Memory Consistency Models

Some material borrowed from Sarita Adve's (UIUC) tutorial on memory consistency models.

Outline

- Need for memory consistency models
- Sequential consistency model
- Relaxed memory models
- Memory coherence
- Conclusions

Uniprocessor execution

- Processors reorder operations to improve performance
- Constraint on reordering: must respect dependences
  - data dependences must be respected: loads/stores to a given memory address must be executed in program order
  - control dependences must be respected
- In particular,
  - stores to different memory locations can be performed out of program order

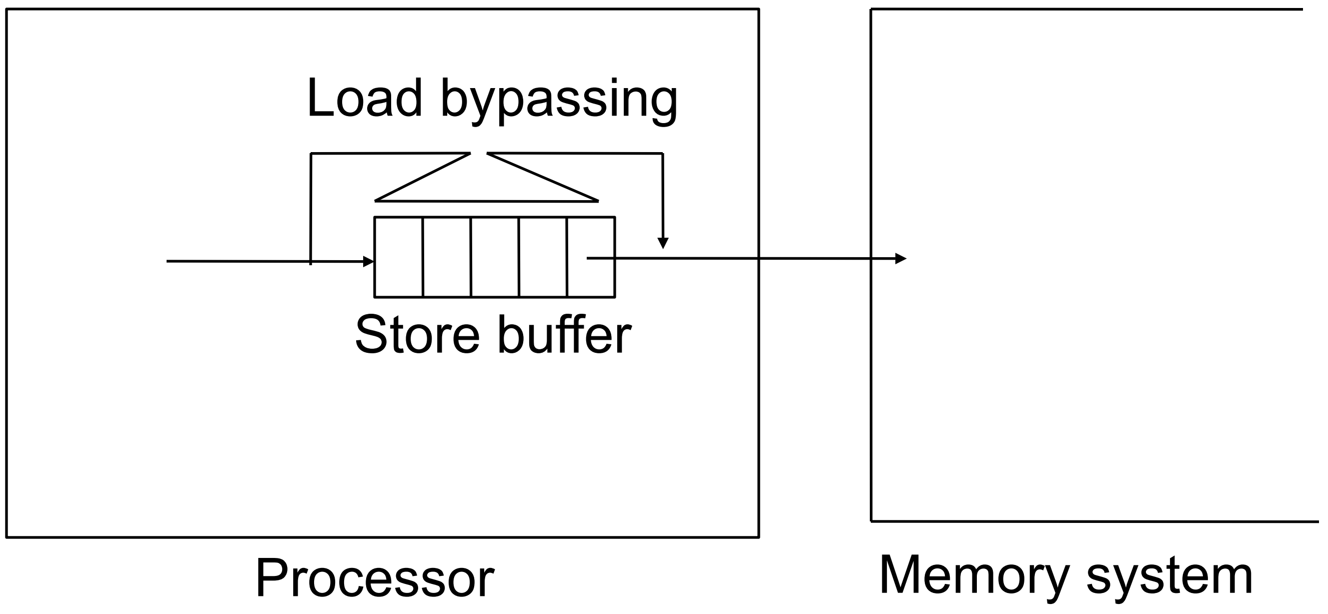
```
store v1, data                     store b1, flag
store b1, flag        <->          store v1, data
```

  - loads to different memory locations can be performed out of program order

```
load flag, r1                      load data,r2
load data, r2         <->          load flag, r1
```

  - load and store to different memory locations can be performed out of program order

Example of hardware reordering



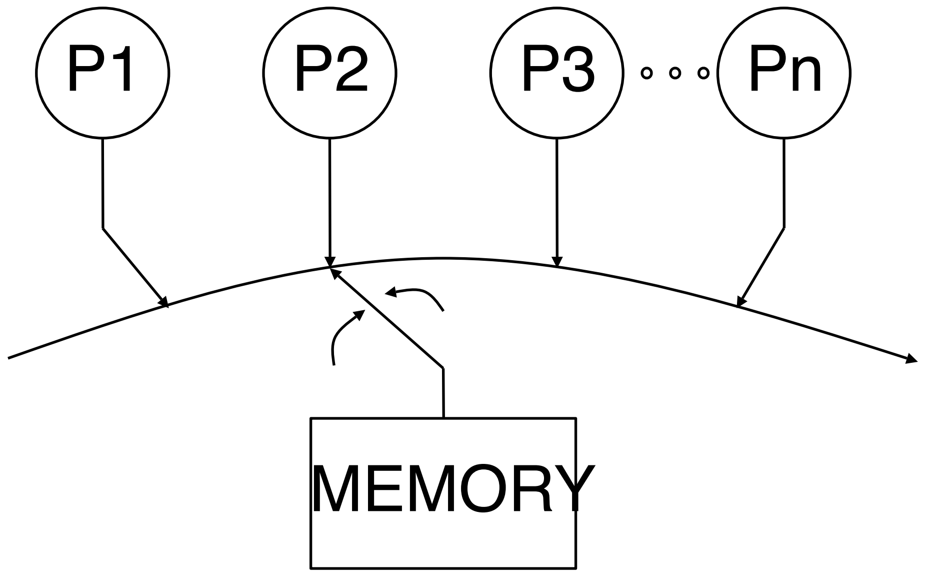
- Store buffer holds store operations that need to be sent to memory
- Loads are higher priority operations than stores since their results are needed to keep processor busy, so they bypass the store buffer
- Load address is checked against addresses in store buffer, so store buffer satisfies load if there is an address match
- Result: load can bypass stores to other addresses

Problem with reorderings

- Reorderings can be performed either by the compiler or by the hardware at runtime
  - static and dynamic instruction reordering

- Problem: uniprocessor operation reordering constrained only by dependences can result in counter-intuitive program behavior in shared-memory multiprocessors.

Simple shared-memory machine model



- All shared-memory locations are stored in global memory.
- Any one processor at a time can grab memory and perform a load or store to a shared-memory location.
- Intuitively, memory operations from the different processors appear to be interleaved in some order at the memory.

Example (I)

Code:

Initially A = Flag = 0

P1

```
A = 23;
Flag = 1;
```

P2

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

Idea:

- P1 writes data into A and sets Flag to tell P2 that data value can be read from A.
- P2 waits till Flag is set and then reads data from A.

Execution Sequence for (I)

Code:

Initially A = Flag = 0

```
P1                P2
A = 23;           while (Flag != 1) {;}
Flag = 1;         ... = A;
```

Possible execution sequence on each processor:

```
P1                P2
Write, A, 23      Read, Flag, 0
Write, Flag, 1    Read, Flag, 1
                  Read, A, ?
```

Problem: If the two writes on processor P1 can be reordered, it is possible for processor P2 to read 0 from variable A.

Example 2

Code: (like Dekker's algorithm)

Initially Flag1 = Flag2 = 0

```
P1                          P2
Flag1 = 1;                  Flag2 = 1;
If (Flag2 == 0)              If (Flag1 == 0)
   critical section            critical section
```

Possible execution sequence on each processor:

```
P1                          P2
Write, Flag1, 1             Write, Flag2, 1
Read, Flag2, 0              Read, Flag1, ??
```

Execution sequence for (II)

Code: (like Dekker's algorithm)

Initially Flag1 = Flag2 = 0

```
P1                          P2
Flag1 = 1;                  Flag2 = 1;
If (Flag2 == 0)              If (Flag1 == 0)
   critical section             critical section
```

Possible execution sequence on each processor:

$$\begin{tikzcd}
\text{P1} & \text{P2} \\
\text{Write, Flag1, 1} & \text{Write, Flag2, 1} \\
\text{Read, Flag2, 0} \arrow[ur] & \text{Read, Flag1, ??}
\end{tikzcd}$$

Most people would say that P2 will read 1 as the value of Flag1.
Since P1 reads 0 as the value of Flag2, P1's read of Flag2 must happen before P2 writes to Flag2. Intuitively, we would expect P1's write of Flag to happen before P2's read of Flag1.

However, this is true only if reads and writes on the same processor to different locations are not reordered by the compiler or the hardware.
Unfortunately, this is very common on most processors (store-buffers with load-bypassing).

Lessons

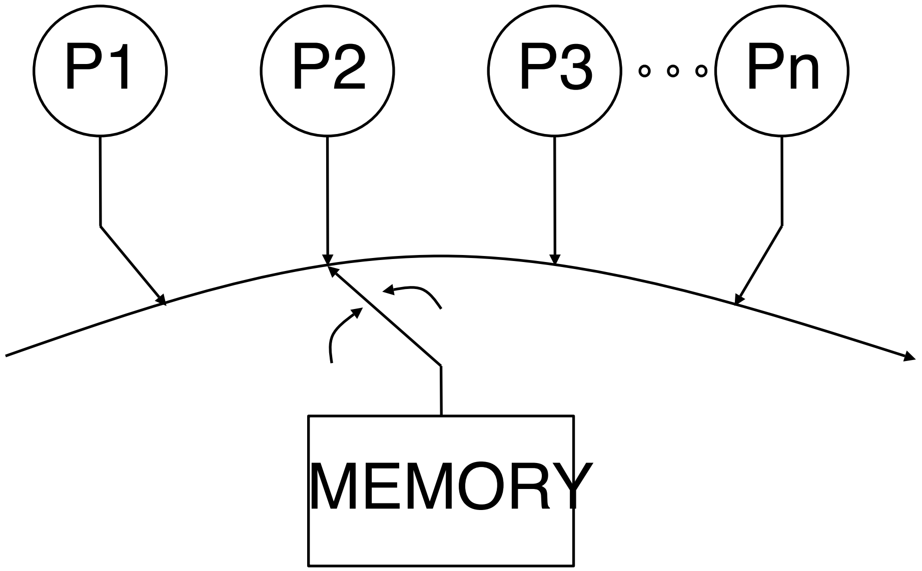
- Uniprocessors can reorder instructions subject only to control and data dependence constraints
- These constraints are not sufficient in shared-memory multiprocessor context
  - simple parallel programs may produce counter-intuitive results
- Question: what constraints must we put on uniprocessor instruction reordering so that
  - shared-memory programming is intuitive
  - but we do not lost uniprocessor performance?
- Many answers to this question
  - answer is called memory consistency model supported by the processor

Consistency models

- Consistency models are not about memory operations from different processors.
- Consistency models are not about dependent memory operations in a single processor’s instruction stream (these are respected even by processors that reorder instructions).
- Consistency models are all about ordering constraints on 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 intuitively reasonable results.

Simple Memory Consistency Model

- Sequential consistency (SC) [Lamport]
  - result of execution is as if memory operations of each process are executed in program order



Program Order

Initially X = 2

```
P1                  P2
…..                 …..
r0=Read(X)          r1=Read(X)
r0=r0+1             r1=r1+1
Write(r0,X)         Write(r1,X)
…..                 ……
```

Possible execution sequences:

```
P1:r0=Read(X)       P2:r1=Read(X)
P2:r1=Read(X)       P2:r1=r1+1
P1:r0=r0+1          P2:Write(r1,X)
P1:Write(r0,X)      P1:r0=Read(X)
P2:r1=r1+1          P1:r0=r0+1
P2:Write(r1,X)      P1:Write(r0,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;
```

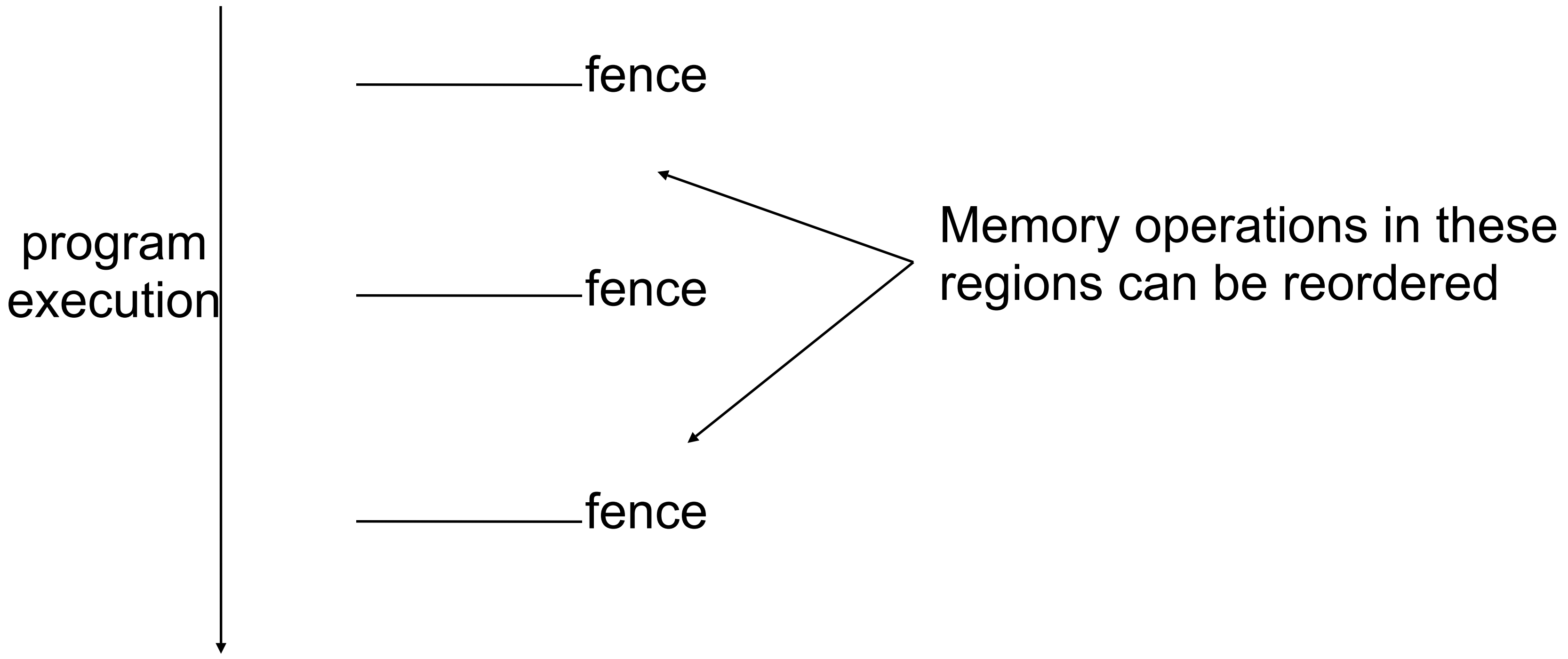
Sequential Consistency

- SC constrains all memory operations:
    - Write → Read
    - Write → Write
    - Read → Read, Write
- Simple model for reasoning about parallel programs
    - You can verify that the examples considered earlier work correctly under sequential consistency.
- However, this simplicity comes at the cost of uniprocessor performance.
- Question: how do we reconcile sequential consistency model with the demands of performance?

Relaxed consistency model: Weak ordering

- Introduce concept of a fence operation:
  - all memory operations before fence in program order must complete before fence is executed
  - all memory operations after fence in program order must wait for fence to complete
  - fences are performed in program order
- Implementation of fence:
  - processor has counter that is incremented when memory op is issued, and decremented when memory op is completed
- Example: PowerPC has SYNC instruction
- Language constructs:
  - OpenMP: flush
  - All synchronization operations like lock and unlock act like a fence

Weak ordering picture



Example (I) revisited

Code:
Initially A = Flag = 0

```
P1                                P2
A = 23;
flush;                            while (Flag != 1) {;}
Flag = 1;                         ... = A;
```

Execution:

- P1 writes data into A
- Flush waits till write to A is completed
- P1 then writes data to Flag
- Therefore, if P2 sees Flag = 1, it is guaranteed that it will read the correct value of A even if memory operations in P1 before flush and memory operations after flush are reordered by the hardware or compiler.

Example II revisited

```
Flag1 = 1;
flush;
if (Flag2 == 0)
   //Critical
```

```
Flag2 = 1;
flush;
if (Flag1 == 0)
   //Critical
```

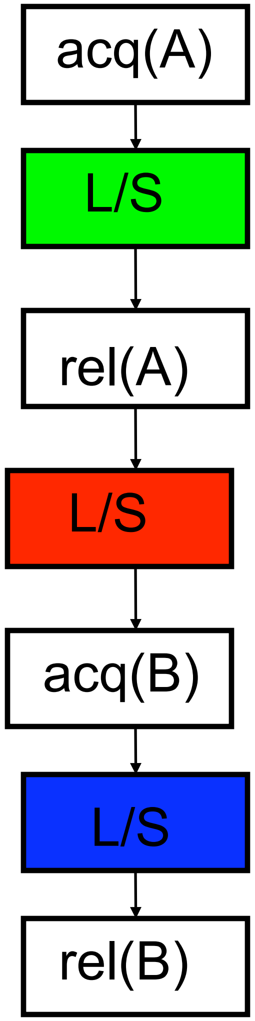
Flushes ensure that reads do not occur before writes

Now can guarantee that both processors will not enter critical section

Another relaxed 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 must complete before release
- However,
  - 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
  - acquire does not wait for accesses preceding it

Example



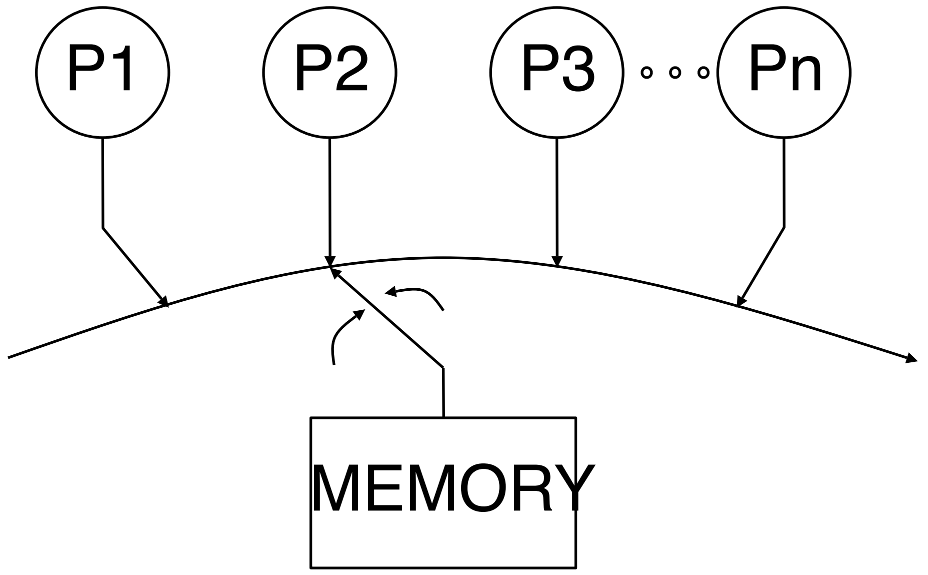
Which operations can be overlapped?

Comments

- In the literature, there are a large number of other consistency models
  - processor consistency
  - Location consistency
  - total store order (TSO)
  - ….
- It is important to remember that all of these are concerned with reordering of independent memory operations within a processor.
- Easy to come up with shared-memory programs that behave differently for each consistency model.
- In practice, weak consistency/release consistency seem to be winning.

Memory coherence

Memory system



• In practice, having a single global shared memory limits performance.
• For good performance, caching is necessary even in uniprocessors.
• Caching introduces new problem in multiprocessor context: memory coherence.

Cache coherence problem

- Shared-memory variables like Flag1 and Flag2 need to be visible to all processors.
- However, if a processor caches such variables in its own cache, updates to the cached version may not be visible to other processors.
- In effect, a single variable at the program level may end up getting “de-cohered” into several ghost locations at the hardware level.
- Coherent memory system: provides illusion that each memory location at the program level is implemented as a single memory location at the architectural level

Understanding Coherence: Example 1

Initially A = B = C = 0

```
P1          P2          P3                          P4
A = 1;      A = 2;      while (B != 1) {;}          while (B != 1) {;}
B = 1;      C = 1;      while (C != 1) {;}          while (C != 1) {;}
                        tmp1 = A;     1             tmp2 = A;     2
```

- 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

Understanding Coherence: Example 2

Initially A = B = 0

```
P1                P2                   P3
A = 1     while (A != 1) ;     while (B != 1) ;
          B = 1;               tmp = A
```

```
P1                P2                   P3
Write, A, 1
                  Read, A, 1
                  Write, B, 1
                                       Read, B, 1
                                       Read, A,   0
```

- Can happen if read returns new value before all copies see it
- All copies must be updated before any processor can access new value.

Write atomicity

These two properties

– writes to same location must be seen in the same order by all processors

– all copies must be updated before any processor can access new value

are known as write atomicity.

Cache Coherence Protocols

- How to find cached copies?
  - Directory-based schemes: look up a directory that keeps track of all cached copies
  - Snoopy-cache schemes: works for bus-based systems
- How to propagate write?
  - *Invalidate* -- Remove old copies from other caches
  - *Update* -- Update old copies in other caches to new values

Summary

- Two problems: memory consistency and memory coherence
- Memory consistency model
  - what instructions is compiler or hardware allowed to reorder?
  - nothing really to do with memory operations from different processors
  - sequential consistency: perform memory operations in program order
  - relaxed consistency models: all of them rely on some notion of a fence operation that demarcates regions within which reordering is permissible
- Memory coherence
  - Preserve the illusion that there is a single logical memory location corresponding to each program variable even though there may be lots of physical memory locations where the variable is stored

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

Outline

- 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 $\rightarrow$ Read
    - Write $\rightarrow$ Write
    - Read $\rightarrow$ 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

Relaxation:	W →R Order	W →W Order	R →RW Order	Read Others' Write Early	Read Own Write Early	Safety Net
IBM 370	✓					serialization instructions
TSO	✓				✓	RMW
PC	✓			✓	✓	RMW
PSO	✓	✓			✓	RMW, STBAR
WO	✓	✓	✓		✓	synchronization
RCsc	✓	✓	✓		✓	release, acquire, nsync, RMW
RCpc	✓	✓	✓	✓	✓	release, acquire, nsync, RMW
Alpha	✓	✓	✓		✓	MB, WMB
RMO	✓	✓	✓		✓	various MEMBARs
PowerPC	✓	✓	✓	✓	✓	SYNC

System-Centric Models: Assessment

- 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

Outline

- 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

An Alternate Programmer-Centric View

- 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?

Which Base Model?

- Choose *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
  -

Programmer-Centric Models

- 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                       P2
A =  23;                 while (Flag != 1)  {;}
B =  37;                  … = B;
Flag = 1;                 … = 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

Data-Race-Free-0: Some Definitions (Cont.)

- (Consider SC executions ⇒ global total order)
- Two conflicting operations race if
  - From different processors
  - Execute one after another (consecutively)

- P1 P2
- Write, A, 23
- Write, B, 37
- Read, Flag, 0
- Write, Flag, 1
- Read, Flag, 1
- Read, B, ___ Read, A, ___
- Races usually "synchronization," others "data"
- Can optimize operations that *never race*

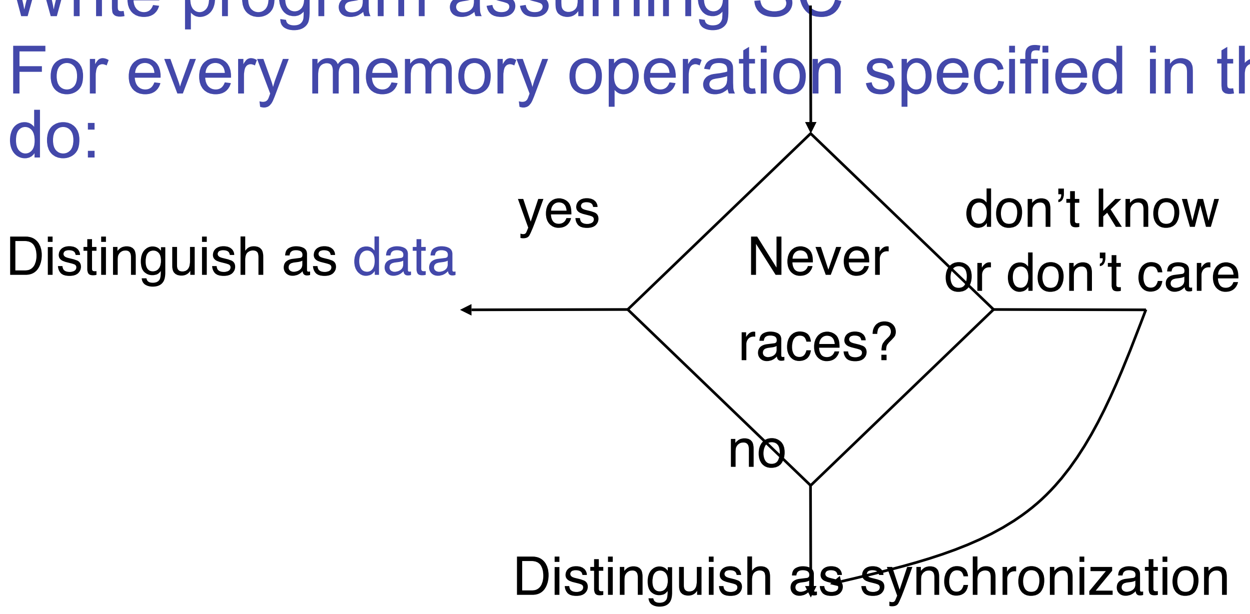
Data-Race-Free-0 (DRF0) Definition

- 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)

Programming with Data-Race-Free-0

- 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:



Programming With Data-Race-Free-0

- 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                              P2
• A =  23;                        while (Flag != 1)  {;}
• B =  37;                        … = B;
• Flag = 1;                        … = A;
```

Distinguishing Individual Memory Operations

- Option 1: Annotations at statement level
  - P1 P2
  - data = ON synchronization = ON
    - A = 23; while (Flag != 1) {;}
    - B = 37; data = ON
  - synchronization = ON … = B;
    - Flag = 1; … = A;
- Option 2: Declarations at variable level
  - synch int: Flag
  - data int: A, B

Distinguishing Individual Memory Operations (Cont.)

- Default declarations
  - To decrease errors
    - Make synchronization default
  - To decrease number of additional labels Make data default

Distinguishing/Labeling Operations for Hardware

- 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

Performance: Data-Race-Free-0 Implementations

- Can prove that we can
  - Reorder, overlap data between consecutive synchronization
  - Make data writes non-atomic

```
• P1                    P2
• A =  23;              while (Flag != 1)  {;}
• B =  37;                   … = B;
• Flag = 1;                  … = A;
```

- ⇒ Weak Ordering obeys Data-Race-Free-0

Data-Race-Free-0 Implementations (Cont.)

- DRF0 also allows more aggressive implementations than WO
- Don't need Data → Read sync, Write sync → Data (like RCsc)

```
• P1                          P2
• A =  23;                    while (Flag != 1)  {;}
• B =  37;                  … = B;
• Flag = 1;                 … = 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

Data-Race-Free-0 vs. Weak Ordering (Cont.)

- 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

Programmer-Centric Models: A Systematic Approach

- In general
  - What software rules are useful?
  - What further optimizations are possible?
- My thesis characterizes
  - Useful rules
  - Possible optimizations
  - Relationship between the above

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