



IBM Software Group

# Challenges to Improving the Performance of Middleware Applications

Mark Stoodley (speaking for myself 😊)  
Toronto Lab, IBM Canada Ltd.

# Overview

- **SPECjvm98 and SPECjbb2000 conclusions**
- **Application Servers and Middleware**
  - Overview
  - Observations from a JIT guy
- **Implications for JIT & VM optimizations**
- **Summary**

## (Some) (Random) Conclusions from SPECjvm & SPECjbb

- Hotspots exist where compile resources can be spent
- Loops are really important for performance
- Compile time doesn't matter much
- Not a lot of active threads
- Virtual and interface call sites are mostly monomorphic
- Interfaces have few implementors
- Class hierarchy is mostly tree-like and static
- Class loading and unloading is not important for performance
- Exceptions not important for performance
- **Not all of these are true for middleware/application servers**

# Application Servers and Middleware Applications

- **One “representative” middleware application is SPECjAppServer2004**
  - Models business processes of Fortune 500 company
  - Throughput metric but must meet quality-of-service metrics (e.g. 90% of orders responded to in < 2 seconds)
  - An official run must last one hour and be representative of a 24-hour run
  - System Under Test (SUT) consists of an application server as well as a database, driven by another machine

# App Server is a TOOL (Truly Object-Oriented Load)

- **Deep class hierarchy**
  - Factoring
  - Delegation
  - Reuse of libraries, components, services, etc.
- **Highly extensible**
  - Interfaces are everywhere
- **Critical path of most applications runs through app server, so lots of TOOL code gets executed**

# Observations I

- **1,000s of classes loaded at startup (startup time!)**
  - Highly interconnected hierarchy graph, many interfaces
- **10,000+ methods execute during a run**
  - Compiles can still happen during steady-state phase
- **Large method bodies, especially main code paths**
- **Flat profile: top method 1%-2% CPU (that's it!)**
- **Active code path is 100s if not 1,000s of methods**

## Observations II

- **Loops exist, but most contain invocations**
- **Deep call chains: Can be 100 frames deep**
- **Lots of app server threads (4-way: ~50 threads)**
- **Creates lots and lots (and lots ...) of objects**
  - Complex task
  - Inefficiencies in/inefficient use of class libraries
  - Data conversions to match APIs to enable reuse
- **Class load and unload events do happen**

## Implications for JIT and VM optimization: It's a big haystack...

- **Large, flat profile: hard to even find the fruit**
  - No hot methods to concentrate compile resources on
  - Startup time matters, so compiles must be effective
- **Lots of interfaces simplifies HUGE s/w design**
  - Interface invocations are common, polymorphic
  - Many implementers breaks some techniques
- **Active code path is huge**
  - Hardware doesn't always help as much as we'd like
  - Sampling-based profiling hits and misses a LOT of code

Implications for JIT and VM optimization:  
Difficult to “know” / “prove” things...

- **Class hierarchy is huge, call chains are deep**
  - Inlining still effective but thresholds quickly reached
  - Peeking doesn't work, can't visit the whole tree
  - Static analysis has problems: stale information, dynamic loading/unloading, changes in execution environment
- **Delegation, factoring move things further apart**
  - Allocations are deeper than frame that captures object
  - More calls become polymorphic (full virtual calls)
  - Restricts the scope of more powerful optimizations

## Implications for JIT and VM optimization: Dynamic collection is tricky...

- **Hard to make collection efficient**
  - Many classes & methods means lots of data
  - Many threads means synchronization limits throughput
  - Ultra-efficient sample storage or reduce sampling rate
  - Reduced sampling rate: fewer samples in given period or need longer period to collect same number of samples
  - BUT can only exploit information at a compile event
- **Still important: well thought-out design essential**

Implications for JIT and VM optimization:  
Things considered 'rare' aren't so rare...

- **“Rare” events actually happen, sometimes often**
  - Exceptions, class loading and unloading
  - Cache entries based on class pointers must be invalidated when class' loader is collected
  - Devirtualized calls must be patched on class loads
- **Other threads can be executing code mid-patch**
- **Correct patching more expensive than naïve patching**
  - correct & quick easier on some processors than others

# Summary

- **SPECjvm98 and SPECjbb2000 do not model performance characteristics of middleware**
- **Many implications for JIT and VM optimizations:**
  - There are no hot spots, so can't focus compile-time
  - Startup time is important so compile-time impact is critical
  - Program knowledge is better “hidden”
  - Data collection frameworks need to scale
  - Many threads: low overhead, lightweight synchronization
  - “Rare” events happen and impact common-case performance

# Contact Information and Acknowledgements

**Mark Stoodley**  
**Toronto Lab, IBM Canada Ltd.**  
**[mstoodle@ca.ibm.com](mailto:mstoodle@ca.ibm.com)**

**With thanks to:**

**Alan Adamson, Mike Fulton, Nikola Grcevski, Matt Hogstrom,  
Derek Inglis, Allan Kielstra, Mattias Persson, Andrew Spyker,  
Enyu Wang**

# Backup Slides

## Moving Forward

- **Middleware benchmarks are a tough target**
  - High entry cost (\$\$,effort,etc.) for analysis
  - Tuning app server, db for h/w setup has huge impact
  - Results will vary by particular h/w, app server, db
- **Alternative: evolve interface between researchers and practitioners**
  - Researchers can report additional helpful data
  - Practitioners can write more “experience” papers
  - Continued search for appropriate and useful benchmarks

## Useful information: some examples

- **Scalability (e.g. dynamic profile collection)**
  - Generated data size, # methods/classes, active threads, **per bm statistics give some idea of variation**
  - Impact of locking: atomic ops? Overheads? # threads?
  - Compile time **with context**
- **Can't analyze everything, of course**
  - Some ideas are still formative
  - Even a little of the data above is better than nothing