Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing

Kenneth Russell
David Detlefs
Sun Microsystems, Inc.

Agenda

- Background and Motivation
- Previous Research
  - IBM's Lock Reservation and extensions
- Contributions of Current Work
- Fast Locking in the Java HotSpot™ VM
  - IllegalMonitorStateException Detection
- Biased Locking
- Epoch-Based Bulk Rebiasing and Revocation
- Results / Conclusion

Background and Motivation

- Java™ programming language supports multithreading at a basic level
  - **`synchronized`** keyword
  - Support for a monitor per object
    - Lock/unlock
    - Wait/notify
- Efficient implementation of these synchronization primitives essential for high performance

Background and Motivation

- Early research utilized property that most synchronization in Java language is uncontended
- Lightweight locking
- Avoid creation of mutex/condvar per Java object
  - Bacon et al, Thin Locks, PLDI 1998
  - Agesen et al, Meta-Lock, OOPSLA 1999
  - Bak et al, U.S. Patent 6,167,424, Issued December 2000
  - Dice, Relaxed Locks, JVM 2001

Background and Motivation

- Lightweight locking uses CPU-level atomic operations
  - Compare-and-swap / compare and exchange
  - 1 or 2 atomic operations per lock/unlock pair depending on algorithm
  - “Inflate” to full heavyweight monitor if contention detected
- Most computers nowadays are multiprocessor
- Atomic operations significantly more expensive
- Essential to further optimize locking

Previous Research

- IBM Research Labs, Tokyo discovered that most locking in Java programs is not only uncontended, but unshared
  - Kawachiya et al, Lock Reservation, OOPSLA 2002
  - Most objects locked / unlocked by exactly one thread in the object's lifetime
- Optimize for this case

Previous Research

- First thread locking the object *reserves* the lock with an atomic operation
  - Subsequent locks / unlocks by that thread use no atomic operations
  - Recursion count in object header detects IllegalMonitorStateException
    - Using non-atomic stores
- If another thread locks the object, relatively expensive *unreservation* required
  - Involves sending signal to reservation owner thread
  - Ensures reservation owner thread not performing concurrent non-atomic stores

Previous Research

- Follow-on research aimed at reducing cost of unreservation
  - Onodera et al, ECOOP 2004
  - Kawachiya, Ph.D thesis, Keio University
- Reduce or eliminate penalty associated with locking by other threads
- Does not optimize case where objects are transferred from one thread to another
  - Atomic operations used for locking/unlocking by other threads than the reservation owner

Contributions of Current Work

- New algorithm for elimination of synchronization-related atomic operations
  - Store-Free Biased Locking (hereafter “biased locking”)
  - Builds on invariants in Java HotSpot VM from Sun Microsystems, Inc.
- **Bulk rebiasing** and **bulk revocation** throttle back the optimization in unprofitable situations
- **Epoch-based** bulk rebiasing supports efficient transfer of sets of objects between threads
  - Optimizes more synchronization patterns than previous work

Fast Locking in the Java HotSpot VM

- Implementation of lightweight locking in Java HotSpot VM maintains certain useful invariants
  - Not aware of other JVMs which maintain these invariants
- Simplifies biased locking algorithm
- Enables optimization of locking of objects transferred between threads

Fast Locking in the Java HotSpot VM

- HotSpot JVM uses a two-word object header
- Mark word
  - Synchronization, GC and hash code information

bitfields				tag bits
hash		age	0	01
ptr to lock record				00
ptr to heavyweight monitor				10
				11
thread id	epoch	age	1	01

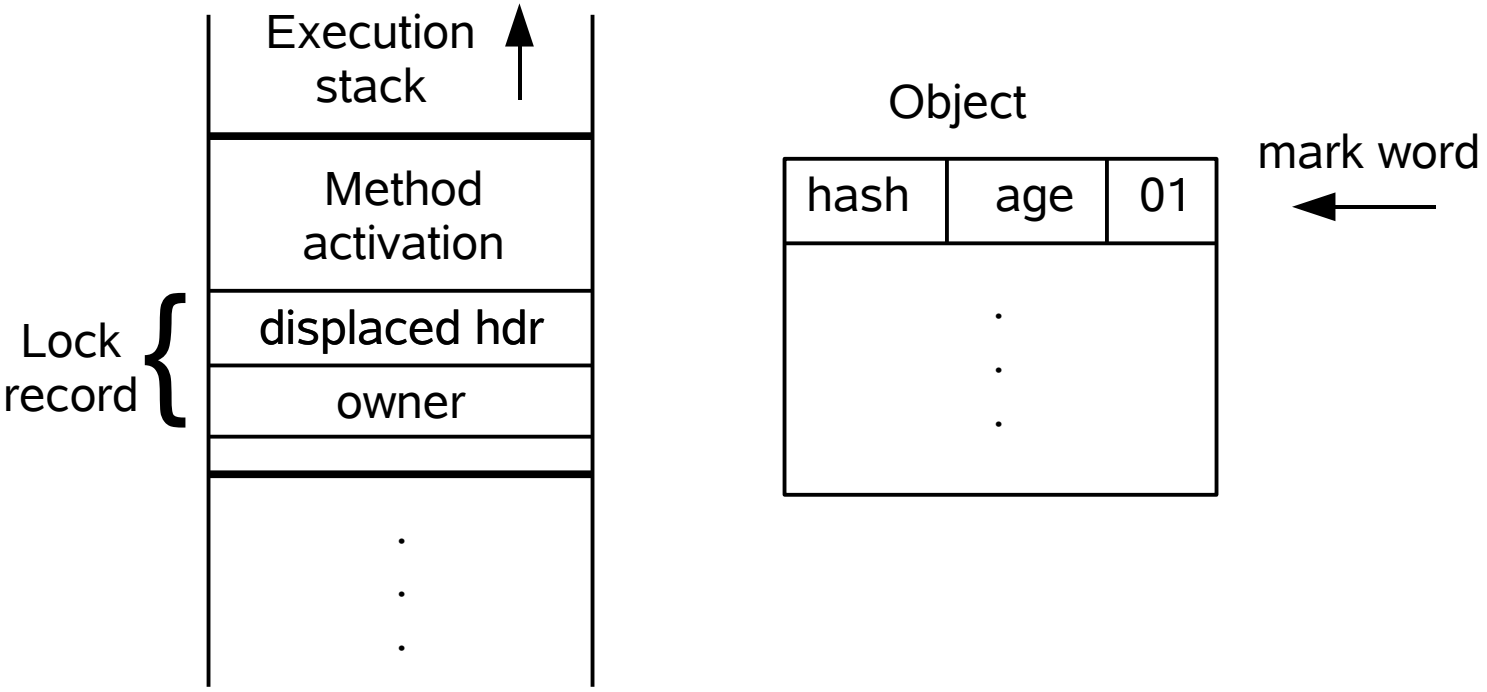
state
unlocked
lightweight locked
inflated
marked for gc
biasable

- Class pointer
  - Type of object

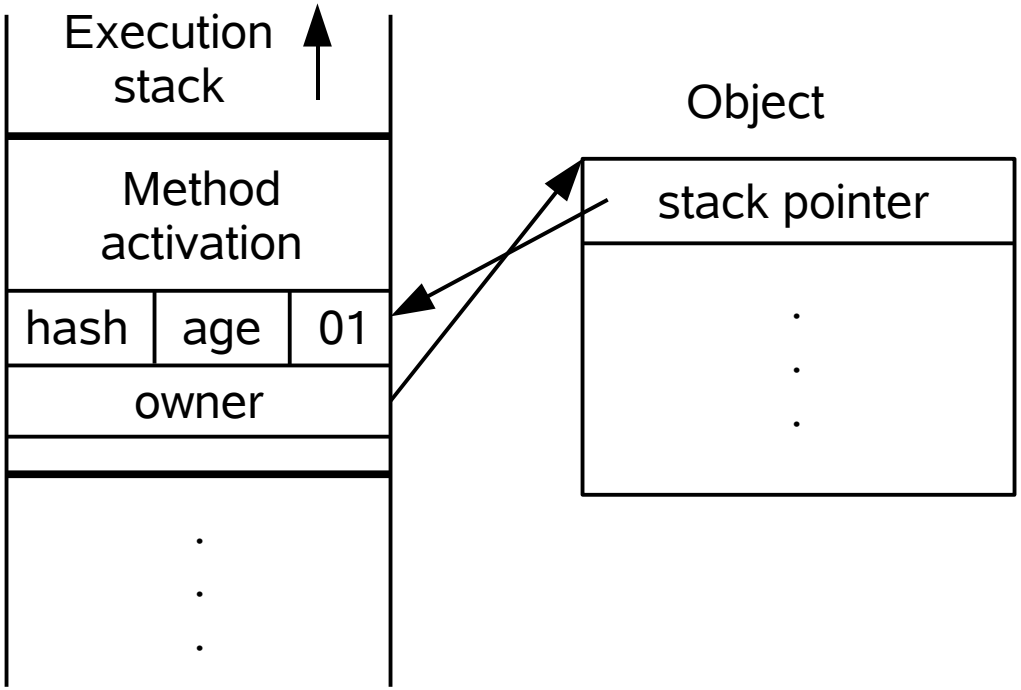
Fast Locking in the Java HotSpot VM

- Fast lock operation copies mark word to on-stack **lock record** in current activation
  - Lock records managed slightly differently for interpreted and compiled activations
  - Run-time system is capable of enumerating all lock records on a given thread's stack
  - Deoptimization, etc.
- Atomically CASs pointer to lock record into object's mark word
  - If successful, lock is owned by this thread
  - Ownership implicit: lock record is in owner's stack

Fast Locking in the Java HotSpot VM



Fast Locking in the Java HotSpot VM



Fast Locking in the Java HotSpot VM

- Fast unlock atomically CASs **displaced** mark word back into object header
  - If success, no contention occurred
  - If failed, monitor was **inflated** into heavyweight case using OS-level locking primitives
    - Enter run-time system, notify waiting threads
- Recursive locks detected during lock operation
  - Value “0” stored into lock record on stack
  - No recursion count stored in mark word

IllegalMonitorStateException Detection

- Java Virtual Machine Specification deliberately vague about when **`IMSE`** detected
  - When too many **`monitorexit`** bytecodes executed for a given object
- JVM may or may not throw **`IMSE`** if **`monitorenter`** bytecode executed and method exited without unlocking
- Intended to allow recursion count-based detection of illegal monitor states
  - Previous work based on this assumption

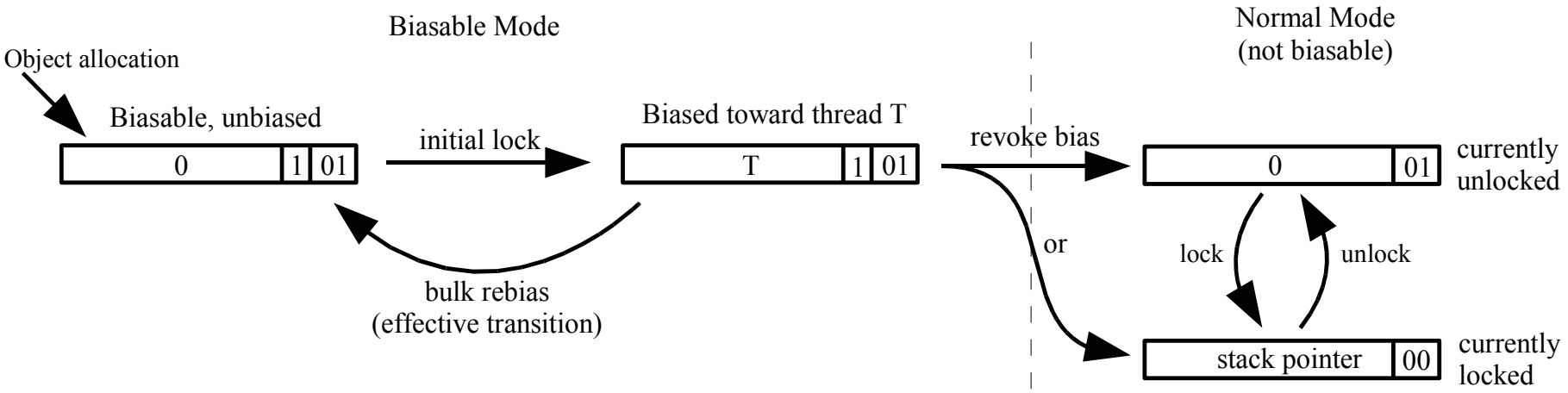
IllegalMonitorStateException Detection

- HotSpot JVM detects **`IMSE`** eagerly in interpreter
  - If objects left locked upon method exit, or **`monitorexit`** executed without paired **`monitorenter`** in current activation
  - JVM unlocks any objects locked in current activation, then throws **`IMSE`**
- Dynamic compilers do not compile code with mismatched monitors
- Eliminates recursion count, and thereby non-atomic stores, from biased locking algorithm
  - Simplifies transfer of biases between threads

Biased Locking

- Newly allocated objects are **biasable** but **unbiased**
- First lock uses CAS to insert thread ID in mark word
- Subsequent locks only compare thread ID to current thread
  - Load-and-test
  - Success case => object locked
  - Unlocks only check to see if object still biasable
- Failure case: revert to original fast locking algorithm
  - May involve potentially expensive **bias revocation**

Biased Locking



Bias Revocation

- Stop bias owner thread at safepoint
  - Similar to how GCs are initiated
- Walk thread's stack
  - Enumerate lock records (if any) for biased object
  - Fix up these and mark word to look like fast locking algorithm in use
- Resume target thread
- Continue with normal fast locking algorithm
  - Including inflation for contended case

Bulk Rebiasing and Revocation

- Individual bias revocations expensive
  - Actually, more expensive than signals used in previous lock reservation work
- Found empirically that biased locking tends to be profitable, or not, largely on a per-class basis
- Try to do bias revocations in bulk instead of individually
  - If biased locking appears to not be profitable for a given class

Bulk Rebiasing and Revocation

- Bulk rebiasing
  - Invalidate all currently held biases for a given class
  - Try to let them settle down into a stable pattern again
  - Handles transfer of sets of objects between threads
    - SPECjbb2000, SPECjbb2005
- Bulk revocation
  - If individual bias revocations persist, disable biased locking for the class
    - All current instances and future allocated instances

Epoch-Based Bulk Rebiasing and Revocation

- Too expensive to iterate the object heap to enumerate all instances of a class
  - Original implementation of these heuristics
- Use epochs to define validity of bias
  - Maintained on per-class basis
- Add in comparison of epoch to biased lock entry

Epoch-Based Bulk Rebiasing and Revocation

```
void lock(Object* obj, Thread* t) {
  int lw = obj->lock_word;
  if (lock_state(lw) == Biased
      && biasable(lw) ==
         obj->class->biasable
      && bias_epoch(lw) ==
         obj->class->bias_epoch
      && bias_owner(lw) == t) {
    return; /7 success
  } else {
    // revoke bias, try to acquire
    // initial bias, fast lock...
```

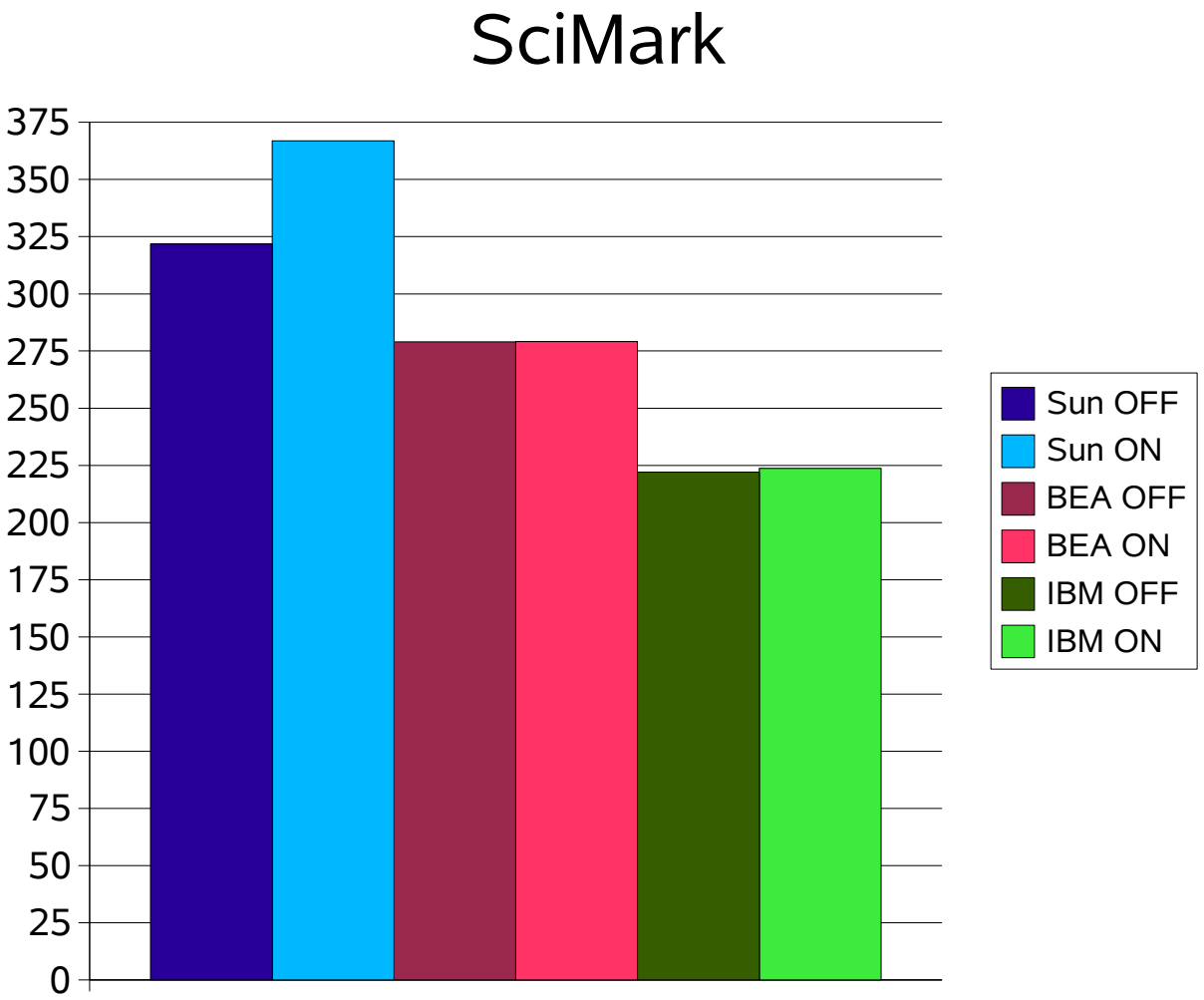
Performance Results

- Sun introduced biased locking in Java SE 6 product roughly mid-2005
- BEA introduced new -XXlazyUnlocking flag roughly six months later
- IBM submitted SPECjbb2005 scores with new -XlockReservation flag in mid-2006
- Sun is currently the only vendor enabling this optimization by default in its product

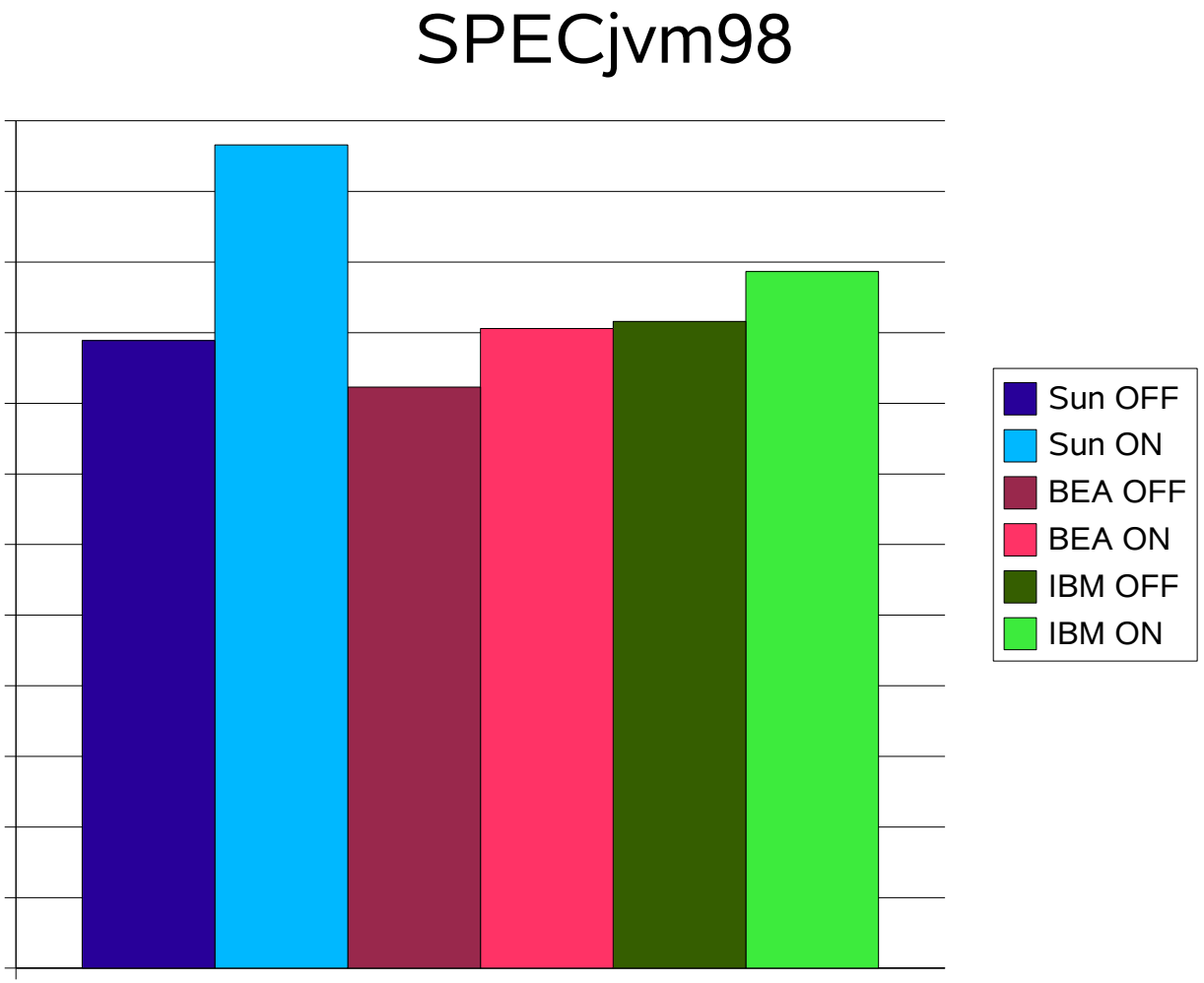
Performance Results

- Data taken from 2-CPU, 3 GHz Intel “Woodcrest”
- Windows Server 2003 R2, Enterprise x64 Edition
- Latest BEA, IBM and Sun JVMs
  - 1500 MB heap for SPECjbb2005
  - IBM, Sun use 32-bit JVMs; BEA, 64-bit with -XXcompressedRefs
  - IBM run with -Xjvm:perf -Xgcpolicy:gencon
  - Default (no) command-line arguments otherwise
- “On”, “off” refer to biased locking, lazy unlocking or lock reservation, respectively

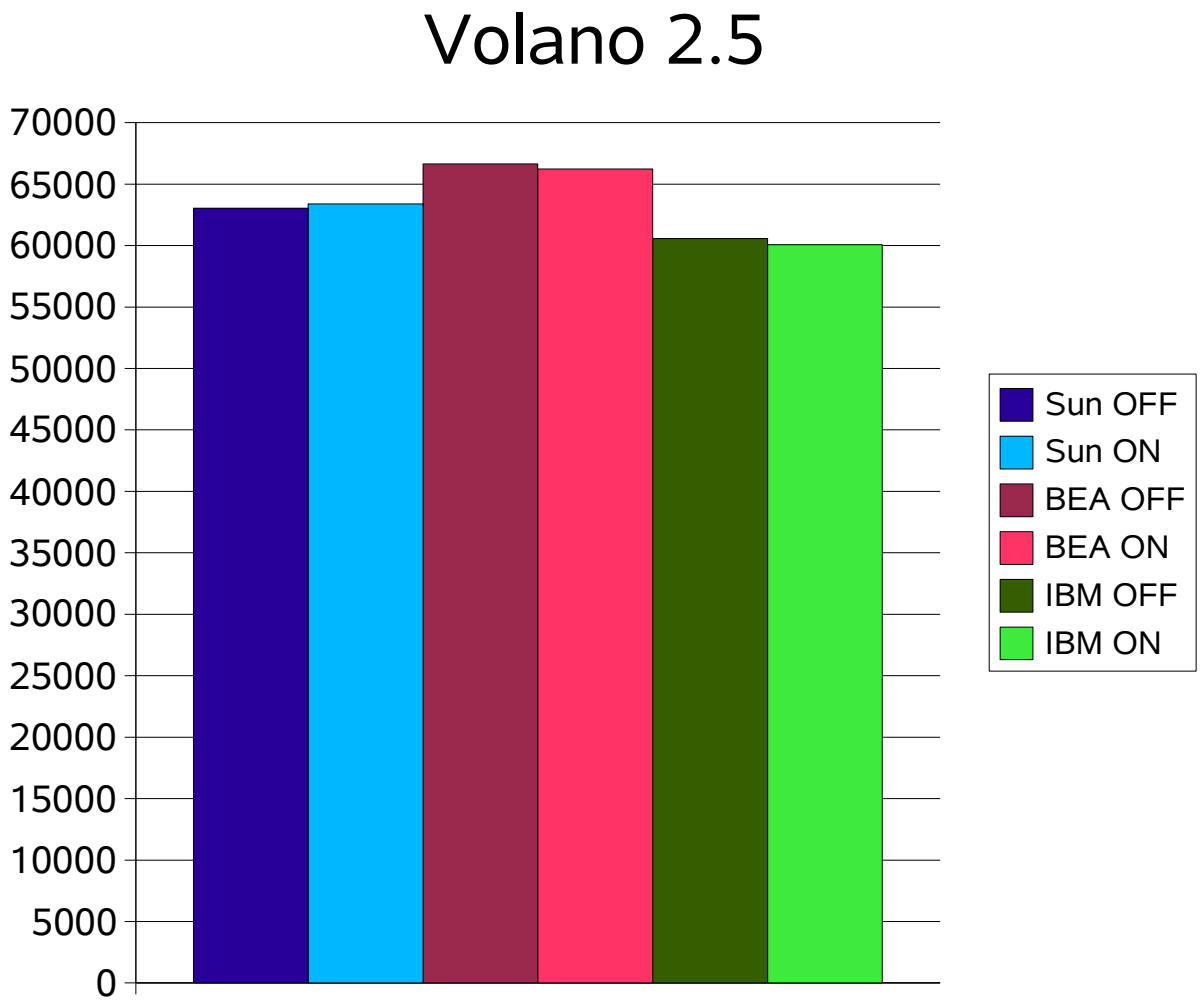
Performance Results



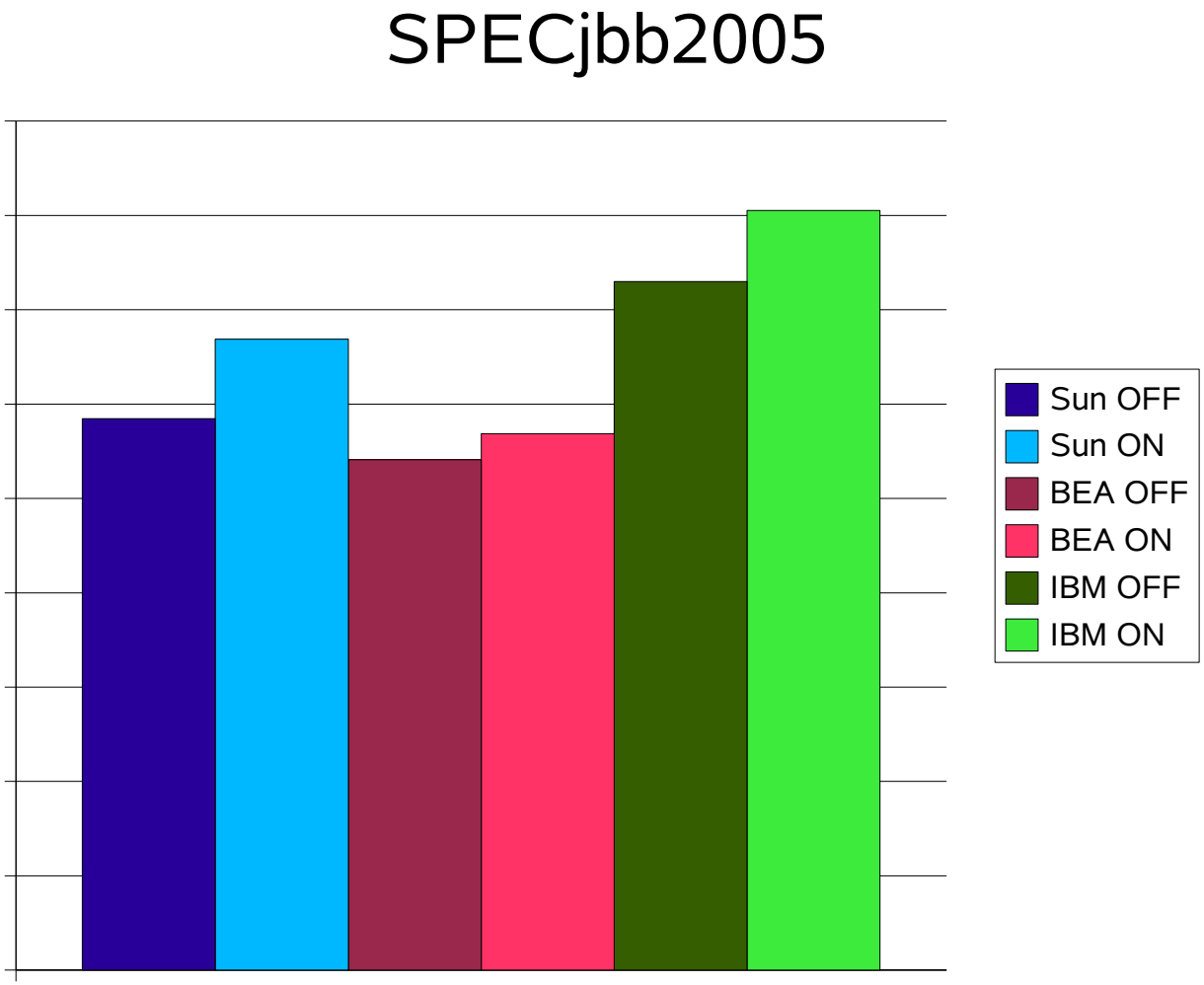
Performance Results



Performance Results



Performance Results



Conclusion

- Biased Locking and Epoch-Based Bulk Rebiasing and Revocation optimize many synchronization patterns in benchmarks and real-world applications
  - Compare favorably to other vendors' implementations
  - Reports from the field of 10% speedups on real apps
- Some remaining downsides
- Future work: optimize synchronization on single objects transferred between threads

Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing

kenneth.russell@sun.com
david.detlefs@alum.mit.edu