Designing Memory Consistency Models for

Shared-Memory Multiprocessors

Sarita V. Adve

Computer Sciences Department

University of Wisconsin-Madison
The Big Picture

Assumptions

Parallel processing important for future
Shared-memory is desirable model

Challenge

To build shared-memory systems that
give high performance
are easy to program
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
The Uniprocessor Model

Program text defines total order = program order

Uniprocessor Model

Memory operations appear to execute one-at-a-time in program order
⇒ Read returns value of last write

BUT uniprocessor hardware
overlap, reorder operations (e.g., write buffers)

Model maintained as long as
maintain control and data dependences

⇒ Easy to use + high performance
Implicit Multiprocessor Model

Sequential Consistency (SC) [Lamport 79]

Each process executes in program order
All operations in some sequential order
(i.e., atomic, one-at-a-time)

Programmers’ view (no buffers or caches)
**SC: Implications**

In practice [Scheurich,Dubois87]

Execute in program order and atomically

But optimizations becoming more important
Many alternative models

Allow hardware optimizations (*hardware-centric*)

BUT 3P criteria

- **Programmability**
- **Portability**
- +/- **Performance**

No common framework

This work gives a *programmer-centric* framework

Enhances 3Ps of many current models

+ **Programmability**
+ **Portability**
+ **Performance**

Exposes design space for future
Thesis Contributions

(I) Programmer-centric view of problem [ISCA90, TPDS93]
Model = Contract
    System gives sequential consistency
    If programmer gives information

(II) Four programmer-centric models [ISCA90, TPDS93, JPDC92]
Enhance 3Ps of many current models

(III) The design space of memory models
Formalize and simplify design process
Expose unexploited potential, new models
Characterize the design space

(IV) Debugging with relaxed models [ISCA91]
Demonstrate use of SC techniques
Outline

Background: Hardware-Centric Models

Programmer-Centric Approach and Four Models

The Design Space

Conclusions
Weak Ordering (WO) [Dubois et al. 86]

Motivation

Ordering important only at synchronization
Can reorder data between synchronization
Distinguish synchronization from data
**Weak Ordering (WO): Definition**

Exposes non-atomicity of memory operations

Op *performs with respect to* processor $P_i$ when

- Op = Write: $P_i$ can read value of Op
- Op = Read: $P_i$ cannot change value of Op

Op *globally performs* when

- Op = Write: Op performed w.r.t all
- Op = Read: Op and write whose value Op returns performed w.r.t all

Definition (“previous” is by program order)

- Synchronization is SC
- Before issuing synchronization, globally perform previous data
- Before issuing data, globally perform previous synchronization
Hardware-Centric Examples (Cont.)

Total Store Ordering (TSO) [SUN 91]
Reads can pass writes +
Writes partially non-atomic (can read early from own write buffer)

Processor consistency (PC) [Gharachorloo 90]
Optimizations of TSO +
Writes fully non-atomic

Release consistency (RCpc) [Gharachorloo 90]
Optimizations of WO +
Can reorder data separated by some synchronization +
Synchronization is PC
**Hardware-Centric Models: Assessment**

Sequential Consistency

??? Weak Ordering

Total Store Ordering

Processor Consistency

Release Consistency (RCsc)

Partial Store Ordering

Release Consistency (RCpc)

BUT,

- Programmability (Lost intuitive interface of SC)
- Portability (Many different models)

+/− Performance (Can do better)

strict better hardware

weaker hardware

+ Better performance than SC [GAG91,92, ZuB92]
Outline

Background: Hardware-Centric Models

Programmer-Centric Approach and Four Models
  The Programmer-Centric Approach
  One Model In Detail
  Overview of Three Models

The Design Space

Conclusions
A Programmer-Centric Approach

Motivation

Many models give informal software rules assuming informal notion of correctness

Why not

Formalize one notion of correctness (base model)
Specify other models as software rules that give appearance of base model

+ Programmability (if base model simple)
+ Portability (programmers see one model)
+ Performance (no unnecessary constraints)

Which base model?
Sequential Consistency Normal Form (SCNF)

Contribution I

*Specify memory model as a contract*

*System gives sequential consistency*

*IF programmer provides some information*

(Sequential Consistency Normal Form)
Four SCNF Models

Contribution II

Four SCNF models (exploit increasing information)

Data-race-free-0 [Adve & Hill 90]

Data-race-free-1 [Adve & Hill 92]

PLpc1

\{ PLpc2 \} (based on joint work [GAG92])

Enhance 3Ps of many current models
Outline

Background: Hardware-Centric Models

**Programmer-Centric Approach and Four Models**

The Programmer-Centric Approach

*One Model In Detail*

*Motivation*

*Definition*

*Programming With Data-Race-Free-0*

*Implementing Data-Race-Free-0*

*Comparison With Weak Ordering*

Overview of Three Models

The Design Space

Conclusions
Clarification: Static vs. Dynamic Issues

Programmer    High-level language program

Compile-time system

Low-level language program

Runtime system

Input 1

Execution E1.1 E 1.2 E 1.j

Input 2

Execution E 2.1 E 2.2

Input i

Execution E i.1 E i.2

Models specify constraints on execution

Models require distinguishing dynamic operations

Programmer must make distinctions in static program
Data-Race-Free-0: Motivation

Different operations have different semantics

\[ P1 \]
\[ A = 100; \]
\[ B = 200; \]
\[ Valid = 1; \]

\[ P2 \]
\[ while (Valid \neq 1) \{ ; \} \]
\[ ... = B; \]
\[ ... = A; \]

Valid = Synchronization; \( A, B \) = Data

Can reorder data operations

Distinguish data and synchronization

Need to

- Characterize data / synchronization
- Prove characterization allows optimizations without violating SC
Data-Race-Free-0: Definitions

(Consider SC executions ⇒ global total order)

Two operations conflict if

access same location
at least one is a write

Two conflicting operations race if

from different processors,
execute one after another (consecutively)

P1
Write, A, 100
Write, B, 200
Write, valid, 1

P2
Read, valid, 0
Read, valid, 1
Read, B, _
Read, A, _

Races usually “synchronization,” others “data”

Can optimize operations that never race
Data-Race-Free-0: Definition

Information required: This operation never races
(in any SC execution)

1. Write program assuming sequential consistency
2. For every memory operation specified in the program do:

   START
   
   distinguish as data
   yes
   never races?
   no
   distinguish as synchronization
   
   don’t know or don’t care

Data-Race-Free-0 Program

All races distinguished as synchronization
(in any SC execution)

Data-Race-Free-0 Model

Guarantees SC to data-race-free-0 programs
Programming With Data-Race-Free-0

SC interface

Knowledge of races needed even with SC

“Don’t-know” option helps

```
P1          P2
A = ...;    data                      while (Valid != 1) {}   synch
B = ...;    data                      ... = B;               data
Valid = 1;  synch                      ... = A;               data
```

To distinguish at high-level, can use annotations

```
P1                      P2
data = ON               synchronization = ON
A = ...;               while (Valid != 1) {}  
B = ...;               data = ON
synchronization = ON    ... = A;           
Valid = 1;              ... = B;           
```

For hardware, can use different reads/writes
**DRF0: Implementations**

Proved that we can

Reorder, overlap data between consecutive synchronization

Make data writes non-atomic

\[
P1\quad \text{while (valid \neq 1) {;}}\quad \text{synch}
\]

\[
P2\quad \text{valid} = \text{valid} + 1;\quad \text{valid} = \text{valid} - 1;
\]

⇒ Weak Ordering obeys DRF0

DRF0 also allows more aggressive hardware

Can postpone writes of \( A, B \) to

Read, \( \text{valid}, 1 \) or to reads of \( A, B \)

[Adve&Hill 90, 93]
Data-Race-Free-0 vs. Weak Ordering

Programmability

- DRF0 programmer can assume SC
- WO requires reasoning with *performs with respect to*, out-of-order execution

Portability

- DRF0 programs correct on more implementations
  - (thesis gives four other than WO)

Performance

- DRF0 allows higher performance implementations

Caveats

- Asynchronous programs
- Theoretically possible to distinguish operations better than DRF0
Other Models: Definitions

1. Write program assuming sequential consistency
2. For every memory operation specified in the program do:

\[ \text{START} \]

- distinguish as \textit{data} yes never races? no distinguish as \textit{synchronization}

- distinguish as \textit{unpairable} yes never orders data? no distinguish as \textit{pairable}

- distinguish as \textit{loop} yes never non-loop? no distinguish as \textit{non-loop}

\textit{PLpc2} further distinguishes \textit{loop} and \textit{data} as atomic/non-atomic
Four SCNF Models: Summary

- **PLpc2**
- **PLpc1**
  - **DATA-RACE-FREE-1**
    - **DATA-RACE-FREE-0**
      - Weak Ordering
      - Lazy Release Consistency
      - Specifications from Thesis
    - Release Consistency (RCsc)
      - Specifications from Thesis
    - Total Store Ordering
    - Partial Store Ordering
      - Alpha
      - Specification from Thesis
  - Processor Consistency
    - Release Consistency (RCpc)
      - IBM 370
      - Specification from Thesis

Programmability, portability, performance for all
Four SCNF Models: Summary

Programmability, portability, performance for all

But can we do better?
Outline

Background: Hardware-Centric Models

Programmer-Centric Approach and Four Models

The Design Space

Conclusions
The Design Space: Motivation

P1 P2 P3

if (Pred) {
    A = 100;
    B = 200;
    F1 = 1;
}

if (not Pred) {
    A = 300;
    B = 400;
    F2 = 1;
}

while ( (F1 != 1) && (F2 != 1) ) {;

A = 100;  ... = B;
B = 200;  ... = A;
F1 = 1;   F2 = 1;
}

Intuition: Reads on F1, F2 can be in parallel

But not allowed by previous models

Are parallel reads of F1, F2 really safe?

How to design model to allow this optimization?

Better goal

When is any optimization safe?

How to design model to allow any optimization?
The Design Space: The Key

Memory Model

Obtain information from programmer
to allow optimizations without violating SC

Previous models

Obtain information for some optimizations
Mostly ad hoc, complex analysis

Can we formalize and simplify the design process?

Key: *Mapping between optimizations and information*
The Design Space

Contribution III

Formalize and simplify design process

What optimizations possible?
What information will make optimization safe?

Expose unexploited potential in design space

New memory models

Characterize the design space
Outline

Background: Hardware-Centric Models

Programmer-Centric Approach and Four Models

The Design Space

Analysis for Mapping

Mapping between Optimizations and Information

Application of mapping: a new memory model

Characterization of design space

Conclusions
Analysis for Designing Memory Models

Program

P1

if (Pred) {
    A = 100;
    B = 200;
    F1 = 1;
}

P2

if (not Pred) {
    A = 300;
    B = 400;
    F2 = 1;
}

while ( (F1 != 1) && (F2 != 1) ) {;

P3

    ... = B;
    ... = A;

    F1 = 1;
    F2 = 1;

}

Execution

P2

Write, A, 300
Write, B, 400
Write, F2, 1

P3

Read, F1, 0
Read, F2, 1
Read, A, _
Read, B, _
Analysis for Designing Memory Models (Cont.)

P2
Write, A, 300
Write, B, 400
Write, F2, 1

P3
Read, F1, 0
Read, F2, 1
Read, B, _
Read, A, _

Ordering Path

Path between conflicting operations using program and conflict orders

(Conflict Order from X to Y if X, Y conflict and X executes before Y)

For SC execution

If there is an ordering path from X to Y, then execute X before Y

⇒ Execute ordering paths safely

(Others have derived different forms)
Analysis for Designing Memory Models (Cont.)

P2
Write, A, 300
Write, B, 400
Write, F2, 1

P3
Read, F1, 0
Read, F2, 1
Read, B, _
Read, A, _

Easy way to get SC

Enforce program order and atomicity on ordering paths

Key Observation

Not all paths need be executed safely

Necessary paths = critical paths †

†Term “critical” inspired by [Shasha&Snir88]
**Example Non-Critical Paths**

- **P2**
  - Write, \( A \), 300
  - Write, \( B \), 400
  - Write, \( F_2 \), 1

- **P3**
  - Read, \( F_1 \), 0
  - Read, \( F_2 \), 1
  - Read, \( B \), _
  - Read, \( A \), _

Between one pair of conflicting operations, only one path is critical

\[ \Rightarrow \text{Operations on } A, B \text{ can be in parallel} \]

Other observations imply other non-critical paths

- Unessential operations
- Self-ordered operations
Mapping Between Optimizations and Information

To get SC,

system must execute critical paths safely

Can optimize non-critical paths
if information indicates non-critical cases

Useful optimization

All critical paths safe (slow)
Some non-critical paths unsafe (fast)

Information to allow optimization

Identify cases where optimization will not make critical paths unsafe
Optimizations and Information: A Problem

BUT

Programmer has info only from SC executions

*Information from SC executions must make non-SC hardware appear SC*

(Key Complexity in Analysis)

Solution: *Control Condition*

Pre-condition on hardware that ensures SC information sufficient

Commonly obeyed, but hard to prove

*Can now analyze only SC executions*
**Application of Mapping**

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (Pred) {          while ((F1 != 1) &amp;&amp; (F2 != 1)) {</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if (not Pred) {   }</td>
<td></td>
</tr>
<tr>
<td>A = 100;              ... = B;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B = 200;              ... = A;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1 = 1;               F2 = 1;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}                     }</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Are parallel reads of \(F_1, F_2\) really safe?

YES

How to design model to allow this optimization?

*One example next*
A New Memory Model

Provide special `signal`, `await` constructs

**Signal** writes location

  e.g., \( F_1 = 1 \)

**Await** loops on one or more locations

  e.g., while \((F_1 != 1 \&\& F_2 != 1)\);\}

Allowed use in any phase of an SC execution

  Only one `signal` per location per `await`

  `Signal,await` location not used by others

Simple analysis reveals

  Two `await` reads of a processor never on critical path

  \( \Rightarrow \) Can do await reads in parallel
Analysis for Two `await` Reads R1, R2

Two concepts

*Unessentials*: Can ignore unsuccessful iterations of synchronization loops

\[
\text{e.g., while (F1 \neq 1 \&\& F2 \neq 1) \{};
\]

*Self-ordered loops*: Can ignore paths from some writes to successful read of synchronization loop

\[
\text{e.g., path from signal to successful await always safe since successful await always after signal}
\]

When will R1 →po R2 be on a critical path?

Case 1: R2 is the last operation on the path

Path begins with signal write for R2

But R2 is self-ordered w.r.t. its signal write

Case 2: R2 is not the last operation on the path

Then next op must be conflicting write

But then R2 is unessential

Implies two `await` reads never on critical path
A New Memory Model (Cont.)

New memory model

System appears SC if

Program uses constructs only as allowed

New memory model allows

Parallel `await` reads

Parallel `signal` writes

Non-atomic `signal` writes

Many other optimizations in thesis

Short and intuitive reasoning
Characterization of Design Space

Key characteristic of model
   Executes certain ordering paths safely
   Called Valid Paths

Generic Memory Model
   If critical paths (of SC executions) are valid paths
   Then system appears SC

Performance potential of model
   How well valid paths capture critical paths?

Programmability and portability of model
   How easy to convert critical paths to valid paths?
Implementing Generic Model

Valid Path Requirement

Valid path from $X$ to $Y \Rightarrow$ all see $X$ before $Y$

- Enforce program order arcs on valid paths
- Make writes on conflict order arcs on valid paths atomic

Control Requirement (allows SC-only info)

- Write must wait until read that "controls" it done
  - Block on write until previous operations resolved
- Writes must terminate
- Loop writes must be coherent
The Design Space: Summary

Formalized and simplified design process
  Mapping between optimizations and information

New memory models with more optimizations
  More reordering, more pipelining,
  more non-atomic updates, fewer acks

A characterization of the design space
  Memory model = valid paths

Not yet done
  How much remaining potential useful?
  Which is the best model?
Overall Conclusions: Previous Work

Memory Model

Affects programmability, portability, performance

Intuitive model: sequential consistency

+ Programmmability
+ Portability
− Performance

Many alternative models: many hardware-centric

− Programmmability
− Portability
+/− Performance
Overall Conclusions: This Work

(I) Programmer-centric approach

Model = Contract

System gives sequential consistency
If programmer gives information

(II) Four programmer-centric models

DRF0, DRF1, PLpc1, PLpc2
Enhance 3Ps of many current models

(III) The design space of memory models

Formalized and simplified design process
Showed unexploited potential, new models
Characterized the design space

(IV) Detecting unidentified races on DRF systems

Can use SC techniques on DRF systems
What Next?

Which is best SCNF model?

Hardware to exploit new parallelism

Compiler benefits

Programming language extensions

Support for debugging, verification

Leave sequential consistency?