Improving the Java Memory Model Using CRF

Jan-Willem Maessen
Arvind
Xiaowei Shen

[jmaessen, arvind]@lcs.mit.edu,
xwshen@us.ibm.com
Java Memory Model: Problems

- Incomplete
  - No semantics for final fields

- Disallows important optimizations
  - Reordering of loads to same location
  - Some reordering are inexpressible in source

- Difficult to understand
  - Memory updates not atomic
Roadmap

- Examples of JMM problems
- Desired Programming Discipline
  - Well-behaved programs
  - Source-level algebraic reasoning
- Translating Java into CRF
- Conclusions
Thread 1
char [ ] a = {‘H’,’i’};
s = new MyString(a);

Thread 2
print(s);

Thread 2 should either print “Hi” or throw an exception

class MyString {
    private final char[ ] theCharacters;
    public MyString( char[ ] value) {
        char[ ] internalValue = value.clone();
        theCharacters = internalValue;
    }
    ...
}
Enabling Optimizations

Thread 1
\[ v = p.f; \]
\[ w = q.f; \]
\[ x = p.f; \]

Thread 2
\[ p.f = 2; \]

Can we replace \( x = p.f \) by \( x = v \) ?

- **Old JMM: No!**
  
  What if \( p==q \)? Reads must be ordered!

- **Proposed JMM: Yes!**
  
  Reads can be reordered
Program behavior is context-sensitive [Pugh99]
The old JMM semantics are simply too convoluted!
The Java Memory Model

[Gosling, Joy, Steele, 1st ed., Ch 17]

Seven axioms define permitted reorderings:
- use and assign occur in program order
- store and write to a location occur in order
- read and load from a location occur in order
Solution: Make Reorderings Explicit

Reorder at the thread level
Make instructions atomic
Plan of action

- Define a desired programming style for Java
- Give high-level description of program behavior
- Capture high-level description in a precise semantics
Java Memory Operations

- **Regular Memory**
  \[ v = \text{LoadR} \ p.f \quad \text{StoreR} \ p.f, v \]

- **Volatile Memory**
  \[ v = \text{LoadV} \ p.f \quad \text{StoreV} \ p.f, v \]

- **Final Memory**
  \[ v = \text{LoadF} \ p.f \quad \text{StoreF} \ p.f, v \]
  \[ \text{EndCon} \]

- **Monitors**
  \[ \text{Enter} \quad | \quad \text{Exit} \]
Regular fields

Constrained only by data dependence

- Load/Store must be protected by monitors
  - If it's shared, it must be locked during access

- Read-only objects can omit synchronization
  - But only when reached through final fields
Final Fields and Constructors

Allow creation of read-only data

- An object must not escape its constructor

- Final fields may be read without synchronization
  - Includes referenced read-only objects
Allow free-form data exchange

- Volatile operations occur in program order
- Volatile loads act like Enter
- Volatile stores act like Exit
- Any field may safely be made volatile
Algebraic Rules

- Source-to-source program manipulation
  - See the effects of reordering
  - Reason about incorrect program behavior

- Captures legal static reorderings

- Easy to reason about interleaved execution

- Implied by dynamic semantics
Load/Store Reordering

- Must respect usual dependencies:
  \[\text{Store } p.f,4; \quad x = \text{Load } p.f; \quad \text{Store } p.f,5;\]

- Regular & Final operations reorder freely:
  \[\text{StoreR } p.f,4; \quad y = \text{LoadF } q.g; \quad x = \text{LoadF } q.g; \quad x = \text{LoadF } q.g; \quad y = \text{LoadF } q.g; \quad \text{StoreR } p.f,4;\]

- Volatile operations do not reorder!
Synchronization

- Any Load/Store may enter synchronization

  LoadR q.f;  
  Enter p.l;  
  LoadR p.f;  
  Exit p.l;  
  LoadR q.g;  

- Non-finals may not escape synchronization

- Enter must be ordered wrt both Enter and Exit.
Other Interactions

- LoadV acts like Enter, StoreV acts like Exit
  - LoadR q.f;
  - LoadV p.v;
  - LoadR g.f;
  - StoreV p.v;
  - LoadR q.g;
  - StoreV p.v;

- EndCon keeps stores in, non-final stores out:
  - StoreF p.f, 5;
  - EndCon;
  - StoreF q.g, p;
  - EndCon;
  - StoreR r.h, p;
  - StoreR r.h, p;
Reordering Around Control Flow

Thread 1

```java
int tmp1 = p.flag;
if (tmp1 == 1) {
    int tmp2 = p.flag;
    system.out.print("yes");
    if (tmp2 == 0) {
        system.out.print("BAD");
    }
}
```

Thread 2

```java
p.flag = 1;
```

**Consequence**

of poor synchronization
Compilation

- *Dependency Analysis* = Reordering
  - Read/write constraints don’t capture reorderings

- *Type & alias analyses* permit read/write reordering
  - Regular, volatile, and final storage are disjoint!

- *Escape analysis* permits local operation reordering

- *Pointer analysis* spots fetches via final pointers
Roadmap

- Examples of JMM problems
- Desired Programming Discipline
  - Well-behaved programs
  - Source-level algebraic reasoning
- Translating Java into CRF
- Conclusions
CRF: A General Representation

Java Threads
(regular, final, volatile, monitors)

Commit-Reconcile & Fences (CRF)

X86  Sparc  PowerPC  Alpha

(Shen, Arvind, Rudolph, ISCA99)
Java to CRF: Regular Memory

\[
\begin{align*}
x &= \text{LoadR p.f}; & \text{Reconcile p.f};
x &= \text{LoadL p.f}; \\
\text{StoreR p.f, y}; & \text{StoreL p.f, y}; & \text{Commit p.f};
\end{align*}
\]
The CRF Model

- **data caching via semantic caches**
  - Cache updates at any time (background)
  - Commit, Reconcile force updates
- **instruction reordering** (controllable via Fence)
- **all operations act atomically**
The Fence Operations

Instructions can be reordered except for:

- Data dependence
- StoreL \(a,v\); Commit \(a\);
- Reconcile \(a\); LoadL \(a\);

\[
\begin{align*}
\text{StoreL}(a_1, v); \\
\text{Commit}(a_1); \\
\text{Fence}_{wr} (a_1, a_2); \\
\text{Reconcile}(a_2); \\
\text{LoadL}(a_2); \\
\text{Fence}_{rr}; \quad \text{Fence}_{rw}; \quad \text{Fence}_{ww};
\end{align*}
\]
Important Properties of CRF

- Safe to add extra Commits & Reconciles
- Safe to add additional Fence operations

Extra operations reduce exhibited behaviors, but preserve correctness

Can use coarse-grain operations, e.g:

\[
\text{Fence}_{rr} p.f, *V; \quad \text{Fence}_{rr} p.f, *VR;
\]

\[
\text{Fence}_{ww} l, *VRL; \quad \text{Fence}_{ww} *, *VR;
\]
Java to CRF: Final Memory

StoreF p.f, x;  
\_  

StoreL p.f, x;
Commit p.f;
Freeze p.f;

y = LoadF p.f;  
\_  

Reconcile p.f;
y = LoadL p.f;
Freeze p.f;
Java to CRF: Volatile Memory

\[ x = \text{LoadV} \ p.f; \]
\[ \text{Fence}_{rr} \ *V, \ p.f; \]
\[ \text{Fence}_{wr} \ *V, \ p.f; \]
\[ \text{Reconcile} \ p.f; \]
\[ x = \text{LoadL} \ p.f; \]
\[ \text{Fence}_{rr} \ p.f, \ *VR; \]
\[ \text{Fence}_{rw} \ p.f, \ *VR; \]

\[ \text{StoreV} \ p.f, \ y; \]
\[ \text{Fence}_{rw} \ *VR, \ p.f; \]
\[ \text{Fence}_{ww} \ *VR, \ p.f; \]
\[ \text{StoreL} \ p.f, \ y; \]
\[ \text{Commit} \ p.f; \]
Java to CRF: Synchronization

Enter $l$;  
-  
\[ \text{Fence}_{ww} *L, l; \]
\[ \text{Lock } l; \]
\[ \text{Fence}_{wr} l, *VR; \]
\[ \text{Fence}_{ww} l, *VRL; \]

Exit $l$;  
-  
\[ \text{Fence}_{ww} *VR, l; \]
\[ \text{Fence}_{rw} *VR, l; \]
\[ \text{Unlock } l; \]

EndCon;  
-  
\[ \text{Fence}_{ww} *,*VR; \]
Allowing Lock Elimination

Enter l; 

Fence_{ww} *L, l;

r = Lock l;

if (r!= currentThread) {
   Fence_{wr} l, *VR;
   Fence_{ww} l, *VRL;
}

- Operations move upward out of lock region
  - Including into preceding lock regions
- Operations cannot move downward
Limits on Reordering

- Some reordering must be dynamic
  - Potential aliasing

- Some reordering is probably purely static
  - Based on analysis

- The boundary of static reordering is fuzzy
  \[ a[x^3 + y^3] \quad a[z^3] \]

- Solution: Flexible dynamic translation
Memory Model Issues Remaining

- Speculation
  - Arbitrary value speculation is the limit point
  - Reordering around control gives us a lot
  - Points between difficult to formalize
  - Biggest open area in memory models

- G-CRF allows non-atomic Commit
  
  *No change in translation needed*
  
  - Is it necessary?
  - Can it be understood
Other Memory Models

- Data-Race-Free and Properly Labeled programs
  [Adve & Gharachorloo, ...]
  - Define a programming style
  - Appearance of sequential consistency

- Location consistency
  [Gao & Sarkar, ...]
  - Order writes per-thread & per-location
  - Set of possible values at each load
Java Issues Remaining

- Run-time system memory model issues
  - New threads start with parent's state
  - GC responsible for its own synchronization
  - EndCon for object pre-initialization
- Thread-safe Library code
  - Code libraries correctly
  - Clarify finalization
  - Fix native code mutating final fields
- Establishing thread-safe Patterns
  - Lock-free caching (double-checking breaks)
  - Freezing mutable objects (Java Beans)
Java Memory Model in CRF

- Precise and easy to understand
  - *Reason about reordering at instruction level*
  - *Intuitive high-level semantics*

- Flexible
  - *Easy to experiment with possible translations*

- Makes optimizations obvious
  - *Reordering expressible in source*

- Simple mapping to a variety of architectures
Acknowledgements

- Bill Pugh
- Guy Steele
- David Detlefs
- Jeremy Manson
- Vivek Sarkar & the Jalapeno group
- The readers of the JMM mailing list
Question Slides
Thread 1

List q = p.next;
if (q == p) {
    List tmp = p.next;
    System.out.print("y");
    List r = tmp.next;
    if (r == null) {
        System.out.print("es");
    }
}

Thread 2

p.next = p;
Another Try

Thread 1

List r = p.next;
List q = p.next;
if (q == p) {
    system.out.print("y");
    if (r == null) {
        system.out.print("es");
    }
}

Thread 2

p.next = p;
CRF: LoadL and StoreL

- LoadL reads from the cache if the address is cached
- StoreL writes into the cache and sets the state to Dirty
CRF: Commit and Reconcile

- Commit completes if the address is not cached in the Dirty state
- Reconcile completes if the address is not cached in Clean
CRF: Background Operations

- **Cache** (retrieve) a copy of an uncached address from memory
- **Writeback** a Dirty copy to memory and set its state Clean
- **Purge** a Clean copy
CRF Extensions: Lock and Unlock

- **Lock** atomically increments the monitor count
- **Unlock** atomically decrements the monitor count

**Diagram:**
- Thread accesses `Cell(a,v,L)`
- Shared memory

[Diagram showing thread accessing `Cell(a,v,L)`]
CRF: Background Locking

- **Locked**: retrieve an exclusive copy of an unheld monitor from memory
- **Unlocked**: return an unheld monitor to memory for others to use
CRF Extensions: Freeze

- Freeze changes cache state to Frozen
- Reconcile can ignore Frozen entries