Don’t Get Caught In the Cold, Warm-up Your JVM

Understand and Eliminate JVM Warm-up Overhead in Data-parallel Systems

David Lion, Adrian Chiu, Hailong Sun*, Xin Zhuang, Nikola Grcevski†, Ding Yuan

University of Toronto, *Beihang University, †Vena Solutions
The JVM is Popular

- Systems are increasingly built on the JVM
- Popular for big data applications
- Increasingly used on latency-sensitive queries
"JVM performance has come a long way, but it will never match native code." - Quora User

"The most obvious outdated Java Performance fallacy is that it is slow." - InfoQ User

"Most work performed by HDFS and MapReduce is I/O, so Java is acceptable." - Hypertable Developer

"If you scale your application Java is likely fast enough for you." - StackOverflow user
An Analysis of JVM Overhead

• Surprisingly, warm-up overhead is the bottleneck
  • Bottlenecks I/O intensive workloads (33% in 1GB HDFS read)
  • Warm-up time stays constant (21s - Spark query)

  “There is a contradiction between parallelizing long running jobs into short tasks, and amortizing JVM warm-up overhead through long tasks.”

• Multi-layer systems aggravate the problem
HotTub: Eliminate Warm-up Overhead

- New JVM that can be reused across jobs

**HDFS 1MB Read**

- OpenJDK 8: 30.08X
- HotTub: 1

**Spark Query 100GB**

- OpenJDK 8: 60
- HotTub: 32
- Speedup: 1.80X

**Hive Query 100GB**

- OpenJDK 8: 35
- HotTub: 19
- Speedup: 1.79X
Outline

JVM Analysis

- Warm-up overhead *bottlenecks* I/O intensive workloads
- Warm-up overhead is constant
- Warm-up overhead increased on multi-layer systems

- HotTub: eliminate warm-up with reuse
  - Demo
Methodology

- Study HDFS, Spark, and Hive on Tez
  - Queries from BigBench [Ghazal’13] (TPC-DS)
  - Server components are fully warmed up
  - 10 node cluster: 16 virtual cores, 128GB RAM, 10GbE

- Instrument OpenJDK 8 to measure warm-up overhead

- Understanding overall slowdown is complicated
  - Blocked time analysis [Ousterhout’15]
  - Subtract warm-up overhead and simulate scheduling
HDFS: I/O Bottleneck is Warm-up

- Warm-up time remains relatively constant across data sizes
- Same classes loaded, same methods JIT-compiled
HDFS: I/O Bottleneck is Warm-up

- Warm-up time remains relatively constant across data sizes
  - Same classes loaded, same methods JIT-compiled

- Warm-up more than 33% of execution for 1GB
Zooming into HDFS Read Overhead

- 1GB file bottlenecked by client initialization warm-up overhead
  - 21% of entire execution is class loading at initialization

- Warm-up overhead dwarfs disk I/O

![HDFS 1GB Sequential Read Graph]

**HDFS 1GB Sequential Read**
- Compiled/native
- Interpreter
- Class loading

**Client init:**
- Red: Class loading
- Blue: Interpreter
- Green: Compiled/native

**Read:**
- Green: Compiled/native
- Blue: Interpreter
- Red: Class loading

**Time (s):**
0 0.5 1 1.5 2 2.5 3 3.5 4 4.5
HDFS Packet Overhead

- Datanode first sends acknowledgement to begin the read
HDFS Packet Overhead

- Client processing acknowledgement is very slow
- Bottleneck is class loading
HDFS Packet Overhead

- Datanode already started to send packets
HDFS Packet Overhead

- The client spends all of its time in warm-up
- CRC checksum interpretation is the bottleneck
**HDFS Packet Overhead**

- Datanode must wait because sendfile buffer is full
HDFS Packet Overhead

- Warm-up overhead slows even the actual I/O path
- 109 packets sent before client finishes processing 3
Spark and Hive Warm-up Overhead

- Warm-up overhead is constant
- Average - Spark: 21s, Hive: 13s
Spark and Hive Warm-up Overhead

- Up to 40% of runtime spent in warm-up overhead
- Spark queries run faster, but have more warm-up overhead
More Layers More Overhead

- Spark client loads 3 times more classes than Hive
  - 19,066 classes

- More classes generally leads to more interpretation time
  - Because more unique methods are invoked
JVM Reuse is Hard at Application Layer

- Spark and Tez already try to reuse JVMs
  - But only within a job

- Many challenges exist when reusing JVMs
  - Need to ensure same classes are used
  - Need to reset all static data
    - 3rd party libraries make this even harder
  - Close file descriptors
  - Kill threads
  - Signals
Outline

- JVM Analysis
  - Warm-up overhead bottlenecks I/O intensive workloads
  - Warm-up overhead is constant
  - Warm-up overhead increased on multi-layer systems

HotTub: eliminate warm-up with reuse
- Demo
HotTub: Reuse JVMs

- Modify OpenJDK 8 to **reuse** warm JVMs
  - Keeps a pool of warm JVM processes
  - Data-parallel systems have lots of opportunity for reuse

- Drop-in replacement
HotTub: Initial Run

- If no JVM exists create a new one
HotTub: Initial Run

- If no JVM exists create a new one
- Run application normally

```
$java

reusable JVM exists?

false

exec new JVM

Run App.
```
HotTub: Initial Run

- If no JVM exists create a new one
- Run application normally
- Reset JVM before adding to the pool
  - Clean up any threads
  - Reset static data to type default
  - Close file descriptors

```
$java
reusable JVM exists?
false
exec new JVM
Run App.
reset JVM
```

JVM pool
HotTub: Reuse Run

- Choose existing JVM from pool
- Ensure loaded classes are correct
- Reinitialize all static data

$java$

Reusable JVM exists?

true → reinit. JVM

false → exec new JVM

JVM pool

Reset JVM

Run App.
HotTub Demo
HotTub Evaluation

- Constant improvement across different workloads on a system
- Reusing a JVM from a different query has similar results
Limitations

- Security: limit reuse to same Linux user
  - Could see loaded classes and compiled methods
  - Similar to timing channel

- Not useful for long running JVMs

- Breaks the JVM specification for class initialization on edge cases
Related Work

• Garbage collection overhead in data-parallel systems
  • Yak [Nguyen'16], Taurus [Maas'16], Broom [Gog'15], etc
  • Not warm-up overhead

• Studies on data-parallel systems
  • [Ousterhout'15], [Pavlo'09], etc
  • Not targeting the JVM

• Studies on the cost of scalability [McSherry'15]

• Work on JVM unrelated to data-parallel systems
Conclusions

- Warm-up overhead is the bottleneck
  - Bottlenecks even I/O intensive workloads
  - Warm-up time stays constant
    - Bad when parallelizing with JVMs
  - Multi-layer systems aggravate warm-up overhead

- HotTub: eliminate warm-up through transparent JVM reuse

- Open sourced at: https://github.com/dsrg-uoft/hottub

Thank You
Extra Slides
HotTub Overheads

- Memory: In our tests an idle JVM took around 1GB memory
  - Can configure pool size (ideally pool is rarely idle)
- Garbage collection: ~200ms
  - Few roots, most objects are dead
    - All stacks ended + Static data set to type default (null)
- Class reinitialization:
  - Spark executor: 400ms, Spark Client: 720ms
  - Hive container: 350ms
  - Not overhead, but cannot be skipped on reuse
Spark and Hive Study

- Complete results
- GC not a factor for these short queries
HotTub Iterations

- 1MB HDFS Read performed repeatedly
- Run 0 is a new JVM
HotTub Cross-Query

- Spark 100GB
- Run training query 4 times then run testing query

<table>
<thead>
<tr>
<th></th>
<th>q11</th>
<th>q14</th>
<th>q15</th>
<th>q09</th>
<th>q01</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q11</td>
<td>1.78</td>
<td>1.67</td>
<td>1.51</td>
<td>1.49</td>
<td>1.55</td>
</tr>
<tr>
<td>q14</td>
<td>1.64</td>
<td>1.65</td>
<td>1.47</td>
<td>1.49</td>
<td>1.50</td>
</tr>
<tr>
<td>q15</td>
<td>1.72</td>
<td>1.67</td>
<td>1.62</td>
<td>1.54</td>
<td>1.62</td>
</tr>
<tr>
<td>q09</td>
<td>1.57</td>
<td>1.59</td>
<td>1.55</td>
<td>1.53</td>
<td>1.53</td>
</tr>
<tr>
<td>q01</td>
<td>1.76</td>
<td>1.74</td>
<td>1.65</td>
<td>1.54</td>
<td>1.74</td>
</tr>
</tbody>
</table>
Custom Class Loaders

- Instance re-created each run
- No consistency issues
- Cannot reuse

```java
Class CustomClassLoader {
    ...
}

Class Test {
    public static void foo() {
        CustomClassLoader ccl = new CustomClassLoader();
        Class clas = ccl.loadClass("classname");
        ...
    }
}
```
HotTub Consistency Limitations

- Timing dependencies
- Unpredictable and dangerous practice
- HotTub initializes classes before runtime

```java
1  Class ClassA {
2      static int a = 0;
3      void foo () { a++; }
4  }
5  Class ClassB {
6      static int b = ClassA.a;
7  }
```

Class A
Loaded
ClassA.a = 0
foo()
ClassA.a = 1
foo()
ClassA.a = 2
Class B
Loaded
ClassB.b = 2
HotTub Consistency Limitations

- Timing dependencies
- foo() could be called multiple times before B is initialized

```java
Class ClassA {
    static int a = 0;
    void foo () { a++; }
}
Class ClassB {
    static int b = ClassA.a;
}
```

- Class dependence cycles
- Problem for normal JVMs too

```java
Class ClassC {
    static int c0 = 10;
    static int c1 = ClassD.d0;
}
Class ClassD {
    // c1 = 10 in HotSpot
    // c1 = 0 in HotTub
    static int d0 = ClassC.c0;
    static int d1 = 12;
}
```
**Instrumentation**

- Goal: track changes between interpreter and compiled code
- Returns are hard to track

<table>
<thead>
<tr>
<th>addr</th>
<th>JVM stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2a0</td>
<td>parameters</td>
</tr>
<tr>
<td>0x2a8</td>
<td>ret addr: 0x8f0</td>
</tr>
</tbody>
</table>

**0x670: ret_handler**

1. push %rax
2. ...
3. push %rdx
4. callq _record_mode_change
5. callq _pop_ret_addr
6. movq %rax, %r11
7. pop %rdx
8. ...
9. pop %rax
10. jmp %r11
Instrumentation

- Copy original return address to thread local stack

```
0x670: ret_handler
1  push %rax
2  ...
3  push %rdx
4  callq _record_mode_change
5  callq _pop_ret_addr
6  movq %rax, %r11
7  pop %rdx
8  ...
9  pop %rax
10 jmp %r11
```
Instrumentation

- Copy our return handler into the return address

<table>
<thead>
<tr>
<th>addr</th>
<th>JVM stack</th>
<th>0x670: ret_handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2a0</td>
<td>parameters</td>
<td></td>
</tr>
<tr>
<td>0x2a8</td>
<td>ret addr: 0x670</td>
<td></td>
</tr>
</tbody>
</table>

```
1  push %rax
2  ...
3  push %rdx
4  callq _record_mode_change
5  callq _pop_ret_addr
6  movq %rax, %r11
7  pop %rdx
8  ...
9  pop %rax
10 jmp %r11
```
Instrumentation

- Return calls into our ret_handler
- Record mode change and pop original return address

<table>
<thead>
<tr>
<th>addr</th>
<th>JVM stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2a0</td>
<td>parameters</td>
</tr>
<tr>
<td>0x2a8</td>
<td>ret addr: 0x670</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>stack addr</th>
<th>ret addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2a8</td>
<td>0x8f0</td>
</tr>
</tbody>
</table>

0x670: ret_handler

1. push %rax
2. ...
3. push %rdx
4. callq _record_mode_change
5. callq _pop_ret_addr
6. movq %rax, %r11
7. pop %rdx
8. ...
9. pop %rax
10. jmp %r11
## Instrumentation

- Jump to original return address

<table>
<thead>
<tr>
<th>addr</th>
<th>JVM stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2a0</td>
<td>parameters</td>
</tr>
<tr>
<td>0x2a8</td>
<td>ret addr: 0x670</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ret addr stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack addr</td>
</tr>
</tbody>
</table>

### 0x670: ret_handler

1. push %rax
2. ...
3. push %rdx
4. callq _record_mode_change
5. callq _pop_ret_addr
6. movq %rax, %r11
7. pop %rdx
8. ...
9. pop %rax
10. jmp %r11 // 0x8f0
Spark and Hive Parallelization

- Parallelization: split long running jobs into short tasks
  - JVM warm-up overhead amortized when long running

- Jobs are smaller and faster than ever
  - 90% of Facebook’s analytics jobs <100GB input
  - Majority of Hadoop workloads read and write <1GB per-task
  - [Ousterhout’13] show a trend in increasingly short running jobs

- Hive on Tez: 1 JVM per task, Spark: 1 JVM per node