Java at Speed: Building a Better JVM

Simon Ritter
Deputy CTO, Azul Systems
azul.com
JVM Performance Graph: Ideal
JVM Performance Graph: Reality

- Bytecodes interpreted
- C1 JIT plus profiling
- C2 JIT with deoptimisations
- Steady optimised state
- GC pauses
Zing: A Better JVM
Azul Zing JVM

- Based on OpenJDK source code
- Passes all Java SE TCK/JCK tests
  - Drop in replacement for other JVMs
- Hotspot collectors replaced with C4
- Works in conjunction with Zing System Tools
  - Only supported on Linux
- Falcon JIT compiler
  - C2 replacement
- ReadyNow! warm up elimination technology
Zing System Tools

- Enables better memory management for JVM
- Memory freed by JVM is returned to kernel
- Allocation of new blocks comes from kernel
  - ZST knows cache status
  - Newly allocated blocks for TLAB are ‘hot’
  - Not like standard JVM
- Other clever tricks
  - Contingency memory
Azul Continuous Concurrent Compacting Collector (C4)
C4 Basics

- Generational (young and old)
  - Uses the same GC collector for both
  - For efficiency rather than pause containment
- Concurrent, parallel and compacting
- No STW compacting fallback
- Algorithm is mark, relocate, remap
Loaded Value Barrier

- Read barrier
  - Tests all object references as they are loaded
- Enforces two invariants
  - Reference is marked through
  - Reference points to correct object position
- Allows for concurrent marking and relocation
- Minimal performance overhead
  - Test and jump (2 instructions)
  - x86 architecture reduces this to one micro-op
Concurrent Mark Phase

- GC Threads
  - Root Set
    - X
    - X
    - X

- App Threads
  - X
  - X
Relocation Phase

A -> A'  B -> B'  C -> C'  D -> D'  E -> E'

Compaction
Quick Release

A -> A'  B -> B'  C -> C'  D -> D'  E -> E'

PHYSICAL

VIRTUAL
Remapping Phase

GC Threads

App Threads

A -> A'  B -> B'  C -> C'  D -> D'  E -> E'

X

X

X
Zing: Big Heaps, No Problem

- Scales to 8Tb heap
  - No degradation in pause times
- Use one big heap, rather than many small heaps
  - Less JVMs means more efficiency
- Zing does not require big heaps
  - But works well with them
Non-Zing GC Tuning Options
GC Tuning Used To Be Hard

Java -Xmx12g -XX:MaxPermSize=64M -XX:PermSize=32M -XX:MaxNewSize=2g
-XX:NewSize=1g -XX:SurvivorRatio=128 -XX:+UseParNewGC
-XX:+UseConcMarkSweepGC -XX:MaxTenuringThreshold=0
-XX:CMSInitiatingOccupancyFraction=60 -XX:+CMSParallelRemarkEnabled
-XX:+UseCMSInitiatingOccupancyOnly -XX:ParallelGCThreads=12
-XX:LargePageSizeInBytes=256m ...

Java -Xms8g -Xmx8g -Xmn2g -XX:PermSize=64M -XX:MaxPermSize=256M
-XX:-OmitStackTraceInFastThrow -XX:SurvivorRatio=2
-XX:-UseAdaptiveSizePolicy -XX:+UseConcMarkSweepGC
-XX:+CMSConcurrentMTEnabled -XX:+CMSParallelRemarkEnabled
-XX:+CMSParallelSurvivorRemarkEnabled
-XX:CMSMaxAbortablePrecleanTime=10000
-XX:+UseCMSInitiatingOccupancyOnly
-XX:CMSInitiatingOccupancyFraction=63 -XX:+UseParNewGC -Xnoclassgc...
GC Tuning Used To Be Hard

Java -Xmx12g -XX:MaxPermSize=64M -XX:PermSize=32M -XX:MaxNewSize=2g
   -XX:NewSize=1g -XX:SurvivorRatio=128 -XX:+UseParNewGC
   -XX:+UseConcMarkSweepGC -XX:MaxTenuringThreshold=0
   -XX:CMSInitiatingOccupancyFraction=60 -XX:+CMSParallelRemarkEnabled
   -XX:+UseCMSInitiatingOccupancyOnly -XX:ParallelGCThreads=12
   -XX:LargePageSizeInBytes=256m ...

Java -Xms8g -Xmx8g -Xmn2g -XX:PermSize=64M -XX:MaxPermSize=256M
   -XX:-OmitStackTraceInFastThrow -XX:SurvivorRatio=2
   -XX:-UseAdaptiveSizePolicy -XX:+UseConcMarkSweepGC
   -XX:+CMSConcurrentMTEnabled -XX:+CMSParallelRemarkEnabled
   -XX:+CMSParallelSurvivorRemarkEnabled
   -XX:CMSMaxAbortablePrecleanTime=10000
   -XX:+UseCMSInitiatingOccupancyOnly
   -XX:CMSInitiatingOccupancyFraction=63 -XX:+UseParNewGC -Xnoclassgc ...

© Copyright Azul Systems 2019
GC Tuning With Zing

java -Xmx1g

java -Xmx10g

java -Xmx100g

java -Xmx2t
Measuring Platform Performance

- **jHiccup**
  - Spends most of its time asleep
    - Minimal effect on performance
    - Wakes every 1 ms
    - Records delta of time it expects to wake up
    - Measured effect is what would be experienced by your application

- Generates histogram log files
  - These can be graphed for easy evaluation
Small Heap, Small Latency

Hazelcast 2-node system with 1Gb heap Hotspot v. Zing
Big Heap, Small Latency

Cassandra with 60Gb heap Hotspot v. Zing
Azul Falcon JIT Compiler
Advancing Adaptive Compilation

- Azul Falcon JVM compiler
  - Based on latest compiler research
  - LLVM project
- Better performance
  - Better intrinsics
  - More inlining
  - Fewer compiler excludes
- Replacement for C2 compiler
Simple Code Example

- Simple array summing loop
  - A modern compiler will use vector operations for this

```java
private int sumLoop(int[] a) {
    int sum = 0;
    for (int i = 0; i < a.length; i++) {
        sum += a[i];
    }
    return sum;
}
```
More Complex Code Example

- Conditional array cell addition loop
  - Hard for compiler to identify for vector instruction use

```java
private void addArraysIfEven(int a[], int b[]) {
    if (a.length != b.length) {
        throw new RuntimeException("length mismatch");
    }
    for (int i = 0; i < a.length; i++) {
        if ((b[i] & 0x1) == 0) {
            a[i] += b[i];
        }
    }
}
```
Traditional JVM JIT

Per element jumps
2 elements per iteration

```java
private void addArraysIfEven(int a[], int b[]) {
    if (a.length != b.length) {
        throw new RuntimeException("length mismatch");
    }
    for (int i = 0; i < a.length; i++) {
        if ((b[i] & 0x1) == 0) {
            a[i] += b[i];
        }
    }
}
```
Falcon JIT

Using AVX2 vector instructions
32 elements per iteration

Broadwell E5-2690-v4
Traditional JVM

Speed
(with contribution by optimization level)

Application Warm-up

Interpreted Tier1 (profiling) Optimized
ReadyNow! Solution

- Save JVM JIT profiling information
  - Classes loaded
  - Classes initialised
  - Instruction profiling data
  - Speculative optimisation failure data
- Data can be gathered over much longer period
  - JVM/JIT profiles quickly
  - Significant reduction in deoptimisations
- Able to load, initialise and compile most code before main()}
Effect Of ReadyNow!

Customer application
ReadyNow! Startup Time

Without ReadyNow!

With ReadyNow!

Class loading, initialising and compile time
Deterministic Compiler

Method for compilation

- Initial IR
  (Method bytecodes & live profile)
- Queries and responses
- Produced machine code

Given identical input
 Guarantees identical output
Add Compile Stashing

- Bytecode frontend
- VM callbacks
- Compiled methods

LLVM IR

Queries

Responses

Machine code

Compiled methods

LLVM

Zing JVM
Compile Stashing Effect

Without Compile Stashing

With Compile Stashing

Up to 80% reduction in compile time and 60% reduction in CPU load
Summary
JVM Performance Graph: Zing
The Zing JVM

- Start fast
- Go faster
- Stay fast

- Simple replacement for other JVMs
  - No recoding necessary

Try Zing free for 30 days:

azul.com/zingtrial
Thank you!

Simon Ritter
Deputy CTO, Azul Systems
azul.com