Experiences with Multi-threading and Dynamic Class Loading in a Java Just-In-Time Compiler

Daryl Maier, Pramod Ramarao, Mark Stoodley, Vijay Sundaresan
TestaRossa JIT compiler team
IBM Toronto Laboratory
Outline

- Overview of J9/TestaRoss
- Brief overview of our paper
  - Class loading/unloading
  - Profiling in a multi-threaded environment
  - Code patching
- Focus on code patching
  - Because it’s cool!
- Summary
J9 virtual machine

- High performance production Java VM from IBM
- Java 1.4.2 and Java 5.0 compliant
- Common code base for J2SE and J2ME products
- Support on 12 different processor/OS platforms
- Used in hundreds of IBM products including
  - Websphere Application Server (WAS 6.x)
  - Rational Application Developer (RAD)
  - DB2
  - XML parsers
TR (TestaRossa) JIT compiler

- Just-In-Time (JIT) compiler for J9 VM
- Fast startup time
- Adaptive compilation: multiple optimization levels
- Target ‘hot spots’ with higher opt level
- Classical and Java-specific optimizations
- Speculative optimizations
  - Low overhead PDF (profiling) framework
  - Code patching in many scenarios
# Program characteristics

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Loaded Classes</th>
<th>Unloaded Classes</th>
<th>Number of threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECjvm98</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>compress</td>
<td>383</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>jess</td>
<td>523</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>db</td>
<td>378</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>javac</td>
<td>537</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>431</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>mtrt</td>
<td>404</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>jack</td>
<td>429</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>SPECjbb2000</td>
<td>1098</td>
<td>0</td>
<td>8*</td>
</tr>
<tr>
<td>Trade6</td>
<td>11639</td>
<td>341</td>
<td>&gt;&gt; 10</td>
</tr>
</tbody>
</table>

- Middleware programs load order of magnitude more classes
- Memory leak: classes must be unloaded on an ongoing basis
- Lots of active threads executing tons of code: no method-level hotspots
- Target only jvm/jbb: ignore critical correctness/performance issues!
The One Page Paper Overview

- Class loading and unloading
  - Unloading a class requires significant clean-up
  - Danger of class references materialized in the code
- Profiling when there are a lot of threads
  - Must ensure timely recompilation and good scalability
- Code patching
  - Resolution, efficient dispatch, recompilation, speculative optimizations
  - Tricky stuff
Code Patching Overview

- Code patching scenarios, from easy to hard
  1. All threads stopped (scalability suffers)
  2. Single site, many active threads
  3. Multiple sites, many active threads
- Patch site alignment problems
- Trade-offs impact designs on each platform
  - e.g. number of PIC slots
Code Patching Example: Intel IA32 Field Resolution

- **Store to unresolved field**
  - Field offset unknown at compile-time

- **When writing instruction, offset initialized to 0**
  - Opcode, operand bytes assume largest offset (4B)
Code Patching Example Cont...

- **Resolution by site-specific generated code**
  - Calls a VM function to resolve the field

```
0x803F
E8 3B 01 00 00

resolveField803F:
0x8180
E8 DF 23 97 00

0x8185
89 82 00 00 00 00
```

Instruction bytes are copied to a site-specific resolution code out of line from the main instruction sequence...

CALL resolveField

and the original bytes are overlaid with a 5-byte CALL that reaches the site-specific resolution code

original instruction bytes
Code Patching Example Cont…

- ‘resolveField’ determines field offset at runtime

The call to the site specific resolution code can now be replaced with the 6 bytes from the original instruction...
Code Patching Example Cont…

- **BUT there’s a problem…**
  - Atomic updates needed to guarantee other threads execute correct code
  - X86 can only patch $2^N$ bytes atomically: example needs 6B

- **Solution: atomically overlay a thread barrier (self loop)**
  - JMP -2 instruction for X86, similar on other platforms

- **Guarantee all processors observe barrier before patching**
  - Only one thread resolves the field
  - MFENCE, CLFLUSH instructions for X86
Code Patching Example Cont...

Spin loop prevents other threads from executing instruction as its being patched

Atomically inserted with LOCK CMPXCHG instruction followed by a memory fence

If the CMPXCHG failed, then branch to 0x803F: another thread put in the self-loop already
Field offset copied into original instruction (spin loop still present)

Followed by memory fence

Finally, spin loop removed with single 2-byte write

We’re done! ...or are we?
Code Patching Example: Still not correct

- **Patched bytes can’t straddle patching boundary**
  - Not all instruction stores guaranteed to be atomically visible
  - Patching boundary is address that can’t be straddled by code locations being patched…empirically:
    - 8-bytes on AMD K7 or K8 cores
    - 32-bytes on Intel Pentium 3
    - 64-bytes on Intel Pentium 4 and EM64T

- **Insert NOPs to align patchable instructions**
  - e.g. spin loop JMP-2 instruction can’t straddle patching boundary
  - Increases code footprint by 1-2% on AMD: need more NOPs
  - Extra NOPs can have surprising performance effects (!)
Code Patching Example Cont…

- Single byte NOP inserted to align spin loop site
  - Patching infrastructure otherwise unaffected

First two bytes no longer straddle a patching boundary
Code Patching On Other Architectures

- **pSeries**
  - Uniform instruction length
  - Challenges:
    - Multiple instructions required for immediate addresses

- **zSeries**
  - Variable instruction length
  - Challenges:
    - Overcoming I-cache coherence costs, efficiency of atomic instructions
Summary

- Middleware applications are highly multi-threaded and load and unload LOTS of classes
  - Implications for patching, profiling, optimization design

- Our paper describes
  - Class unloading pain
  - Profiling correctly when lots of threads around
  - Code patching trickiness
Backup Slides
Contributions of Our Paper

- Highlight issues relevant in a production JIT compiler running multi-threaded and/or large applications
  - Class loading and unloading
  - Best code patching techniques vary by platform
  - Low overhead profiling with multiple active threads

- Describe our solutions to these problems
Class loading and CHTable

- **Class loading is not a ‘stop the world’ event**
  - Allows other Java threads to make progress while one thread loads a class
  - Allows compilation thread to compile while classes are being loaded

- **JIT compiler maintains a class hierarchy table**
  - Superclass/interface relationships are updated
  - Compensate for violated run time assumptions
  - All updates performed after acquiring CHTable lock
  - Compiler does not hold CHTable lock throughout a compilation
  - Compile-time CHTable queries must acquire CHTable lock
Class loading and CHTable

- **JIT compiler optimizations using class hierarchy table**
  - Guarded devirtualization
    - Conditionally convert virtual call to direct call
    - Assumption is registered in the CHTable
    - If assumption is violated, compensate at run time
    - Patch code to execute the backup code (virtual call)
  - Invariant argument pre-existence
    - Devirtualize virtual calls using an invariant parameter as a receiver
    - Re-compile the method containing devirtualized calls if assumption is violated

- **Class hierarchy might have changed while a compilation was in progress**
  - Acquire CHTable lock just before binary code is generated
  - Generate binary code
  - Compensate for any assumptions violated during the compilation
  - Release CHTable lock
Garbage collection and class unloading in the J9 VM

- **Class unloading**: memory allocated for class is re-claimed and the class ‘dies’

- **Class unloading done during garbage collection**

- **Garbage collection is a ‘stop the world’ event in J9**
  - Co-operative model (Java threads execute ‘yield points’ to check if GC is pending)
  - Java classes are also objects on the heap and can therefore be collected (and unloaded)
  - Class objects are never ‘moved’, i.e. a class is always at the same address throughout it’s lifetime

- **All classes in a class loader unloaded together**

- **A class is unloaded when**
  - No objects of that class type are ‘live’ on the Java heap
  - No loaded bytecodes explicitly refer to the class
Class unloading in the J9 VM

Impacts the JIT compiler significantly

- Class hierarchy table
- Profiling data
- Compilation issues
- Code memory reclamation
- Persistent data reclamation
- ‘Materialized’ references in generated code
Class unloading and ‘materialized’ references

Interface I { public void foo(); }

class C1 implements I
{
    public void foo() { System.out.println("In C1.foo"); }
}

class C2 implements I
{
    public void foo() { System.out.println("In C2.foo"); }
}
Class unloading and ‘materialized’ references

    public I createC1orC2(int x) {
        if (x % 2)
            return new C1();
        else
            return new C2();
    }

    public void bar() {
        x++;
        I obj = this.createC1orC2(x);
        obj.foo(); // Polymorphic interface call
    }
Class unloading and ‘materialized’ references

De-virtualized interface call conditionally

```java
public void bar() {
    x++;
    I obj = this.createClorC2(x);
    if (obj.class == C1)    // ‘materialized’ reference to C1
        C1.foo();    // called with obj as the receiver object
    else if (obj.class == C2)    // ‘materialized’ reference to C2
        C2.foo();   // called with obj as the receiver object
    else
        obj.foo(); // Polymorphic interface call
}
```
Class unloading and ‘materialized’ references

After replacing ‘materialized’ reference when C1 is unloaded

```java
public void bar() {
    x++;
    I obj = this.createC1orC2(x);
    if (obj.class == -1)   // 'materialized' reference to C1 changed
        C1.foo();   // called with obj as the receiver object
    else if (obj.class == C2)  // 'materialized' reference to C2
        C2.foo();   // called with obj as the receiver object
    else
        obj.foo(); // Polymorphic interface call
}
```
Class unloading and ‘materialized’ references

- List of code locations containing ‘materialized’ references is maintained for each class
- Addition to the list is done both at compile time and at run time
- Only add to the list if the class loader of ‘materialized’ class is different from the class loader of some other class referred to in the constant pool
  - Compare with class loader of method being compiled
  - Compare with class loader of super class/interface referred to in the constant pool
- Patching can be done without any race conditions because all threads have yielded for a GC
Class unloading and CHTable

- Remove unloaded classes from superclasses/interfaces in CHTable
- Grouping unloading requests avoids excessive traversals over data structures
  - Problematic scenario
    - Interface I is implemented by N classes
    - Each implemented class loaded by a different class loader (N class loaders)
    - Each class loader is unloaded and CHTable updates are performed independently
    - O(N²) to remove all implementors of I
    - We have seen N ~ 10,000 in customer applications
Class unloading and compilation

- **Asynchronous compilation**
  - Java threads queue methods for compilation and continue executing (in most cases)
- **Class containing a queued method could be unloaded before it is actually compiled**
  - Solution: Walk the compilation queue every time a class is unloaded and delete methods that belonging to the class
- **Class might be unloaded when a compilation is in progress**
  - Solution: Check if an unloaded class was used by the compilation in any manner; if so, abort the compilation
Class unloading and profiling

- Minimize work at run time and instead, move work to compile time as much as possible
- Profile data is for Java bytecodes that have been unloaded
  • Raw data is generated while program runs
  • Periodically, raw data is read and 'processed'
  • Bytecodes that generated raw data might have been unloaded
  • Solution: purge all raw data when class unloading occurs
  • What about 'processed' data for unloaded code?
  • Solution: maintain bytecode address range for unloaded code and avoid returning information from compile-time queries for profiling information for bytecodes in that range

- Profile data contains references to unloaded classes
  • Keep track of unloaded classes' addresses
  • Avoid returning class whose address matched an unloaded class
- Alternatives
  • Cleanse profiling data as unloading occurs (costly at run time?)
Class unloading and memory reclamation

- Common tasks like serialization sometimes create class loaders with short lifetimes
- Unbounded memory increase over time (server applications can run for days)
- Re-claim code and data memory for compiled method(s) in unloaded class
- Problem: Might involve expensive searches each time at run time
- Solution: Maintain per-class loader information about compiled methods and persistent data
  - Example: check if ‘any’ method belonging to an unloaded class loader was compiled
Profiling

- **When is profiling done**
  - Profile methods deemed to be 'hot' based on sampling

- **When a method is chosen to be profiled**
  - Compile the method with embedded profiling code
  - Execute the method body for a while collecting data
  - Recompile the method using profiling data
Profiling in the TR JIT

- Loosely based on Jikes RVM approach
  - Arnold et al (PLDI 2001)
- Compiler creates a clone of the method to be profiled
  - Clone contains the profiling code
- Transition paths at equivalent points allow flow of control between two bodies
  - Original method body executes more frequently
Profiling in the TR JIT (cont…)

- Profiling approach
  - Every \( M \) execution paths in the non-profiled version, transition to profiled version
  - Execute one execution path in the profiled version and transition back to non-profiled version
  - Do these steps \( N \) times

- ‘\( M \)’ is the profiling PERIOD
  - 19, 29, 47… (increasing number of back edges)

- ‘\( N \)’ is the profiling COUNT
  - 100, 625, 1250, 2500 … (increasing number of back edges)
Preliminaries

- **“Async checks”**
  - Inserted at each loop back edge to test if thread needs to yield to GC
  - Profiler uses async checks to mark loop back edges

- **“Execution Path”**
  - Starts at method entry or an async check
  - Ends at method entry or an async check

- **After one execution path is completed in profiled version, return to non-profiled version**
  - Ensures execution is not stuck in a loop in profiled version
Preliminaries (cont...)

Execution Path

1 --> 2 --> 6
6 --> 3 --> 5
5 --> 4 --> 5
6 --> 7 ... --> 1
5 --> 6
Profiling Transitions

Non-profiled

1
2
3
4
5
6
7

Profiled

1
2
3
4
5
6
7
Profiling Transitions (cont...)

METHOD ENTRY
if (recompCounter < 0)
    RECOMPILE

profilingCount < 0

profilingPeriod--

asyncChk
profilingPeriod < 0

N1:
...program code...

asyncChk
goto N1

P1:
profilingPeriod = PERIOD
profilingCount--

profilingCount > 0

recompCounter--
profilingPeriod = MAXINT

P2:
...program code...
Effects of Multi-threading

- Recompilation may not occur for a long time

Initially

Thread1: count = 0 period = MAXINT
Thread2: period--
Thread3: at method entry
Thread Interaction

Thread3: set initial values

count = 29 period = 625

Thread3

......

Thread2

profilingPeriod--

......

Thread1: read period (MAXINT)

Thread1:

period-- => period = MAXINT-1

Thread1: count = 0 period = MAXINT

profilingCount < 0

recompCounter-- profilingPeriod = MAXINT

count = 29 period = MAXINT-1
transition won't occur for a long time!
Effects of Multi-threading

- Poor scalability with increasing number of threads
  - Multiple threads could transition to profiling code
  - Possibility of threads manipulating ‘period’ multiple times
  - Guarantee of profiling path being executed once every PERIOD paths no longer true

- Imprecision in basic block profiling counts
  - Multiple threads may manipulate basic block counts
  - Basic block counts may no longer reflect the hotness of an execution path
Profiling in the TR JIT

- To improve scalability, use synchronization to access global 'period' and 'count' variables

- **At Method Entry**
  - Synchronization is used to read global variables into thread-local storage
  - Basic block counters are also thread-local

- **At Method Exit**
  - Global variables are updated from thread-local storage at each method exit under synchronization

- **Adds overhead**
  - Each thread has now to allocate extra storage
  - Two locking operations introduce runtime overhead
Results

- **Statistics of stack usage and runtime overhead of synchronization in profiled methods**
  - Only period and count variables are allocated as thread-local
  - All counters are allocated as thread-local (including basic block counts)
- **Average stack usage increase was 14.7% across SPECjvm98 and SPECJbb2000**
- **Runtime overhead was negligible**
## Stack usage

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>NumSlots</th>
<th>% Increase</th>
<th>NumSlots</th>
<th>% Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>_201_compress</td>
<td>81</td>
<td>87</td>
<td>7.4</td>
<td>348</td>
</tr>
<tr>
<td>_202_jess</td>
<td>100</td>
<td>105</td>
<td>5.0</td>
<td>832</td>
</tr>
<tr>
<td>_209_db</td>
<td>50</td>
<td>58</td>
<td>16.0</td>
<td>260</td>
</tr>
<tr>
<td>_213_javac</td>
<td>258</td>
<td>293</td>
<td>13.5</td>
<td>2462</td>
</tr>
<tr>
<td>_222_mpegaudio</td>
<td>111</td>
<td>121</td>
<td>9.0</td>
<td>311</td>
</tr>
<tr>
<td>_227_mtrt</td>
<td>99</td>
<td>112</td>
<td>13.1</td>
<td>763</td>
</tr>
<tr>
<td>_228_jack</td>
<td>148</td>
<td>211</td>
<td>42.5</td>
<td>1462</td>
</tr>
<tr>
<td>SPECjbb2000</td>
<td>183</td>
<td>203</td>
<td>10.9</td>
<td>954</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td><strong>14.7</strong></td>
<td><strong>562</strong></td>
</tr>
</tbody>
</table>
Results (cont…)

- Only _202_jess shows some overhead
  - Contains many small methods that get profiled
- Runtime overhead in the two multi-threaded benchmarks were negligible