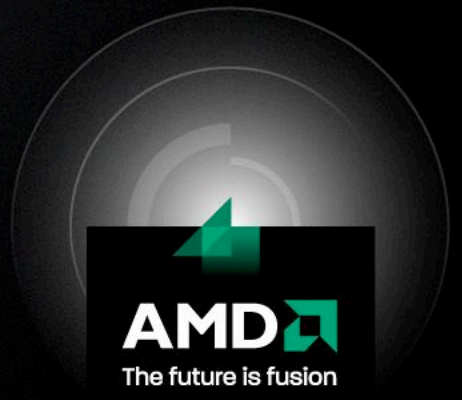
Improvements in OpenJDK useful for JVM Languages

JVM Language Summit 2010

Eric Caspole
AMD Java Labs
July 2010



Overview

- AMD Java Labs working on OpenJDK
  - Contributions for better performance and diagnostics
  - Get project ideas from customer cases or benchmarks

- AMD changes added to OpenJDK since November 2009
  - JVMTI change for better performance while JDWP debugging
  - JVMTI extension to export info for inlining in JITed methods
  - Unload older methods in code cache if it gets full
  - Make compiled method sweeper concurrent with application

JDWP Debugging Improvement

- JVM must be started with jdwp agent if you ever want to attach a debugger, typical command line:
  `agentlib:jdwp=transport=dt_socket,address=8000,server=y,suspend=n`

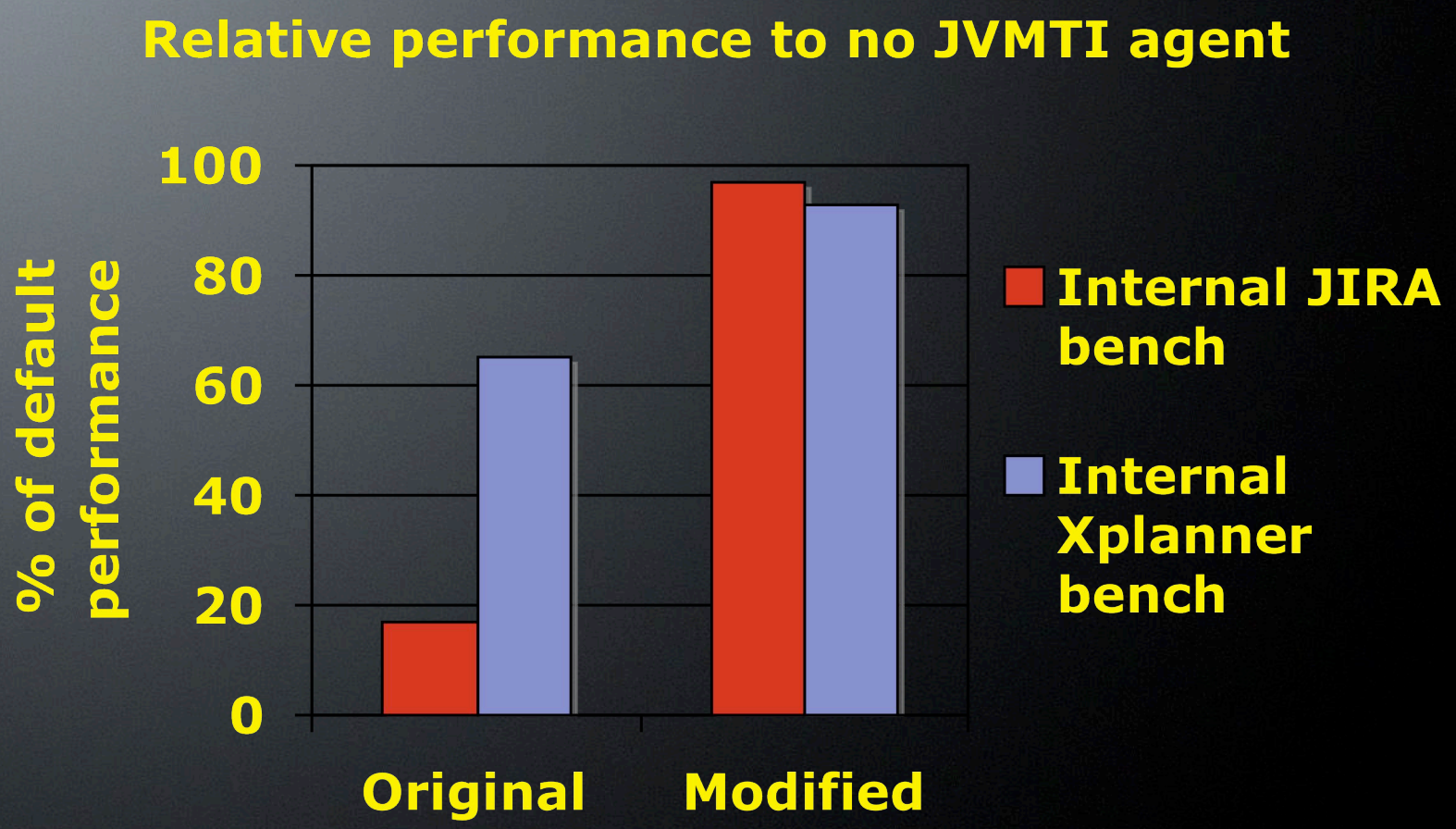
- Some customers run in production with JDWP agent on!
  - "Just in case" need to attach a JVMTI debugger later
  - Expectation: little or no performance penalty if debugger not attached
    - On many workloads this is true

- JDWP agent enables JVMTI event notifications when the debugger attaches, but must enable JVMTI capabilities at boot time.
  - Some capabilities were affecting OpenJDK codegen, even when no debugger attached

JVMTI can_generate_exceptions Capability

- JDWP enables this JVMTI capability at startup
- Exception throw handling always prepared to send an event
- Throws took slow path even if no debugger attached
  - Caused a deoptimization and revert to interpreter each time
  - Huge performance penalty on throwy applications
- Our change allows full-speed exception-path operation until debugger is attached
- Events generated only when attached, per-thread basis
- Speed up code/build/debug work cycles
- Automatically on in latest builds, no option required

JDWP Debugging - Performance Results



JVMTI Compiled Method Inlining Info

- With JVMTI, the JVM emits events to agents by callbacks
- CompiledMethodLoad when compiled method is installed
  - Useful for profiling and monitoring
  - Shows method id, address of code, and map to BCIs

- Why would we want to see inline details in JVMTI?
  - Fine-grain perf tuning
  - Understanding how Hotspot compiles your app

JVMTI Compiled Method Inlining Info

- Need to pass more info describing inline sites to agent
- An unused parameter compile_info was provided in the callback API

```
void JNICALL compiledMethodLoad (jvmtiEnv *jvmti_env, jmethodID method,
                jint code_size, const void* code_addr, jint map_length,
                const jvmtiAddrLocationMap* map,
                const void* compile_info)
```

JVMTI Compiled Method Inlining Info

- We used the compile_info param to emit inline info
  - Contains a ptr to a jvmtiCompiledMethodLoadInlineRecord
  - Built on a base struct for other uses of compile_info

```
typedef struct _PCStackInfo {
  void* pc;              /* the pc address for this compiled method */
  jint numstackframes;   /* number of methods on the stack */
  jmethodID* methods;    /* array of numstackframes method ids */
  jint* bcis;            /* array of numstackframes bytecode indices */
} PCStackInfo;

typedef struct _jvmtiCompiledMethodLoadInlineRecord {
  jvmtiCompiledMethodLoadRecordHeader header;  /* common header for casting */
  jint numpcs;           /* number of pc descriptors in this nmethod */
  PCStackInfo* pcinfo;   /* array of numpcs pc descriptors */
} jvmtiCompiledMethodLoadInlineRecord;
```

Quick Overview of Code Cache/Sweeping

- JITed methods are allocated in the code cache
  - Don’t get moved by GC, but contain fields that get GCed
  - Stick around until invalidated or owning class is unloaded
  - Code Cache has fixed upper limit, it can’t grow forever
  - 48MB server, 32MB client by default in JDK 6 for x86
- Java app threads run doing work then block at a safepoint
- Safepoints may happen for GC or runtime reasons
- Most safepoint work happens running on the VM thread

Quick Overview of Code Cache/Sweeping

- Invalidated compiled methods reclaimed by "sweeper"
- Sweeper runs during each safepoint on VM thread
- Discarded methods go through phases of aging
  - Non-entrant: activations may still exist, need to keep it
  - Zombie: we are sure no activations exist, can flush it
- Methods get marked non-entrant when compile-time assumptions become invalid
  - Callers will enter a stub and go back into the runtime
  - Method will get recompiled again later
- Compiling/sweeping occurring frequently as the application runs

Description of the "Code Cache Full" Problem

- The code cache has a fixed upper size
- Compiler is shut off if code cache gets full
  - New code runs interpreted-only
  - Existing compiled methods remain active
  - No way to turn compiler back on if space should clear up
- In a large application, tens of thousands of methods will get compiled as time goes by
- Many J2EE app servers offer hot (re)deployment of web apps
- New apps should each be in their own class loader
  - A class loader is a playpen so apps cannot see each other
  - Everything in one class loader gets unloaded together

Description of the "Code Cache Full" Problem

- Ideally, old instance of web app will get garbage collected
- App server or app coding error may prevent unloading
  - Everything in that class loader context remains alive
  - Compiled methods from old instances don’t get unloaded
- Code cache becomes full, reducing application performance
- No message is emitted when compiler is shut off
- Mysterious slowdowns are the best slowdowns
- Only solution was to restart the application

Code Cache Full - Description of the Fix

- Decided to target the older half of active compiled methods for aggressive unloading
  - Will unload methods only used during app startup
  - Will address the app redeployment issue
  - Assume most recently compiled is the hot code
- Not really necessary to unload all of those
  - Default max code cache size is 48MB
  - Probably some hot methods are in the older half
  - Want to sustain good performance
- What to do?

Speculative Disconnection

- Disconnect the compiled code from the JVM metadata representing the java method
- Callers notice compiled code ptr is null, enter runtime to find destination
  - Uses the usual path for resolving a method
  - Target could be interpreted or compiled
- Resolve code determines target method is disconnected
  - Reconnects the link from metadata->compiled code
  - Method goes back to the normal state
- Methods not restored in this way will soon be marked non-entrant and reclaimed by normal sweeping

Speculative Disconnection

- Works with both server and client compilers
- Hottest methods likely to avoid being flushed
- Applications spend more time running compiled code
- Performance largely unaffected when unloading happens
- Pause time comparable or better than scavenge GC
- Use new HotSpot option -XX:+UseCodeCacheFlushing

Nmethod Sweeper - Description of the Problem

- Applications are getting larger
  - More and more compiled code
  - Housekeeping of the code takes longer
- New CPUs have lots of cores
  - Safepoint time degrades throughput more and more
  - Want to get app threads back to work quickly
- Sweeper runs a little during each safepoint
  - Scans thread stacks to find methods in active frames
  - Sweeps the code cache to delete discarded nmethods

Nmethod Sweeper - Description of the Problem

- Sweep times can be 10+ ms even on latest CPUs
- Can happen during every safepoint depending on app
- Want to shorten safepoints as much as possible

Nmethod Sweeper - Description of the Change

- Stack scan continues to run in the safepoint
- Code cache sweeping runs concurrently
- Moves majority of work out of safepoint
  - Performed by compiler threads
  - Possibly sweep before taking a new compile task
  - Compiling can run on other threads during sweep
- Retrofit code cache unloading to be compatible

Summary

- These changes available in latest OpenJDK builds

- JVMTI change for better performance while JDWP debugging
  - Find the problem faster
- JVMTI extension to export info for inlining in JITed methods
  - Find the hotspot more easily
- Unload older methods in code cache if it gets full
  - Use -XX:+UseCodeCacheUnloading
- Make compiled method sweeper concurrent with app
  - Less safepoint time increases potential throughput

Resources and Links

JVMTI Inlining Article:
http://developer.amd.com/documentation/articles/pages/JVMTIEventPiggybacking.aspx

JDWP Debugging Article:
http://developer.amd.com/documentation/articles/pages/Java-Performance-Debugging-Enabled.aspx

Blog describing Code Cache Unloading:
http://blogs.amd.com/developer/2010/04/12/better-uptime-for-long-running-java-applications/

Disclaimer & Attribution