New Compiler Optimizations in the Java HotSpot™ Virtual Machine

Steve Dever
Steve Goldman
Kenneth Russell
Sun Microsystems, Inc.

TS-3412

Goal of This Talk

Learn how new dynamic compiler optimizations make Java-based programs run faster

Agenda

- Background
- Synchronization Optimizations
- Escape Analysis
- Tiered Compilation and Other Optimizations
- Future Plans
- Conclusion

Agenda

- **Background**
- Synchronization Optimizations
- Escape Analysis
- Tiered Compilation and Other Optimizations
- Future Plans
- Conclusion

Brief Introduction to the Java HotSpot VM

- Sun Microsystems' flagship Java™ Virtual Machine implementation (JVM) for the desktop
- Roots in Smalltalk and Self
- Focus on object-oriented optimizations
  - Deep inlining
  - Class Hierarchy Analysis
  - Virtual call inlining
- Aggressive optimization
- Dynamic deoptimization

Brief Introduction to the Java HotSpot VM

- Two flavors: client and server
- Same infrastructure
- Java HotSpot client compiler focuses on compile speed
- Java HotSpot server compiler focuses on peak performance
    - More later on eliminating this distinction

Brief Introduction to the Java HotSpot VM

- Rest of this talk focuses on new optimizations being done by the client and server compilers
- Should largely be unnoticeable to the Java programmer
- May still be useful to understand more of inner workings of underlying Java virtual machine implementation

Agenda

Background

Synchronization Optimizations

Escape Analysis

Tiered Compilation and Other Optimizations

Future Plans

Conclusion

Locking in the Java Programming Language

- In Java language, every object is potentially a monitor
  - `synchronized` keyword
- All modern Java VMs incorporate lightweight locking
  - Avoid associating an OS-level mutex/condition variable pair with each Java-based object
  - Use atomic operations to enter and exit monitor
  - Fall back to heavyweight OS locks if contended
- Differences in encodings and protocols
- Effective because most locking is uncontended

Locking in the Java Programming Language

- In Java SE 5.0, **`java.util.concurrent`** locks introduced
  - New locks and primitives for building new locks
  - These optimizations do not apply to this class of locks

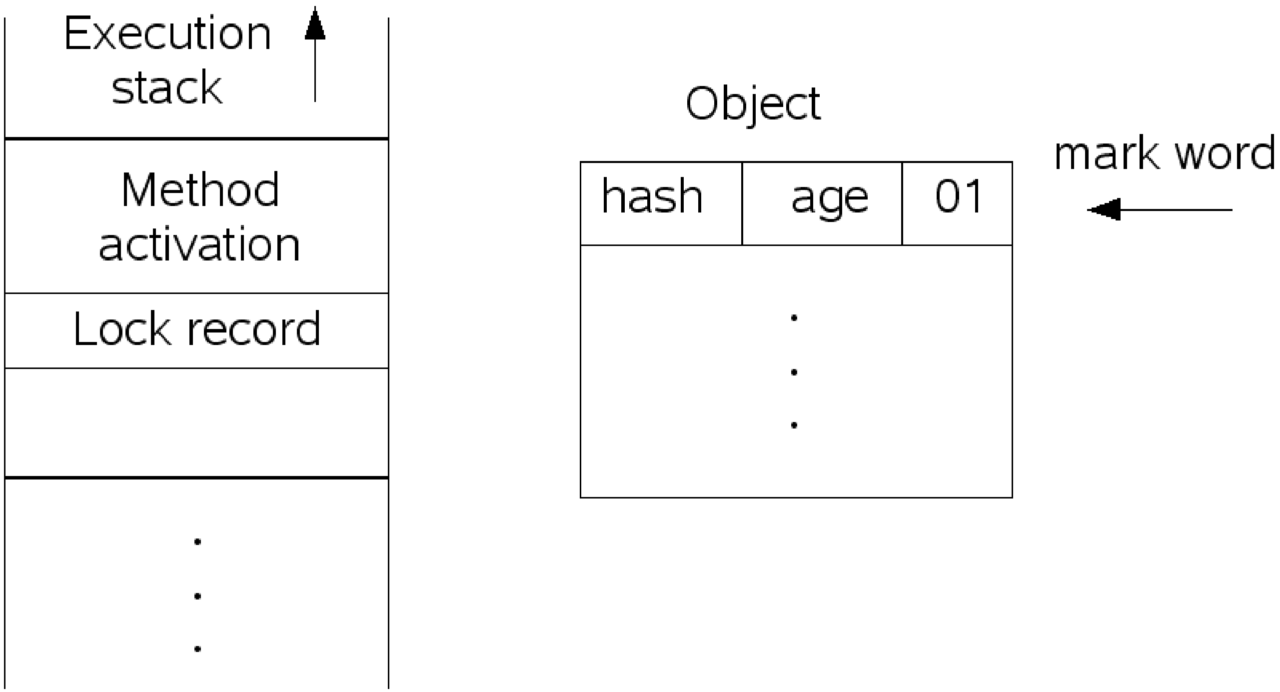
Overview of Lightweight Locking in Java HotSpot VM

- First word of every object is the mark word
- Used for synchronization and garbage collection
  - Also holds identity hash code if computed
- Low two bits of mark word indicate synchronization state
  - 01 > unlocked
  - 00 > lightweight locked
  - 10 > inflated (heavyweight locked)
  - 11 > marked for GC

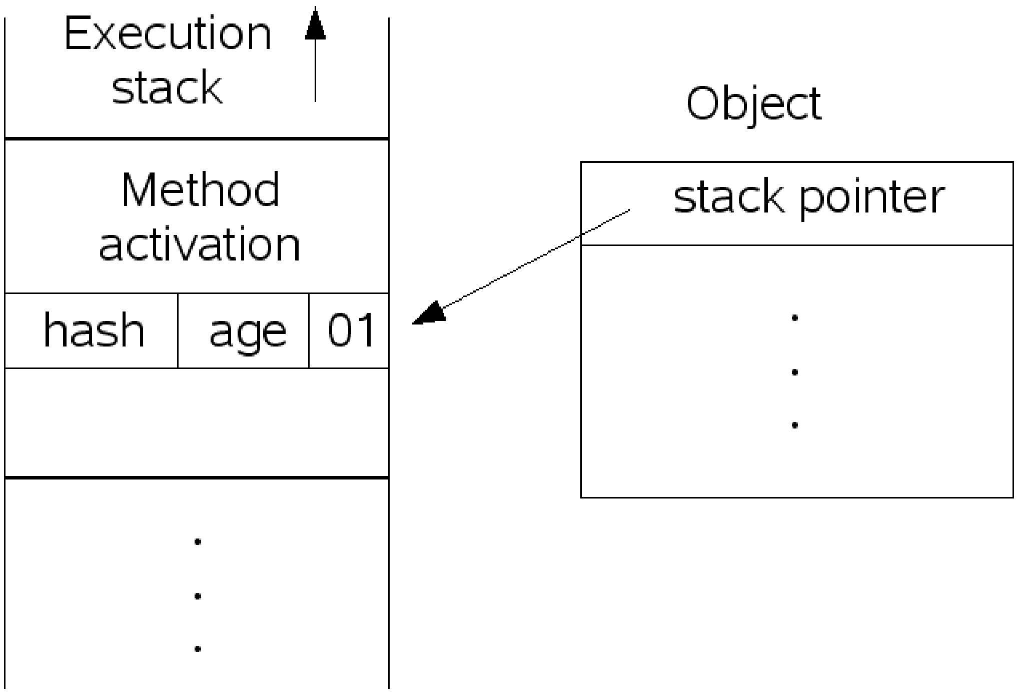
Overview of Lightweight Locking in Java HotSpot VM

- When object locked, mark word copied to stack into lock record
  - Displaced mark
- Atomic compare-and-swap (CAS) instruction used to make object point to on-stack lock record
- If CAS succeeds, thread owns lock
  - If fails, lock inflated–contention
- Lock records track which objects locked by the currently-executing method
  - Can walk stack of a thread to iterate locked objects

Locking Diagram



Locking Diagram



Overview of Lightweight Locking in Java HotSpot VM

- When object unlocked, CAS used to put displaced mark back in object
  - If fails, contention occurred
  - Notify waiting threads that monitor has exited

Observations

- Even atomic instructions can be relatively expensive on multiprocessors
- Most locking not only uncontended, but also performed by the same thread repeatedly
  - cf. Kawachiya et al, “Lock Reservation”, OOPSLA 2002
- Make it cheaper for a single thread to reacquire a lock
  - Trade-off of making it more expensive for another thread to acquire the same lock

Biased Locking

- First lock of an object biases it toward the thread which locked it
    - New encoding in mark word of object
- Subsequent locks and unlocks by same thread are very cheap
    - No atomic operations
    - Load-and-test to make sure still biased toward current thread
- Bias revoked if another thread locks same object
    - Expensive for individual objects

Bias Revocation

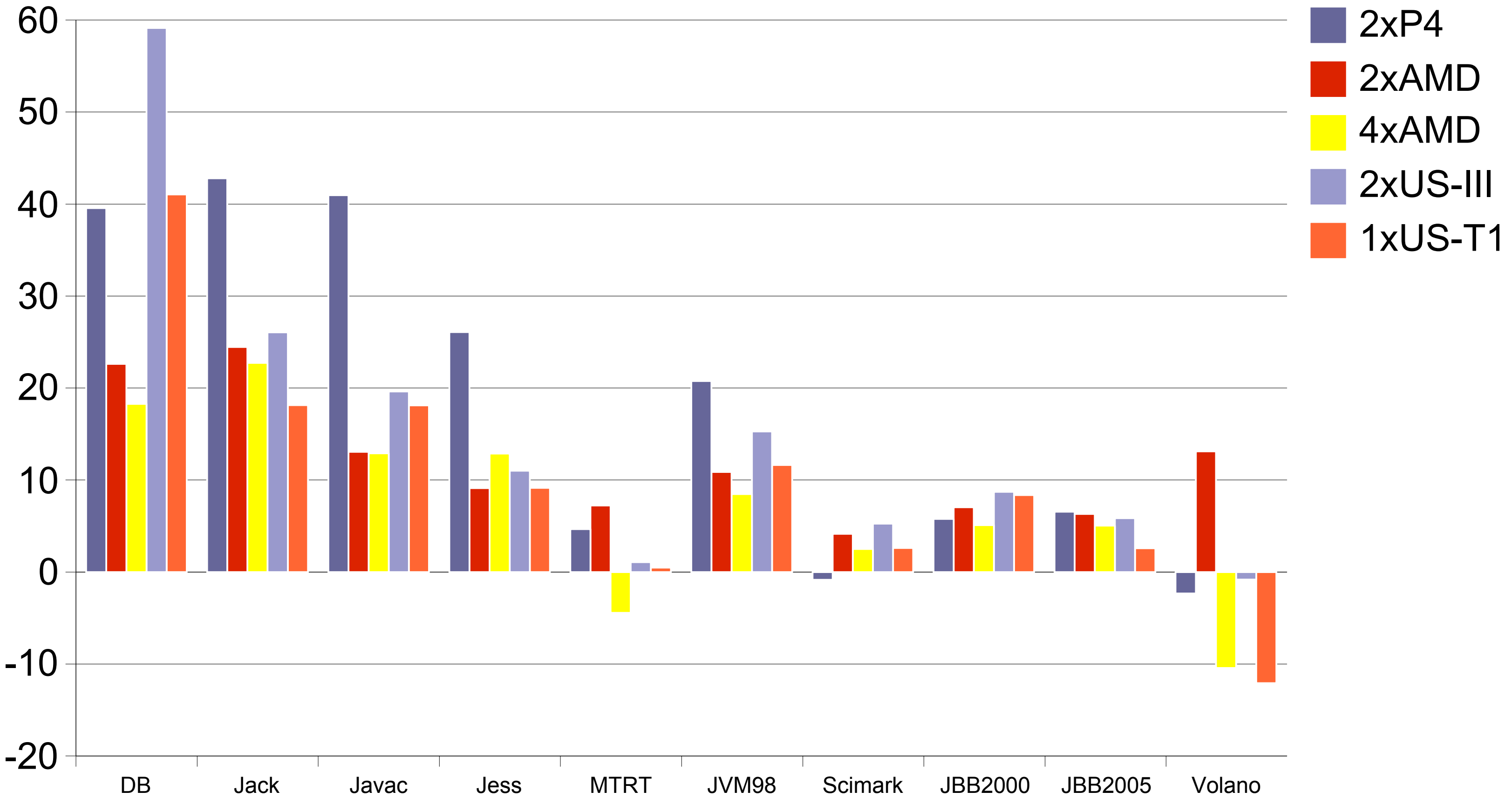
- Stop thread owning the object’s bias
- Walk stack enumerating lock records
- Fill in lock records for object, if any
- Update object’s mark word
  - Point at highest lock record if currently locked
  - Write in unlocked value if not currently locked
- Continue with normal CAS-based locking
- Obviously fairly expensive

Bulk Rebiasing and Revocation

- Detect if many revocations occurring for a given data type
- Try invalidating all biases for objects of that type
  - Allows them to rebias themselves to a new thread
  - Amortizes cost of individual revocations
  - Multiple such bulk rebias operations permitted
- If individual revocations persist, disable biased locking for that data type
- Minimize the downside of the optimization while retaining the benefits

Results

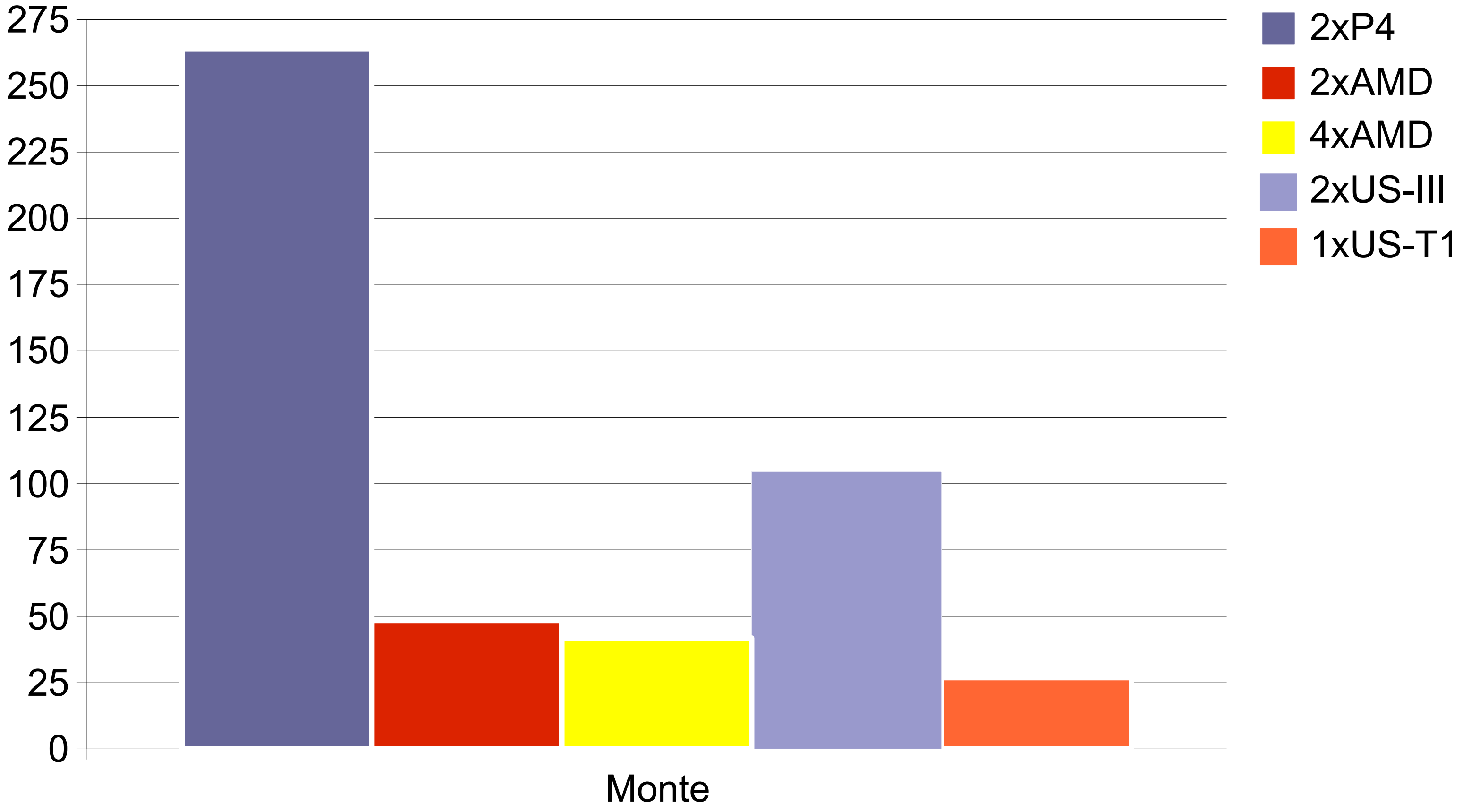
Percentage Increase/Decrease in Benchmark Scores



Source: Sun Microsystems, Inc.

Results

Percentage Increase/Decrease in Benchmark Scores



Source: Sun Microsystems, Inc.

Summary

- Biased locking improves uncontended synchronization performance
    - Still a fairly aggressive optimization
- Have attempted to minimize any performance penalties of biased locking
    - `-XX:-UseBiasedLocking` to disable completely
- Please provide feedback on Mustang forums
    - `http://mustang.dev.java.net/`

Summary

- Additional optimizations in Java Platform, Standard Edition 6 (Java SE 6) to improve contended synchronization performance
- Escape analysis and lock coarsening further improve synchronization speed
    - More later

Agenda

Background

Synchronization Optimizations

Escape Analysis

Tiered Compilation and Other Optimizations

Future Plans

Conclusion

Escape Analysis

- **Problem:** In general, when compiling and optimizing a method, we must assume that other threads and methods called can make arbitrary changes to any Java-based object

Escape Analysis

- Problem: In general, when compiling and optimizing a method, we must assume that other threads and methods called can make arbitrary changes to any Java-based object
- For objects allocated in a method these assumptions can be relaxed if we can prove that it does not ESCAPE the code being compiled

Non-Escaping Objects

- Allocated in the method being compiled
- Are not a subclass of Thread
- Do not have a finalizer
- Are not stored to a static field or a field of an escaped object
- Are not passed as an argument to a method call unless we know that the called method does not cause it to escape

Escape Example

```
class Escape1 {
  Integer val;
  Escape1 next;
  Escape1(Integer v) { val = v;}

  void example() {
    Integer i1 = new Integer(1);
    Integer i2 = new Integer(2);
    Integer i3 = new Integer(3);
    Escape1 e1 = new Escape1(i1);
    Escape1 e2 = new Escape1(i2);
    Escape1 e3 = new Escape1(i3);

    e1.next = e2;
    next = e2;  //  e2 and i2 escape via “this”
    e2.next = e3;  // e3 and i3 escape
  }
}
```

Optimization Possibilities

- Eliminate locking on the object
- Optimize field references
- In some cases can allocate object on the stack frame instead of the heap

Common Occurrences of Non-escaping Objects

- Autoboxing of method arguments (if called method is inlined.)
- Iterators over Collections
- StringBuilder objects created for String concatenation

Tracking Object Stores

- The analysis in the server compiler is based on:
  - J. Choi, M. Gupta, M. Serrano, V Sreedhar, S. Midkiff, Escape Analysis for Java, OOPSLA99, 1999

Tracking Object Stores

- For all ptr. values in a method, computes the set of objects that it could point to
- Initialize allocations to non-escaping and all other pointer values as escaping
- Mark anything a ptr. value could point to as escaping when it is:
  - Stored into a field of an escaped object
  - Passed as an argument to a method which causes the argument to escape

Tracking Object Stores

- The paper describes 2 algorithms:
  - Flow-insensitive—identifies objects which do not escape over the entire method
  - Flow-sensitive—identifies objects which do not escape over regions of a method
- The flow-sensitive algorithm requires more memory and may interact with other compiler optimizations

Tracking Object Stores

- The server compiler currently implements the flow-insensitive algorithm
- We have a prototype of the flow-sensitive version and are evaluating whether the extra complexity give sufficiently better code

Tracking Method Arguments

- If a called method is not inlined, we must track whether it causes any of its arguments to escape
- Without this tracking, we must make the pessimistic assumption that all arguments escape. This eliminated most of the optimization opportunities from escape analysis

Tracking Method Arguments

- Since a called method may not have been compiled yet, we can not rely on the compiler

Tracking Method Arguments

- We have a bytecode escape estimator which was implemented by two researchers from the Johannes Kepler University Linz as part of their work described in:
    - T. Kotzmann, H. Mössenböck, Escape analysis in the context of dynamic compilation and deoptimization, Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments, 2005 http://portal.acm.org/citation.cfm?doid=1064979.1064996

Tracking Method Arguments

- The escape estimator scans the bytecodes of a method and produces a conservative estimate of which arguments escape
- It also tracks whether the return value of the scanned method is an unescaped object
- Records the results of the scan for later use

Field Optimization Without Escape Analysis

```
class Escape2 {
  int fld1, fld2;

  Escape2(int v1, int v2) { fld1 = v1; fld2 = v2; }

  static void bigMethod() {
    ...  // a large method too big to inline
  }

  static int example(int v1, int v2) {
    Escape2  e1 = new Escape2(v1, 10);
    Escape2  e2 = new Escape2(v2, 5 - v1);

    bigMethod();// must assume fields of e1 & e2 can change

    return e1.fld1 + e2.fld1;  // need to reload values
  }
}
```

Field Optimization with Escape Analysis

```
class Escape2 {
  int fld1, fld2;

  Escape2(int v1, int v2) { fld1 = v1; fld2 = v2; }

  static void bigMethod() {
    ...  // a large method too big to inline
  }

  static int example(int v1, int v2) {
    Escape2  e1 = new Escape2(v1, 10);
    Escape2  e2 = new Escape2(v2, 5 - v1);

    bigMethod();  // cannot change fields of e1 & e2

    return e1.fld1 + e2.fld2; // returns v1 + (5 - v1) =  5
  }
}
```

Performance Results

- Lock elision provided no significant performance benefit over and above biased locking and lock coarsening (described later)
- Performance benefit of other optimizations made possible by escape analysis is continuing

Implementation Status

- Java SE 6 has escape analysis and lock elision in the server compiler
- It is off by default, it can be enabled with the -XX:+UseEscapeAnalysis flag
- Java SE 7 will have further optimizations
- There are currently no plans to release a client compiler with escape analysis

Agenda

Background

Synchronization Optimizations

Escape Analysis

Tiered Compilation and Other Optimizations

Future Plans

Conclusion

Lock Coarsening

- Dynamically we often see a lock being released and immediately acquired
- Idea is to eliminate the closely separated release and acquire
- Doing this in non-loop code does not impact fairness
- Not obvious at source level as the locks are either synchronized methods or locks within the called method
- Inlining exposes the closely paired locks

Lock Coarsening

- Assume p is simple predicate (no exception possible) and S is a synchronized method
- Release from first call can be removed if acquire is removed from then and else path
- Release in then/else can be removed if acquire is removed from final call

```
S();
if (p)
   S();
else
   S();
S();
```

Lock Coarsening

- Release from first call can be removed if acquire/release is removed from then path
- Acquire is removed from final call

```
S();
if (p)
   S();
S();
```

Lock Coarsening

- Acquire/release could be removed from then path if we moved the release from initial call to after the then join point
- This case is not currently handled

```
S();
if (p)
   S();
```

Lock Coarsening—Results

- Removes 20% of all dynamic locks in single warehouse run of specjbb2000
- Improves score on specjbb2000 by 2%
- Scimark Monte Carlo subtest score improved by 60%!

Source: Sun Microsystems, Inc.

Array Copy Stubs

- System.arraycopy is heavily used in the JDK™ libraries as well as application code
- Compilers inlined System.arraycopy but they tended to be pessimistic about aliasing and alignment
- As a result performance was okay but not great

Array Copy Stubs

- In Mustang (and backported to 5.0u5) hand coded assembly stubs written for each type size assuming no overlap
- Compiler generates one simple test to decide
  - Overlap? Same code as previously
  - No Overlap? Call stub

Array Copy Stubs—Results

- System.arraycopy microbenchmarks
  - Slight degradation for small (1–4 elements)
  - > 100% improvement for modest number of elements (20+)
- 4% increase of specjbb2000 score on SPARC® hardware
- 1+% increase of specjbb2000 score on AMD64

Source: Sun Microsystems, Inc.

Tiered Compilation

- Client compiler is good at startup and short apps
  - Inferior performance for longer running apps
- Server compiler is good at long apps
  - Inferior startup performance
- Single JVM with both compilers
- Like an automatic transmission—
  - Startup with client compiler
  - Cruise with server compiler

Tiered Compilation—Issues

- Different calling conventions
  - A method compiled by client compiler can't call method created by server compiler or vice versa
- Different runtime interfaces
  - OopMaps were incompatible

Tiered Compilation

- Different calling conventions
  - Each compiler had separate code to describe calling conventions
- In Mustang shared code maps a signature into a description of the registers and/or stack slots used to pass parameters
- As a result methods generated by different compilers can call each other

Tiered Compilation

- Adapters convert from interpreter calling convention to compiled convention (i2c) and vice versa (c2i)
- Server compiler compiled adapters as separate code objects
- Server compiler used a separate thread for adapter compilation
- Client compiler built the code into the compiled Java method

Tiered Compilation

- In Mustang adapter code is generated by shared code
- A single adapter code object contains the i2c and c2i for each signature seen
- Reduction in generated code compared to client style of adapters
- Reduction in server compiler code and one less JVM thread

Tiered Compilation

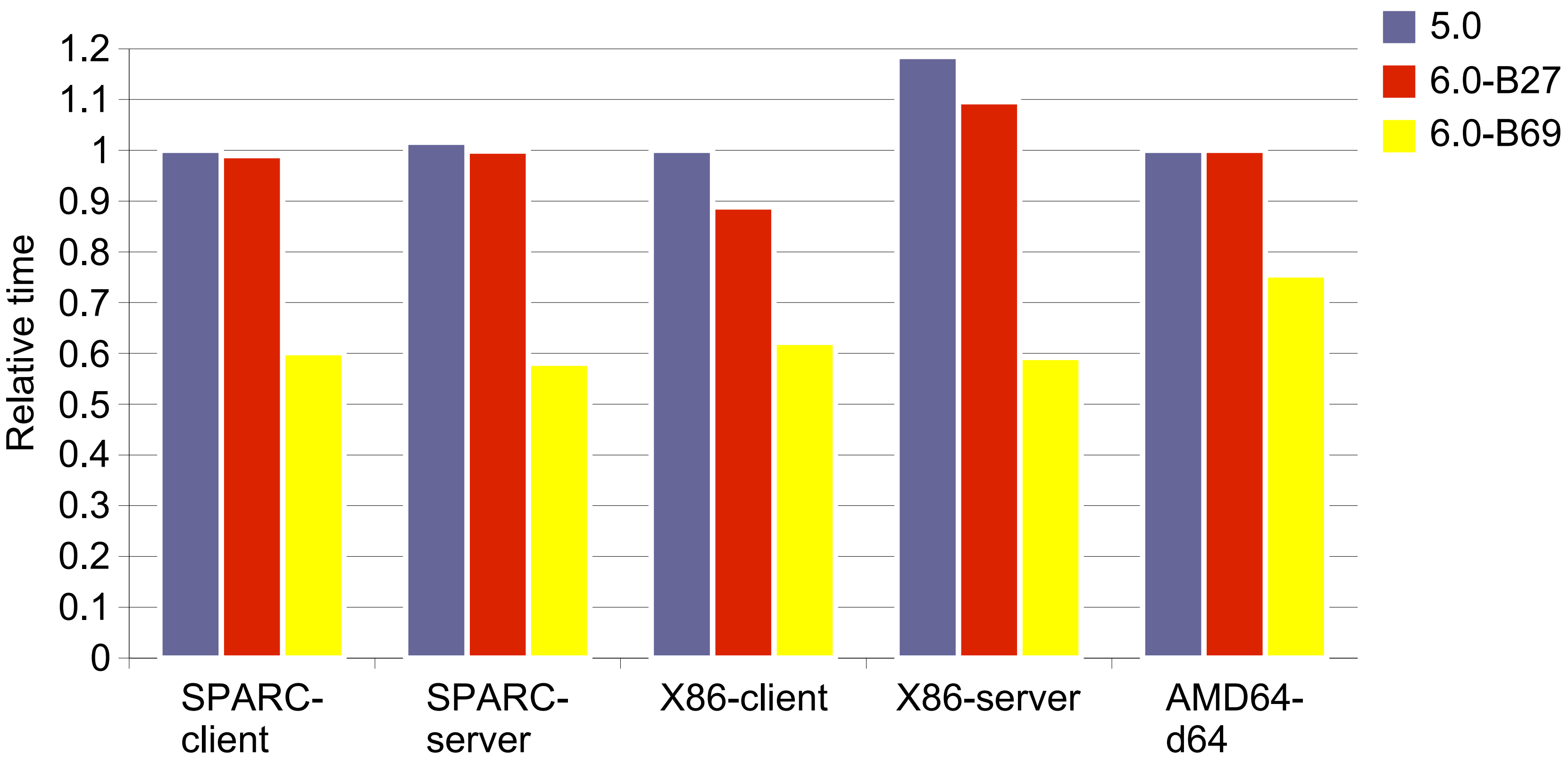
- Each compiler had distinct code for generating wrapper code for Java native methods
    - Transition from Java code to native and return requires precisely ordered thread state changes
- Client compiler code was straight forward and easy to modify
- Server code was difficult to understand and hard to get correct

Tiered Compilation

- In Mustang Java native method wrappers are produced by shared code
- Simple to modify
- Easy to experiment with new state transitions
- Better generated code

Tiered Compilation

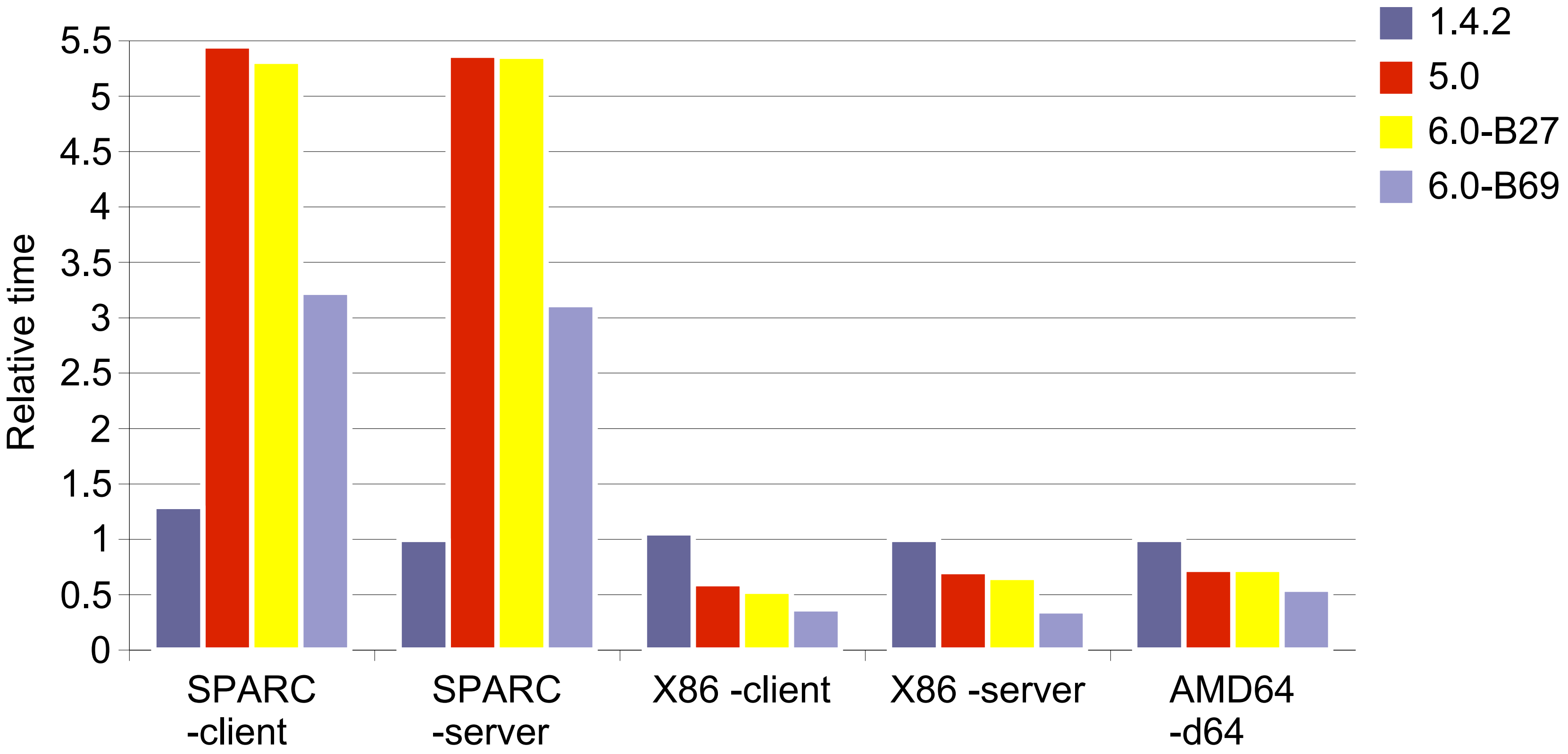
JNI Micro Benchmark



Source: Sun Microsystems, Inc.

Tiered Compilation

JNI Micro Benchmark



Source: Sun Microsystems, Inc.

Tiered Compilation—Remaining Work

- Merging runtime stubs
  - IC miss handler
  - Deoptimization
  - Exception handling
- Policy decisions
  - When to deopt/recompile
  - When to collect profile data
- Client compiler for 64bit platforms

Conclusion

- More performance improvements coming
  - Finish tiered compilation
  - More use of escape analysis results
  - Faster call out to JNI
- Try it out
  - http://mustang.dev.java.net/

For More Information

BOF-0197 Java HotSpot VM Q&A

- Thursday 7:30 PM North Meeting Room 121/124/125

Q&A

New Compiler Optimiztions in the Java HotSpot™ Virtual Machine

Steve Dever
Steve Goldman
Kenneth Russell
Sun Microsystems, Inc.

TS-3412