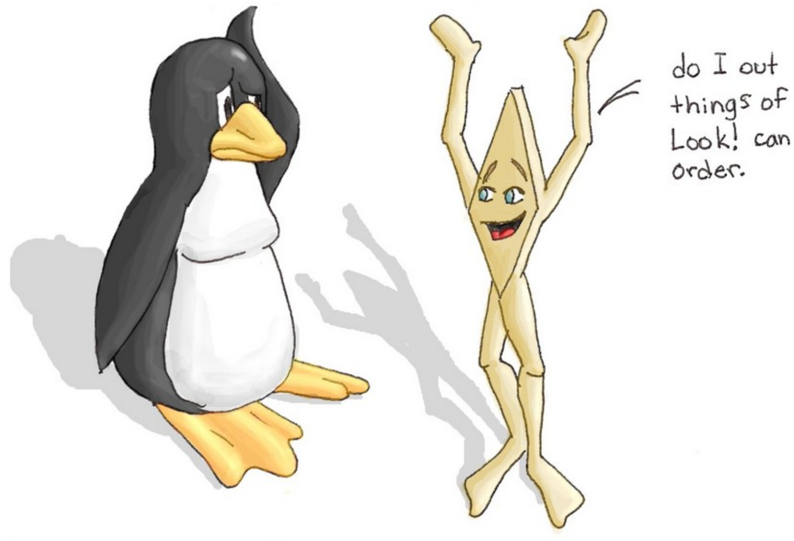
EECS 570

Lecture 13

End-to-End SC



Winter 2025

Prof. Satish Narayanasamy

http://www.eecs.umich.edu/courses/eecs570/

Slides developed in part by Profs. Adve, Falsafi, Hill, Lebeck, Martin, Narayanasamy, Nowatzyk, Reinhardt, Singh, Smith, Torrellas and Wenisch.

Announcements

Midterm exam – 26th Wednesday 3p-4:20p

Old exams posted on the website

Readings

For this week:

Sorin, Hill, Wood. A Primer on Memory Consistency and Cache Coherence. Synthesis Lectures, 2011.Chapter 3

A Safety-First Approach to Memory Models. Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, Madanlal Musuvathi. IEEE Micro, Top Picks from the 2012 Computer Architecture Conferences, May/June 2013.

Skim this paper: [The Silently Shifting Semicolon.]

Daniel Marino, Todd D. Millstein, Madanlal Musuvathi, Satish Narayanasamy, Abhayendra Singh, Madan Musuvathi
1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA | May 2015
Publication

Language-Level
DRF-0 Vs SC
Memory Model

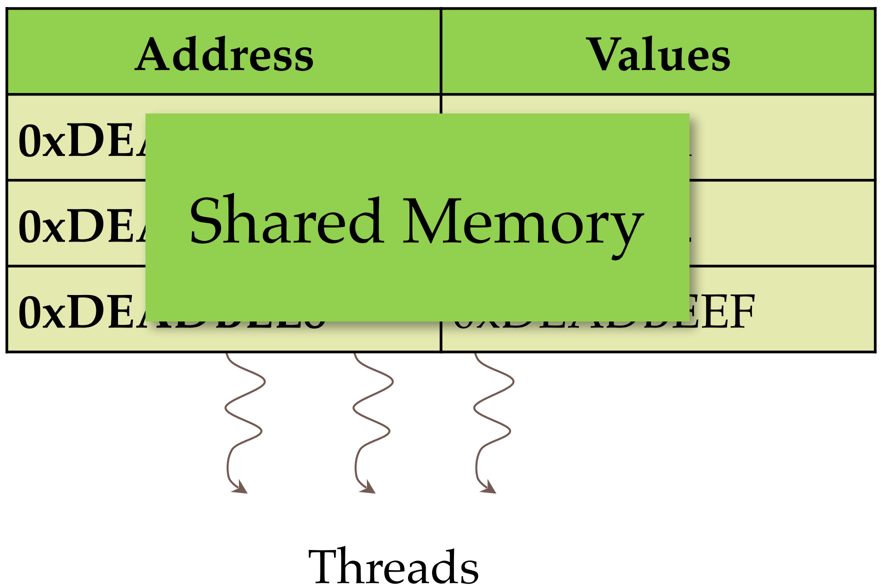
Program Order

a thread

A ; B

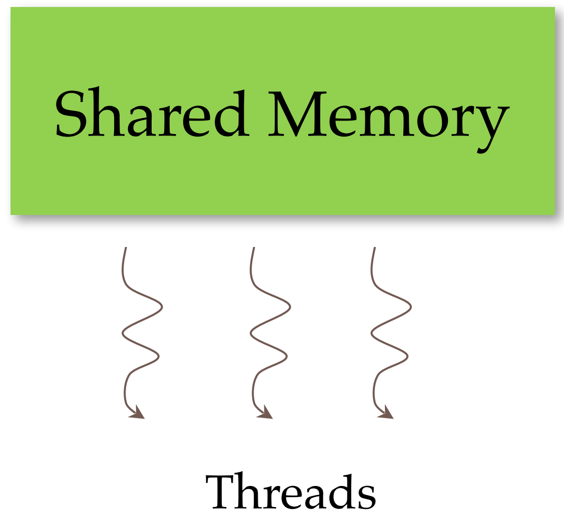
Execute A and then B

Atomic Shared Memory



Memory is a map from address to values
with reads/writes taking effect immediately

Intuitive Concurrency Semantics



Atomic

Program order

Memory model that guarantees this is called sequential consistency

Sequential Consistency

```
X* x = null;
bool flag = false;
```

```
// Producer Thread          // Consumer Thread
A: x = new X();             C: while(!flag);
B: flag = true;             D: x->f++;
```

sequential consistency (SC)

[Lamport 1979]

memory operations appear to occur in some global order consistent with the program order

Intuitive reasoning fails in C++/Java

```
X* x = null;
bool flag = false;

// Producer Thread          // Consumer Thread
A: x = new X();             C: while(!flag);
B: flag = true;             D: x->f++;
```

In C++ model this can crash!

Intuitive reasoning fails in C++/Java

```
X* x = null;
bool flag = false;
```

```
// Producer                 // Consumer
A: x = new X();             C: while(!flag);
B: flag = true;             D: x->f++;
```

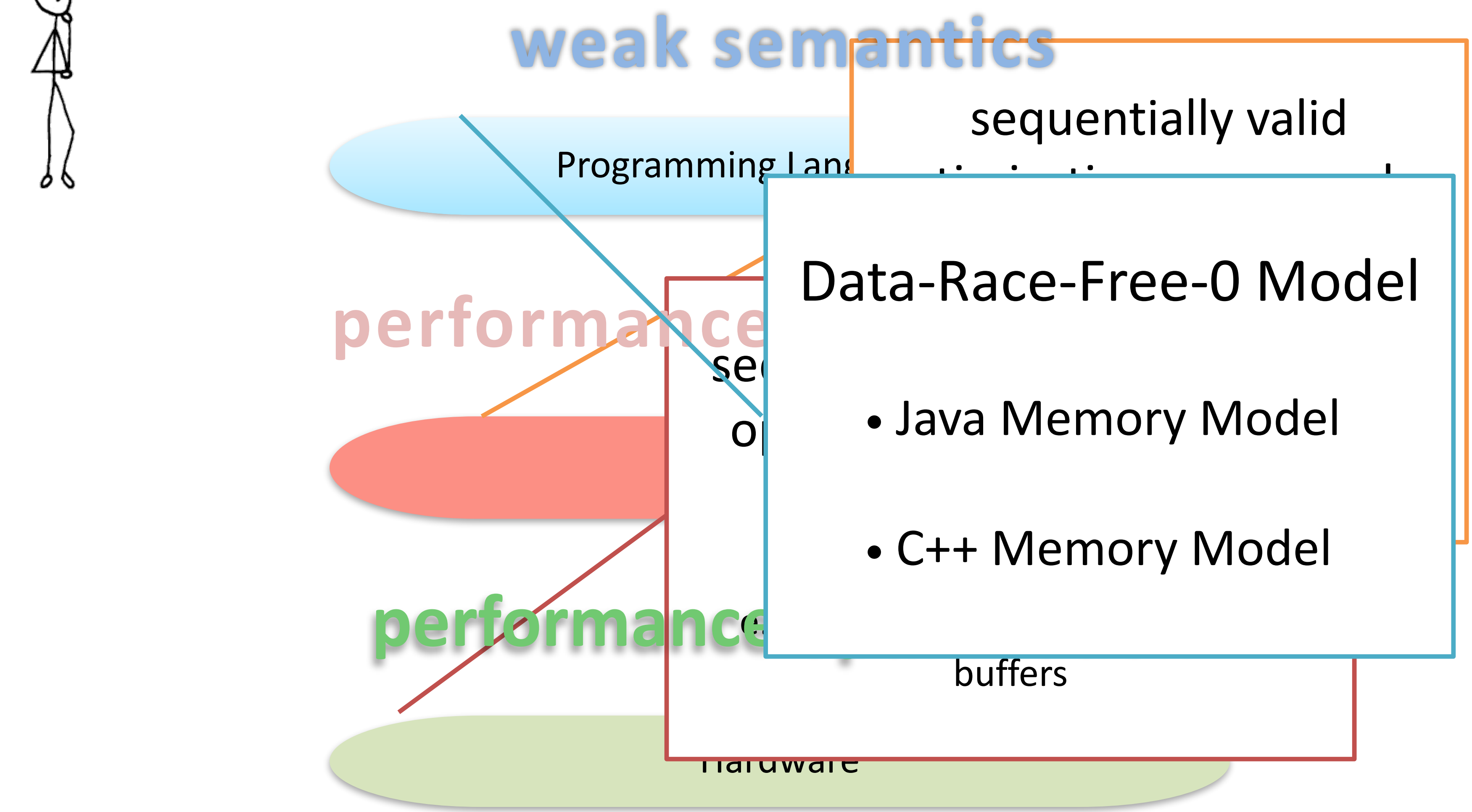
Optimizing Compiler and Hardware

Null Dereference!

B doesn't depend on A.

It might be faster to reorder them!

Why are accesses reordered?



A Short Detour: Data Races

A program has a data race if it has an execution in which two conflicting accesses to memory are simultaneously ready to execute.

access the same memory location
at least one is a write

Data Race

```
// T                    ad u
A: x                    e(!flag);
B: flag = true          D: x->f++;
```

Useful Data Races

- Data races are essential for implementing shared-memory synchronization

```
AcquireLock(){
    while (lock == 1) {}
    t = CAS (lock, 0, 1);
    if (!t) retry;
}
```

```
ReleaseLock() {
    lock = 0;
}
```

Data Race Free Memory Model

A program is data-race-free if all data races are appropriately annotated (`volatile/atomic`)

DRF0

[Adve & Hill 1990]

SC behavior for data-race-free programs, weak or no semantics otherwise

Java Memory Model (JMM)

[Manson et al. 2005]

C++0x Memory Model

[Boehm & Adve 2008]

DRF0-compliant Program

```
            X* x = null;
atomic bool flag = false;

A: x = new X();          C: while(!flag);
B: flag = true;          D: x->f++;
```

- DRF0 guarantees SC

  …. only if data-race-free (all *unsafe* accesses are annotated)

- What if there is one data-race?

  …. all bets are off (e.g., compiler can output an empty binary!)

Data-Races are Common

- Unintentional data-races
  - Easy to accidentally introduce a data race
    - forget to grab a lock
    - grab the wrong lock
    - forget a `volatile` annotation
    - …

- Intentional data-races
  - 100s of "benign" data-races in legacy code

[Narayanasamy et al. PLDI 2007]

Data Races with no Race Condition (assuming SC)

- Single writer multiple readers

```
// Thread t            // Thread u
A:  time++;            B:  l = time;
```

Data Races with no Race Condition (assuming SC)

- Lazy initialization

```
// Thread t                      // Thread u
   if( p == 0 )                  if( p == 0 )
      p = init();                   p = init();
```

Intentional Data Races

- ~97% of data races are not errors under SC
  - Experience from one Microsoft internal data-race detection study [Narayanasamy et al. PLDI'07]
- The main reason to annotate data races is to protect against compiler/hardware optimizations

Data Race Detection is Not a Solution

- Current static data-race detectors are not sound *and* precise
  - typically only handle locks, conservative due to aliasing, ...
- Dynamic analysis is costly
  - DRFx: throw exception on a data-race [Marino'10]
  - Either slow (8x) or requires complex hardware
- Legacy issues

Deficiencies of DRF0

weak or no semantics for data-racy programs

+

no easy way to identify & reject racy programs

problematic for

DEBUGGABILITY

programmer must assume non-SC

Analogous to unsafe languages: relying on programmer infallibility

optimi

jump to arbitrary code!

[Boehm et al., PLDI 2008]

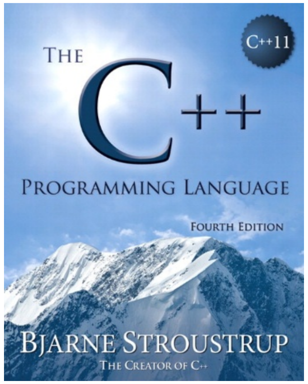
RECTNESS

n safety at the cost of complexity

[Ševčík&Aspinall, ECOOP 2008]

Languages, compilers, processors are adopting DRF0

Not a strong foundation



[Foundations of the C++ concurrency memory model]

[The Java memory model]

[Rust: Atomics - The Rustonomicon]

[The Go Memory Model]

Language-level SC: A Safety-First Approach

Program order and shared memory are important abstractions

Modern languages should protect them

All programs, buggy or otherwise, should have SC semantics

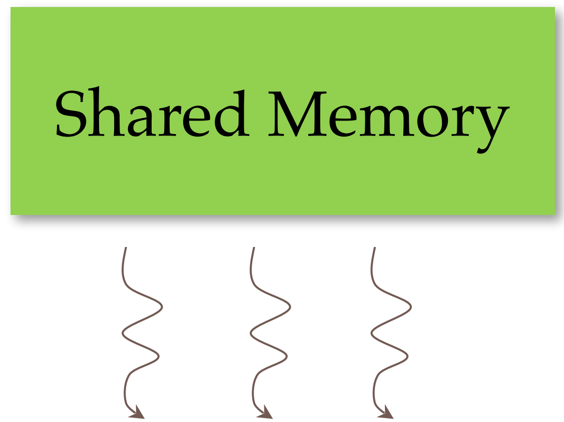
Efficiently supporting Language-Level SC

What is the Cost of SC?

SC prevents essentially all compiler and hardware optimizations.

And thus SC is impractical. *Or is it?*

Review: SC



Atomic

Program order

Threads

Memory model that guarantees this is called sequential consistency

Sequential Consistency

```
X* x = null;
bool flag = false;
```

```
// Producer Thread          // Consumer Thread
A: x = new X();             C: while(!flag);
B: flag = true;             D: x->f++;
```

sequential consistency (SC)

[Lamport 1979]

memory operations appear to occur in some global order consistent with the program order

End-to-End Sequential Consistency

Efficient language-level SC is feasible with hardware-software cooperation

- SC-Preserving compiler (Language specification)
- SC-Preserving hardware (Hardware ISA)

SC-Preserving Compiler

SC-Preserving Definition

- A SC-preserving compiler ensures that

  every SC-behavior of the binary

  is a SC-behavior of the source

- Guarantees end-to-end SC when the binary is run on SC-hardware

An SC-Preserving C Compiler

modified LLVM[Lattner & Adve 2004] to be SC-preserving

- obvious idea: restrict optimizations so they never reorder shared accesses
- simple, small modifications to the base compiler
- slowdown on x86: average of **3.8%**
    - PARSEC, SPLASH-2, SPEC CINT2006

Many ~~SOME~~ OPTIMIZATIONS PRESERVE SC

all optimizations on locals and compiler temporaries

```
for(i=0;i<3;i++)
      X++;
```

loop unrolling

```
X++;X++;X++
```

```
foo();
bar();
baz();
```

function inlining

```
bar(){X++;}
```

```
foo();
X++;
baz();
```

arithmetic reassociation

stack slot coloring

```
t=X*4;
```

arithmetic simplification

```
t=X<<2;
```

unreachable code elim.

dead argument elim.

loop rotation

loop unswitching

virtual to physical register allocation

correlated val prop

tail call elim

scalar replication

allocating locals to virtual registers

Optimizations That Break SC

- Example: Common Subexpression Elimination (CSE)

```
L1: t = X*5;
L2: u = Y;
L3: v = X*5;
```

```
L1: t = X*5;
L2: u = Y;
L3: v = t;
```

t,u,v are local variables

X,Y are possibly shared

Common Subexpression Elimination is not SC-Preserving

Init: X = Y = 0;

```
L1: t = X*5;
L2: u = Y;
L3: v = X*5;
```

```
M1: X =
1;
M2: Y =
1;
```

u == 1 ➔ v == 5

Init: X = Y = 0;

```
L1: t = X*5;
L2: u = Y;
L3: v = t;
```

```
M1: X =
1;
M2: Y =
1;
```

possibly u == 1 && v == 0

IMPLEMENTING CSE IN A SC-PRESERVING COMPILER

```
L1: t = X*5;
L2: u = Y;
L3: v = X*5;
```

→

```
L1: t = X*5;
L2: u = Y;
L3: v = t;
```

- Enable this transformation when
  - X is a *safe* variable, or
  - Y is a *safe* variable
- Identifying *safe* variables:
  - Compiler generated temporaries
  - Stack allocated variables whose address is not taken
- More *safe* variables?

A SC-PRESERVING LLVM COMPILER FOR C PROGRAMS

- Enable transformations on *safe* variables
- Enable transformations involving a single shared variable
  - e.g. t= X; u=X; v=X; → t=X; u=t; v=t;
- Enable trace-preserving optimizations
  - These do not change the order of memory operations
  - e.g. loop unrolling, procedure inlining, control-flow simplification, dead-code elimination,…
- Modified each of ~70 passes in LLVM to be SC-preserving

Experiments using LLVM

- baseline

  stock LLVM compiler with standard optimizations (-O3)

- no optimizations

  disable all LLVM optimization passes

- naïve SC-preserving

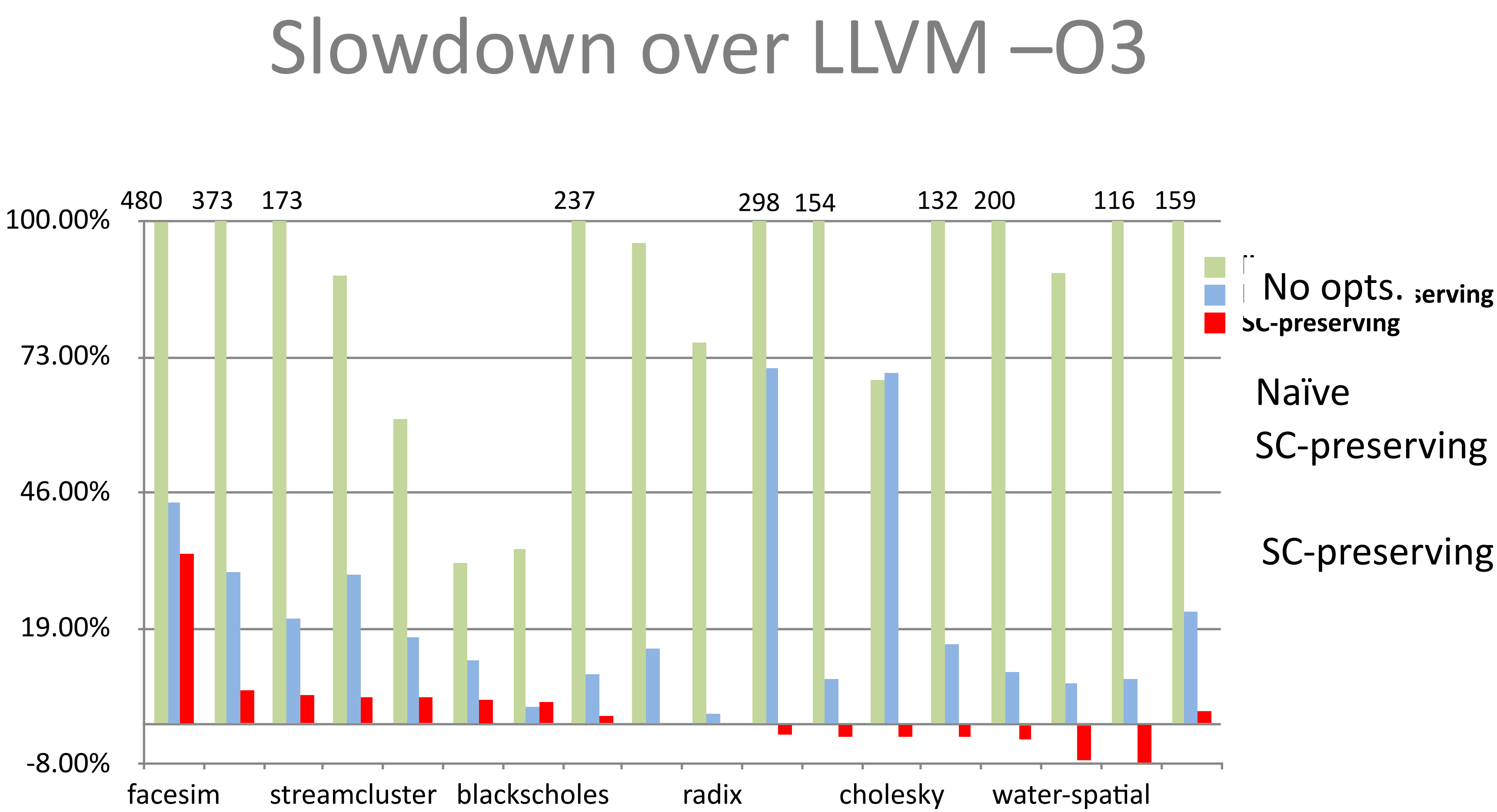
  disable LLVM passes that possibly reorder memory accesses

- SC-preserving

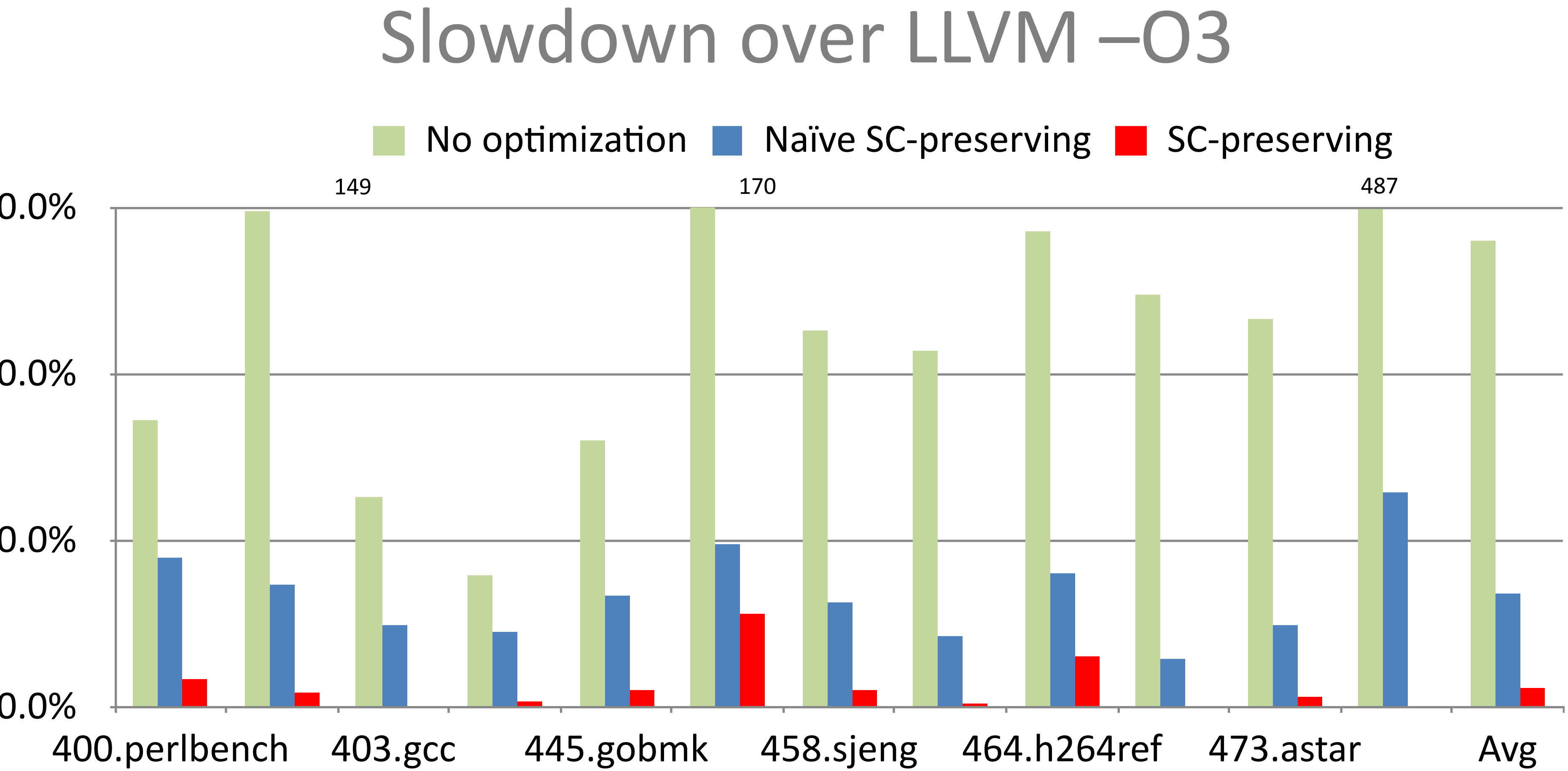
  use modified LLVM passes that avoid reordering shared memory accesses

- ran compiled programs on 8-core Intel Xeon

Parallel Benchmarks



SPEC INTEGER 2006



How Far Can A SC-Preserving Compiler Go?

```
float s, *x, *y;
int i;
s=0;
for( i=0; i<n; i+
+ ){
   s += (x[i]-y[i])
          * (x[i]-
y[i]);
}
```

no opt.

```
float s, *x, *y;
int i;
s=0;
for( i=0; i<n; i++ ){
    s += (*(x + i*sizeof(float))
–
             *(y +
i*sizeof(float))) *
            (*(x + i*sizeof(float))
–
             *(y +
i*sizeof(float)));
}
```

SC pres

```
float s, *x, *y;
float *px, *py, *e;

s=0; py=y; e = &x[n]
for( px=x; px<e; px++, py+
+){
   s += (*px-*py)
        * (*px-*py);
}
```

full opt

```
float s, *x, *y;
float *px, *py, *e, t;

s=0; py=y; e = &x[n]
for( px=x; px<e; px++, py+
+){
   t = (*px-*py);
   s += t*t;
}
```

Many "Can't-Live-Without" Optimizations are Eager-Load Optimizations

- Eagerly perform loads or use values from previous loads or stores

Common Subexpression Elimination

```
L1: t = X*5;
L2: u = Y;
L3: v = X*5;
```

→

```
L1: t = X*5;
L2: u = Y;
L3: v = t;
```

Constant/copy Propagation

```
L1: X = 2;
L2: u = Y;
L3: v = X*5;
```

→

```
L1: X = 2;
L2: u = Y;
L3: v = 10;
```

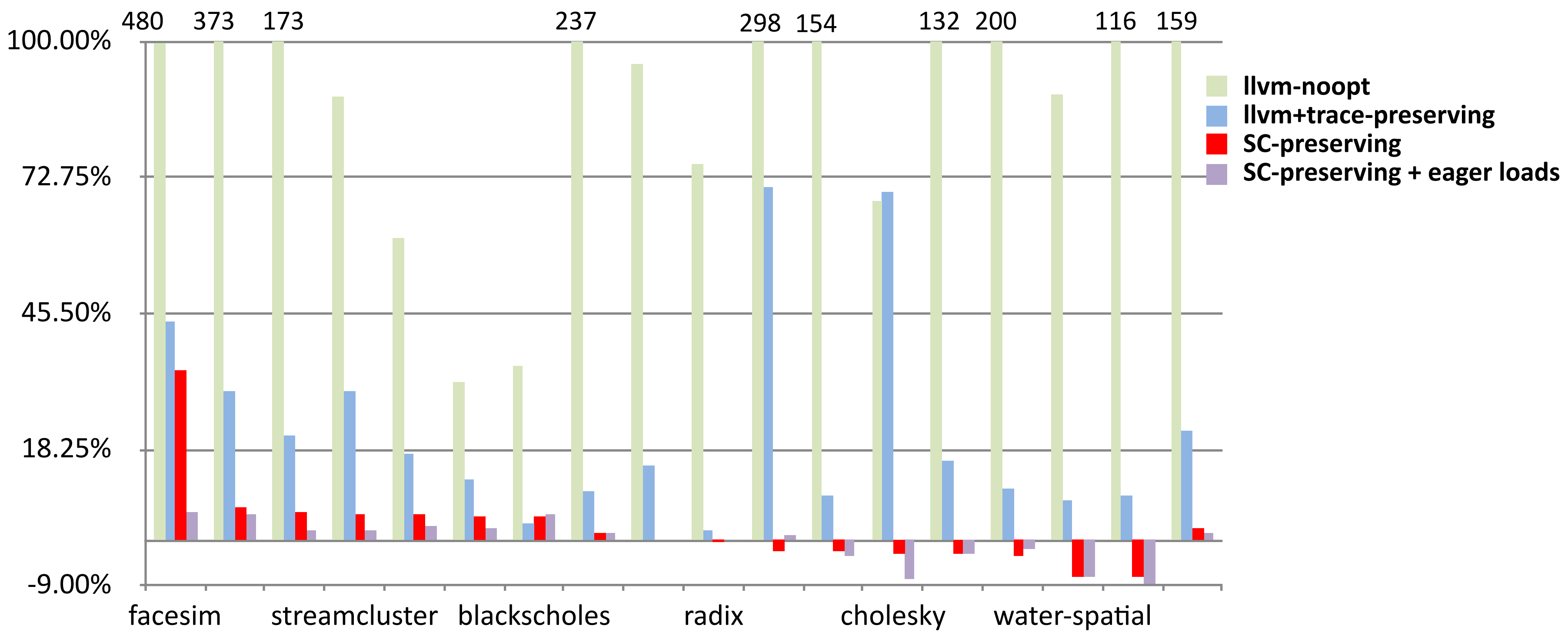
Loop-invariant Code Motion

```
L1:
L2: for(…)
L3:    t =
X*5;
```

→

```
L1: u = X*5;
L2: for(…)
L3:    t =
u;
```

PERFORMANCE OVERHEAD



Allowing eager-load optimizations alone reduces max overhead to 6%

SPECULATIVELY PERFORMING EAGER-LOAD OPTIMIZATIONS

```
L1: t = X*5;
L2: u = Y;
L3: v = X*5;
```

→

```
L1: t = monitor.load(X, tag)
* 5;
L2: u = Y;
L3: v = t;
C4: if
(interference.check(tag))
C5:    v = X*5;
```

- On monitor.load, hardware starts tracking coherence messages on X's cache line
- The interference check fails if X's cache line has been downgraded since the monitor.load
- In our implementation, a single instruction checks interference on up to 32 tags

CONCLUSION ON SC-PRESERVING COMPILER

- Efficient SC-preserving compiler is feasible with careful engineering
- Hardware support can enable eager-load optimizations without violating SC

SC-Preserving Hardware

SC: HARDWARE

Formal Requirements:

Before LOAD is performed w.r.t. any other processor,
all prior LOADs must be globally performed and
all prior STOREs must be performed

Before STORE is performed w.r.t. any other processor,
all prior LOADs globally performed and
all previous STORE be performed.

Every CPU issues memory ops in program order

In simple words:

SC: Perform memory operations in program order

Naïve SC Processor Design

Requirement: Perform memory operations in program order

Need

coherence

store atomicity

\+ memory ordering restrictions

Review: Coherence

A Memory System is Coherent if

- can serialize all operations to that location such that,
- operations performed by any processor to a location appear in program order (<p)
- value returned by a read is value written by last store to that location

There is broad consensus that coherence is a good idea.

Store Atomicity: Example 1

```
              A=0   B=0
P1           P2                  P3
A=1;         while (A==0);       while (B==0);
             B = 1;              print A;
```

Intuition says: P3 prints A=1

- But, with caches:
  - A=0 initially cached at P3 in shared state
  - Invalidation for A arrives at P2; sends out B=1
  - Invalidation for B arrives at P3
  - P3 prints A=0 before invalidation from P1 arrives

Many past commercial systems allow this behavior

- Key issue here: store atomicity
  - Do new values reach all nodes at the same time?

Store Atomicity: Example 2

- Store atomicity –All nodes will agree on the order that writes happen

```
           A=0   B=0
P1         P2          P3                P4
A=1;       B = 1;      Ld B -> r1;       Ld A -> r1;
                       Ld A -> r2;       Ld B -> r2;
```

- Under store-atomicity, what results are (im-)possible?

Implementing Store Atomicity

- On a bus…
  - Trivial (mostly); store is globally performed when it reaches the bus
- With invalidation-based directory coherence…
  - Writer cannot reveal new value till all invalidations are ack'd
- With update-based coherence…
  - Hard to achieve… updates must be ordered across all nodes
- With multiprocessors & shared caches
  - Cores that share a cache must not see one another's writes! (ugly!)

SC Memory Ordering Constraint

Memory ordering constraints:

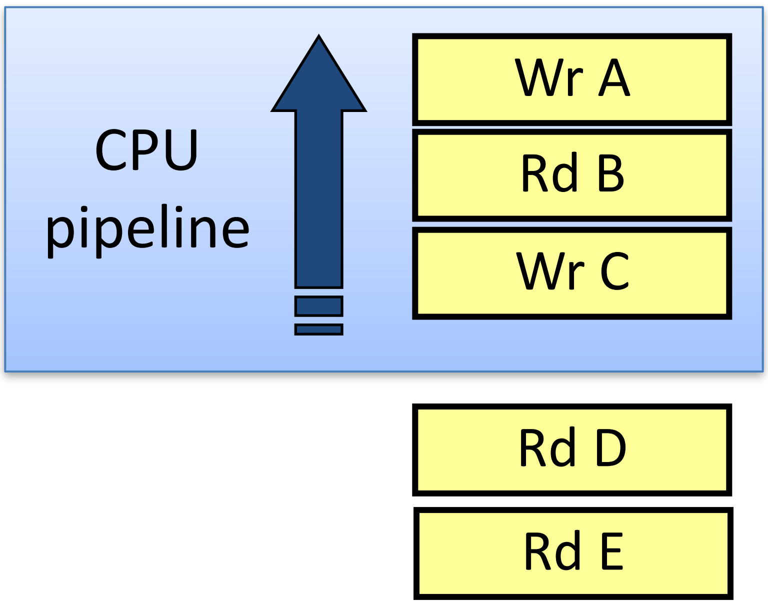
- Processor core waits for store to complete, before issuing next memory op
- Processor core waits for load to complete, before issuing next op

Problem: Too slow …

Optimizations

- Non-Binding store prefetching
  - A non-binding prefetch is effectively a no-op as far as memory model is concerned
- Speculative cores (e.g., branch prediction)
  - Squashed loads/stores due to any misspeculation made to look like non-binding prefetches
- In-window speculation
- Out-of-window speculation

Execution in strict SC

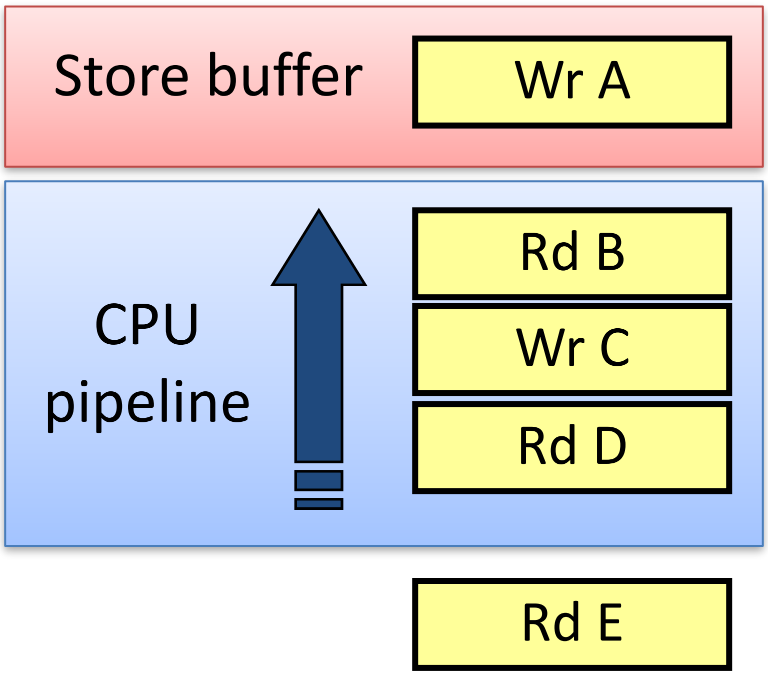


Wr A	Miss
Rd B	Idle
Wr C	Idle
Rd D	Not fetched
Rd E	Not fetched

- Miss on Wr A stalls all unrelated accesses

Memory accesses issue one-at-a-time

SC + Store Buffer

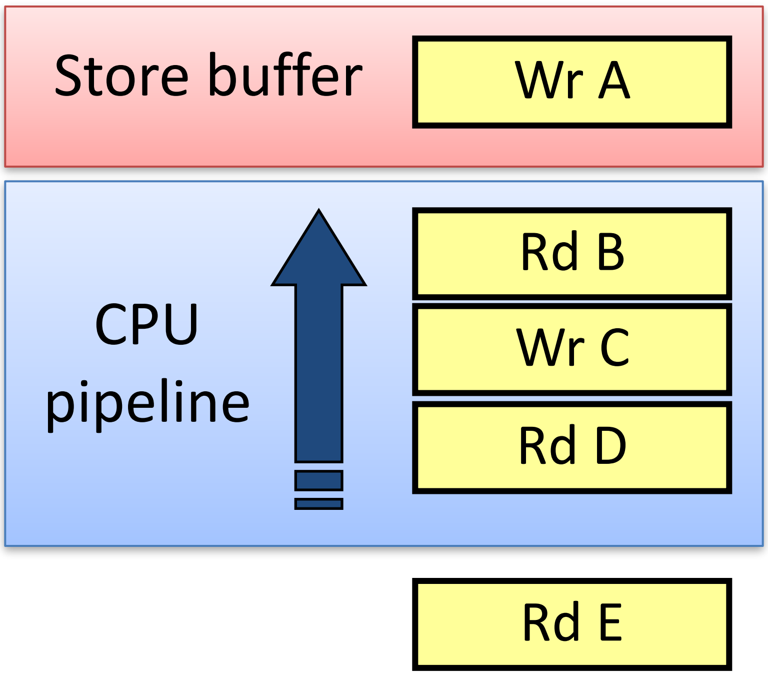


Wr A	Miss
Rd B	Idle
Wr C	Idle
Rd D	Idle
Rd E	Not fetched

- Removes pending stores from ROB…
- …but still no memory parallelism

SC + Store Buffer + Store Prefetching

[Gharachorloo 91]

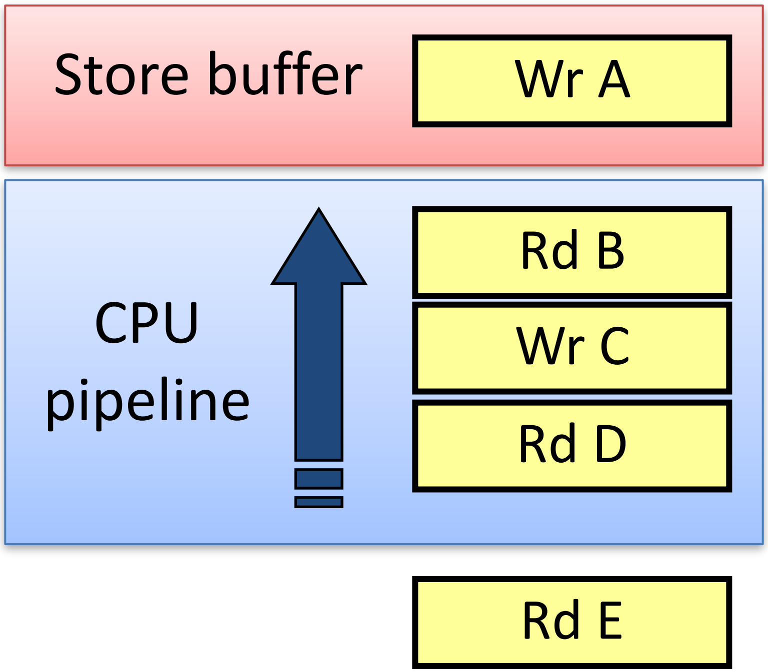


Wr A	Miss
Rd B	Idle
Wr C	Prefetch
Rd D	Idle
Rd E	Not fetched

- Key Idea: Separate fetching write permission from writing to the cache
  - “Store prefetch” performs coherence ops in advance
  - Commit value to cache when write leaves ROB
- May need to re-request store permission upon commit

MIPS R10K:
SC + SB + Prefetch + In-window Load Speculation

[Gharachorloo 91]



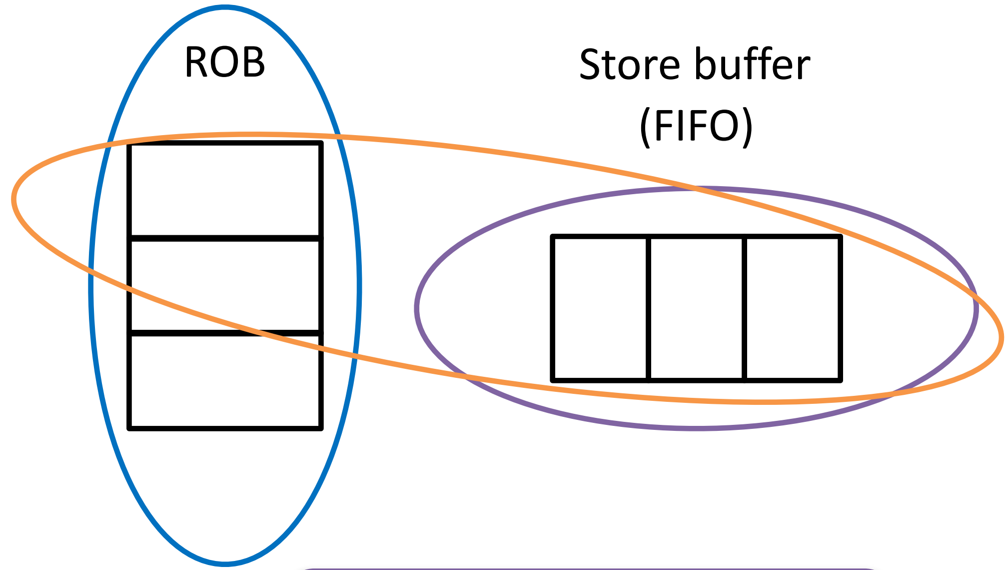
Wr A	Miss
Rd B	Complete
Wr C	Prefetch
Rd D	Miss
Rd E	Not fetched

- Key Idea: Perform load speculatively, use branch rewind to roll back if the value of the load changes
  - Invalidation messages “snoop” load-store queue
    - If invalidation “hits” a complete load, rewind & re-execute
    - Alternative implementation – redo all loads in program order at retirement (“Value”-based ordering) [Cain & Lipasti 04]

SC hardware overhead

Memory operations in pipeline can not be executed out-of-order

Speculative execution of loads in execution window [Gharachorloo et al., 1991]



Loads must wait for store buffer drain

Stores must retire in-order

Several speculative and non-speculative optimizations have addressed this problem

SC-Preserving Store Buffer --- Non-Speculative

OPPORTUNITY

- *Safe* and *Unsafe* accesses
  - Private or read-only shared accesses are *safe*

- No need to enforce memory model constraints for *safe* accesses [Shasha & Snir, 1988; Adve, 1993]

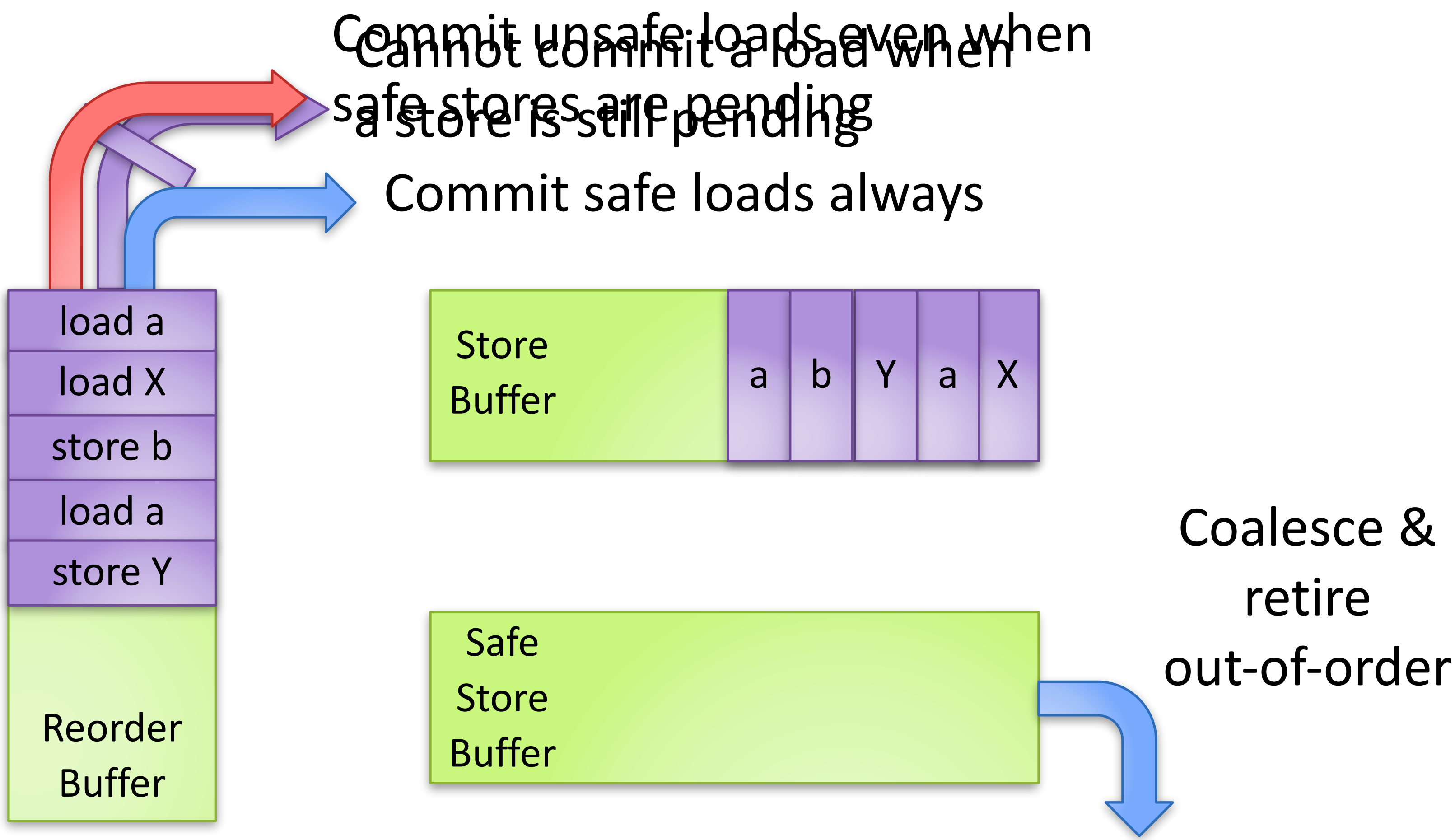
- Large fraction of memory accesses are safe [Hardvellas et al., 2009; Cuesta et al., 2011]

Memory Access Classification

- Two complementary access classification schemes
  - Static compiler analysis
  - Dynamic page protection mechanism

Old Rules Conventional SC	New Rules Access Type Aware SC
Loads must wait for store buffer drain	*Unsafe* loads must wait for *unsafe* store buffer drain
Stores must retire in-order	*Unsafe* stores must retire in-order

SC Hardware Design



Two Store Buffers: Correctness Challenge

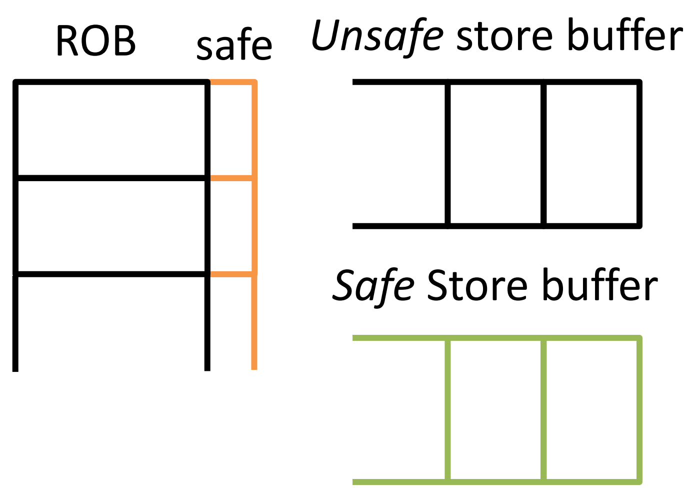
- Uniform-Type assumption
  - All accesses to a memory location are of the same type
- Store-to-load forwarding
  - Safe loads look-up only safe store buffer and *vice-versa*
- Store-to-store program order
  - Stores to a location will be committed into same store buffer
- Challenge:
  - Safe/unsafe classifier may transiently violate Uniform-Type assumption
  - Ensure correct store-to-load and store-to-store semantics

Static Classifier

- Conservative analysis to identify *safe* accesses
  – Non-escaping function locals, temporaries, and literals

- ISA is extended to indicate type

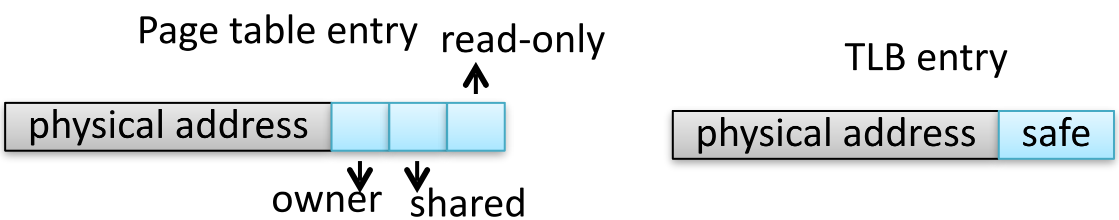
- Safe bit is set at decode



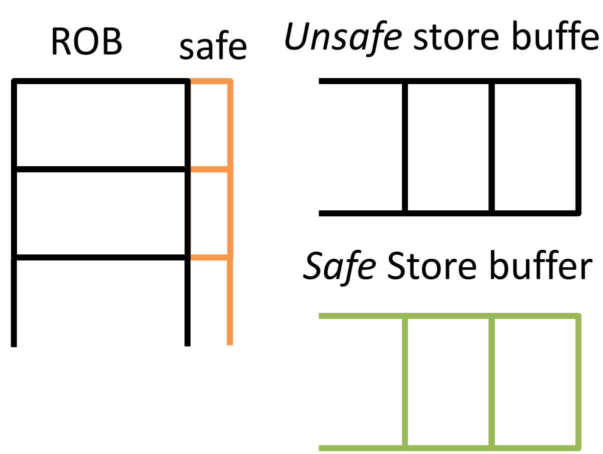
Dynamic Classifier

- Dynamically classify pages as safe or unsafe
  - Extend page protection to thread level
  - Extend TLB to track page type

[Dunlap et al. VEE'08]



Safe bit set during address translation



Dynamic Classifier: Ensuring Correctness

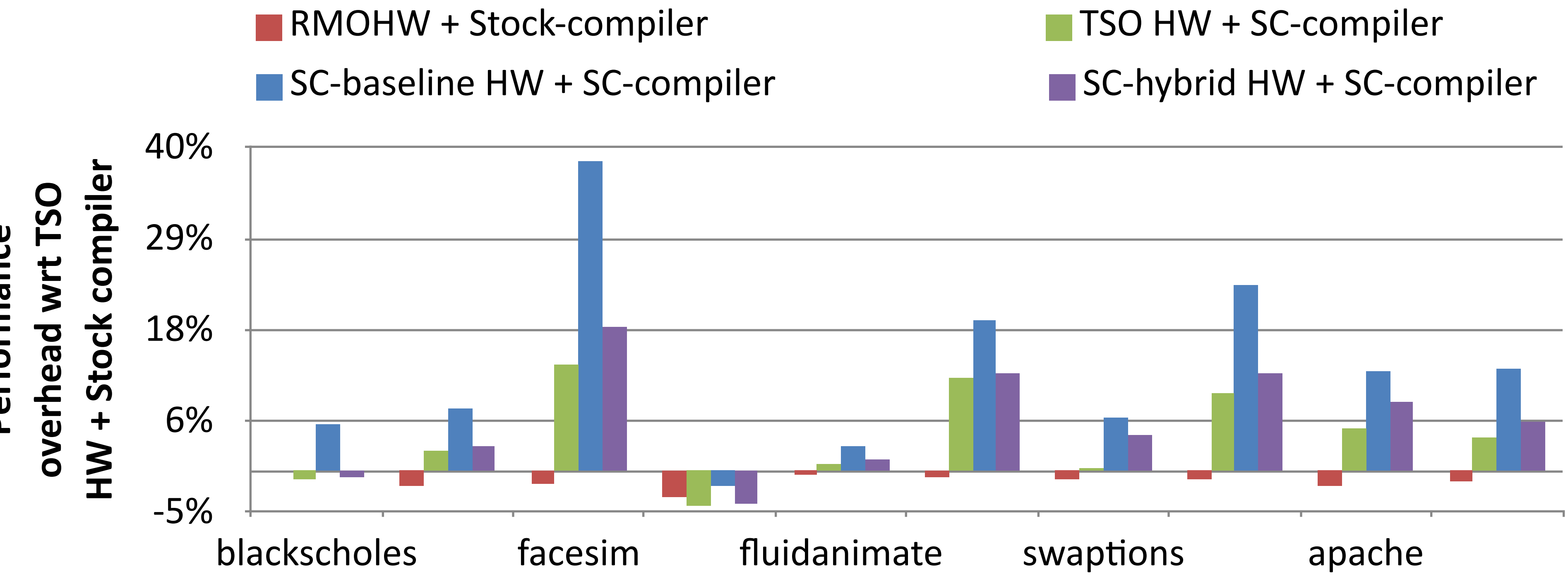
- Problem: A page's type can change from safe to unsafe
  - Access-type assumption may be violated
  - SC may be violated

- Solution:
  - Drain store buffers of processors that last accessed the page

Simulation Methodology

- LLVM compiler extensions
  - SC-preserving
  - Support for static classification
- Hardware simulator
  - Simics based FeS2, x86_64
- Benchmarks:
  - PARSEC, SPLASH-2, Apache web server (SURGE)
- Compare End-To-End SC to
  - Stock LLVM on TSO
  - Stock LLVM on RMO

Cost of End-to-End SC



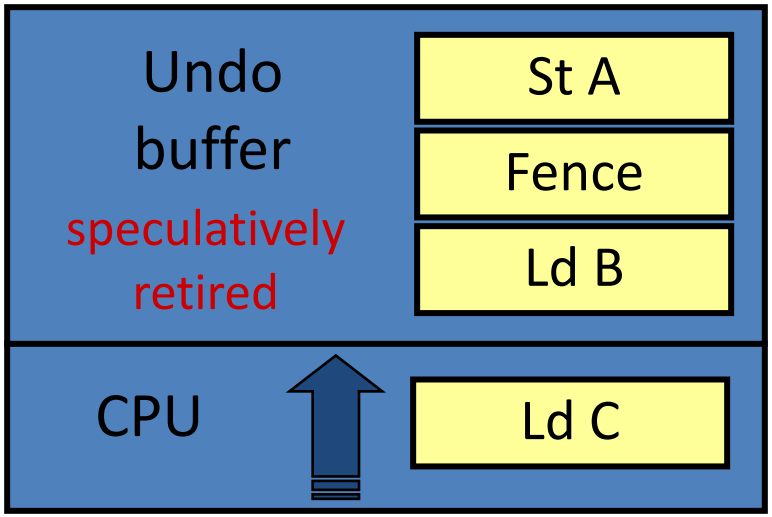
Average performance cost of end-to-end SC is 6.2% w.r.t stock compiler on TSO

End-to-End Sequential Consistency

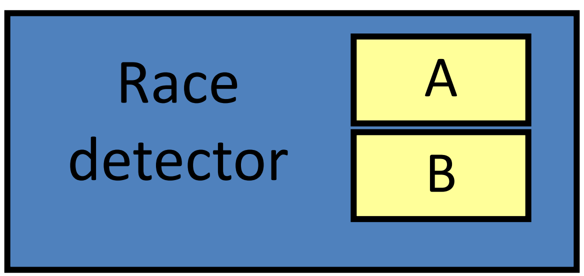
- DRF0: assume a memory access is *safe* by default
- SC: assumes a memory access is *unsafe* by default
- SC-Preserving compiler
  - Optimizations that break SC don't buy much performance
  - Exposing hardware load speculation enables more optimizations
- SC Hardware
  - Identify safe accesses using compiler and OS
  - Relax memory ordering constraints for safe accesses
- Overhead over stock: avg. ~6%

Out-of-Window Speculation
Early HW solutions

[Ranganathan 97] [Gniady 99]



- Log all instructions
  - **Large storage requirement**
- Read old value before store
  - **Extra L1 traffic**



- Assoc. search on external req.
  - **Limited capacity**

Early solutions require impractical mechanisms

InvisiFence

[Blundell et al. ISCA 2009]

- Key departure: apply to weakly-ordered system
  - Straightforward hardware; fewest stalls to address
- Augment with familiar deep speculation mechanisms
  - Violation detection: read/write bits in cache
  - Version management: clean to L2 before 1st write
- Result: eliminate fence stalls (up to 13% speedup)
  - No fine-grained (per-store) tracking
  - Fast & simple commit and rollback
  - Conventional memory system
- For strong ordering: speculate more ("implicit fences")
  - Bonus: can even eliminate LSQ snooping! (a la [Ceze'07])