The JSR-133 Cookbook for Compiler Writers

by Doug Lea, with help from members of the JMM mailing list.

dl@cs.oswego.edu.

Preface: This document is now of historical interest only. Some of the accounts here of processor support for ordering and atomics are obsolete and should not be relied on. For updated accounts, see, among other sources, the guide to JDK9+ memory order modes. For current processor support, see C++ mappings.

This is an unofficial guide to implementing the new Java Memory Model (JMM) specified by JSR-133 . It provides at most brief backgrounds about why various rules exist, instead concentrating on their consequences for compilers and JVMs with respect to instruction reorderings, multiprocessor barrier instructions, and atomic operations. It includes a set of recommended recipes for complying to JSR-133. This guide is "unofficial" because it includes interpretations of particular processor properties and specifications. We cannot guarantee that the intepretations are correct. Also, processor specifications and implementations may change over time.

Reorderings

For a compiler writer, the JMM mainly consists of rules disallowing reorderings of certain instructions that access fields (where "fields" include array elements) as well as monitors (locks).

Volatiles and Monitors

The main JMM rules for volatiles and monitors can be viewed as a matrix with cells indicating that you cannot reorder instructions associated with particular sequences of bytecodes. This table is not itself the JMM specification; it is just a useful way of viewing its main consequences for compilers and runtime systems.

Can Reorder	2nd operation		
1st operation	Normal Load Normal Store	Volatile Load MonitorEnter	Volatile Store MonitorExit
Normal Load Normal Store			No
Volatile Load MonitorEnter	No	No	No
Volatile store MonitorExit		No	No

Where:

- Normal Loads are getfield, getstatic, array load of non-volatile fields.
- Normal Stores are putfield, putstatic, array store of non-volatile fields
- Volatile Loads are getfield, getstatic of volatile fields that are accessible by multiple threads
- Volatile Stores are putfield, putstatic of volatile fields that are accessible by multiple threads
- MonitorEnters (including entry to synchronized methods) are for lock objects accessible by multiple threads.
- MonitorExits (including exit from synchronized methods) are for lock objects accessible by multiple threads.

The cells for Normal Loads are the same as for Normal Stores, those for Volatile Loads are the same as MonitorEnter, and those for Volatile Stores are same as MonitorExit, so they are collapsed together here (but are expanded out as needed in subsequent tables). We consider here only variables that are readable and writable as an atomic unit -- that is, no bit fields, unaligned accesses, or accesses larger than word sizes available on a platform.

Any number of other operations might be present between the indicated 1st and 2nd operations in the table. So, for example, the "No" in cell [Normal Store, Volatile Store] says that a non-volatile store cannot be reordered with ANY subsequent volatile store; at least any that can make a difference in multithreaded program semantics.

The JSR-133 specification is worded such that the rules for both volatiles and monitors apply only to those that may be accessed by multiple threads. If a compiler can somehow (usually only with great effort) prove that a lock is only accessible from a single thread, it may be eliminated. Similarly, a volatile field provably accessible from only a single thread acts as a normal field. More fine-grained analyses and optimizations are also possible, for example, those relying on provable inaccessibility from multiple threads only during certain intervals.

Blank cells in the table mean that the reordering is allowed if the accesses aren't otherwise dependent with respect to basic Java semantics (as specified in the JLS). For example even though the table doesn't say so, you can't reorder a load with a subsequent store to the same location. But you can reorder a load and store to two distinct locations, and may wish to do so in the course of various compiler transformations and optimizations. This includes cases that aren't usually thought of as reorderings; for example reusing a computed value based on a loaded field rather than reloading and recomputing the value acts as a reordering. However, the JMM spec permits transformations that eliminate avoidable dependencies, and in turn allow reorderings.

In all cases, permitted reorderings must maintain minimal Java safety properties even when accesses are incorrectly synchronized by programmers: All observed field values must be either the default zero/null "pre-construction" values, or those written by some thread. This usually entails zeroing all heap memory holding objects before it is used in constructors and never reordering other loads with the zeroing stores. A good way to do this is to zero out reclaimed memory within the garbage collector. See the JSR-133 spec for rules dealing with other corner cases surrounding safety guarantees.

The rules and properties described here are for accesses to Java-level fields. In practice, these will additionally interact with accesses to internal bookkeeping fields and data, for example object headers, GC tables, and dynamically generated code.

Final Fields

Loads and Stores of final fields act as "normal" accesses with respect to locks and volatiles, but impose two additional reordering rules:

1. A store of a final field (inside a constructor) and, if the field is a reference, any store that this final can reference, cannot be reordered with a subsequent store (outside that constructor) of the reference to the object holding that field into a variable accessible to other threads. For example, you cannot reorder
   ```
   x.finalField = v; ... ; sharedRef = x;
   ```
   This comes into play for example when inlining constructors, where "..." spans the logical end of the constructor. You cannot move stores of finals within constructors down below a store outside of the constructor that might make the object visible to other threads. (As seen below, this may also require issuing a barrier). Similarly, you cannot reorder either of the first two with the third assignment in:
   ```
   v.afield = 1; x.finalField = v; ... ; sharedRef = x;
   ```
2. The initial load (i.e., the very first encounter by a thread) of a final field cannot be reordered with the initial load of the reference to the object containing the final field. This comes into play in:
   ```
   x = sharedRef; ... ; i = x.finalField;
   ```
   A compiler would never reorder these since they are dependent, but there can be consequences of this rule on some processors.

These rules imply that reliable use of final fields by Java programmers requires that the load of a shared reference to an object with a final field itself be synchronized, volatile, or final, or derived from such a load, thus ultimately ordering the initializing stores in constructors with subsequent uses outside constructors.

Memory Barriers

Compilers and processors must both obey reordering rules. No particular effort is required to ensure that uniprocessors maintain proper ordering, since they all guarantee "as-if-sequential" consistency. But on multiprocessors, guaranteeing conformance often requires emitting barrier instructions. Even if a compiler optimizes away a field access (for example because a loaded value is not used), barriers must still be generated as if the access were still present. (Although see below about independently optimizing away barriers.)

Memory barriers are only indirectly related to higher-level notions described in memory models such as "acquire" and "release". And memory barriers are not themselves "synchronization barriers". And memory barriers are unrelated to the kinds of "write barriers" used in some garbage collectors. Memory barrier instructions directly control only the interaction of a CPU with its cache, with its write-buffer that holds stores waiting to be flushed to memory, and/or its buffer of waiting loads or speculatively executed instructions. These effects may lead to further interaction among caches, main memory and other processors. But there is nothing in the JMM that mandates any particular form of communication across processors so long as stores eventually become globally performed; i.e., visible across all processors, and that loads retrieve them when they are visible.

Categories

Nearly all processors support at least a coarse-grained barrier instruction, often just called a Fence, that guarantees that all loads and stores initiated before the fence will be strictly ordered before any load or store initiated after the fence. This is usually among the most time-consuming instructions on any given processor (often nearly as, or even more expensive than atomic instructions). Most processors additionally support more fine-grained barriers.

A property of memory barriers that takes some getting used to is that they apply BETWEEN memory accesses. Despite the names given for barrier instructions on some processors, the right/best barrier to use depends on the kinds of accesses it separates. Here's a common categorization of barrier types that maps pretty well to specific instructions (sometimes no-ops) on existing processors:

LoadLoad Barriers
: The sequence: `Load1; LoadLoad; Load2`
ensures that Load1's data are loaded before data accessed by Load2 and all subsequent load instructions are loaded. In general, explicit LoadLoad barriers are needed on processors that perform speculative loads and/or out-of-order processing in which waiting load instructions can bypass waiting stores. On processors that guarantee to always preserve load ordering, the barriers amount to no-ops.

StoreStore Barriers
: The sequence: `Store1; StoreStore; Store2`
ensures that Store1's data are visible to other processors (i.e., flushed to memory) before the data associated with Store2 and all subsequent store instructions. In general, StoreStore barriers are needed on processors that do not otherwise guarantee strict ordering of flushes from write buffers and/or caches to other processors or main memory.

LoadStore Barriers

The sequence: `Load1; LoadStore; Store2`
ensures that Load1's data are loaded before all data associated with Store2 and subsequent store instructions are flushed. LoadStore barriers are needed only on those out-of-order procesors in which waiting store instructions can bypass loads.

StoreLoad Barriers
The sequence: `Store1; StoreLoad; Load2`
ensures that Store1's data are made visible to other processors (i.e., flushed to main memory) before data accessed by Load2 and all subsequent load instructions are loaded. StoreLoad barriers protect against a subsequent load incorrectly using Store1's data value rather than that from a more recent store to the same location performed by a different processor. Because of this, on the processors discussed below, a StoreLoad is strictly necessary only for separating stores from subsequent loads of the same location(s) as were stored before the barrier. StoreLoad barriers are needed on nearly all recent multiprocessors, and are usually the most expensive kind. Part of the reason they are expensive is that they must disable mechanisms that ordinarily bypass cache to satisfy loads from write-buffers. This might be implemented by letting the buffer fully flush, among other possible stalls.

On all processors discussed below, it turns out that instructions that perform StoreLoad also obtain the other three barrier effects, so StoreLoad can serve as a general-purpose (but usually expensive) Fence. (This is an empirical fact, not a necessity.) The opposite doesn't hold though. It is NOT usually the case that issuing any combination of other barriers gives the equivalent of a StoreLoad.

The following table shows how these barriers correspond to JSR-133 ordering rules.

Required barriers	2nd operation			
1st operation	Normal Load	Normal Store	Volatile Load MonitorEnter	Volatile Store MonitorExit
Normal Load				LoadStore
Normal Store				StoreStore
Volatile Load MonitorEnter	LoadLoad	LoadStore	LoadLoad	LoadStore
Volatile Store MonitorExit			StoreLoad	StoreStore

Plus the special final-field rule requiring a StoreStore barrier in
```
x.finalField = v; StoreStore; sharedRef = x;
```

Here's an example showing placements.

Java	Instructions
`class X {`	
`  int a, b;`	
`  volatile int v, u;`	
`  void f() {`	
`    int i, j;`	
`    i = a;`	`load a`
`    j = b;`	`load b`
`    i = v;`	`load v`
	`    LoadLoad`
`    j = u;`	`load u`
	`    LoadStore`
`    a = i;`	`store a`
`    b = j;`	`store b`
	`    StoreStore`
`    v = i;`	`store v`
	`    StoreStore`
`    u = j;`	`store u`
	`    StoreLoad`
`    i = u;`	`load u`
	`    LoadLoad`
	`    LoadStore`
`    j = b;`	`load b`
`    a = i;`	`store a`
`  }`	
`}`	

Data Dependency and Barriers

The need for LoadLoad and LoadStore barriers on some processors interacts with their ordering guarantees for dependent instructions. On some (most) processors, a load or store that is dependent on the value of a previous load are ordered by the processor without need for an explicit barrier. This commonly arises in two kinds of cases, indirection:
```
Load x; Load x.field
```
and control
```
Load x; if (predicate(x)) Load or Store y;
```

Processors that do NOT respect indirection ordering in particular require barriers for final field access for references initially obtained through shared references:
```
x = sharedRef; ... ; LoadLoad; i = x.finalField;
```

Conversely, as discussed below, processors that DO respect data dependencies provide several opportunities to optimize away LoadLoad and LoadStore barrier instructions that would otherwise need to be issued. (However, dependency does NOT automatically remove the need for StoreLoad barriers on any processor.)

Interactions with Atomic Instructions

The kinds of barriers needed on different processors further interact with implementation of MonitorEnter and MonitorExit. Locking and/or unlocking usually entail the use of atomic conditional update operations CompareAndSwap (CAS) or LoadLinked/StoreConditional (LL/SC) that have the semantics of performing a volatile load followed by a volatile store. While CAS or LL/SC minimally suffice, some processors also support other atomic instructions (for example, an unconditional exchange) that can sometimes be used instead of or in conjunction with atomic conditional updates.

On all processors, atomic operations protect against read-after-write problems for the locations being read/updated. (Otherwise standard loop-until-success constructions wouldn't work in the desired way.) But processors differ in whether atomic instructions provide more general barrier properties than the implicit StoreLoad for their target locations. On some processors these instructions also intrinsically perform barriers that would otherwise be needed for MonitorEnter/Exit; on others some or all of these barriers must be specifically issued.

Volatiles and Monitors have to be separated to disentangle these effects, giving:

Required Barriers	2nd operation					
1st operation	Normal Load	Normal Store	Volatile Load	Volatile Store	MonitorEnter	MonitorExit
Normal Load				LoadStore		LoadStore
Normal Store				StoreStore		StoreExit
Volatile Load	LoadLoad	LoadStore	LoadLoad	LoadStore	LoadEnter	LoadExit
Volatile Store			StoreLoad	StoreStore	StoreEnter	StoreExit
MonitorEnter	EnterLoad	EnterStore	EnterLoad	EnterStore	EnterEnter	EnterExit
MonitorExit			ExitLoad	ExitStore	ExitEnter	ExitExit

Plus the special final-field rule requiring a StoreStore barrier in:

```
x.finalField = v; StoreStore; sharedRef = x;
```

In this table, "Enter" is the same as "Load" and "Exit" is the same as "Store", unless overridden by the use and nature of atomic instructions. In particular:

- EnterLoad is needed on entry to any synchronized block/method that performs a load. It is the same as LoadLoad unless an atomic instruction is used in MonitorEnter and itself provides a barrier with at least the properties of LoadLoad, in which case it is a no-op.
- StoreExit is needed on exit of any synchronized block/method that performs a store. It is the same as StoreStore unless an atomic instruction is used in MonitorExit and itself provides a barrier with at least the properties of StoreStore, in which case it is a no-op.
  ExitEnter is the same as StoreLoad unless atomic instructions are used in MonitorExit and/or MonitorEnter and at least one of these provide a barrier with at least the properties of StoreLoad, in which case it is a no-op.

The other types are specializations that are unlikely to play a role in compilation (see below) and/or reduce to no-ops on current processors. For example, EnterEnter is needed to separate nested MonitorEnters when there are no intervening loads or stores. Here's an example showing placements of most types:

```
Java                                  Instructions
class X {
   int a;
   volatile int v;
   void f() {
       int i;
       synchronized(this) {
                                      enter
           i = a;                          EnterLoad
           a = i;                          EnterStore
       }                              load a
                                      store a
                                           LoadExit
                                           StoreExit
                                      exit
       synchronized(this) {                ExitEnter
           synchronized(this) {       enter
           }                               EnterEnter
       }                              enter
                                           EnterExit
                                      exit
                                           ExitExit
                                      exit
                                           ExitEnter
                                           ExitLoad
       i = v;                         load v
       synchronized(this) {                LoadEnter
       }                              enter
                                           EnterExit
                                      exit
```

`        v = i;`
`        synchronized(this) {`
`        }`
`    }`
`}`
	ExitEnter
	ExitStore
	store v
	StoreEnter
	enter
	EnterExit
	exit

Java-level access to atomic conditional update operations will be available in JDK1.5 via JSR-166 (concurrency utilities) so compilers will need to issue associated code, using a variant of the above table that collapses MonitorEnter and MonitorExit -- semantically, and sometimes in practice, these Java-level atomic updates act as if they are surrounded by locks.

Multiprocessors

Here's a listing of processors that are commonly used in MPs, along with links to documents providing information about them. (Some require some clicking around from the linked site and/or free registration to access manuals). This isn't an exhaustive list, but it includes processors used in all current and near-future multiprocessor Java implementations I know of. The list and the properties of processors decribed below are not definitive. In some cases I'm just reporting what I read, and could have misread. Several reference manuals are not very clear about some properties relevant to the JMM. Please help make it definitive.

Good sources of hardware-specific information about barriers and related properties of machines not listed here are Hans Boehm's atomic_ops library, the Linux Kernel Source, and Linux Scalability Effort. Barriers needed in the linux kernel correspond in straightforward ways to those discussed here, and have been ported to most processors. For descriptions of the underlying models supported on different processors, see Sarita Adve et al, Recent Advances in Memory Consistency Models for Hardware Shared-Memory Systems and Sarita Adve and Kourosh Gharachorloo, Shared Memory Consistency Models: A Tutorial.

sparc-TSO
: Ultrasparc 1, 2, 3 (sparcv9) in TSO (Total Store Order) mode. Ultra3s only support TSO mode. (RMO mode in Ultra1/2 is never used so can be ignored.) See UltraSPARC III Cu User's Manual and The SPARC Architecture Manual, Version 9 .

x86 (and x64)
: Intel 486+, as well as AMD and apparently others. There was a flurry of re-specs in 2005-2009, but the current specs are nearly identical to TSO, differing mainly only in supporting different cache modes, and dealing with corner cases such as unaligned accesses and special forms of instructions. See The IA-32 Intel Architecture Software Developers Manuals: System Programming Guide and AMD Architecture Programmer's Manual Programming.

ia64
: Itanium. See Intel Itanium Architecture Software Developer's Manual, Volume 2: System Architecture

ppc (POWER)
: All versions have the same basic memory model, but the names and definition of some memory barrier instructions changed over time. The listed versions have been current since Power4; see architecture manuals for details. See MPC603e RISC Microprocessor Users Manual, MPC7410/MPC7400 RISC Microprocessor Users Manual , Book II of PowerPC Architecture Book, PowerPC Microprocessor Family: Software reference manual, Book E- Enhanced PowerPC Architecture, EREF: A Reference for Motorola Book E and the e500 Core. For discussion of barriers see IBM article on power4 barriers, and IBM article on powerpc barriers.

arm
: Version 7+. See ARM processor specifications

alpha
: 21264x and I think all others. See Alpha Architecture Handbook

pa-risc
: HP pa-risc implementations. See the pa-risc 2.0 Architecture manual.

Here's how these processors support barriers and atomics:

Processor	**LoadStore**	**LoadLoad**	**StoreStore**	**StoreLoad**	**Data dependency orders loads?**	**Atomic Conditional**	**Other Atomics**	**Atomics provide barrier?**
sparc-TSO	no-op	no-op	no-op	membar (StoreLoad)	yes	CAS: casa	swap, ldstub	full
x86	no-op	no-op	no-op	mfence or cpuid or locked insn	yes	CAS: cmpxchg	xchg, locked insn	full
ia64	combine with st.rel or ld.acq	ld.acq	st.rel	mf	yes	CAS: cmpxchg	xchg, fetchadd	target + acq/rel
arm	dmb (see below)	dmb (see below)	dmb-st	dmb	indirection only	LL/SC: ldrex/strex		target only
ppc	lwsync (see below)	hwsync (see below)	lwsync	hwsync	indirection only	LL/SC: ldarx/stwcx		target only
alpha	mb	mb	wmb	mb	no	LL/SC: ldx_l/stx_c		target only

pa-risc	no-op	no-op	no-op	no-op	yes	build from ldcw	ldcw	(NA)

Notes

- Some of the listed barrier instructions have stronger properties than actually needed in the indicated cells, but seem to be the cheapest way to get desired effects.
- The listed barrier instructions are those designed for use with normal program memory, but not necessarily other special forms/modes of caching and memory used for IO and system tasks. For example, on x86-SPO, StoreStore barriers ("sfence") are needed with WriteCombining (WC) caching mode, which is designed for use in system-level bulk transfers etc. OSes use Writeback mode for programs and data, which doesn't require StoreStore barriers.
- On x86, any lock-prefixed instruction can be used as a StoreLoad barrier. (The form used in linux kernels is the no-op `lock; addl $0,0(%%esp)`.) Versions supporting the "SSE2" extensions (Pentium4 and later) support the mfence instruction which seems preferable unless a lock-prefixed instruction like CAS is needed anyway. The cpuid instruction also works but is slower.
- On ia64, LoadStore, LoadLoad and StoreStore barriers are folded into special forms of load and store instructions -- there aren't separate instructions. ld.acq acts as (load; LoadLoad+LoadStore) and st.rel acts as (LoadStore+StoreStore; store). Neither of these provide a StoreLoad barrier -- you need a separate mf barrier instruction for that.
- On both ARM and ppc, there may be opportunities to replace load fences in the presence of data dependencies with non-fence-based instruction sequences. Sequences and cases in which they apply are described in work by the Cambridge Relaxed Memory Concurrency Group.
- The sparc membar instruction supports all four barrier modes, as well as combinations of modes. But only the StoreLoad mode is ever needed in TSO. On some UltraSparcs, any membar instruction produces the effects of a StoreLoad, regardless of mode.
- The x86 processors supporting "streaming SIMD" SSE2 extensions require LoadLoad "lfence" only only in connection with these streaming instructions.
- Although the pa-risc specification does not mandate it, all HP pa-risc implementations are sequentially consistent, so have no memory barrier instructions.
- The only atomic primitive on pa-risc is ldcw, a form of test-and-set, from which you would need to build up atomic conditional updates using techniques such as those in the HP white paper on spinlocks.
- CAS and LL/SC take multiple forms on different processors, differing only with respect to field width, minimially including 4 and 8 byte versions.
- On sparc and x86, CAS has implicit preceding and trailing full StoreLoad barriers. The sparcv9 architecture manual says CAS need not have post-StoreLoad barrier property, but the chip manuals indicate that it does on ultrasparcs.
- On ppc and alpha, LL/SC have implicit barriers only with respect to the locations being loaded/stored, but don't have more general barrier properties.
- The ia64 cmpxchg instruction also has implicit barriers with respect to the locations being loaded/stored, but additionally takes an optional .acq (post-LoadLoad+LoadStore) or .rel (pre-StoreStore+LoadStore) modifier. The form cmpxchg.acq can be used for MonitorEnter, and cmpxchg.rel for MonitorExit. In those cases where exits and enters are not guaranteed to be matched, an ExitEnter (StoreLoad) barrier may also be needed.
- Sparc, x86 and ia64 support unconditional-exchange (swap, xchg). Sparc ldstub is a one-byte test-and-set. ia64 fetchadd returns previous value and adds to it. On x86, several instructions (for example add-to-memory) can be lock-prefixed, causing them to act atomically.

Recipes

Uniprocessors

If you are generating code that is guaranteed to only run on a uniprocessor, then you can probably skip the rest of this section. Because uniprocessors preserve apparent sequential consistency, you never need to issue barriers unless object memory is somehow shared with asynchrononously accessible IO memory. This might occur with specially mapped java.nio buffers, but probably only in ways that affect internal JVM support code, not Java code. Also, it is conceivable that some special barriers would be needed if context switching doesn't entail sufficient synchronization.

Inserting Barriers

Barrier instructions apply between different kinds of accesses as they occur during execution of a program. Finding an "optimal" placement that minimizes the total number of executed barriers is all but impossible. Compilers often cannot tell if a given load or store will be preceded or followed by another that requires a barrier; for example, when a volatile store is followed by a return. The easiest conservative strategy is to assume that the kind of access requiring the "heaviest" kind of barrier will occur when generating code for any given load, store, lock, or unlock:

1. Issue a StoreStore barrier before each volatile store.
   (On ia64 you must instead fold this and most barriers into corresponding load or store instructions.)
2. Issue a StoreStore barrier after all stores but before return from any constructor for any class with a final field.
3. Issue a StoreLoad barrier after each volatile store.
   Note that you could instead issue one before each volatile load, but this would be slower for typical programs using volatiles in which reads greatly outnumber writes. Alternatively, if available, you can implement volatile store as an atomic instruction (for example XCHG on x86) and omit the barrier. This may be more efficient if atomic instructions are cheaper than StoreLoad barriers.
4. Issue LoadLoad and LoadStore barriers after each volatile load.
   On processors that preserve data dependent ordering, you need not issue a barrier if the next access instruction is dependent on the value of the load. In particular, you do not need a barrier after a load of a volatile reference if the subsequent instruction is a null-check or load of a field of that reference.
5. Issue an ExitEnter barrier either before each MonitorEnter or after each MonitorExit.
   (As discussed above, ExitEnter is a no-op if either MonitorExit or MonitorEnter uses an atomic instruction that supplies the equivalent of a StoreLoad barrier. Similarly for others involving Enter and Exit in the remaining steps.)
6. Issue EnterLoad and EnterStore barriers after each MonitorEnter.
7. Issue StoreExit and LoadExit barriers before each MonitorExit.
8. If on a processor that does not intrinsically provide ordering on indirect loads, issue a LoadLoad barrier before each load of a final field. (Some alternative strategies are discussed in this JMM list posting, and this description of linux data dependent barriers.)

Many of these barriers usually reduce to no-ops. In fact, most of them reduce to no-ops, but in different ways under different processors and locking schemes. For the simplest examples, basic conformance to JSR-133 on x86 or sparc-TSO using CAS for locking amounts only to placing a StoreLoad barrier after volatile stores.

Removing Barriers

The conservative strategy above is likely to perform acceptably for many programs. The main performance issues surrounding volatiles occur for the StoreLoad barriers associated with stores. These ought to be relatively rare -- the main reason for using volatiles in concurrent programs is to avoid the need to use locks around reads, which is only an issue when reads greatly overwhelm writes. But this strategy can be improved in at least the following ways:

- Removing redundant barriers. The above tables indicate that barriers can be eliminated as follows:

Original			=>	Transformed		
1st	ops	2nd	=>	1st	ops	2nd
LoadLoad	[no loads]	LoadLoad	=>		[no loads]	LoadLoad
LoadLoad	[no loads]	StoreLoad	=>		[no loads]	StoreLoad
StoreStore	[no stores]	StoreStore	=>		[no stores]	StoreStore
StoreStore	[no stores]	StoreLoad	=>		[no stores]	StoreLoad
StoreLoad	[no loads]	LoadLoad	=>	StoreLoad	[no loads]	
StoreLoad	[no stores]	StoreStore	=>	StoreLoad	[no stores]	
StoreLoad	[no volatile loads]	StoreLoad	=>		[no volatile loads]	StoreLoad

  Similar eliminations can be used for interactions with locks, but depend on how locks are implemented. Doing all this in the presence of loops, calls, and branches is left as an exercise for the reader. :-)

- Rearranging code (within the allowed constraints) to further enable removing LoadLoad and LoadStore barriers that are not needed because of data dependencies on processors that preserve such orderings.

- Moving the point in the instruction stream that the barriers are issued, to improve scheduling, so long as they still occur somewhere in the interval they are required.

- Removing barriers that aren't needed because there is no possibility that multiple threads could rely on them; for example volatiles that are provably visible only from a single thread. Also, removing some barriers when it can be proven that threads can only store or only load certain fields. All this usually requires a fair amount of analysis.

Miscellany

JSR-133 also addresses a few other issues that may entail barriers in more specialized cases:

- Thread.start() requires barriers ensuring that the started thread sees all stores visible to the caller at the call point. Conversely, Thread.join() requires barriers ensuring that the caller sees all stores by the terminating thread. These are normally generated by the synchronization entailed in implementations of these constructs.

- Static final initialization requires StoreStore barriers that are normally entailed in mechanics needed to obey Java class loading and initialization rules.

- Ensuring default zero/null initial field values normally entails barriers, synchronization, and/or low-level cache control within garbage collectors.

- JVM-private routines that "magically" set System.in, System.out, and System.err outside of constructors or static initializers need special attention since they are special legacy exceptions to JMM rules for final fields.
- Similarly, internal JVM deserialization code that sets final fields normally requires a StoreStore barrier.
- Finalization support may require barriers (within garbage collectors) to ensure that Object.finalize code sees all stores to all fields prior to the objects becoming unreferenced. This is usually ensured via the synchronization used to add and remove references in reference queues.
- Calls to and returns from JNI routines may require barriers, although this seems to be a quality of implementation issue.
- Most processors have other synchronizing instructions designed primarily for use with IO and OS actions. These don't impact JMM issues directly, but may be involved in IO, class loading, and dynamic code generation.

Acknowledgments

Thanks to Bill Pugh, Dave Dice, Jeremy Manson, Kourosh Gharachorloo, Tim Harris, Cliff Click, Allan Kielstra, Yue Yang, Hans Boehm, Kevin Normoyle, Juergen Kreileder, Alexander Terekhov, Tom Deneau, Clark Verbrugge, Peter Kessler, Peter Sewell, Jan Vitek, and Richard Grisenthwaite for corrections and suggestions.

A translation of this page is available in japanese.

Doug Lea

Last modified: Tue Mar 22 07:11:36 EDT 2011