Memory Consistency Models

Sarita Adve

Department of Computer Science
University of Illinois at Urbana-Champaign
sadve@cs.uiuc.edu

Ack: Previous tutorials with Kourosh Gharachorloo
(some additional slides by KP in September 01)
What is a memory consistency model?

Implicit memory model – sequential consistency

Relaxed memory models (system-centric)

Programmer-centric approach for relaxed models

Application to Java

Conclusions
Memory Consistency Model: Definition

Memory consistency model

Order in which memory operations will appear to execute
⇒ What value can a read return?

Affects ease-of-programming and performance
Sequential consistency (SC) [Lamport]

Result of an execution appears as if

- All operations executed in some *sequential order*
- Memory operations of each process in *program order*

No caches, no write buffers
Sequential consistency (SC) [Lamport]

Result of an execution appears as if

- All operations executed in some sequential order
- Memory operations of each process in program order

Two aspects:

Program order

Atomicity

No caches, no write buffers
Initially $X = 2$

Possible execution sequences:

- P1: $r_0 = \text{Read}(X)$
- P2: $r_1 = \text{Read}(x)$
- P2: $r_1 = \text{Read}(X)$
- P2: $r_1 = r_1 + 1$
- P1: $r_0 = r_0 + 1$
- P2: $r_1 = r_1 + 1$
- P1: $\text{Write}(r_0, X)$
- P2: $\text{Write}(r_1, X)$
- P1: $\text{Write}(r_0, X)$
- P2: $\text{Write}(r_1, X)$

$x = 3$

$x = 4$
Atomic Operations

- sequential consistency has nothing to do with atomicity as shown by example on previous slide

- atomicity: use atomic operations such as exchange
  - exchange(r,M): swap contents of register r and location M

r0 = 1;

do exchange(r0,S) while (r0 != 0); //S is memory location
  //enter critical section

  .....

  //exit critical section

S = 0;
Initially Flag1 = Flag2 = 0

P1
Flag1 = 1
if (Flag2 == 0)
    critical section

P2
Flag2 = 1
if (Flag1 == 0)
    critical section

Execution:

P1
(\textit{Operation, Location, Value})
Write, Flag1, 1
Read, Flag2, 0

P2
(\textit{Operation, Location, Value})
Write, Flag2, 1
Read, Flag1, ___
Initially Flag1 = Flag2 = 0

P1
 Flag1 = 1
if (Flag2 == 0)
    critical section

P2
 Flag2 = 1
if (Flag1 == 0)
    critical section

Execution:

P1
(Operation, Location, Value)
Write, Flag1, 1
Read, Flag2, 0

P2
(Operation, Location, Value)
Write, Flag2, 1
Read, Flag1, ____
Initially Flag1 = Flag2 = 0

P1
Flag1 = 1
if (Flag2 == 0)
    critical section

P2
Flag2 = 1
if (Flag1 == 0)
    critical section

Execution:
P1
(Operation, Location, Value)
Write, Flag1, 1
Read, Flag2, 0

P2
(Operation, Location, Value)
Write, Flag2, 1
Read, Flag1, 0
Understanding Program Order – Example 1

P1
Write, Flag1, 1
Read, Flag2, 0

P2
Write, Flag2, 1
Read, Flag1,

Can happen if

• Write buffers with read bypassing
• Overlap, reorder write followed by read in h/w or compiler
• Allocate Flag1 or Flag2 in registers

On AlphaServer, NUMA-Q, T3D/T3E, Ultra Enterprise Server
Initially $A = \text{Flag} = 0$

P1
A = 23;
Flag = 1;

P2
while (Flag != 1) {};
... = A;

P1
Write, A, 23
Write, Flag, 1

P2
Read, Flag, 0
Read, Flag, 1
Read, A, ____
Initially \( A = \text{Flag} = 0 \)

\begin{align*}
P1 & \\
A & = 23; \\
\text{Flag} & = 1; \\
P1 & \\
\text{Write, A, 23} \\
\text{Write, Flag, 1} \\
\end{align*}

\begin{align*}
P1 & \\
P2 & \text{while (Flag} \neq 1) \{;\} \\
... & = A; \\
P2 & \\
\text{Read, Flag, 0} \\
\text{Read, Flag, 1} \\
\text{Read, A, } & 0 \\
\end{align*}
Initially $A = \text{Flag} = 0$

P1
$A = 23;$
Flag = 1;

P2
while (Flag != 1) {};
...

$\ldots = A;$

P1
Write, $A$, 23
Write, Flag, 1

P2
Read, Flag, 0
Read, Flag, 1
Read, $A$, 0

Can happen if

Overlap or reorder writes or reads in hardware or compiler

On AlphaServer, T3D/T3E
SC limits program order relaxation:

- Write → Read
- Write → Write
- Read → Read, Write
Sequential Consistency

SC constrains all memory operations:
- Write → Read
- Write → Write
- Read → Read, Write

- Simple model for reasoning about parallel programs
- But, intuitively reasonable reordering of memory operations in a uniprocessor may violate sequential consistency model

Modern microprocessors reorder operations all the time to obtain performance (write buffers, overlapped writes, non-blocking reads...).

Question: how do we reconcile sequential consistency model with the demands of performance?
A mechanism needed to propagate a write to other copies

⇒ Cache coherence protocol
- Sequential consistency is not really about memory operations from different processors (although we do need to make sure memory operations are atomic).

- Sequential consistency is not really about dependent memory operations in a single processor’s instruction stream (these are respected even by processors that reorder instructions).

- The problem of relaxing sequential consistency is really all about independent memory operations in a single processor’s instruction stream that have some high-level dependence (such as locks guarding data) that should be respected to obtain correct results.
- Weak ordering:
  - Divide memory operations into data operations and synchronization operations
  - Synchronization operations act like a fence:
    - All data operations before synch in program order must complete before synch is executed
    - All data operations after synch in program order must wait for synch to complete
    - Synchs are performed in program order
  - Implementation of fence: processor has counter that is incremented when data op is issued, and decremented when data op is completed
  - Example: PowerPC has SYNC instruction (caveat: semantics somewhat more complex than what we have described…)

*Relaxing Program Orders*
Another model: Release consistency

- Further relaxation of weak consistency
  - Synchronization accesses are divided into
    - Acquires: operations like lock
    - Release: operations like unlock
  - Semantics of acquire:
    - Acquire must complete before all following memory accesses
  - Semantics of release:
    - All memory operations before release are complete
    - But accesses after release in program order do not have to wait for release
    - Operations which follow release and which need to wait must be protected by an acquire
How to propagate write?

*Invalidated* -- Remove old copies from other caches

*Updated* -- Update old copies in other caches to new values
Initially $A = B = C = 0$

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$A = 1;$</td>
<td>$A = 2;$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>$B = 1;$</td>
<td>$C = 1;$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\text{while } (B \neq 1) {;}$</td>
<td>$\text{while } (B \neq 1) {;}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\text{while } (C \neq 1) {;}$</td>
<td>$\text{while } (C \neq 1) {;}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\text{tmp1} = A;$</td>
<td>$\text{tmp2} = A;$</td>
</tr>
</tbody>
</table>
**Understanding Atomicity - Example 1**

Initially $A = B = C = 0$

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A = 1;$</td>
<td>$A = 2;$</td>
<td>while $(B !\neq 1)$ {};</td>
<td>while $(B !\neq 1)$ {};</td>
</tr>
<tr>
<td>$B = 1;$</td>
<td>$C = 1;$</td>
<td>while $(C !\neq 1)$ {};</td>
<td>while $(C !\neq 1)$ {};</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp1 = A;</td>
<td>tmp2 = A;</td>
</tr>
</tbody>
</table>

Can happen if updates of $A$ reach P3 and P4 in different order

Coherence protocol must serialize writes to same location
  (Writes to same location should be seen in same order by all)
Initially $A = B = 0$

P1

A = 1

while (A != 1) ;

B = 1;

tmp = A

P2

while (B != 1) ;

P3

Write, A, 1

Read, A, 1

Write, B, 1

Read, B, 1

Read, A,

Can happen if read returns new value before all copies see it

Read-others’-write early optimization unsafe
Program Order and Write Atomicity Example

Initially all locations = 0

P1                                   P2
Flag1 = 1;                           Flag2 = 1;

... = Flag2;  0                      ... = Flag1;  

Can happen if read early from write buffer
Program Order and Write Atomicity Example

Initially all locations = 0

P1                                        P2
Flag1 = 1;                                  Flag2 = 1;
A = 1;                                      A = 2;
... = A;                                    ... = A;
... = Flag2;                               ... = Flag1;  
... = Flag2;  0                             ... = Flag1;  


Program Order and Write Atomicity Example

Initially all locations = 0

P1
Flag1 = 1; A = 1; ... = A; ... = Flag2; 0

P2
Flag2 = 1; A = 2; ... = A; 1 ... = Flag1; 0

Can happen if read early from write buffer

“Read-own-write early” optimization can be unsafe
SC Summary

SC limits

Program order relaxation:

Write → Read
Write → Write
Read → Read, Write

Read others’ write early
Read own write early
Unserialized writes to the same location

Alternative

Give up sequential consistency
Use relaxed models
Note: Aggressive Implementations of SC

Can actually do optimizations with SC with some care
    Hardware has been fairly successful
    Limited success with compiler

But not an issue here
    Many current architectures do not give SC
    Compiler optimizations on SC still limited
What is a memory consistency model?

Implicit memory model

Relaxed memory models (system-centric)

Programmer-centric approach for relaxed models

Application to Java

Conclusions
Classification for Relaxed Models

Typically described as system optimizations - system-centric

Optimizations

Program order relaxation:

Write → Read
Write → Write
Read → Read, Write

Read others’ write early
Read own write early

All models provide safety net

All models maintain uniprocessor data and control dependences, write serialization
### Some Current System-Centric Models

<table>
<thead>
<tr>
<th>Relaxation:</th>
<th>W $\rightarrow$ R Order</th>
<th>W $\rightarrow$ W Order</th>
<th>R $\rightarrow$ RW Order</th>
<th>Read Others’ Write Early</th>
<th>Read Own Write Early</th>
<th>Safety Net</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM 370</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>serialization instructions</td>
</tr>
<tr>
<td>TSO</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>RMW</td>
</tr>
<tr>
<td>PC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>RMW</td>
</tr>
<tr>
<td>PSO</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>RMW, STBAR</td>
</tr>
<tr>
<td>WO</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>synchronization</td>
</tr>
<tr>
<td>RCsc</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>release, acquire, nsync, RMW</td>
</tr>
<tr>
<td>RCpc</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>release, acquire, nsync, RMW</td>
</tr>
<tr>
<td>Alpha</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>MB, WMB</td>
</tr>
<tr>
<td>RMO</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>various MEMBARs</td>
</tr>
<tr>
<td>PowerPC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SYNC</td>
</tr>
</tbody>
</table>
System-centric models provide higher performance than SC

BUT 3P criteria

Programmability?
Lost intuitive interface of SC

Portability?
Many different models

Performance?
Can we do better?

Need a higher level of abstraction
What is a memory consistency model?

Implicit memory model - sequential consistency

Relaxed memory models (system-centric)

Programmer-centric approach for relaxed models

Application to Java

Conclusions
Many models give informal software rules for correct results

BUT

Rules are often ambiguous when generally applied
What is a correct result?

Why not

Formalize one notion of correctness – the base model

Relaxed model =
Software rules that give appearance of base model

Which base model? What rules? What if don’t obey rules?
Choose \textit{sequential consistency} as base model

Specify memory model as a contract
  System gives sequential consistency
  IF programmer obeys certain rules

+ Programmability
+ Performance
+ Portability

[Adve and Hill, Gharachorloo, Gupta, and Hennessy]
What Software Rules?

Rules must

Pertain to program behavior on SC system
Enable optimizations without violating SC

Possible rules

Prohibit certain access patterns
Ask for certain information
Use given constructs in prescribed ways

Examples coming up
What if a Program Violates Rules?

What about programs that don’t obey the rules?

Option 1: Provide a system-centric specification
   But this path has pitfalls

Option 2: Avoid system-centric specification
   Only guarantee a read returns value written to its location
Several models proposed

Motivated by previous system-centric optimizations (and more)

This talk

Data-race-free-0 (DRF0) / properly-labeled-1 model

Application to Java
The Data-Race-Free-0 Model: Motivation

Different operations have different semantics

P1
A = 23;
B = 37;
Flag = 1;

P2
while (Flag != 1) {;
... = B;
... = A;

Flag = Synchronization; A, B = Data

Can reorder data operations

Distinguish data and synchronization

Need to
- Characterize data / synchronization
- Prove characterization allows optimizations w/o violating SC
Data-Race-Free-0: Some Definitions

Two operations conflict if

- Access same location
- At least one is a write
(Consider SC executions $\Rightarrow$ global total order)

Two conflicting operations race if
- From different processors
- Execute one after another (consecutively)

P1
Write, A, 23
Write, B, 37
Write, Flag, 1

P2
Read, Flag, 0
Read, Flag, 1
Read, B, ___
Read, A, ___

Races usually “synchronization,” others “data”

Can optimize operations that never race
Data-Race-Free-0 (DRF0) Definition

Data-Race-Free-0 Program

All accesses distinguished as either synchronization or data

All races distinguished as synchronization

(in any SC execution)

Data-Race-Free-0 Model

Guarantees SC to data-race-free-0 programs

(For others, reads return value of some write to the location)
Information required:

*This operation never races* (in any SC execution)

1. Write program assuming SC

2. For every memory operation specified in the program do:

   - Distinguish as *data*
   - *Never races?*
     - yes
     - no
     - don’t know or don’t care

Distinguish as *synchronization*
Programmer’s interface is sequential consistency

Knowledge of races needed even with SC

“Don't-know” option helps
Distinguishing/Labeling Memory Operations

Need to distinguish/label operations at all levels

- High-level language
- Hardware

Compiler must translate language label to hardware label

Tradeoffs at all levels

- Flexibility
- Ease-of-use
- Performance
- Interaction with other level
Language Support for Distinguishing Accesses

Synchronization with special constructs

Support to distinguish individual accesses
**Synchronization with Special Constructs**

Example: `synchronized` in Java

Programmer must ensure races limited to the special constructs

Provided construct may be inappropriate for some races

E.g., producer-consumer with Java

```
P1
A = 23;
P2
while (Flag != 1) {;
B = 37;
Flag = 1;
... = B;
... = A;
```
Option 1: Annotations at statement level

P1
- data = ON
- A = 23;
- {;;}
- B = 37;
- synchronization = ON

P2
- synchronization = ON
- while (Flag != 1)
- data = ON
- ... = B;
- ... = A;
- Flag = 1;

Option 2: Declarations at variable level

synch int: Flag

data int: A, B
Default declarations

To decrease errors
  Make synchronization default

To decrease number of additional labels
  Make data default
Different flavors of load/store
- E.g., ld.acq, st.rel in IA-64

Fences or memory barrier instructions
- Most popular today
  E.g., MB/WMB in Alpha, MEMBAR in SPARC V9
  - For DRF0, insert appropriate fence before/after synch
  - Extra instruction for all synchronization
    Default = synchronization can give bad performance

Special instructions for synchronization
- E.g., Compare&Swap
Interactions Between Language and Hardware

• If hardware uses fences, language should not encourage default of synchronization

• If hardware only distinguishes based on special instructions, language should not distinguish individual operations

• Languages other than Java do not provide explicit support, high-level programmers directly use hardware fences
Can prove that we can
Reorder, overlap data between consecutive synchronization
Make data writes non-atomic

\[
\begin{array}{l}
\text{P1} \\
A = 23; \\
B = 37; \\
\text{Flag} = 1; \\
\text{P2} \\
\text{while (Flag !== 1) } \\
\end{array}
\]

\[
\begin{array}{l}
\text{... = B;} \\
\text{... = A;} \\
\end{array}
\]

⇒ Weak Ordering obeys Data-Race-Free-0
DRF0 also allows more aggressive implementations than WO

- Don't need Data → Read sync, Write sync → Data (like RCsc)
  
  P1
  A = 23;
  B = 37;
  Flag = 1;

  P2
  while (Flag != 1) {;}
  ... = B;
  ... = A;

- Can postpone writes of A, B to Read, Flag, 1

- Can postpone writes of A, B to reads of A, B

- Can exploit last two observations with
  
  Lazy invalidations
  
  Lazy release consistency on software DSMs
Portability: DRF0 Program on System-Centric Models

WO - Direct port

Alpha, RMO - Precede synch write with fence, follow synch read with fence, fence between synch write and read

RCsc - Synchronization = competing

IBM 370, TSO, PC - Replace synch reads with read-modify-writes

PSO - Replace synch reads with read-modify-writes, precede synch write with STBAR

PowerPC - Combination of Alpha/RMO and TSO/PC

RCpc - Combination of RCsc and PC
Data-Race-Free-0 vs. Weak Ordering

Programmability
- DRF0 programmer can assume SC
- WO requires reasoning with out-of-order, non-atomicity

Performance
- DRF0 allows higher performance implementations

Portability
- DRF0 programs correct on more implementations than WO
- DRF0 programs can be run correctly on all system-centric models discussed earlier
Caveats

• Asynchronous programs
• Theoretically possible to distinguish operations better than DRF0 for a given system
Programmer-Centric Models: Summary

The idea

Programmer follows prescribed rules (for behavior on SC)
System gives SC

For programmer

Reason with SC
Enhanced portability

For system designers

More flexibility
In general

• What software rules are useful?
• What further optimizations are possible?

My thesis characterizes

• Useful rules
• Possible optimizations
• Relationship between the above
What is a memory consistency model?

Implicit memory model - sequential consistency

Relaxed memory models (system-centric)

Programmer-centric approach for relaxed models

Application to Java

Conclusions
Defining a Programmer-Centric Java Model

Identify rules for Java programs to get SC behavior
Let’s call such programs **correct** Java programs

Identify minimal guarantees for incorrect programs
Return value written by some write to that location

Reasonableness tests
• Rules should not prohibit common programming idioms
• Confirm all needed systems appear SC to correct programs

Develop system-centric spec
May require mapping from Java rules to rules for hardware

Verify mapping doesn’t inhibit performance for key idioms
Rules for Correct Java Programs

Option 1: No “data races”
(all races from accesses to implement synchronized)
+ Works well on all hardware
- Prohibits common idioms

Option 2: All variables in a data race are declared volatile
+ Any program can be correct by making all volatile
- On Sun, PowerPC, Alpha, IA-64, fences required:
  • After volatile read, monitorenter
  • Before volatile write, monitorexit
  • Between volatile write and volatile read
    Often fences for volatile unnecessary
Motivation

```java
String getFoo() {
    if (foo == null)
        foo = new String(..whatever..);
    return foo;
}
```

Making foo volatile makes this SC, but all foo.X need fences

Option 3:

Provide synch annotations at statement level

For every data race, variable is volatile or statement is synch

Fences like option 2 – but only first read of foo.X needs fence
Rules for Correct Java Programs – Option 4

String getFoo() {
    if (foo == null)
        foo = new String(..whatever..);
    return foo;
}

If access is in races that are always from write to read, then access needs fewer fences

Call such a race WR-race and provide a **WR-race** label

On current machines, fences required:

- **After** WR-race read, volatile read, monitorenter
- **Before** WR-race write, volatile write, monitorexit
- Between volatile write and volatile read

No fence before WR-race read or after WR-race write
Formally define

- Programs for which want SC
- Other idioms we want “working correctly”
- Reasonable behavior for other programs

Develop system-centric constraints for above \textit{and no more}

Follow previous “reasonableness tests”

Use systematic framework, lots of gotchas - another talk!

(e.g., Adve and Gharachorloo theses)
Conclusions

Sequential consistency limits performance optimizations

System-centric relaxed memory models harder to program

Programmer-centric approach for relaxed models
  Software obeys rules, system gives SC

Application to Java
  Can develop software rules for SC for idioms of interest
  Easier for programmers than system-centric specification