Rethinking Shared-Memory Languages and Hardware

Sarita V. Adve
University of Illinois
sadve@illinois.edu

Memory Consistency Models

Parallelism for the masses!
Shared-memory most common
Memory model = Legal values for reads
Memory Consistency Models

Parallelism for the masses!

Shared-memory most common

Memory model = Legal values for reads
Memory Consistency Models

Parallelism for the masses!

Shared-memory most common

Memory model = Legal values for reads
Memory Consistency Models

Parallelism for the masses!

Shared-memory most common

Memory model = Legal values for reads
Memory Consistency Models

Parallelism for the masses!

Shared-memory most common

Memory model = Legal values for reads
20 Years of Memory Models … & Beyond

- Memory model is at the heart of concurrency semantics
  - 20 year journey from confusion to convergence at last!
  - Hard lessons learned
  - Implications for future software and hardware

- Current way to specify concurrency semantics is too hard
  - Fundamentally broken for software and hardware

- Must rethink parallel languages and hardware
  - E.g., Deterministic Parallel Java (DPJ) language, DeNovo architecture
• **Memory Models**
  – Desirable properties
  – *State-of-the-art: Data-race-free, Java, C++*
  – Implications

• **Deterministic Parallel Java (DPJ)***

• **DeNovo***

• **Conclusions***
**What is a Memory Model?**

- **Memory model** defines what values a read can return.

---

Initially $A=B=C=\text{Flag}=0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A = 26$</td>
<td>while ($\text{Flag} \neq 1$) {}</td>
</tr>
<tr>
<td>$B = 90$</td>
<td>$r1 = B \leftarrow 90$</td>
</tr>
<tr>
<td>\ldots</td>
<td>$r2 = A \leftarrow 26$</td>
</tr>
<tr>
<td>$\text{Flag} = 1$</td>
<td>\ldots</td>
</tr>
</tbody>
</table>
Desirable Properties of a Memory Model

- Memory model is an interface between a program and its transformers

- Weakest system component exposed to the programmer

- Must satisfy 3 P properties
  - Programmability, Performance, Portability

  **Challenge: hard to satisfy all 3 Ps**
Programmability – SC [Lamport79]

- **Programmability**: Sequential consistency (SC) most intuitive
  - Operations of a single thread in program order
  - All operations in a total order or atomic

- **But Performance?**
  - Recent (complex) hardware techniques boost performance with SC
  - But compiler transformations still inhibited

- **But Portability?**
  - Almost all hardware, compilers violate SC today

⇒ SC not practical, but...
Parallel programming too hard even with SC

- Programmers (want to) write well structured code
- Explicit synchronization, no data races

Thread 1                                     Thread 2
Lock(L)                                        Lock(L)
Read Data1                                    Read Data2
Write Data2                                   Write Data1
...                                            ...
Unlock(L)                                     Unlock(L)

- SC for such programs much easier: can reorder data accesses

⇒ Data-race-free model  [AdveHill90]

- SC for data-race-free programs
- No guarantees for programs with data races
Definition of a Data Race

- Distinguish between data and non-data (synchronization) accesses

- Only need to define for SC executions ⇒ total order

- Two memory accesses form a race if
  - From different threads, to same location, at least one is a write
  - Occur one after another

  Thread 1
  - Write, A, 26
  - Write, B, 90
  - Write, Flag, 1

  Thread 2
  - Read, Flag, 0
  - Read, Flag, 1
  - Read, B, 90
  - Read, A, 26

- A race with a data access is a data race

- Data-race-free-program = No data race in any SC execution
**Data-Race-Free Model**

Data-race-free model = SC for data-race-free programs

- **Does not preclude races** for wait-free constructs, etc.
  * Requires races be explicitly identified as synchronization
  * E.g., use volatile variables in Java, atomics in C++

- **Dekker’s algorithm**

  Initially Flag1 = Flag2 = 0

  volatile Flag1, Flag2

  **Thread1**  **Thread2**
  
  Flag1 = 1  
  if Flag2 == 0  if Flag1 == 0
  //critical section  //critical section

  SC prohibits both loads returning 0
Data-Race-Free Approach

• Programmer’s model: SC for data-race-free programs

• Programmability
  – Simplicity of SC, for data-race-free programs

• Performance
  – Specifies **minimal** constraints (for SC-centric view)

• Portability
  – Language must provide way to identify races
  – Hardware must provide way to preserve ordering on races
  – Compiler must translate correctly
1990's in Practice (The Memory Models Mess)

- **Hardware**
  - Implementation/performance-centric view
  - Different vendors had different models – most non-SC
    * Alpha, Sun, x86, Itanium, IBM, AMD, HP, Cray, …
  - Various ordering guarantees + fences to impose other orders
  - Many ambiguities - due to complexity, by design(?)

- **High-level languages**
  - Most shared-memory programming with Pthreads, OpenMP
    * Incomplete, ambiguous model specs
    * Memory model property of language, not library [Boehm05]
  - Java – commercially successful language with threads
    * Chapter 17 of Java language spec on memory model
    * But hard to interpret, badly broken [Schuster et al., Pugh et al.]
2000 – 2004: Java Memory Model

• ~2000: Bill Pugh publicized fatal flaws in Java model

• Lobbied Sun to form expert group to revise Java model

• Open process via mailing list
  – Diverse participants
  – Took 5 years of intense, spirited debates
  – Many competing models
  – Final consensus model approved in 2005 for Java 5.0

[MansonPughAdve POPL 2005]
Java Memory Model Highlights

• Quick agreement that SC for data-race-free was required

• Missing piece: Semantics for programs with data races
  – Java cannot have undefined semantics for ANY program
  – Must ensure safety/security guarantees
    * Limit damage from data races in untrusted code

• Goal: Satisfy security/safety, w/ maximum system flexibility
  – Problem: “security/safety, limited damage” w/ threads very vague

    ... and hard!
Initially X=Y=0

Thread 1

\[ r1 = X \]
\[ Y = r1 \]

Thread 2

\[ r2 = Y \]
\[ X = r2 \]

Is \( r1=r2=42 \) allowed?
Initially X=Y=0

Thread 1

42

r1 = X

42

Y = r1

Is r1=r2=42 allowed? YES!

Thread 2

42

r2 = Y

42

X = r2

Data races produce causality loop!

- Definition of a causality loop was surprisingly hard
- Common compiler optimizations seem to violate “causality”
Java Memory Model Highlights

- Final model based on consensus, but complex
  - Programmers can (must) use “SC for data-race-free”
  - But system designers must deal with complexity
  - Correctness tools, racy programs, debuggers, ...??
  - Bugs discovered [SevcikAspinall08] ... remain unresolved
2005 - : C++, Microsoft Prism, Multicore

• ~2005: Hans Boehm initiated C++ concurrency model
  – Prior status: no threads in C++, most concurrency w/ Pthreads

• Microsoft concurrently started its own internal effort

• C++ easier than Java because it is unsafe
  – Data-race-free is plausible model

• BUT multicore ⇒ New h/w optimizations, more scrutiny
  – Mismatched h/w, programming views became painfully obvious
    * Fences define per-thread order, synch orders multiple threads
  – Debate that SC for data-race-free inefficient w/ hardware models
Hardware Implications of Data-Race-Free

- Synchronization (volatiles/atomics) must appear SC
  - Each thread’s synch must appear in program order
    
    ```
    synch Flag1, Flag2
    
    Thread 1                                      Thread 2
    Flag1 = 1                                     Flag2 = 1
    
    Fence                                           Fence
    
    if Flag2 == 0                   if Flag1 == 0
    
    critical section                  critical section
    
    SC ⇒ both reads cannot return 0
    ```

- Requires efficient fences between synch stores/loads
- All synchs must appear in a total order (atomic)
### Implications of Atomic Synch Writes

**Independent reads, independent writes (IRIW):**

Initially $X=Y=0$

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$X = 1$</td>
<td>$Y = 1$</td>
<td>$\ldots = Y \leftarrow 1$</td>
<td>$\ldots = X \leftarrow 1$</td>
</tr>
<tr>
<td></td>
<td>fence</td>
<td>fence</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\ldots = X \leftarrow 0$</td>
<td>$\ldots = Y \leftarrow 0$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$SC \Rightarrow$ no thread sees new value until old copies invalidated

- Shared caches w/ hyperthreading/multicore make this harder
- Programmers don’t usually use IRIW
- Why pay cost for SC in h/w if not useful to s/w?
• 2006: Pressure from hardware vendors to remove SC baseline

• But what is alternative?
  – Must allow some hardware optimizations
  – But must be teachable to undergrads

• Showed such an alternative (probably) does not exist
C++ Compromise

- Default C++ model is data-race-free [BoehmAdve PLDI 2008]

- But
  - Some systems need expensive fence for SC
  - Some programmers really want more flexibility
    * C++ specifies low-level (complex) model only for experts
    * Not advertising this
Lessons Learned

• SC for data-race-free minimal baseline

• Specifying semantics for programs with data races is HARD
  – But “no semantics for data races” also has problems
    * Not an option for safe languages; debugging; correctness checking tools

• Hardware-software mismatch for some code
  – “Simple” optimizations have unintended consequences

⇒ State-of-the-art is fundamentally broken
Lessons Learned

- SC for data-race-free minimal baseline
- Specifying semantics for programs with data races is HARD
  - But “no semantics for data races” also has problems
    * Not an option for safe languages; ebugging; correctness checking tools

- Hardware-software mismatch for some code
  - “Simple” optimizations have unintended consequences

⇒ State-of-the-art is fundamentally broken

Banish shared-memory?
Lessons Learned

• SC for data-race-free minimal baseline

• Specifying semantics for programs with data races is HARD
  – But “no semantics for data races” also has problems
    * Not an option for safe languages; debugging; correctness checking tools

  Banish wild shared-memory!

  Need disciplined shared memory!

⇒ State-of-the-art is fundamentally broken

• We need
  – Higher-level disciplined programming models that enforce discipline
  – Hardware co-designed with high-level models
What is Shared-Memory?

Shared-Memory =

Global address space

+ Implicit, anywhere communication, synchronization
What is Shared-Memory?

Shared-Memory =

Global address space

+ Implicit, anywhere communication, synchronization
What is Shared-Memory?

**Wild Shared-Memory =**

- **Global address space**
- **Implicit, anywhere communication, synchronization**
What is Shared-Memory?

**Wild Shared-Memory** =

Global address space +

Implicit, anywhere communication, synchronization
What is Shared-Memory?

**Disciplined** Shared-Memory =

Global address space

+ Implicit, anywhere communication, synchronization

Explicit, structured side-effects
Benefits of Explicit Effects

• **Strong safety properties**
  - Determinism-by-default
    * Sequential reasoning, parallel performance model
  - Safe non-determinism only when explicitly requested
    * Data-race-freedom, strong isolation, serializability, composition
  - Simplifies test/debug, composability, maintainability, …

• **Efficiency: power, complexity, performance**
  - Simplify coherence and consistency
  - Optimize communication and storage layout
    * Memory hierarchy driven by explicit effects vs. cache lines

⇒ Simple programming model AND
Power-, complexity-, performance-scalable hardware
**Our Approach**

- **Deterministic Parallel Java (DPJ) [Vikram Adve et al.]**
  - No data races, determinism-by-default, safe non-determinism
  - Simple semantics, safety, and composability

- **DeNovo [Sarita Adve et al.]**
  - Simple coherence and consistency
  - Software-driven coherence, communication, data layout
  - Power-, complexity-, performance-scalable hardware
Outline

- Memory Models
  - Desirable properties
  - State-of-the-art: Data-race-free, Java, C++
  - Implications

- Deterministic Parallel Java (DPJ)

- DeNovo

- Conclusions
DPJ Project Overview

• **Deterministic-by-default parallel language [OOPSLA09]**
  – Extension of sequential Java; fully Java-compatible
  – Structured parallel control: nested fork-join
  – **Novel region-based type and effect system**
  – Speedups close to hand-written Java programs
  – Expressive enough for irregular, dynamic parallelism

• **Disciplined support for non-deterministic code [POPL11]**
  – Non-deterministic, deterministic code can co-exist safely
  – Explicit, data race-free, isolated

• **Semi-automatic tool for effect annotations [ASE09]**

• **Encapsulating frameworks, unchecked code [ECOOP11]**

• **Software:** http://dpj.cs.illinois.edu/
Regions and Effects

- **Region**: a name for a set of memory locations
  - Programmer assigns a region to each field and array cell
  - Regions partition the heap

- **Effect**: a read or write on a region
  - Programmer summarizes effects of method bodies

- **Compiler checks that**
  - Region types are consistent, effect summaries are correct
  - Parallel tasks are non-interfering (no conflicts)
  - Simple, modular type checking (no inter-procedural …)

- **Programs that type-check are guaranteed determinism**

- **Side benefit**: regions, effects are valuable documentation
class Pair {
    region One, Two;
    int one in One;
    int two in Two;
    void setOne(int one) writes One {
        this.one = one;
    }
    void setTwo(int two) writes Two {
        this.two = two;
    }
    void setOneTwo(int one, int two) writes One; writes Two {
        cobegin {
            setOne(one); // writes One
            setTwo(two); // writes Two
        }
    }
}

Region names have static scope (one per class)

<table>
<thead>
<tr>
<th></th>
<th>one</th>
<th>two</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pair</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair.One</td>
<td>one</td>
<td>3</td>
</tr>
<tr>
<td>Pair.Two</td>
<td>two</td>
<td>42</td>
</tr>
</tbody>
</table>

Declaring and using region names
class Pair {
    region One, Two;
    int one in One;
    int two in Two;
    void setOne(int one) writes One {
        this.one = one;
    }
    void setTwo(int two) writes Two {
        this.two = two;
    }
    void setOneTwo(int one, int two) writes One; writes Two {
        cobegin {
            setOne(one);  // writes One
            setTwo(two);  // writes Two
        }
    }
}

Writing method effect summaries
class Pair {
region One, Two;
int one in One;
int two in Two;
void setOne(int one) writes One {
    this.one = one;
}
void setTwo(int two) writes Two {
    this.two = two;
}
void setOneTwo(int one, int two) writes One; writes Two {
    cobegin {
        setOne(one); // writes One
        setTwo(two); // writes Two
    }
}
}
class Tree<region P> {  
    region L, R;  
    int data in P  
    Tree<P:L> left;  
    Tree<P:R> right;  
    int increment() writes P:* {  
        ++data; // infer: writes P  
        cobegin {  
            left.increment(); // infer: writes P:L:*  
            right.increment(); // infer: writes P:R:*  
        }  
    }  
}
Safe Non-Determinism

• Intentional non-determinism is sometimes desirable
  – Branch-and-bound; graph algorithms; clustering
  – Will often be combined with deterministic algorithms

• DPJ mechanisms
  – foreach\_nd, cobegin\_nd
  – Atomic sections and atomic effects
  – Only atomic effects within non-deterministic tasks can interfere

• Guarantees
  – Explicit: Non-determinism cannot happen by accident
  – Data race-free: Guaranteed for all legal programs
  – Isolated: Deterministic, non-det parts isolated, composable
Outline

• Memory Models
  – Desirable properties
  – State-of-the-art: Data-race-free, Java, C++
  – Implications

• Deterministic Parallel Java (DPJ)

• DeNovo

• Conclusions
**DeNovo Goals**

- **If software is disciplined, how to build hardware?**
  - Goal: power-, complexity-, performance-scalability

- **Strategy:**
  - Many emerging software systems with disciplined shared-memory
    - DeNovo uses DPJ as driver
    - End-goal: language-oblivious interface
  - Focus so far on deterministic codes
    - Common and best case
    - Extending to safe non-determinism, legacy codes
  - Hardware scope: full memory hierarchy
    - Coherence, consistency, communication, data layout, off-chip memory
• Coherence, consistency, communication

  – Complexity
    * Subtle races and numerous transient states in the protocol
    * Hard to extend for optimizations

  – Storage overhead
    * Directory overhead for sharer lists

  – Performance and power inefficiencies
    * Invalidation and ack messages
    * False sharing
    * Indirection through the directory
    * Suboptimal communication granularity of cache line …
Results So Far

• **Simplicity**
  – Compared DeNovo protocol complexity with MESI
  – 15X fewer reachable states, 20X faster with model checking

• **Extensibility**
  – Direct cache-to-cache transfer
  – Flexible communication granularity

• **Storage overhead**
  – No storage overhead for directory information
  – Storage overheads beat MESI after tens of cores and scale beyond

• **Performance/Power**
  – Up to 75% reduction in memory stall time
  – Up to 72% reduction in network traffic
Guaranteed determinism

⇒ Read returns value of \textit{last} write in sequential order

1. Same task in this parallel phase

2. Or before this parallel phase
Cache Coherence

- Coherence Enforcement
  1. Invalidate stale copies in caches
  2. Track up-to-date copy

- Explicit effects
  - Compiler knows all regions written in this parallel phase
  - Cache can self-invalidate before next parallel phase
    * Invalidates data in writeable regions not accessed by itself

- Registration
  - Directory keeps track of one up-to-date copy
  - Writer updates before next parallel phase
Basic DeNovo Coherence

• Assume (for now): Private L1, shared L2; single word line
  – Data-race freedom at word granularity

• L2 data arrays double as directory
  – Keep valid data or registered core id, no space overhead

• L1/L2 states
  - Invalid
  - Read → Valid
  - Write
  - Registered

• Touched bit set only if read in the phase
class S_type {
    X in DeNovo-region ;
    Y in DeNovo-region ;
}

S_type S[size];

... Phase1 writes { // DeNovo effect
    foreach i in 0, size {
        S[i].X = ...;
    }
    self_invalidate();
}
Addressing Limitations

• Addressing current limitations
  – Complexity
    * Subtle races and numerous transient states in the protocol ✔
    * Hard to extend for optimizations
  – Storage overhead
    * Directory overhead for sharer lists ✔
  – Performance and power overhead
    * Invalidation and acknowledgment messages ✔
    * False-sharing ✔
    * Indirection through the directory
    * Suboptimal communication granularity of cache line …
Practical DeNovo Coherence

• Basic protocol impractical
  – High tag storage overhead (a tag per word)

• Address/Transfer granularity > Coherence granularity

• DeNovo Line-based protocol
  – Traditional software-oblivious spatial locality
  – Coherence granularity still at word
    * no word-level false-sharing

Line Merging

<table>
<thead>
<tr>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
</tr>
<tr>
<td>V</td>
</tr>
<tr>
<td>V</td>
</tr>
<tr>
<td>R</td>
</tr>
</tbody>
</table>

*Line Merging*
DeNovo overhead is scalable and beats MESI after 29 cores
Addressing Limitations

- **Addressing current limitations**
  - **Complexity**
    - Subtle races and numerous transient states in the protocol ✓
    - Hard to extend for optimizations
  - **Storage overhead** ✓
    - Directory overhead for sharer lists
  - **Performance and power overhead**
    - Invalidation and ack messages ✓
    - False-sharing ✓
    - Indirection through the directory
    - Suboptimal communication granularity of cache line ...
• Traditional directory-based protocols
  ⇒ Sharer-lists always contain all the true sharers

• DeNovo protocol
  ⇒ Registry points to latest copy at end of phase

⇒ Valid data can be copied around freely
• Basic with Direct cache-to-cache transfer
  – Get data directly from producer
  – Through prediction and/or software-assistance
  – Convert 3-hop misses to 2-hop misses
• **Basic with Flexible communication**
  
  – Software-directed data transfer
  
  – Transfer “relevant” data together

  – **Effect of AoS-to-SoA transformation w/o programmer/compiler**
• **Basic with Flexible communication**

  – *Software-directed data transfer*
  
  – *Transfer “relevant” data together*
  
  – *Effect of AoS-to-SoA transformation w/o programmer/compiler*
Evaluation

- **Simplicity**
  - Formal verification of coherence protocol
  - Comparing reachable states

- **Performance/Power**
  - Simulation experiments

- **Extensibility**
  - DeNovo extensions
Protocol Verification

- DeNovo vs. MESI word with Murphi model checking

- Correctness
  - Three bugs in DeNovo protocol
    * Mistakes in translation from high level spec
    * Simple to fix
  - Six bugs in MESI protocol
    * Two deadlock scenarios
    * Unhandled races due to L1 writebacks
    * Several days to fix

- Complexity
  - 15x fewer reachable states for DeNovo
  - 20x difference in the runtime
Memory Stall Time

- Mem Hit
- Remote L1 Hit
- L2 Hit

Memory Stall Time

- FFT
- LU
- Barnes
- kdFalse
- kdPad
- Bodytrack
DeNovo vs. MESI word: simplicity doesn’t reduce performance
- **DeNovo vs. MESI word**: simplicity doesn’t reduce performance
- **DeNovo line much better than MESI line with false sharing**
• DeNovo vs. MESI word: simplicity doesn’t reduce performance
• DeNovo line much better than MESI line with false sharing
• Benefit of lines is app-dependent
• DeNovo vs. MESI word: simplicity doesn’t reduce performance
• DeNovo line much better than MESI line with false sharing
• Benefit of lines is app-dependent
• DeNovo with flexible transfer is best: up to 75% reduction vs. MESI line
DeNovo has less network traffic than MESI

Up to 72% reduction
DeNovo Summary

- **Simplicity**
  - Compared DeNovo protocol complexity with MESI
  - 15X fewer reachable states, 20X faster with model checking

- **Extensibility**
  - Direct cache-to-cache transfer
  - Flexible communication granularity

- **Storage overhead**
  - No storage overhead for directory information
  - Storage overheads beat MESI after tens of cores and scale beyond

- **Performance/Power**
  - Up to 75% reduction in memory stall time
  - Up to 72% reduction in network traffic

- **Future work:** Data layout, off-chip mem, non-det/legacy codes, …
• Current way to specify shared-memory semantics fundamentally broken
  – Best we can do is SC for data-race-free programs
  – But not good enough
    * Cannot hide from programs with data races
    * Mismatched h/w-s/w: simple optimizations give unintended consequences

• Need
  – High-level disciplined models that enforce discipline
  – Hardware co-designed with high-level model

• Previous memory models convergence from similar process
  – But this time, let’s co-design software and hardware
Conclusions (2 of 2)

Deterministic Parallel Java (DPJ) [Vikram Adve et al.]
- No data races, determinism-by-default, safe non-determinism
- Simple semantics, safety, and composability

DeNovo [Sarita Adve et al.]
- Simple coherence and consistency
- Software-driven coherence, communication, data layout
- Power-, complexity-, performance-scalable hardware

Future work: LOTS!