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

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

Memory Consistency Model: Definition

Memory consistency model

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

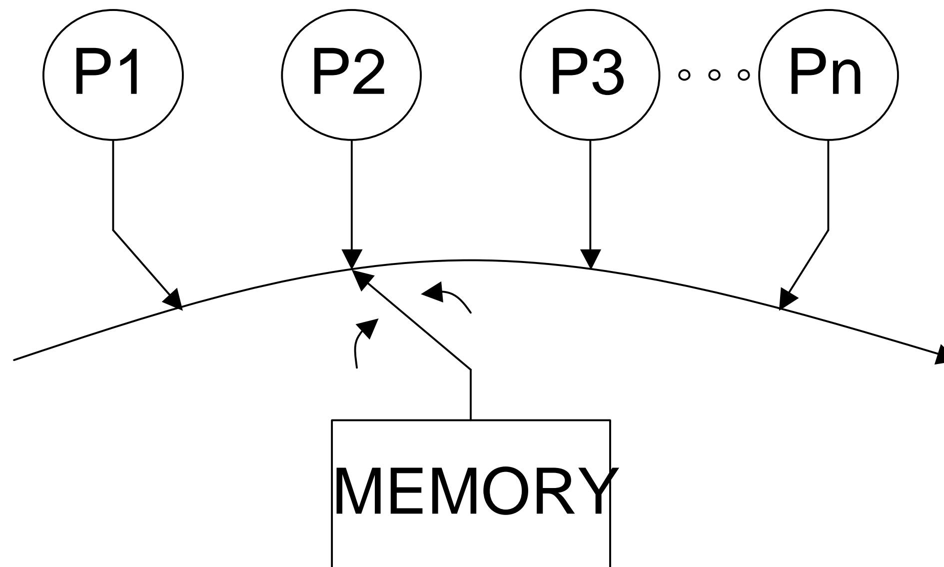
Affects ease-of-programming and performance

Implicit Memory Model

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



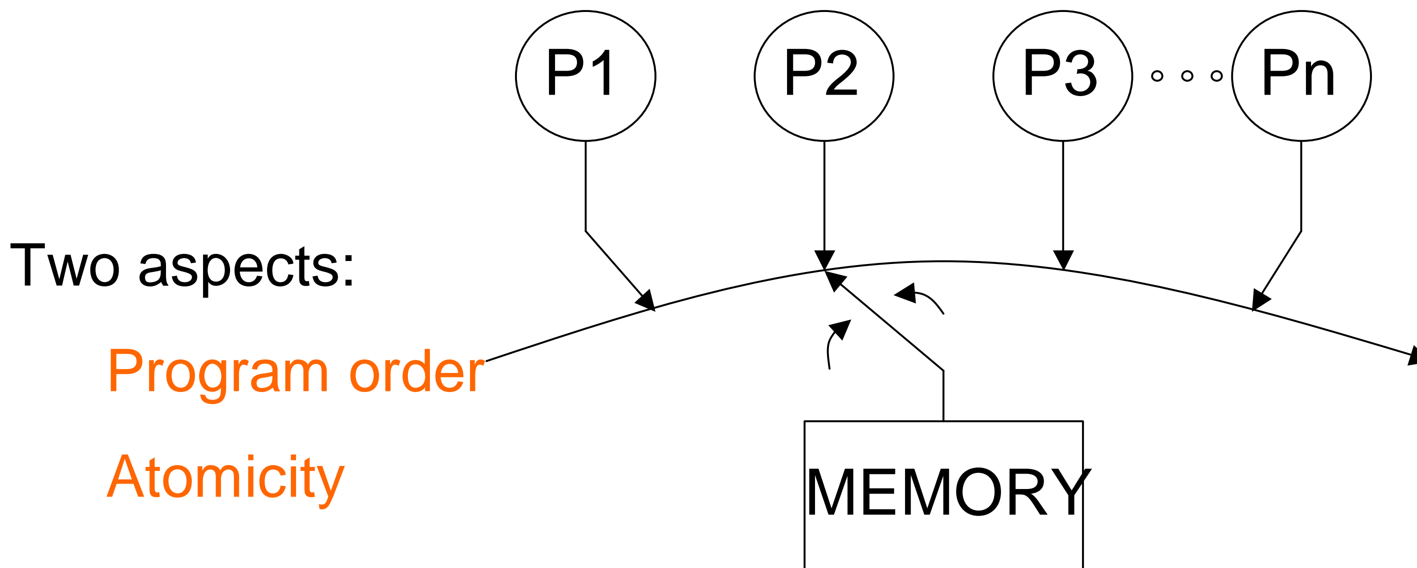
No caches, no write buffers

Implicit Memory Model

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

Understanding Program Order – Example 1

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, ___

Understanding Program Order – Example 1

Initially Flag1 = Flag2 = 0

P1

Flag1 = 1

if (Flag2 == 0)

critical section

P2

Flag2 = 1

if (Flag1 == 0)

critical section

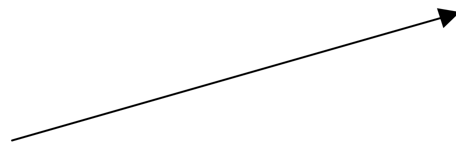
Execution:

P1

(Operation, Location, Value)

Write, Flag1, 1

Read, Flag2, 0



P2

(Operation, Location, Value)

Write, Flag2, 1

Read, Flag1, ____

Understanding Program Order – Example 1

Initially Flag1 = Flag2 = 0

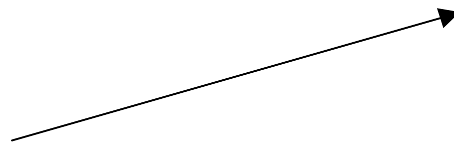
P1
Flag1 = 1
if (Flag2 == 0)
    critical section

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

Execution:

P1
(Operation, Location, Value)
Write, Flag1, 1

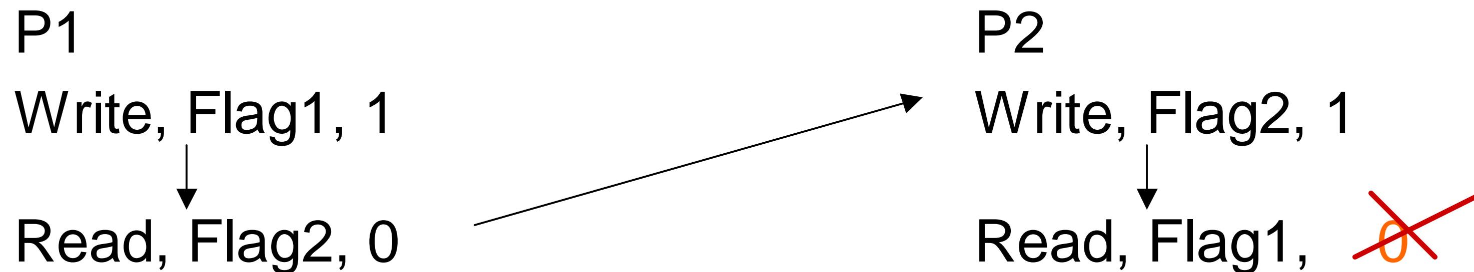
Read, Flag2, 0



P2
(Operation, Location, Value)
Write, Flag2, 1

Read, Flag1, ~~0~~

Understanding Program Order – Example 1



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

Understanding Program Order - Example 2

Initially A = Flag = 0

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

P1	P2
Write, A, 23	Read, Flag, 0
Write, Flag, 1	
	Read, Flag, 1
	Read, A, ____

Understanding Program Order - Example 2

Initially A = 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, ~~0~~

Understanding Program Order - Example 2

Initially A = Flag = 0

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

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

Can happen if

Overlap or reorder writes or reads in hardware or compiler

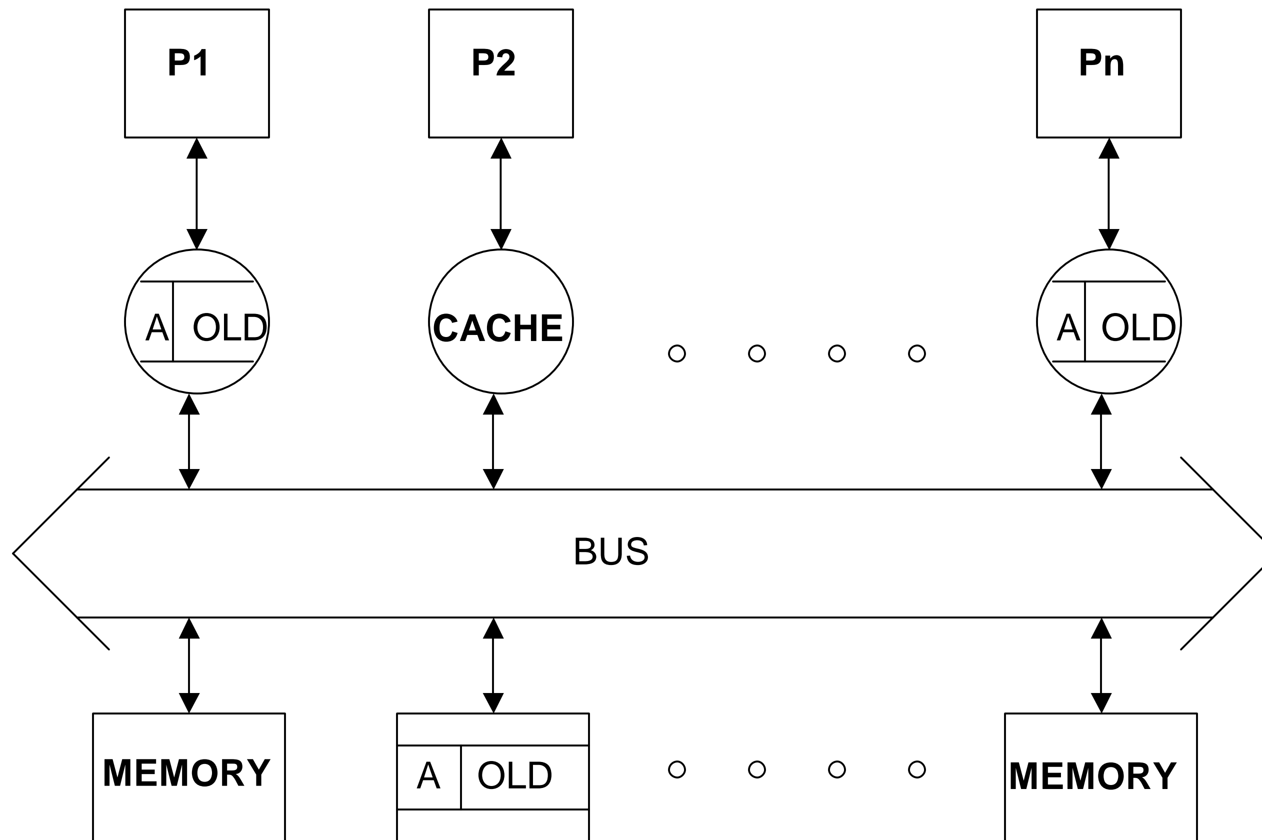
On AlphaServer, T3D/T3E

Understanding Program Order: Summary

SC limits program order relaxation:

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

Understanding Atomicity – Caches 101



A mechanism needed to propagate a write to other copies

⇒ Cache coherence protocol

Cache Coherence Protocols

How to propagate write?

Invalidate -- Remove old copies from other caches

Update -- Update old copies in other caches to new values

Understanding Atomicity - 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;	tmp2 = A;

Understanding Atomicity - 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 Atomicity - 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

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; 0

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; 0	... = Flag1; ~~0~~

Program Order and Write Atomicity Example

Initially all locations = 0

P1

Flag1 = 1;

A = 1;

... = A; 1

... = Flag2; 0

P2

Flag2 = 1;

A = 2;

... = A; 2

... = 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

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 $\Rightarrow$ 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)

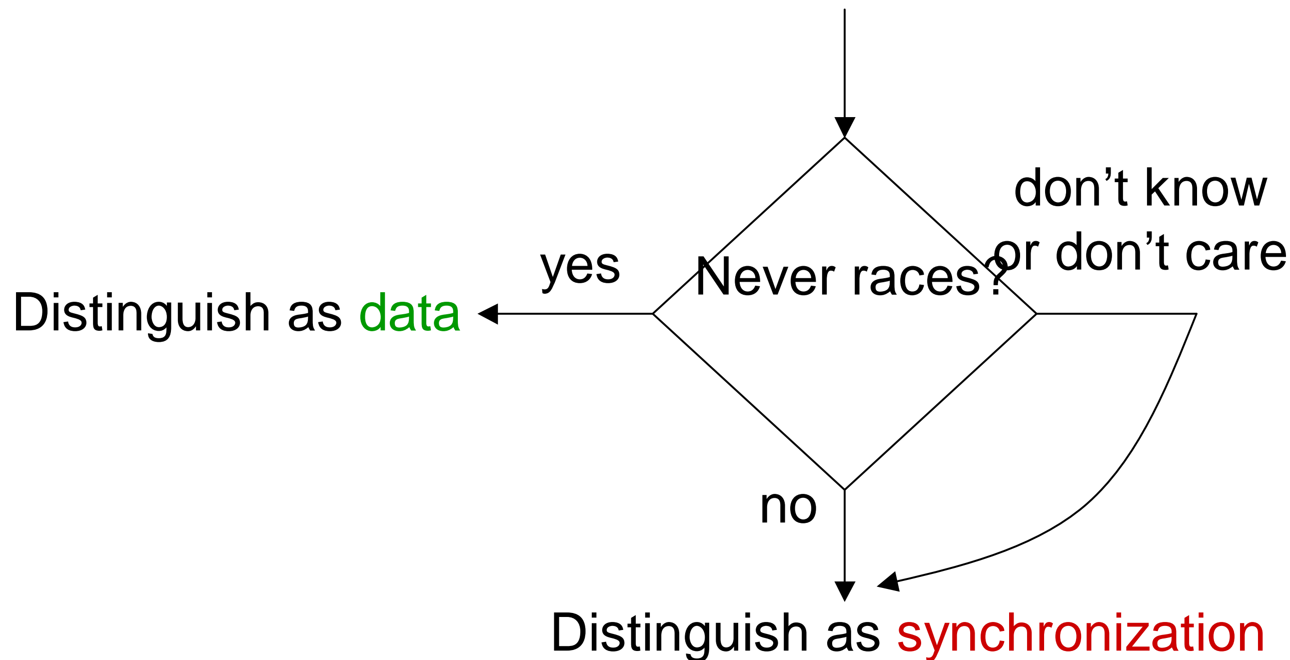
Programming with Data-Race-Free-0

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;

$\Rightarrow$ 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

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

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

Rules for Correct Programs – Option 3

Motivation

```
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

If Insist on System-Centric Route …

Formally define

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

Develop system-centric constraints for above *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