Kris Mok, Software Engineer, Taobao
@rednaxelafx
莫枢 / “撒迦”

JVM @ Taobao

Agenda

Customization

Tuning

JVM @ Taobao

Open Source

Training

INTRODUCTION

Java Strengths

- Good abstraction
- Good performance
- Good tooling (IDE, profiler, etc.)
- Easy to recruit good programmers

Java Weaknesses

- Tension between “abstraction leak” and performance
  - Abstraction and performance don’t always come together
- More control/info over GC and object overhead wanted sometimes

Our Team

- Domain-Specific Computing Team
  - performance- and efficency-oriented
  - specific solutions to specific problems
  - do the low-level plumbing to leverage new technologies
  - we’re hiring!

Our Team (cont.)

- Current Focus
  - JVM-level customization/tuning
    - based on HotSpot Express 20 from OpenJDK
  - Dedicated compression card integration with Hadoop

JVM CUSTOMIZATION @ TAOBAO

Themes

- Performance
- Monitoring/Diagnostics
- Stability

Tradeoffs

- Would like to make as little impact on existing Java application code as possible
- But if the performance/efficiency gains are significant enough, we're willing to make extensions to the VM/core libs

JVM Customizations

- GC Invisible Heap (GCIH)
- JNI Wrapper improvement
- New instructions
- PrintGCReason / CMS bug fix
- ArrayAllocationWarningSize
- Change VM argument defaults
- etc.

Case 1: in-memory cache

- Certain data is computed offline and then fed to online systems in a read-only, “cache” fashion

in-memory cache

- Fastest way to access them is to
  - put them in-process, in-memory,
  - access as normal Java objects,
  - no serialization/JNI involved per access

in-memory cache

- Large, static, long-live data in the GC heap
  - may lead to long GC pauses at full GC,
  - or long overall concurrent GC cycle
- What if we take them out of the GC heap?
  - but without having to serialize them?

GC Inivisible Heap

- “GC Invisible Heap” (GCIH)
  - an extension to HotSpot VM
  - an in-process, in-memory heap space
  - not managed by the GC
  - stores normal Java objects
- Currently works with ParNew+CMS

GCIH interface

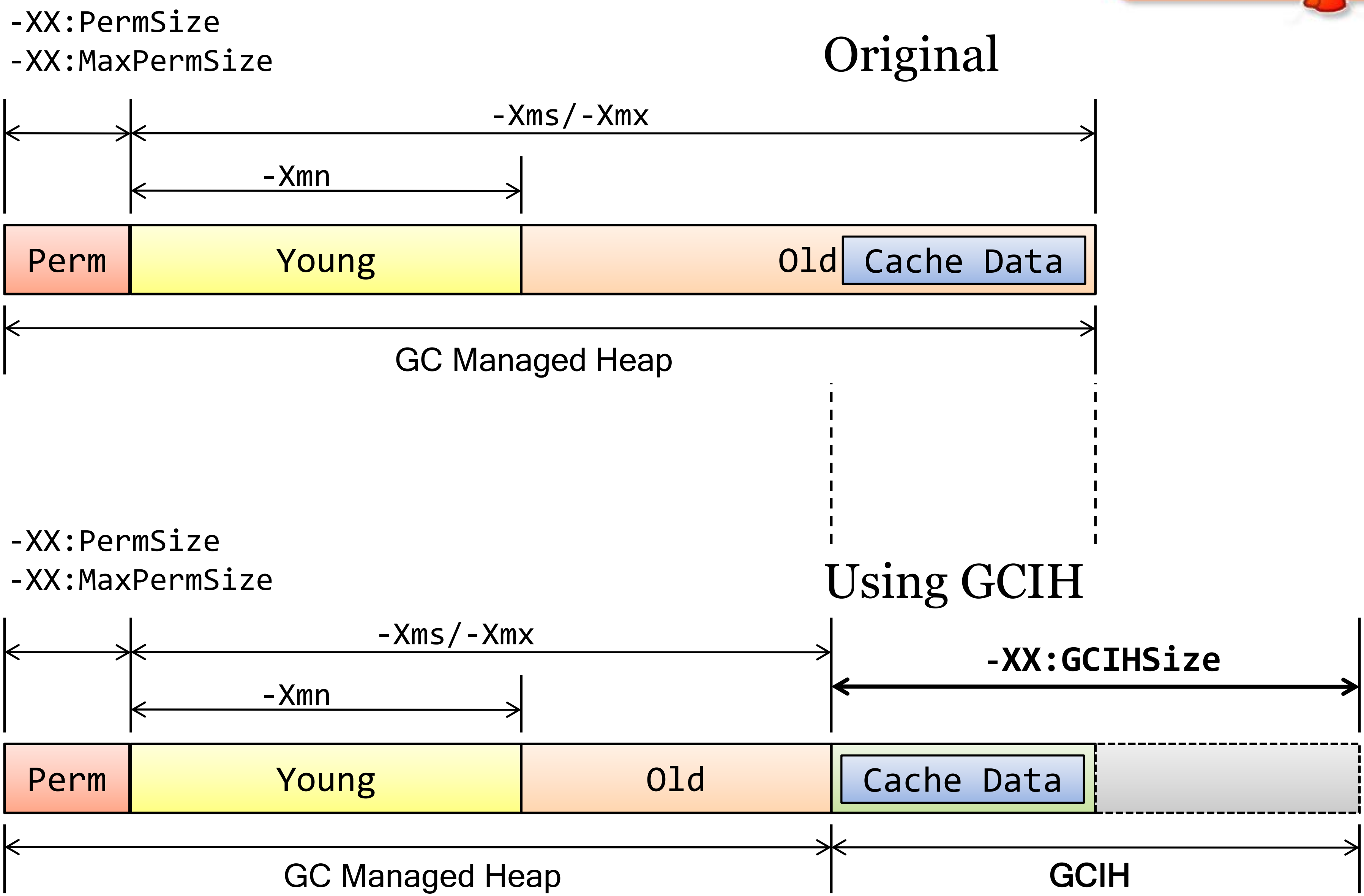
- “`moveIn(Object root)`”
  - given the root of an object graph, move the whole graph out of GC heap and into GCIH
- “`moveOut()`”
  - GCIH space reset to a clean state
  - abandon all data in current GCIH space
  - (earlier version) move the object graph back into GC heap

GCIH interface (cont.)

- Current restrictions
  - data in GCIH should be read-only
  - objects in GCIH may not be used as monitors
  - no outgoing references allowed
- Restrictions may be relaxed in the future

GCIH interface (cont.)

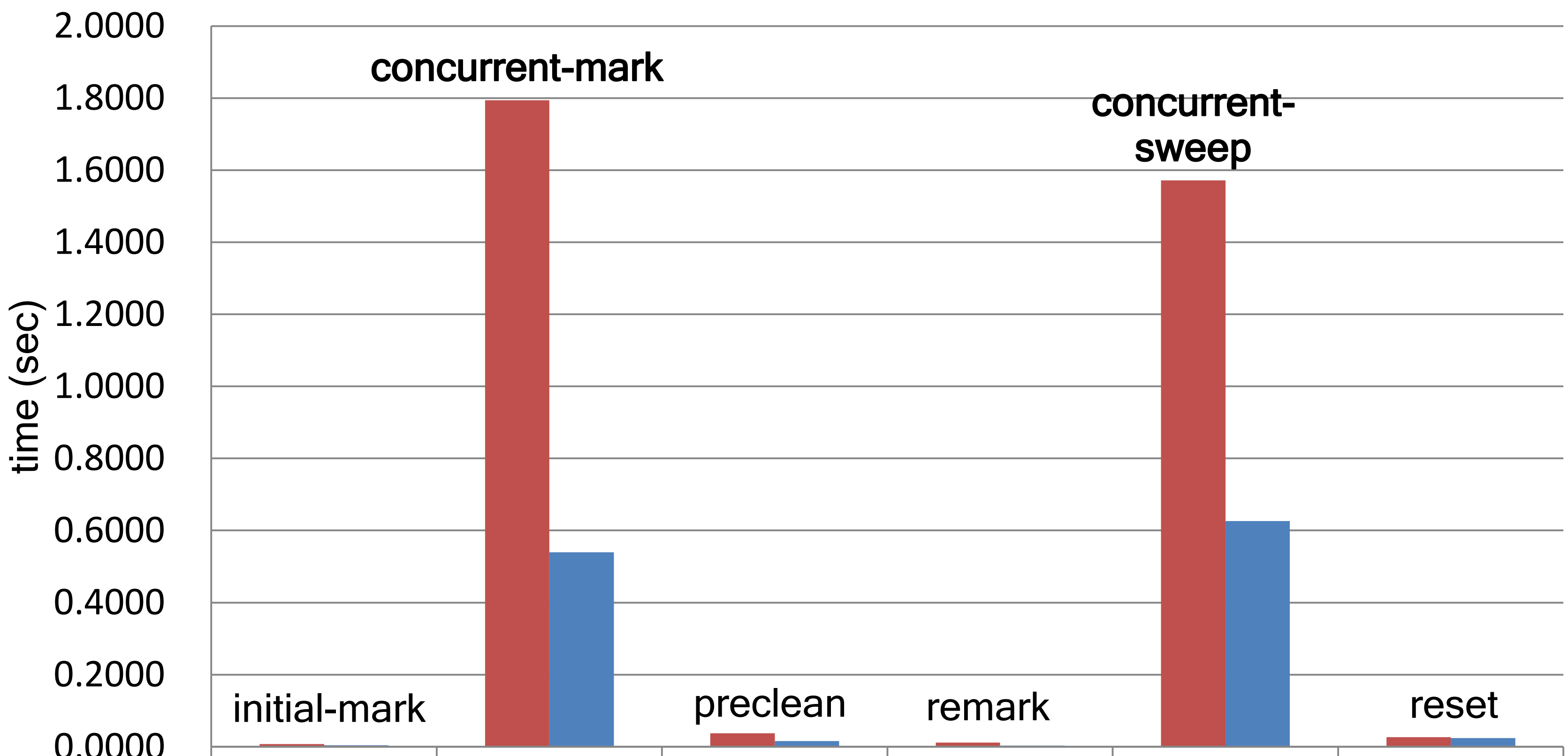
- To update data
  - `moveOut` - update - `moveIn`


-XX:PermSize
-XX:MaxPermSize
Original
-Xms/-Xmx
-Xmn
Perm
Young
Old
Cache Data
GC Managed Heap
-XX:PermSize
-XX:MaxPermSize
Using GCIH
-Xms/-Xmx
-XX:GCIHSize
-Xmn
Perm
Young
Old
Cache Data
GC Managed Heap
GCIH

Actual performance

- Reduces stop-the-world full GC pause time
- Reduces concurrent-mark and concurrent-sweep time
  - but the two stop-the-world phases of CMS aren’t necessarily significantly faster

Total time of CMS GC phases



	1	2	3	4	5	6
Original	0.0072	1.7943	0.0373	0.0118	1.5717	0.0263
w/GCIH	0.0043	0.5400	0.0159	0.0035	0.6266	0.0240

Alternatives

GCIH

- × extension to the JVM
- √ in-process, in-memory
- √ not under GC control
- √ direct access of Java objects
- √ no JNI overhead on access
- √ object graph is in better locality

BigMemory

- √ runs on standard JVM
- √ in-process, in-memory
- √ not under GC control
- × serialize/deserialize Java objects
- × JNI overhead on access
- × N/A

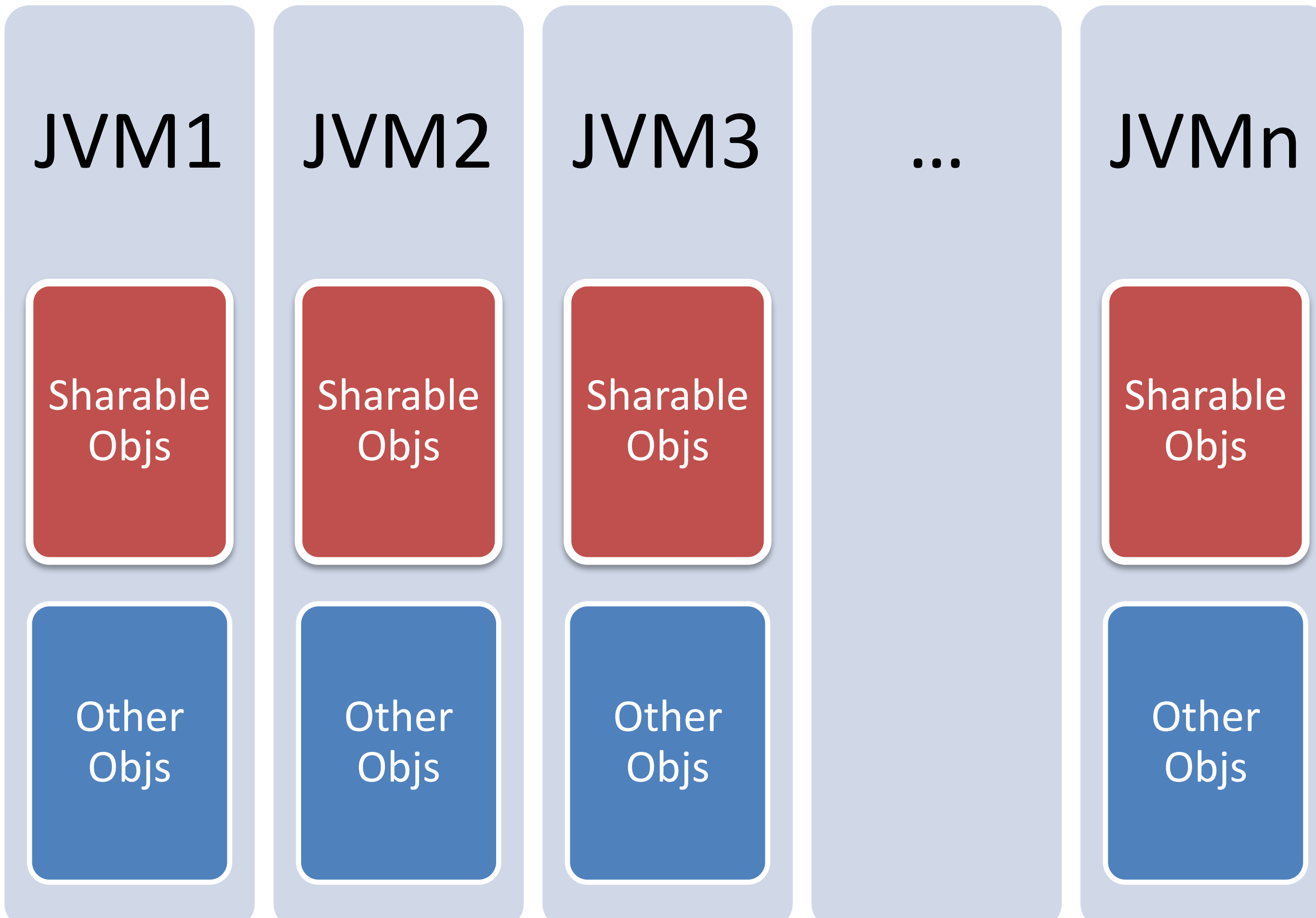
GCIH future

- still in early stage of development now
- may try to make the API surface more like RTSJ

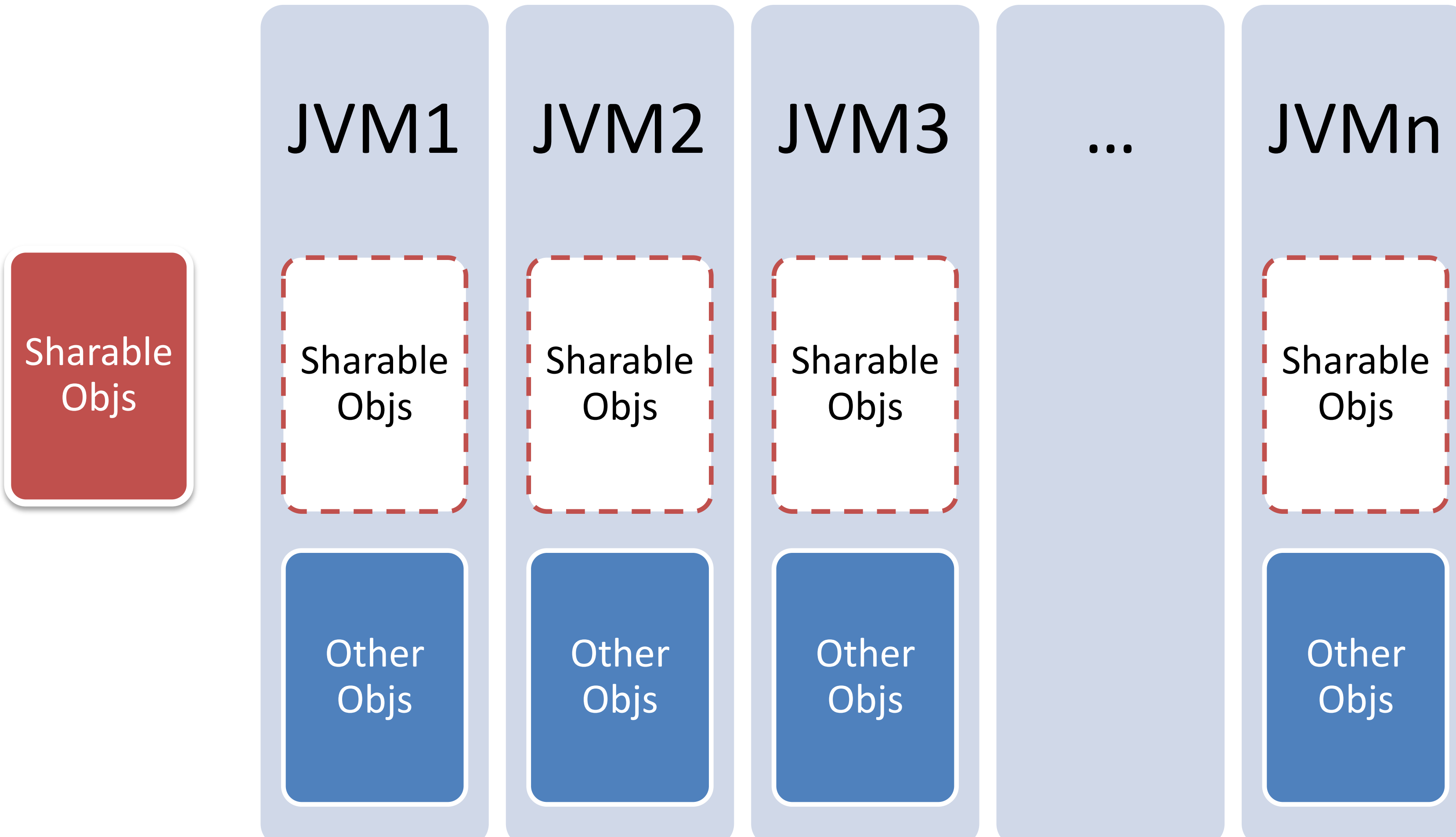
Experimental: object data sharing

- Sharing of GCIH between JVMs on the same box
- Real-world application:
  - A kind special Map/Reduce jobs uses a big piece of precomputed cache data
  - Multiple homogenous jobs run on the same machine, using the same cache data
  - can save memory to run more jobs on a machine, when CPU isn't the bottleneck

Before sharing



After sharing



Case 2: JNI overhead

- JNI carries a lot overhead at invocation boundaries
- JNI invocations involves calling JNI native wrappers in the VM

JNI wrapper

- Wrappers are in hand-written assembler
- But not necessarily always well-tuned
- Look for opportunities to optimize for common cases

Wrapper example

```
...
0x00002aaaab19be92:     cmpl   $0x0,0x30(%r15) // check the suspend flag
0x00002aaaab19be9a:     je     0x2aaaab19bec6
0x00002aaaab19bea0:     mov    %rax,-0x8(%rbp)
0x00002aaaab19bea4:     mov    %r15,%rdi
0x00002aaaab19bea7:     mov    %rsp,%r12
0x00002aaaab19beaa:     sub    $0x0,%rsp
0x00002aaaab19beae:     and    $0xfffffffffffffff0,%rsp
0x00002aaaab19beb2:     mov    $0x2b7d73bcbda0,%r10
0x00002aaaab19bebc:     rex.WB callq  *%r10
0x00002aaaab19bebf:     mov    %r12,%rsp
0x00002aaaab19bec2:     mov    -0x8(%rbp),%rax
0x00002aaaab19bec6:     movl   $0x8,0x238(%r15)  //change thread state to
thread in java
... //continue
```

Wrapper example (cont.)

- The common case
  - Threads are more unlikely to be suspended when running through this wrapper
- Optimize for the common case
  - move the logic that handles suspended state out-of-line

Modified wrapper example

```
...
0x00002aaaab19be3a:      cmpl   $0x0,0x30(%r15) // check the suspend flag
0x00002aaaab19be42:      jne    0x2aaaab19bf52
0x00002aaaab19be48:      movl   $0x8,0x238(%r15) //change thread state to
thread in java

... //continue

0x00002aaaab19bf52:      mov    %rax,-0x8(%rbp)
0x00002aaaab19bf56:      mov    %r15,%rdi
0x00002aaaab19bf59:      mov    %rsp,%r12
0x00002aaaab19bf5c:      sub    $0x0,%rsp
0x00002aaaab19bf60:      and    $0xfffffffffffffff0,%rsp
0x00002aaaab19bf64:      mov    $0x2ae3772aae70,%r10
0x00002aaaab19bf6e:      rex.WB callq  *%r10
0x00002aaaab19bf71:      mov    %r12,%rsp
0x00002aaaab19bf74:      mov    -0x8(%rbp),%rax
0x00002aaaab19bf78:      jmpq   0x2aaaab19be48
...
```

Performance

- 5%-10% improvement of raw JNI invocation performance on various microarchitectures

Case 3: new instructions

- SSE 4.2 brings new instructions
  - e.g. CRC32c
- We’re using Westmere now
- Should take advantage of SSE 4.2

CRC32 / CRC32C

- CRC32
  - well known, commonly used checksum
  - used in HDFS
  - JDK’s impl uses zlib, through JNI
- CRC32c
  - an variant of CRC32
  - hardware support by SSE 4.2

Intrinsify CRC32c

- Add new intrinsic methods to directly support CRC32c instruction in HotSpot VM
- Hardware accelerated
- To be used in modified HDFS
- Completely avoids JNI overhead
  - HADOOP-7446 still carries JNI overhead

Other intrinsics

- May intrinsify other operation in the future
  - AES-NI
  - Others interested?

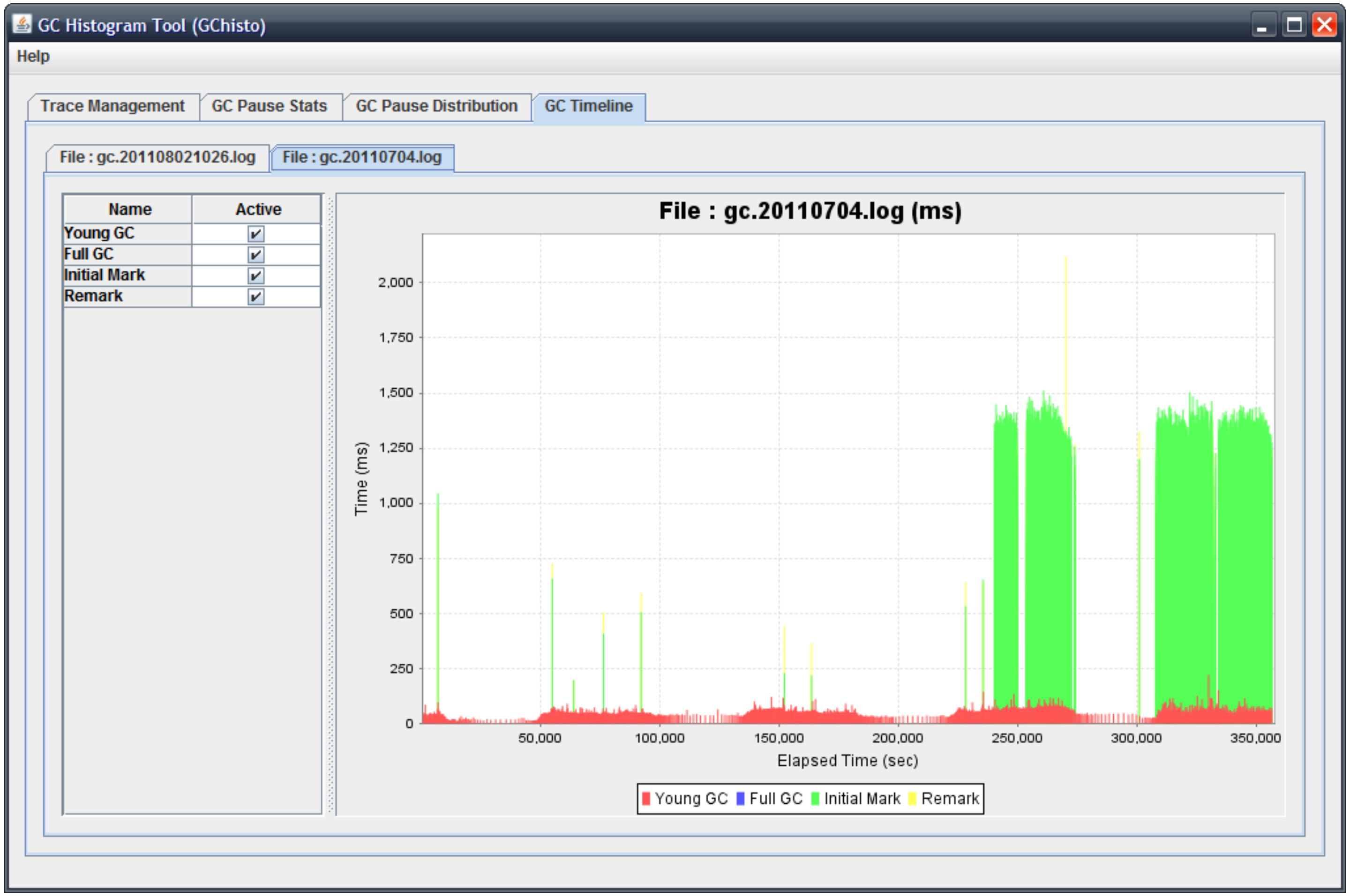
Case 4: frequent CMS GC

- An app experienced back-to-back CMS GC cycles after running for a few days
- The Java heaps were far from full
- What's going on?

The GC Log

```
2011-06-30T19:40:03.487+0800: 26.958: [GC 26.958: [ParNew: 1747712K->40832K(1922432K), 0.0887510 secs] 1747712K->40832K(4019584K), 0.0888740 secs] [Times: user=0.19 sys=0.00, real=0.09 secs]
2011-06-30T19:41:20.301+0800: 103.771: [GC 103.771: [ParNew: 1788544K->109881K(1922432K), 0.0910540 secs] 1788544K->109881K(4019584K), 0.0911960 secs] [Times: user=0.24 sys=0.07, real=0.09 secs]
2011-06-30T19:42:04.940+0800: 148.410: [GC [1 CMS-initial-mark: 0K(2097152K)] 998393K(4019584K), 0.4745760 secs] [Times: user=0.47 sys=0.00, real=0.46 secs]
2011-06-30T19:42:05.416+0800: 148.886: [CMS-concurrent-mark-start]
```

GC log visualized



The tool used here is GCHisto from Tony Printezis

Need more info

- `-XX:+PrintGCReason` to the rescue
  - added this new flag to the VM
  - print the direct cause of a GC cycle

The GC Log

```
2011-06-30T19:40:03.487+0800: 26.958: [GC 26.958: [ParNew: 1747712K->40832K(1922432K), 0.0887510 secs] 1747712K->40832K(4019584K), 0.0888740 secs] [Times: user=0.19 sys=0.00, real=0.09 secs]
2011-06-30T19:41:20.301+0800: 103.771: [GC 103.771: [ParNew: 1788544K->109881K(1922432K), 0.0910540 secs] 1788544K->109881K(4019584K), 0.0911960 secs] [Times: user=0.24 sys=0.07, real=0.09 secs]
 CMS Perm: collect because of occupancy 0.920845 / 0.920000
CMS perm gen initiated
2011-06-30T19:42:04.940+0800: 148.410: [GC [1 CMS-initial-mark: 0K(2097152K)] 998393K(4019584K), 0.4745760 secs] [Times: user=0.47 sys=0.00, real=0.46 secs]
2011-06-30T19:42:05.416+0800: 148.886: [CMS-concurrent-mark-start]
```

- Relevant VM arguments
  - `-XX:PermSize=96m -XX:MaxPermSize=256m`

- The problem was caused by bad interaction between CMS GC triggering and PermGen expansion
  - Thanks, Ramki!

- The (partial) fix

```
// Support for concurrent collection policy decisions.
bool CompactibleFreeListSpace::should_concurrent_collect() const {
  // In the future we might want to add in frgamentation stats --
  // including erosion of the "mountain" into this decision as well.
  return !adaptive_freelists() && linearAllocationWouldFail();
  return false;
}
```

After the change

Case 5: huge objects

- An app bug allocated a huge object, causing unexpected OOM
- Where did it come from?

huge objects and arrays

- Most Java objects are small
- Huge objects usually happen to be arrays
- A lot of collection objects use arrays as backing storage
  - `ArrayLists`, `HashMaps`, etc.
- Tracking huge array allocation can help locate huge allocation problems

```
product(intx, ArrayAllocationWarningSize, 512*M, \
        "array allocation with size larger than" \
        "this (bytes) will be given a warning"   \
        "into the GC log")                       \
```

Demo

```java
import java.util.ArrayList;

public class Demo {
  private static void foo() {
    new ArrayList<Object>(128 * 1024 * 1024);
  }

  public static void main(String[] args) {
    foo();
  }
}
```

Demo

```
$ java Demo
==WARNNING==  allocating large array:
thread_id[0x0000000059374800], thread_name[main],
array_size[536870928 bytes], array_length[134217728 elememts]
        at java.util.ArrayList.<init>(ArrayList.java:112)
        at Demo.foo(Demo.java:5)
        at Demo.main(Demo.java:9)
```

Case 6: bad optimizations?

- Some loop optimization bugs were found before launch of Oracle JDK 7
- Actually, they exist in recent JDK 6, too
  - some of the fixes weren't in until JDK6u29
  - can't wait until an official update with the fixes
  - roll our own workaround

Workarounds

- Explicitly set `-XX:-UseLoopPredicate` when using recent JDK 6
- Or …

Workarounds (cont.)

- Change the defaults of the opt flags to turn them off

```
product(bool, UseLoopPredicate, true false,                                    \
  "Generate a predicate to select fast/slow loop versions") \
```

A Case Study

JVM TUNING @ TAOBAO

JVM Tuning

- Most JVM tuning efforts are spent on memory related issues
  - we do too
  - lots of reading material available
- Let’s look at something else
  - use JVM internal knowledge to guide tuning

Case: Velocity template compilation

- An internal project seeks to compile Velocity templates into Java bytecodes

Compilation process

- Parse *.vm source into AST
  - reuse original parser and AST from Velocity
- Traverse the AST and generate Java source code as target
  - works like macro expansion
- Use Java Compiler API to generate bytecodes

Example

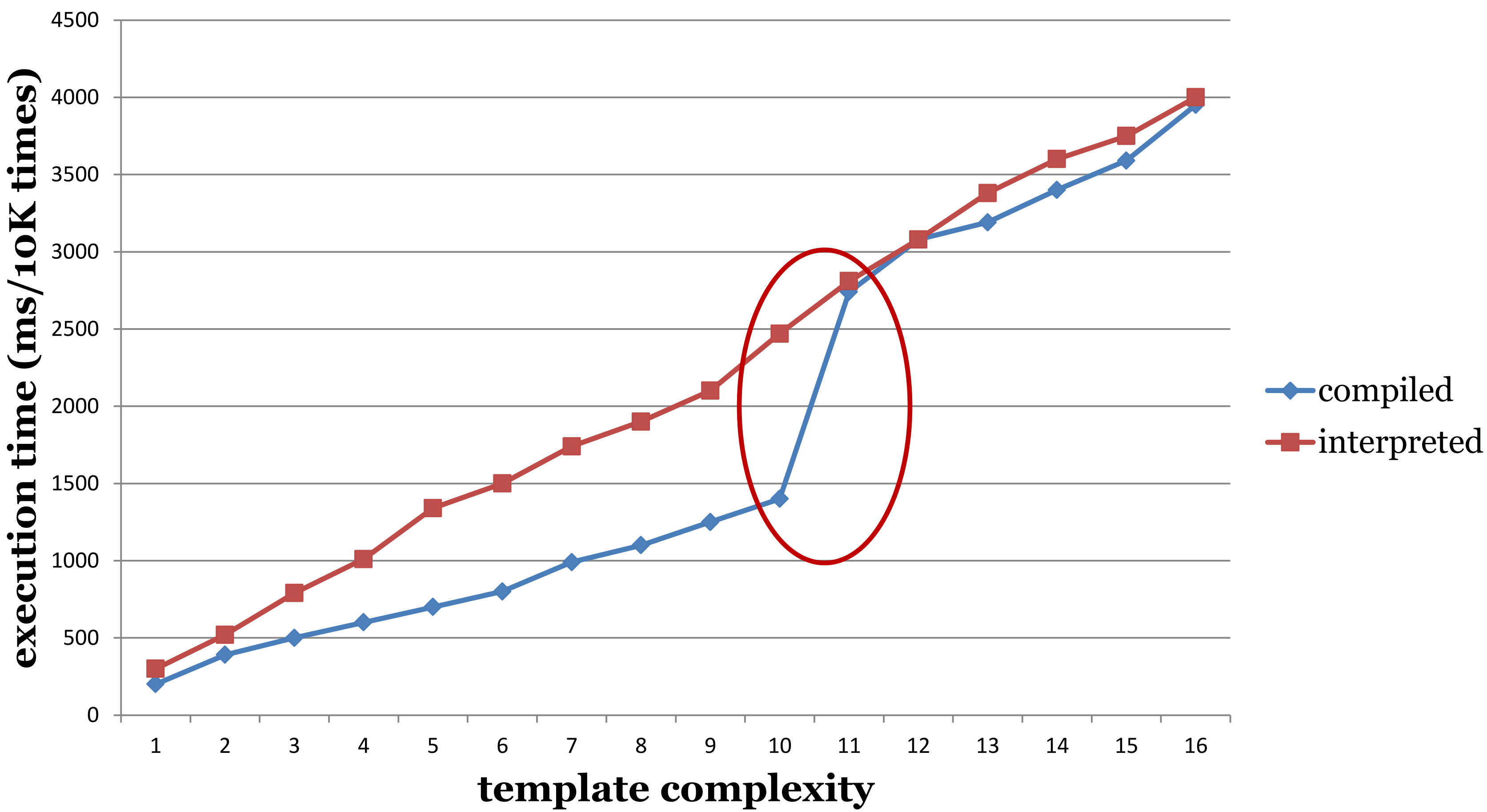
Velocity template source

```
Check $dev.Name out!
```

generated Java source

```
_writer.write("Check ");
_writer.write(
  _context.get(_context.get("dev"),
  "Name", Integer.valueOf(26795951)));
_writer.write(" out!");
```

Performance: interpreted vs. compiled



Problem

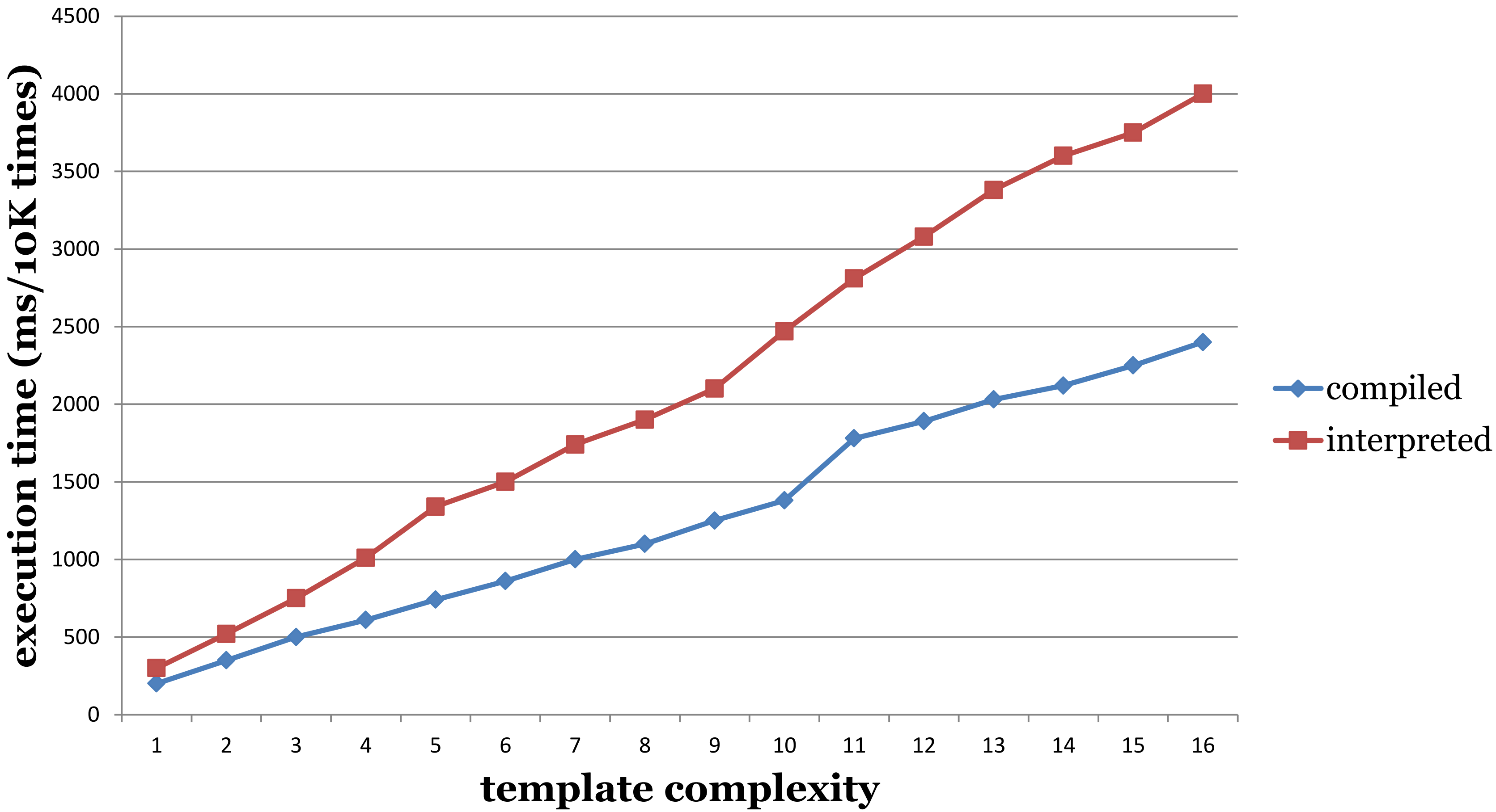
- In the compiled version
  - 1 “complexity” ≈ 800 bytes of bytecode
  - So 11 “complexities” > 8000 bytes of bytecode

Compiled templates larger than “11” are not JIT'd!

```
develop(intx, HugeMethodLimit, 8000,                \
  "don't compile methods larger than"               \
  "this if +DontCompileHugeMethods")                \
product(bool, DontCompileHugeMethods, true,         \
  "don't compile methods > HugeMethodLimit")        \
```

Case Study Summary

-XX:-DontCompileHugeMethods



JVM OPEN SOURCE @ TAOBAO

Open Source

- Participate in OpenJDK
  - Already submitted 4 patches into the HotSpot VM and its Serviceability Agent
  - Active on OpenJDK mailing-lists
- Sign the OCA
  - Work in progress, almost there
  - Submit more patches after OCA is accepted
- Future open sourcing of custom modifications

Open Source (cont.)

- The submitted patches
  - 7050685: jsdbproc64.sh has a typo in the package name
  - 7058036: FieldsAllocationStyle=2 does not work in 32-bit VM
  - 7060619: C1 should respect inline and dontinline directives from CompilerOracle
  - 7072527: CMS: JMM GC counters overcount in some cases
- Due to restrictions in contribution process, more significant patches cannot be submitted until our OCA is accepted

JVM TRAINING
@ TAOBAO

JVM Training

- Regular internal courses on
  - JVM internals
  - JVM tuning
  - JVM troubleshooting
- Discussion group for people interested in JVM internals

QUESTIONS?

Kris Mok, Software Engineer, Taobao
@rednaxelafx
莫枢 / “撒迦”


QCon
北京站 · 2012年4月18~20日
www.qconbeijing.com（11月启动）
QCon杭州站官网和资料
www.qconhangzhou.com
全球企业开发大会
INTERNATIONAL
SOFTWARE DEVELOPMENT
CONFERENCE