
    writeBuf(sock, out);
}
```

Sample execution (async)

```java
procRequest(Socket sock) {
    Buffer in = future readBuf(sock);
    Request req =
    Buffer out
}
```

Read the buffer in new thread

```java
procRequest(Socket sock) {
    Buffer in =
    Request req
    Buffer out
    spawn thread
    readBuf(sock)
    result = ...
    return result;
}
```

Placeholder to caller

```java
procRequest(Socket sock) {
    Buffer in =
    Request req =
    Buffer out
    readBuf(sock)
    result = ...
    return result;
}
```

Calculate result in child

```java
procRequest(Socket sock) {
    Buffer in =
    Request req =
    Buffer out
    readBuf(sock)
    result = ...
    return result;
}
```

Store in placeholder

```java
procRequest(Socket sock) {
    Buffer in =
    Request req =
    Buffer out
    readBuf(sock)
    result = ...
    return result;
}
```
Child finished

```
procRequest(Socket sock)
  Buffer in =
  Request req =
  Buffer out
```

Next call

```
procRequest(Socket sock)
  Buffer in =
  Request req =
  Buffer out
```

translate(Buffer in)
  Request result;
  ... in.foo() ...
  return result;

Problems

```
procRequest(Socket sock)
  Buffer in =
  Request req =
  Buffer out
```

Callee and caller type not correct (should be Future or Object)

```
translate(Buffer in)
  Request result;
  ... in.foo() ...
  return result;
```

Problems

```
procRequest(Socket sock)
  Buffer in =
  Request req =
  Buffer out
```

Callee operations on argument assume a Buffer, not a Future

```
translate(Buffer in)
  Request result;
  ... in.foo() ...
  return result;
```

Problems

```
procRequest(Socket sock)
  Buffer in =
  Request req =
  Buffer out
```

Callee operations might violate transparency

```
translate(Buffer in)
  Request result;
  ... map.add(in, result)
  return result;
```

```
java.util.concurrent
```

- Concurrency library in Java 1.5
  ```
  public interface Future<T> {
  T get();
  }
  ```
  ```
  public class FutureTask<T> implements Future<T> {
  ... }
  ```
java.util.concurrent

- Could convert our HTTP program by hand to use this library, but
  - Would take a lot of code rewriting
    - Adjust the types, insert code to spawn the thread, to extract the underlying object from the future when needed, catch any exceptions that could be thrown...
    - Makes it hard to change policies later
    - What if I later want only one of the methods to be async?
  - Might result in inadvertent transparency violation

Proxy Design Pattern

- The proxy and object share an interface
- Addresses typing and code problems, but
  - Still might have to change the program to introduce an interface type, rather than the concrete type
  - Interfaces only name methods
    - Thus field accessors disallowed
  - Does not solve the transparency problems
    - Still can use ==, instanceof, etc. to distinguish between the object and its proxy

Solution: Proxy Programming Framework

[Pratikakis et al 2004]

- User indicates
  - where proxies are introduced, e.g. by future annotations on method calls
  - what to do when a proxy’s underlying object is required, e.g. when calling a method or extracting a field from a proxy
- An automatic program transformation inserts necessary code
  - For proxy introduction and coercion, avoiding transparency violations

Benefits

- No code changes needed by hand
- Policies can be changed easily
- Prevents violations of transparency
- Has applications beyond futures
  - Tracking of security-sensitive data
  - Not-null types
  - Stack allocation of objects

Summary of Approach

- Formalization of analysis and transformation
  - Formally proven correct
- Prototype implementation
  - Built on the SOOT Java bytecode analysis toolkit
- Experimental evaluation, considering
  - Analysis running time
  - Quality of generated code

Three-Stage Transformation

- Inference
  - Generate constraint graph describing how proxies could flow through the program.
- Constraint solving
  - Solve the constraints, identifying where coercions are needed.
- Transformation
  - Rewrite any classes requiring coercions, type changes, etc.
Inference

- Each type has **qualifier proxy or nonproxy**
  - Like final, but never appears in source programs
  - **proxy** indicates the value **may be** a proxy
  - **nonproxy** indicates it is **definitely not** a proxy
  - **nonproxy < proxy**
- **Qualifier inference** is used to assign qualifiers to types in the program, based on
  - Where proxies are introduced
  - Where non-proxies are required
  - How values flow between these locations

**Constraint Solving**

- **Standard**
  - Based on graph reachability
- If a possible **proxy** indeed flows to a location requiring a **nonproxy**, there will be a path between the two in the graph.
  - Requires a coercion as **proxy < nonproxy**

**Transformation**

- For each class that
  - Requires a coercion
  - Introduces a proxy
- ... rewrite the class as necessary to insert code to implement them
  - Code provided by the user
- Must avoid transparency violations
  - Forward calls to .equals(), .hashCode(), etc.

**Before Analysis: procRequest**

```java
procRequest(Socket sock) {
    Buffer in = future readBuf(sock);
    Request req = future translate(in);
    Buffer out = future process(req);
    writeBuf(sock, out);
}
```

**Inference constraints**

```java
procRequest(Socket sock) {
    Buffer in = proxy readBuf(sock);
    Request req = proxy translate(in);
    Buffer out = proxy process(req);
    writeBuf(sock, out);
}
```
After transformation

```java
procRequest(Socket sock) {
    Object in = new Proxy {
        private Object result;
        public void run() {
            result = readBuf(sock); }
        public synchronized Object get() {
            // return result; }
        public boolean equals(Object o) {
            return get().equals(o); }
    };
    TPK.run((Runnable)in);
    Object req = new Proxy {
        translate(in);
    }
    Object out = new Proxy {
        process(req);
    }
    writeBuf(sock, out);
}
```

Before Analysis: translate

```java
Request translate(Buffer in) {
    Request result;
    in.foo();
    return result;
}
```

Inference Constraints

```java
Request translate(Buffer in) {
    Request result;
    translate(in);
    nonproxy in.foo();
    return result;
}
```

After transformation

```java
Request translate(Object inF) {
    Request result;
    String in =
    (String)(inF instanceof Proxy ?
    inF.get() :
    inF);
    in.foo();
    return result;
}
```

Example

```
procRequest(Socket_sock)  String in
    String req
    Request req
    String out
translate(Object inF)  String in = inF.get();
    in.foo();
```

After executing coercion

```
procRequest(Socket_sock)  String in
    String req
    Request req
    String out
translate(Object inF)  String in =
    in.foo();
```
User control of Analysis

- Analysis determines where coercions are needed, then rewrites classes.
- What code to insert depends on the proxy being used; provided by the user
  - Can support lazy computation using the same code for coercions, with proxy.get() to run the invocation.

Analysis Characterization

- Analysis is context-insensitive, path-insensitive, and partly flow-sensitive (only with regard to coercions).
- Operates on whole program
  - User can control whether standard class libraries should also be rewritten

Other Applications

- Checking for transparency violations
  - Follow the flow of design-pattern proxies (which use an interface)
  - Require identity-revealing operations to be only on non-proxies
    - Argument to ==
    - Argument to instanceof
    - Argument to downcast
  - If any coercions are needed, reveals potential transparency violation

Other Applications

- Not-null types
  - Two qualifiers null and notnull
    - notnull + null
  - Coercion implemented as null-check
- Stack-allocated objects
  - Two qualifiers stack and nonstack
    - stack + nonstack
  - Coercions introduced when
    - Assigning nonstack to a field or return value
    - Performing an identity-revealing operation (e.g. hashcode)

Implementation

- Modified the SOOT bytecode analysis framework
  - Three-address code, SSA-like intermediate representation called Jimple
  - Extended Jimple with opcode to indicate proxy introduction
- User-provided classes dictate what expression forms may require coercions and how they are implemented

Experiments

- Overhead of inserted dynamic checks
- Cost of running the analysis
- Benefits to target applications
Dynamic Check Overhead

Object p, o = ...;
for (int i = 0; i<N; i++) {
    p = o; p.m();
}

<table>
<thead>
<tr>
<th>Case</th>
<th>test (s)</th>
<th>tot (s)</th>
<th>per-check (ns)</th>
<th>% over</th>
</tr>
</thead>
<tbody>
<tr>
<td>no claim</td>
<td>2.154</td>
<td>n/a</td>
<td>n/a</td>
<td>100%</td>
</tr>
<tr>
<td>spurious claim</td>
<td>2.401</td>
<td>35</td>
<td>10%</td>
<td>65%</td>
</tr>
<tr>
<td>necessary claim</td>
<td>3.567</td>
<td>141</td>
<td>10%</td>
<td>65%</td>
</tr>
</tbody>
</table>

Sample Application: Async RMI

Service findService(LocalPeer self, String name) {
    Service s = self.getService(name);
    if (s != null) return s;
    Async.invoke(self.forward(...));
    return getRemoteService(self.name);
}

Sample Application: Async RMI

Service findService(LocalPeer self, String name) {
    Service s = self.getService(name);
    if (s != null) return s;
    Async.invoke(...self.forward(...));
    return Lazy.invoke(
        getRemoteService(self.name));
}

Sample Application: Async RMI

<table>
<thead>
<tr>
<th>Version</th>
<th>Services requested and used</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orig.</td>
<td></td>
<td>11</td>
<td>22</td>
<td>30</td>
<td>41</td>
<td>54</td>
<td>68</td>
<td>78</td>
<td>80</td>
<td>86</td>
<td>94</td>
</tr>
<tr>
<td>FS</td>
<td></td>
<td>13</td>
<td>24</td>
<td>32</td>
<td>43</td>
<td>55</td>
<td>71</td>
<td>79</td>
<td>81</td>
<td>84</td>
<td>87</td>
</tr>
<tr>
<td>Async + delay</td>
<td></td>
<td>600</td>
<td>692</td>
<td>268</td>
<td>370</td>
<td>462</td>
<td>562</td>
<td>647</td>
<td>738</td>
<td>826</td>
<td>914</td>
</tr>
<tr>
<td>Async + delay</td>
<td></td>
<td>107</td>
<td>110</td>
<td>120</td>
<td>124</td>
<td>137</td>
<td>128</td>
<td>143</td>
<td>151</td>
<td>136</td>
<td>136</td>
</tr>
</tbody>
</table>

- Adding asynchrony provides a performance benefit for higher-latency networks when messages can be retrieved in parallel
- Otherwise, network latency dominates, so asynchrony not helpful

Async RMI Analysis Time

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Time</th>
<th>classes w/ fut.</th>
<th>re-written</th>
<th>claims</th>
</tr>
</thead>
<tbody>
<tr>
<td>FS</td>
<td>139</td>
<td>1319</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>spark</td>
<td>126</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

- Flow-insensitive version adds little cost to points-to analysis
- Flow-sensitive version adds greater cost
  - Currently over-eagerly introduces flow-sensitive nodes; can be more on-demand
Other Applications

- Checking for transparency violations of design-pattern proxies
  - In SOAP/RMI library (2087 classes analyzed)
  - In SOOT framework (2510 classes analyzed)
- Chose various locations to introduce a design-pattern proxy
  - Found that doing so would have introduced as many as 7 transparency violations.

Summary

- Proxy programming framework provides a way to introduce futures to Java transparently
  - Write the annotation as in Multilisp
  - Compiler inserts code to touch possible futures, with some optimizations
  - Ensures placeholder not mistaken for original object
- Next up: worrying about side effects ...

Futures - Safety

If sequential program \( P \) is annotated with futures to yield concurrent program \( P_f \), then the observable behavior of \( P \) is equivalent to \( P_f \)

- Logical serial order trivially satisfied when no side-effects
- Problems arise with mutation of shared data

Running Example

<table>
<thead>
<tr>
<th>Account s; // savings</th>
<th>Account c; // checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Future f = F(monthlyTotal());</td>
<td>浮 monthlyTotal()</td>
</tr>
<tr>
<td>transfer(50);</td>
<td>s.addInterest(0.10);</td>
</tr>
<tr>
<td>global = f.get();</td>
<td>return c.balance()+s.balance();</td>
</tr>
<tr>
<td>void transfer (float amount)</td>
<td></td>
</tr>
<tr>
<td>s.withdraw(amount);</td>
<td></td>
</tr>
<tr>
<td>c.deposit(amount);</td>
<td></td>
</tr>
</tbody>
</table>

Terminology

Account s; // savings
Account c; // checking
Future f = F(monthlyTotal());
transfer(50);
global = f.get();

monthlyTotal() — transfer() — get()
**Terminology**

Account s; // savings
Account c; // checking

Future f = F(monthlyTotal());
transfer(50);
global = f.get();

float monthlyTotal () {
  s.addInterest(0.10);
  return c.balance()+s.balance();
}

void transfer (float amount) {
  s.withdraw(amount);
  c.deposit(amount);
}

**Logical Serial Order**

Account s; // savings
Account c; // checking

Future f = F(monthlyTotal());
transfer(50);
global = f.get();

float monthlyTotal () {
  s.addInterest(0.10);
  return c.balance()+s.balance();
}

void transfer (float amount) {
  s.withdraw(amount);
  c.deposit(amount);
}

s = 100  c = 100  global = 0

T_F

future
continuation

T_M

**Logical Serial Order**

Account s; // savings
Account c; // checking

Future f = F(monthlyTotal());
transfer(50);
global = f.get();

float monthlyTotal () {
  s.addInterest(0.10);
  return c.balance()+s.balance();
}

void transfer (float amount) {
  s.withdraw(amount);
  c.deposit(amount);
}

s = 110  c = 100  global = 0

T_F

rd(s)  rd(c)  rd(s)

future
continuation

T_M

**Arbitrary Interleaving**

Account s; // savings
Account c; // checking

Future f = F(monthlyTotal());
transfer(50);
global = f.get();

float monthlyTotal () {
  s.addInterest(0.10);
  return c.balance()+s.balance();
}

void transfer (float amount) {
  s.withdraw(amount);
  c.deposit(amount);
}

s = 100  c = 100  global = 0

T_F

future
continuation

T_M

**Arbitrary Interleaving**

Account s; // savings
Account c; // checking

Future f = F(monthlyTotal());
transfer(50);
global = f.get();

float monthlyTotal () {
  s.addInterest(0.10);
  return c.balance()+s.balance();
}

void transfer (float amount) {
  s.withdraw(amount);
  c.deposit(amount);
}

s = 100  c = 100  global = 0

T_F

future
continuation

T_M
Arbitrary Interleaving

Account s: // savings
Account c: // checking
Future f = F(monthlyTotal());
transfer(50);
global = f.get();

float monthlyTotal() {
    s.addInterest(0.10);
    return c.balance()+s.balance();
}
void transfer(float amount) {
    s.withdraw(amount);
    c.deposit(amount);
}
s = 100, c = 100, global = 0

SERIAL
s=60

Arbitrary Interleaving

Account s: // savings
Account c: // checking
Future f = F(monthlyTotal());
transfer(50);
global = f.get();

s = 50, c = 100, global = 0

SERIAL
s=100

What Happened?

- Concurrency of shared updates led to unexpected behavior
- Updates from continuation leaked into future
  - monthlyTotal() should not see results of transfer()
- Results computed by future were not available for continuation
  - transfer() supposed to see results of monthlyTotal()

Two Kinds of Violations

- **Forward Dependency Violation**
  - Continuation does not observe an effect of the future computation when it should have serially (or observes the wrong one)
- **Backward Dependency Violation**
  - Future does observe an effect of the continuation when it would not have serially
Avoiding Safety Violations
[Welc et al 2005]

- Formal framework for reasoning about safe futures
  - Proof that schedules that do not exhibit forward or backward dependency violations are equivalent to serial
- Implementation that ensures safe schedules
  - Uses optimistic techniques

Implementation Overview

- Data accesses hashed into read and write maps. Maps used by continuation to detect conflicts for accesses from its future
  - Detects forward dependency violations
- Versions used by future to prevent seeing updates by its continuation
  - Prevents backward dependency violations
- Automatic roll-back when conflict detected

Safe Execution

Account s; // savings
Account c; // checking
Future f = F(monthlyTotal());
transfer(50);
global = f.get();

s = 100  c = 100  global = 0

Future f = F(monthlyTotal());
return c.balance()+s.balance();
void transfer (float amount) {
  s.withdraw(amount);
c.deposit(amount); }

Safe Execution

float monthlyTotal () {
s.addInterest(0.10);
return c.balance()+s.balance();
}
void transfer (float amount) {
s.withdraw(amount);
c.deposit(amount); }
Safe Execution

Account s; // savings
Account c; // checking

Future f = f(monthlyTotal());
transfer(50);
global = f.get();
sr♯ = 110
sr♮ = 50
s = 100
c = 100
global = 0

^r
future
rd(s) wr(s♯)
continuation
rd(c) wr(c♮)

97

Safe Execution

Account s; // savings
Account c; // checking

Future f = f(monthlyTotal());
transfer(50);
global = f.get();
sr♯ = 110
sr♮ = 50
s = 100
c = 100
global = 0

^r
future
rd(s) wr(s♯) rd(c) wr(c♮)
continuation
rd(s) wr(s♮)

98

Safe Execution

Account s; // savings
Account c; // checking

Future f = f(monthlyTotal());
transfer(50);
global = f.get();
sr♯ = 110
sr♮ = 50
s = 100
c = 100
global = 0

^r
future
rd(s) wr(s♯) rd(c) wr(c♮)
continuation
rd(s) wr(s♮)

99

Safe Execution

Account s; // savings
Account c; // checking

Future f = f(monthlyTotal());
transfer(50);
global = f.get();
sr♯ = 110
sr♮ = 50
c = 150
s = 100
c = 100
global = 0

^r
future
rd(s) wr(s♯) rd(c) wr(c♮)
continuation
rd(s) wr(s♮)

100

Safe Execution

Account s; // savings
Account c; // checking

Future f = f(monthlyTotal());
transfer(50);
global = f.get();
sr♯ = 110
sr♮ = 50
c = 150
s = 100
c = 100
global = 0

^r
future
rd(s) wr(s♯) rd(c) wr(c♮)
continuation
rd(s) wr(s♮) rd(c) wr(c♮)

101

Safe Execution

Account s; // savings
Account c; // checking

Future f = f(monthlyTotal());
transfer(50);
global = f.get();
sr♯ = 110
sr♮ = 50
c = 150
s = 100
c = 100
global = 0

^r
future
rd(s) wr(s♯) rd(c) wr(c♮)
continuation
rd(s) wr(s♮) rd(c) wr(c♮)

102
Safe Execution

Account s; // savings
Account c; // checking
Future f = F(monthlyTotal());
transfer(50);
global = f.get();

float monthlyTotal () { return c.balance() + s.balance(); }
void transfer (float amount) {
    s.withdraw(amount);
c.deposit(amount);
}

s = 60   c = 150  global = 0

continuation

get

Prototype Implementation

- Based on IBM's Jikes RVM
- Compiler-injected read and write barriers to intercept shared data accesses
- Bytecode rewriting plus run-time support for automatic roll-back
- Modification of object headers
  - Version access via forwarding pointers

Barrier Optimizations

- Goal: omit barriers on loads of primitive values
- Problem: accesses through stale on-stack references
- Solution: update references on stack using modified GC stack scanning procedure
  - At version creation
  - At pre-specified "synchronization" points
Automatic Rollback

- Discard versions
- Futures:
  - evaluated within separate thread so just re-run
- Continuations:
  - Rewrite bytecodes to save state at start
  - On rollback throw revoke exception
  - Modify run-time to unwind revoke exceptions without running user handlers
  - Handler restores state and restarts continuation

Challenges

- Continuations escaping method scope
  - Perform get early
- Serial order for multiple futures
  - Different threads for separate futures
  - The same thread for all continuations
  - Nested futures
- Interaction with existing mechanisms
  - Java threads, native methods may fail safety

Benchmarks

- Selected Java Grande benchmarks
- Modified Multi-User OO7 benchmark
  - Standard OO7 design database
  - Multi-level hierarchy of composite parts
  - Shared and private modules
  - Mixed-mode read/write traversals
- Configuration
  - 700MHz Pentium 3 (4 CPUs)
  - Average of 5 "hot" runs (no compilation)

Java Grande - 4 Futures

OO7 - 4 Futures
All reads to shared module

OO7 - 1 Future
All reads to shared module
Conclusions

- Futures are a lighter-weight alternative to programming for parallelism
- Multilisp pioneered the idea
- Applying to Java requires more work
  - Proxy inference
  - Safety checking

Future Work

- Better run-time support
  - Lazy task creation a la Multilisp
- Safety checking for non-serial futures
  - HTTP example rejected by safety checking scheme
- Incremental analysis for better software development

Further Reading

- Static Analysis
  - Points-to analysis (many)
  - Qualifier inference (Foster et al.)
  - Value flow analysis (Heintze and Tardieu)

Further Reading

- Parallelization (Rinard et al.)
- Transactional memory (Herlihy et al., Shavit-Touitou)
- Atomicity (Flanagan et al., Harris et al.)
- Traditional lock optimizations (Bacon et al.)
- Lock-free data structures (Rajwar-Goodman, Jensen et al.)

It’s break time!