Performance Management for Web Services with Aspect-Oriented Programming

Ron Bodkin
rbodkin@newaspects.com
New Aspects of Software
Agenda

- Overview
- Monitoring performance
- Measuring overhead
- Optimizing performance
- Conclusion
Performance and Failure

- Architectures are increasing in complexity
- Many parts can fail
  - Web services bring the complexity of multiple distributed components to many applications
  - Database connections failures
  - Libraries bring leverage... but increase interfaces that are less understood & might change
- Help desk calls is a depressingly common metric
- How can you detect & resolve production problems in a timely fashion?
Goal: Detect Problems Automatically

- Step one: capturing key system statistics like
  - How many requests?
  - How long do they take?
  - How much time is spent in database calls?
  - Which requests are slow?
  - What happens when the system is slow?

- Can be used for
  - Real-time alerts
  - Management consoles
  - Offline analysis
  - Low-overhead tuning
Traditional Approaches…

- Manually gather & correlate low-level statistics (JMX, log files, etc.)
- Instrument code for
  - Web service implementations
  - Web request handlers (Struts, Servlets, …)
  - Web service invocations
  - Database invocations
  - XML processing calls
  - etc.
- Wire in specialized interceptors (when available)
  - JAX-RPC Handlers
  - Servlet Filters, Struts Request Processor
  - Hibernate Filters
  - etc.
- “Enterprise” bytecode instrumentation tools
Problems

- Lots of boilerplate code
- Need to correlate information across layers
  - What is causing the bad query?
  - Where do I generate a bad service request?
  - What bean handles this request?
- Need to revise
  - Want to adapt monitoring based on experience
- But it’s already painful to add monitoring once

Performance monitoring is crosscutting
AOP Summary

- Crosscutting is natural
  - The problem has been a lack of support
- Aspects provide *modular support*
- Evolutionary step: structured $\rightarrow$ objects $\rightarrow$ aspects
- Aspects use
  - *pointcuts* to match ...
  - *join points* in executing programs
  - *advice* runs automatically before and/or after a pointcut
- More AOP background at *Using Aspect-Oriented Programming to Implement Web Services*
Looking Inside the Glassbox Inspector

Open source: https://glassbox-inspector.dev.java.net/
Part 2: Monitoring Performance
Monitoring Architecture

[Diagram showing the Monitoring Architecture with components such as Container, Application 1, Application 2, JMXClient, MonitoringAgent, JDBC Driver, JDBC, Hibernate, Domain, Spring MVC, Web Services, Struts, Servlets, Web Services, JDBC, and HTTP connections.]

Ron Bodkin — Performance Management for Web Services with Aspect-Oriented Programming
public aspect HttpServletMonitor {
  public pointcut monitoredOperation(Object operation) :
      execution(void HttpServlet.do*(..)) && this(operation);

  void around(Object operation) : monitoredOperation(operation) {
    long start = getTime();

    proceed(operation);

    PerfStats stats = lookupStats(operation);
    stats.recordExecution(start, getTime());
  }

  public long getTime() {
    return System.currentTimeMillis();
  }

  ...
}
protected PerfStats lookupStats(Object operation) {
    Class keyClass = operation.getClass();
    synchronized(operations) {
        stats = (PerfStats)operations.get(keyClass);
        if (stats == null) {
            stats = statsFactory.createTopLevelOperationStats(
                HttpServlet.class, keyClass);
            operations.put(keyClass, stats);
        }
    }
    return stats;
}

private Map<Class, PerfStats> operations =
    new WeakIdentityHashMap();
private PerfStatsFactory statsFactory;

...
Performance Statistics

```java
public interface PerfStats {
    /** Record that a single execution occurred. */
    void recordExecution(int time);
    void reset();
    int getAccumulatedTime();
    int getMaxTime();
    int getCount();
}

public class PerfStatsImpl implements PerfStats {
    public void recordExecution(int time) {
        accumulatedTime += time;
        maxTime = Math.max(time, maxTime);
        count++;
    }

    public void reset() {
        accumulatedTime=0L;
        maxTime=0L;
        ...
    }

    Prefers low overhead to absolute accuracy…
    not synchronized
```
Exposing Data...

- Various choices
  - Log files
  - Database summarization & analysis
  - Online access
- I will show use of JMX to provide live access
Java Management Extensions (JMX)

- The standard API for Java management
- JMX Remote allows remote client access
- Widely available
  - Embedded in many containers
  - Freely available RI, open source tools like MX4J
  - Java 5 embeds implementations (& JConsole)
public aspect JmxManagement {
    public interface ManagedBean {
        String getOperationName();
        Object getMBean();
    }

    /** After constructing a ManagedBean, register it */
    after() returning (ManagedBean bean):
    call(ManagedBean+.new(..)) {
        String keyName = bean.getOperationName();
        ObjectName objectName =
            new ObjectName("glassbox.inspector:" + keyName);

        Object mBean = bean.getMBean();
        if (mBean != null) {
            mbeanServer.registerMBean(mBean, objectName);
        }
    } // use IoC to configure mbeanServer
}
Exposing Statistics Through JMX

```java
public aspect StatsJmxManagement {
    public interface PerfStatsMBean extends ManagedBean {
        int getAccumulatedTime();
        int getMaxTime();
        int getCount();
        void reset();
    }

    declare parents: PerfStats implements PerfStatsMBean;

    public DynamicMBean PerfStats.getMBean() {
        try {
            RequiredModelMBean mBean = new RequiredModelMBean();
            mBean.setModelMBeanInfo(assembler.getMBeanInfo(this,
                getOperationName()));
            mBean.setManagedResource(this, "ObjectName");
            return mBean;
        } catch (Exception e) { // handled by ErrorHandling aspect
            throw new AspectConfigException("can't register", e);
        }
    }
} ...
```
Exposing Statistics Through JMX

(Continued)

```java
public String PerfStats.getOperationName() {
    StringBuffer keyStr = new StringBuffer("operation=");
    if (key instanceof Class) {
        keyStr.append(((Class)key).getName());
    } else {
        keyStr.append(key.toString());
    }
    JmxManagement.jmxEncode(keyStr, OFFSET);
    keyStr.append(""");
    return keyStr.toString();
}

// use Spring utility to create dynamic MBean
private static Class[] managedIfaces = { PerfStatsMBean.class };
static InterfaceBasedMBeanInfoAssembler assembler;
static {
    assembler = new InterfaceBasedMBeanInfoAssembler();
    assembler.setManagedInterfaces(managedIfaces);
}
...
Expanded, Correlated Monitoring
Base Monitoring Aspect

```
public abstract aspect AbstractRequestMonitor {
    public pointcut requestExecution(RequestContext context) :
        execution(* RequestContext.execute(..)) && this(context);

    public pointcut inRequest(RequestContext context) :
        cflow(requestExecution(context));

    // track parent/child relationships for contexts
    after(RequestContext parent) returning (RequestContext child) :
        call(RequestContext+.new(..)) && inRequest(parent) {
            child.setParent(parent);
        }
...
... Using the *Worker Object* Pattern

```java
/** Worker object: holds context for a monitored request. */
public abstract class RequestContext {
    public final Object execute() {
        long start = getTime();

        Object result = doExecute();

        PerfStats stats = lookupStats();
        if (stats != null) {
            stats.recordExecution(start, getTime());
        }
        return result;
    }

    // template methods
    public abstract Object doExecute();
    protected abstract PerfStats lookupStats();
}
```
Two Main State Management Options

- Worker objects
  - Can not be pooled for more efficiency
  - Pros: language support, modular code, extensibility
  - Cons: requires extension to track new state, forces non-inlined around advice

- ThreadLocal stack of context
  - Pros: more control
  - Cons: lower-level, requires more data structure
Generalized Operation Monitoring

```java
public abstract aspect AbstractOperationMonitor extends AbstractRequestMonitor {
    public pointcut classControllerExec(Object controller); 

    Object around(final Object controller) {
        classControllerExec(controller) {
            RequestContext requestContext = new OperationRequestContext() {
                public Object doExecute() {
                    return proceed(controller);
                }

                protected Object getKey() {
                    return controller.getClass();
                }
            };
            return requestContext.execute();
        }
    }
}
```
Generalized Operation Monitoring

protected abstract class OperationRequestContext extends RequestContext {
    public PerfStats lookupStats() {
        if (getParent() != null) {
            OperationStats parentStats =
                (OperationStats)getParent().getStats();
            return parentStats.getOperationStats(getKey());
        }
        return getTopLevelStats(getKey());
    }

    ...
protected OperationStats getTopLevelStats(Object key) {
    OperationStats stats;
    synchronized(topLevelOperations) {
        stats = (OperationStats)topLevelOperations.get(key);
        if (stats == null) {
            stats = perfStatsFactory.createTopLevelOperationStats(            key, getContextName(controller));
            topLevelOperations.put(key, stats);
        }
    }
    return stats;
}
public aspect ServletMonitor extends AbstractOperationMonitor {
  public pointcut servletService(Servlet servlet) :
      execution(void Servlet.service(..)) && this(servlet);

  public pointcut httpServletDo(HttpServlet servlet) :
      execution(void HttpServlet.do*(..)) && this(servlet);

  public pointcut jspService(JspPage page) :
      execution(* _jspService(..)) && this(page);

  public pointcut classControllerExec(Object controller) :
      (servletService(*) || httpServletDo(*) || jspService(*)) &&
      this(controller);

  protected String getContextName(Object controller) {
    Servlet servlet = (Servlet)controller;
    return servlet.getServletConfig().getServletContext().
    getServletContextName();
  }

  ...
}
Monitoring JDBC Information

- It’s fairly simple to monitor calls from, *e.g.*, a line of code, a class or a method
- But tracking what is called requires *context*
  - Database metadata (*e.g.*, URL) for connection
    - *e.g.*, pooled data source connections
  - SQL string for (prepared) statement
  - Connection for statements
    - Even with proxies (*e.g.*, Hibernate, CMP)
- This adds responsibilities to our aspects
public aspect JdbcConnectionMonitor extends AbstractRequestMonitor {
    public pointcut dataSourceConnectionCall(DataSource dataSource) :
        call(Connection+ DataSource.getConnection(..)) &&
        target(dataSource);

    public pointcut directConnectionCall(String url) :
        (call(Connection+ Driver.connect(..)) ||
        call(Connection+ DriverManager.getConnection(..))) &&
        args(url, ..);

    public pointcut nestedConnectionCall() :
        cflowbelow(dataSourceConnectionCall(*) ||
                  directConnectionCall(*));

    ...
...  
Connection around(final DataSource dataSource) :
    dataSourceConnectionCall(dataSource) &&
    !nestedConnectionCall() {
        RequestContext requestContext = new ConnectionReqCtx() {
            public Object doExecute() {
                accessingConnection(dataSource);
                Connection connection = proceed(dataSource);
                return addConnection(connection);
            }
        };
        return (Connection)requestContext.execute();
    }

/** Associates connections with their database names */
private Map/*<Connection,String>*/ connections =
    new WeakIdentityHashMap();
...
Worker Object for JDBC Connections

```java
protected abstract class ConnectionReqCtx extends RequestContext {
    private ResourceStats dbStats;
    private String databaseName;

    protected void accessingConnection(final DataSource ds) {
        databaseName = getNameForDS(ds);
        if (getParent() != null) {
            OperationStats opStats =
                (OperationStats)getParent().getStats();
            dbStats = opStats.getDatabaseStats(databaseName);
        }
    }

    protected Connection addConnection(final Connection conn) {
        synchronized(connections) {
            connections.put(connection, databaseName);
        }
        return connection;
    }

    protected PerfStats lookupStats() { return dbStats; }
};
```
Information Tracking

- These Aspects use *Weak Identity* Hash Maps
- Weak References
  - Allow tracked objects to be garbage collected
  - Since singleton aspect isn’t garbage collected...
  - This is important to avoid memory leaks
- Identity Map
  - Tracks by system identity hashcode (same reference)
  - Avoids calling application-specific hashCode()
  - This prevents any exceptions (*e.g.*, closed iBatis connection)
  - Changed value as state changes
- Might make inter-type declaration to store metadata for each Connection instance
  - Requires weaving into JDBC code
public aspect JdbcStatementMonitor extends AbstractRequestMonitor {
	public pointcut statementExec(Statement statement) :
	call(* java.sql..*execute*(..)) && target(statement);

	public pointcut callCreateStatement(Connection connection):
	call(Statement+ Connection.*(..)) && target(connection);


private Map/*<Statement,Connection>*/ statementCreators =
new WeakIdentityHashMap();

private Map/*<Statement,String>*/ statementSql =
new WeakIdentityHashMap();

...
JDBC Statement Tracking

```java
...
// associate connection called to create statement
after(Connection connection) returning (Statement statement):
    callCreateStatement(connection) {
        synchronized (JdbcStatementMonitor.this) {
            statementCreators.put(statement, connection);
        }
    }

after(String sql) returning (PreparedStatement statement):
    callCreatePreparedStatement(sql) {
        setUpStatement(statement, sql);
    }

private synchronized void setUpStatement(Statement statement, String sql) {
    statementSql.put(statement, sql);
}
...
```
public abstract class StatementRequestContext() {
    protected PerfStats lookupStats() { // omits null checks
        synchronized (JdbcStatementMonitor.this) {
            connection = (Connection)statementCreators.get(statement);
            sql = (String) statementSql.get(statement);
        }
        String databaseName = JdbcConnectionMonitor.aspectOf().
            getDatabaseName(connection);
        OperationStats opStats =
            (OperationStats)getParent().getStats();
        ResourceStats dbStats =
            opStats.getDatabaseStats(databaseName);
        return dbStats.getRequestStats(sql);
    }
}
Consuming WS with JAX RPC Proxy

```java
public aspect RemoteCallMonitor extends AbstractResourceMonitor {
    public pointcut remoteProxyCall(Object recipient) :
        call(public * Remote+.*(..) throws RemoteException) &&
        target(recipient) && !within(glassbox.inspector..*);

    Object around(final Object recipient) :
        remoteProxyCall(recipient) {
            RequestContext requestContext = new ResourceRequestContext() {
                public PerfStats lookupStats() {
                    String key =
                        "jaxrpc:"+recipient.getClass().getName()+"."+
                        thisJoinPointStaticPart.getSignature().getName();
                    return lookupResourceStats(key);
                }
            }

            ...
        }
}
```
Monitor Providing Web Service

```java
public aspect AxisOperationMonitor extends AbstractOperationMonitor {
    public pointcut axisRpcMethodInvocation(Object receiver, Method method) :
        execution(* invokeMethod(..)) && this(RPCProvider) && args(*, method, receiver, ..);

    protected pointcut methodControllerExec(Object controller, Method method) :
        axisRpcMethodInvocation(controller, method);

    ...
}
```

Similar in nature to using handlers
But this approach lets you correlate resource use within a monitoring operation
And use common technology for each layer
The aspect monitoring can be enabled & disabled dynamically…
Tracking Failures

```java
public abstract aspect AbstractRequestMonitor {
    ...
    after(RequestContext requestContext) throwing (Throwable t) :
        requestExecution(requestContext) {
            logInfo("exiting request context with exception ", t);
            PerfStats stats = requestContext.getStats();
            if (stats != null) {
                requestContext.recordFailure(t);
            }
        }
}
```

Catches, *e.g.*, Axis faults
Also track error return codes in Servlets
Even check for classes of errors (SQLException / Axis fault)
Then expose failure count in JMX statistics
Error Isolation:
Basic Approach

```java
public aspect ErrorHandling {
    public pointcut handlingScope() :
        scope() && adviceexecution();

    public pointcut scope() :
        within(glassbox.inspector..*) && !within(ErrorHandling);

    Object around() : handlingScope() {
        try {
            return proceed();
        } catch (Throwable e) {
            // log but don't rethrow – don’t poison underlying code
            handleError(e, thisJoinPointStaticPart);
            return null;
        }
    }
}
```
Monitoring XML Processing
Select monitoring

- Simplest: global flag
  ```
  && if(enabled)
  ```
  Can also write advice to disable monitoring advice

- Next: per monitor flag
  - Same options, just per aspect or advice

- Best: *sampling rate* (across whole request)
  - 0 in $n$ is disabled
  - $n$ in $n$ is fully enabled
  ```
  && if(shouldSample())
  ```
  ```
  boolean shouldSample() { return (ctr++ % freq < limit) }
  ```

- Natural to manage (set) with JMX
The Troubleshooting Death Spiral

It can take 8 days, 5 people, and 6 conference calls to troubleshoot what turns out to be a simple problem.

1. Finger pointing
2. Get subsystem owner involved
3. Collect data
4. Hypothesize which subsystem broke
5. Conference calls
Common Technology Challenges

Episodic

- Slow processing
- Contention under load
- Stale connections
- Memory leak
- Errors poison resources
- Component interactions
- Resource unavailable (e.g., network)

Systematic

- Missing index
- Scales badly with increased data
- Resource hog
- Death by 1000 cuts
- Overloaded environment
- Configuration error
Slow Query

- A single database statement takes too long...

Common Causes

- SQL not tuned
- Database contention
- Optimizer changing plan
- Database schema change (e.g., missing key) in some or all databases (script not run)

Challenges

- What operation causes the slow query
- Who is responsible
- Overall database performance may be fine
- Many requests may perform well
Thread Contention

- Multiple threads (or processes) queuing for locks
  - e.g., Java synchronized methods
  - Worst case is deadlocks

- Common Causes
  - Too much synchronization (e.g., locking at many layers)
  - Performing unnecessary slow work with a lock (e.g., remote calls)
  - Forcing readers to queue when only writers need exclusive access

- Challenges
  - Synchronization is *needed* to avert race conditions
Connection Failure

- Inability to establish connection to database

**Common Causes**
- Exhaustion of pool from abandoned connections under load
- Network and database outages
  - Immature implementations may not reestablish, leading to stale pooled connections. May result in intermittent failures.
- Changed configuration

**Challenges**
- Detecting transient failures (bad interactions)
- Consistent configuration (*e.g.*, in deployment)
- Detecting connection failure as a root cause
  - In the noise of cascading system failures
  - Frameworks that don’t chain exceptions
Transient Problems

- Interactions change system state gradually over time
- Simple
  - How could there be a null pointer exception here?
- Complex
  - Connections failed and were reestablished
  - Internal state became inconsistent
  - Those requests put bad data into session
  - Requests subsequently crash
- Require *dynamic context* to understand
  - Otherwise very hard to recreate
- The hardest kind!
Part 3: Measuring Overhead for monitoring and for AOP
Measuring Monitoring Overhead

- Local Windows Client
  - 50 simulated concurrent users

- Workstation Configuration
  - 2.8GHz Pentium M, 2GB RAM, Win XP SP2
  - Tomcat 5.5.9
  - JRockIt 1.5.0_03 VM

- Application
  - iBatis 1.3 JPetstore
  - AspectJ 1.5.0 dev build Oct. 21
### Measured Overhead

<table>
<thead>
<tr>
<th>Monitoring</th>
<th>End to End Response Time (ms)</th>
<th>Server Start Up Time (s)</th>
<th>Heap Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabled</td>
<td>20</td>
<td>15</td>
<td>135</td>
</tr>
<tr>
<td>Disabled (at Runtime)</td>
<td>10</td>
<td></td>
<td>130</td>
</tr>
<tr>
<td>Not Deployed</td>
<td>10</td>
<td>5</td>
<td>65</td>
</tr>
</tbody>
</table>
Other AOP Performance Measurements

- Micro-benchmarks
  - AOP Benchmark from AspectWerkz
  - abc compiler benchmarks
  - AspectJ internal benchmark suite
  - Haupt & Mezini: micro-measurements

- System Benchmarks
  - Hugunin & Hilsdale: XSLTMark tracing
    - 22% slower when tuned for basic policy
    - 24% faster with static field optimization
  - Zhang & Jacobsen: ORBacus refactoring
    - 8% slower with all aspects included
    - 6% faster without unnecessary aspects
Performance Results Summary

- **AspectJ performance usually within 5% of handwritten equivalent logic**
  - Typical advice overhead 10-80 ns/invocation
  - If it’s important, test and tune
  - AOP can often make crosscutting optimizations...

- **Small caveats**
  - Use of inefficient methods (e.g., Class.getName())
  - Generally performance issues can be resolved (and AOP also enables better optimization)
  - Older AspectJ lacks some optimizations (pertypewithin, lazy thisJoinPoint, cflow)
  - Build or weave times noticeably slower
Part 4: Improving Performance
Optimizing with AOP

- AOP can also help to
  - Identify bottlenecks (when optimizing/tuning)
  - Improve performance

- AOP complements
  - Good system architecture
  - Good (improved) algorithms
  - Profilers
AOP to Improve Performance

- **Caching**
  - Storing previously calculated values

- **Prefetching**
  - Retrieving information before it’s required

- **Changing Concurrency**
  - Altering locking or conflict detection for parallel use

- **Changing query strategy**
  - Using indexes or simpler queries in a common case

- **Batching distributed requests**
  - Sending many updates in one message
A Constant Cache

```java
public aspect CacheTaxCalculation {
    private Double Account.cachedTaxRate;

    double around(Account account) :
        execution(double calcTaxRate(..)) && this(account) {
            double value;
            if (account.cachedTaxRate == null) {
                value = proceed(account);
                account.cachedTaxRate = new Double(value);
            } else {
                value = account.cachedTaxRate.doubleValue();
            }

            return value;
        }
}
```
An Event-Based Invalidating Cache

```java
public aspect CacheTaxCalculation {
    private Object Account.cachedTaxRate;

    double around(Account account) :
        // same as before...

    before(Account account) : set(* Account+.*) &&
        target(account) && !within(CacheTaxCalculation) {
            account.cachedTaxRate = null;
        }
}
```

Invalidate the cache if Account changes state  
*i.e.* if a field within Account changes  
Could can extend to use observer (*e.g.*, if `TaxRegulation` changes)
A Time-Based Invalidating Cache

```java
public aspect CacheTaxCalculation {
    private Double Account.cachedTaxRate;
    private long Account.cacheWriteTime;
    private static int INVALIDATION_TIME = 5000; // 5s

    double around(Account account) :
        execution(double calcTaxRate(..)) && this(account) {
            double value;
            if (!account.hasValidCache()) {
                account.cachedTaxRate = new Double(proceed(account));
                account.cacheWriteTime = System.currentTimeMillis();
            }
            return account.cachedTaxRate.doubleValue();
        }

    ...
```
Checking Cache Expiry

```java
public boolean Account.hasValidCache() {
    return account.cachedTaxRate != null &&
    account.cacheWriteTime + INVALIDATION_TIME >
    System.currentTimeMillis();
}
```
Part 5: Conclusion
Slow Query

- Useful Information
  - Database statistics
  - Thread dumps / profiler / database connection monitor
  - Application server statistics

- Need to determine
  - What environments are slow?
  - What situations cause slow queries (what code paths/data)?
  - What has changed?
Thread Contention

- Useful Information
  - Thread dumps
  - Application server statistics
  - System-wide performance

- Need to determine
  - What kinds of load cause contention?
  - What environments are slow?
  - What has changed?
Connection Failure

- Useful Information
  - Application configuration
  - Database and network monitoring output
  - Database statistics
  - Thread dumps / profiler / database monitor
  - Server Logs
  - Application server statistics (e.g., pool sizes)

- Need to determine
  - Correlations?
  - What has changed?
Web Services Monitoring Revisited

Running the CrankyPetStore Demo based on iBATIS JPetStore
Use of AOP

- AspectJ scales up to real world problems
- There’s a lot more to know than simple logging!
- Maturing
  - Tools & technology (*e.g.*, load-time weaving)
  - Designs, patterns & best practices
  - Load-time weaving is a major advance
- Recommend incremental adoption
Conclusion

- Real need to track performance ...
  - With increasing scale ...
    and integration complexity
  - With correlated data and exposed
  - To effectively analyze in real time

- AOP is a big improvement over traditional approaches
  - Modularity instead of scattered and tangled, redundant, unreliable, ad hoc monitoring
  - Allows visibility, testability, maintainability, reuse

- Glassbox Inspector
  https://glassbox-inspector.dev.java.net/

- Glassbox http://www.glassbox.com

- Performance monitoring with AspectJ, Part 1 article at
Thank You!

Ron Bodkin
rbodkin@newaspects.com
New Aspects of Software