A Programming-Languages View of Data Races

Dan Grossman
University of Washington

Prepared for the
2012 Microsoft Research Summer School on Concurrency
St. Petersburg, Russia

Goals

- Broad overview of data races
  - What they are [not]
  - Why they complicate language semantics
  - Data-race detection, especially dynamic detection

- Difference between low-level and high-level data races
  - How to use low-level data races to detect high-level data races
  - [Recent work with Benjamin Wood, Luis Ceze: MSPC2010 + under submission]

Meta

- Much of presentation is background
  - Prepared for a summer school
  - Much not my work
  - Will not carefully cite references: better to omit most than cite most ☺
  - There are surely > 100 good papers on data races, which should be accessible after this material

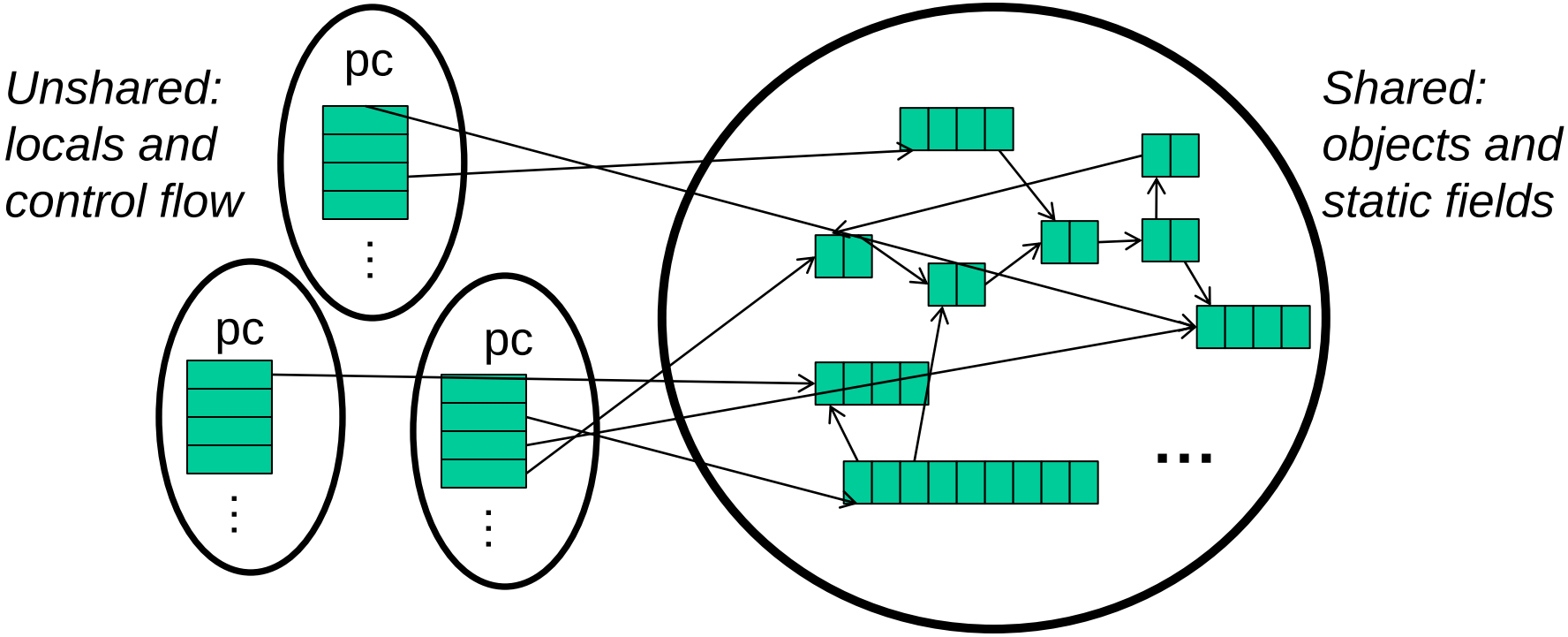
- Some most-important-references listed at end

- Some materials generously adapted from Joseph Devietti, Stephen Freund, Vijay Menon, Hal Perkins, Benjamin Wood

Outline

- What are data races
- Memory-consistency models: why data races muddy semantics
- Approaches to data-race detection
  - Static vs. dynamic
  - Sound vs. complete vs. both vs. neither
  - Locksets
  - Vector clocks and FastTrack
  - Other recent approaches
- Low-level vs. high-level data races
  - Low-level detection is wrong for detecting high-level data races
  - Abstracting low-level data races to remove this gap

Shared memory



Assume programming model with shared memory and explicit threads

- Technical definition of data races assumes shared memory
- Not claiming shared memory is the best model, but a prevalent model that needs support and understanding

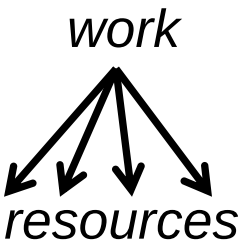
Digression: Why Use Threads?

First distinguish *parallelism* from *concurrency*

- Terms may not yet be standard, but distinction is essential

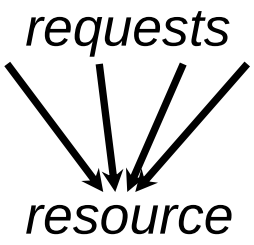
Parallelism:

Use extra resources to solve a problem faster



Concurrency:

Correctly and efficiently manage access to shared resources



An analogy

CS1 idea: A program is like a recipe for a cook

- One cook who does one thing at a time! (*Sequential*)

Parallelism:

- Have lots of potatoes to slice?
- Hire helpers, hand out potatoes and knives
- But too many chefs and you spend all your time coordinating

Concurrency:

- Lots of cooks making different things, but only 4 stove burners
- Want to allow access to all 4 burners, but not cause spills or incorrect burner settings

Back to shared memory…

Natural to confuse parallelism and concurrency:

- Common to use threads to divide work (parallelism) and to provide responsiveness to external events (concurrency)

- If parallel computations need access to shared resources, then the concurrency needs managing
  - Library client thinks parallelism
  - Library implementor thinks concurrency

- Shared memory (which leads to data races) relevant to both:
  - Parallelism: Communicate arguments/results to/from workers
  - Concurrency: Shared resource often resides in memory

Data races, informally

[More formal definition to follow]

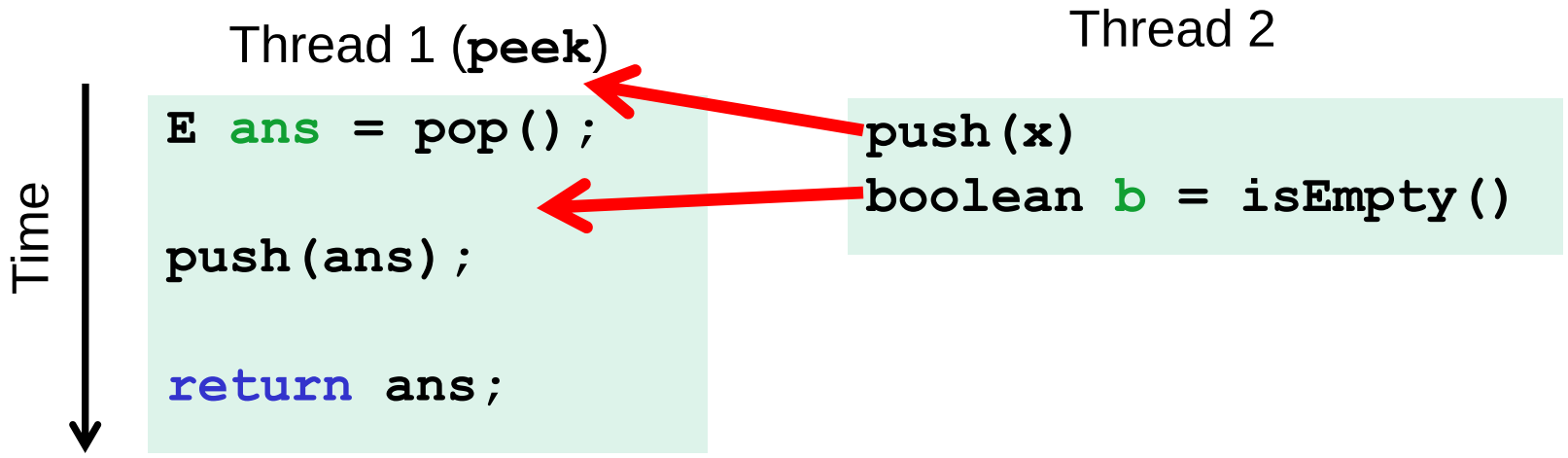
"*race condition*" means two different things

- *Data race:* Two threads read/write, write/read, or write/write the same location without intervening synchronization
  - So two conflicting accesses could happen "at the same time"
  - Better name not used: *simultaneous access error*

- *Bad interleaving:* Application error due to thread scheduling
  - Different order would not produce error
  - A data-race free program can have bad interleavings

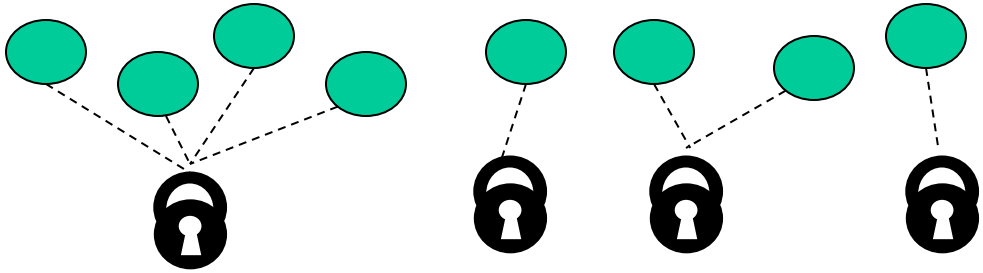
Bad interleaving example

```
class Stack<E> {
  … // state used by isEmpty, push, pop
  synchronized boolean isEmpty() { … }
  synchronized void push(E val) { … }
  synchronized E pop() { … }
  E peek() { // this is wrong
     E ans = pop();
     push(ans);
     return ans;
  }
}
```



Consistent locking

If all mutable, thread-shared memory is *consistently guarded by some lock*, then data races are impossible



But:

- Bad interleavings can remain: programmer must make *critical sections* large enough
- Consistent locking is *sufficient* but not *necessary*
  - A tool detecting consistent-locking violations might report "problems" even if no data races are possible

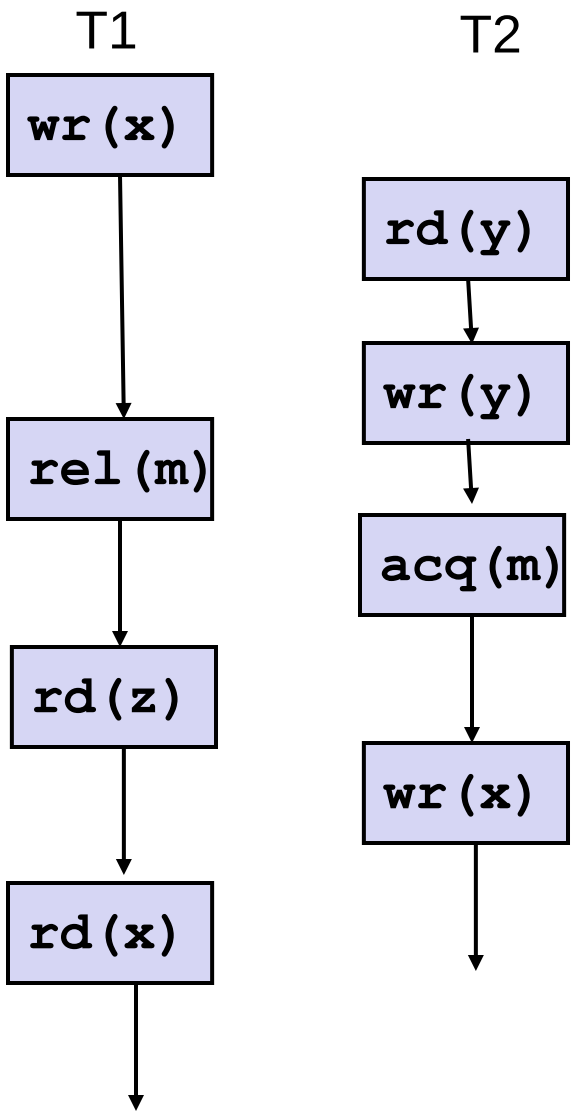
Data races, more formally

Let threads T1, …, Tn perform *actions*:

- *Read* shared location *x*
- *Write* shared location *x*
- [Successfully] *Acquire* lock *m*
- *Release* lock *m*
- Thread-local actions (local variables, control flow, arithmetic)
  - Will ignore these

Order in one thread is *program order*

- Legal orders given by language's single-threaded semantics + reads

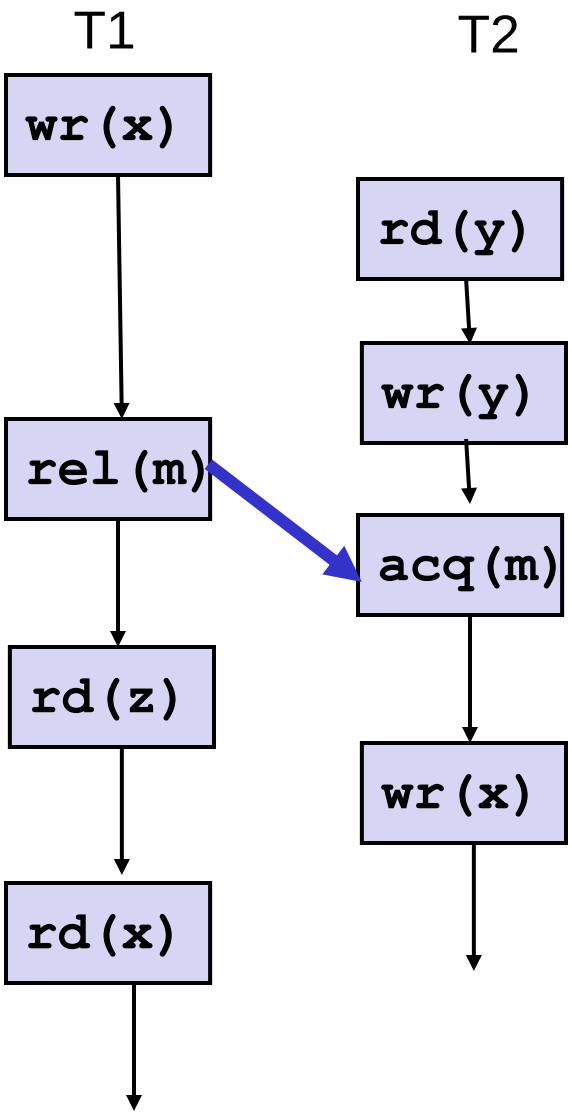


Data races, more formally

Execution [trace] is a partial order over actions *a1* < *a2*

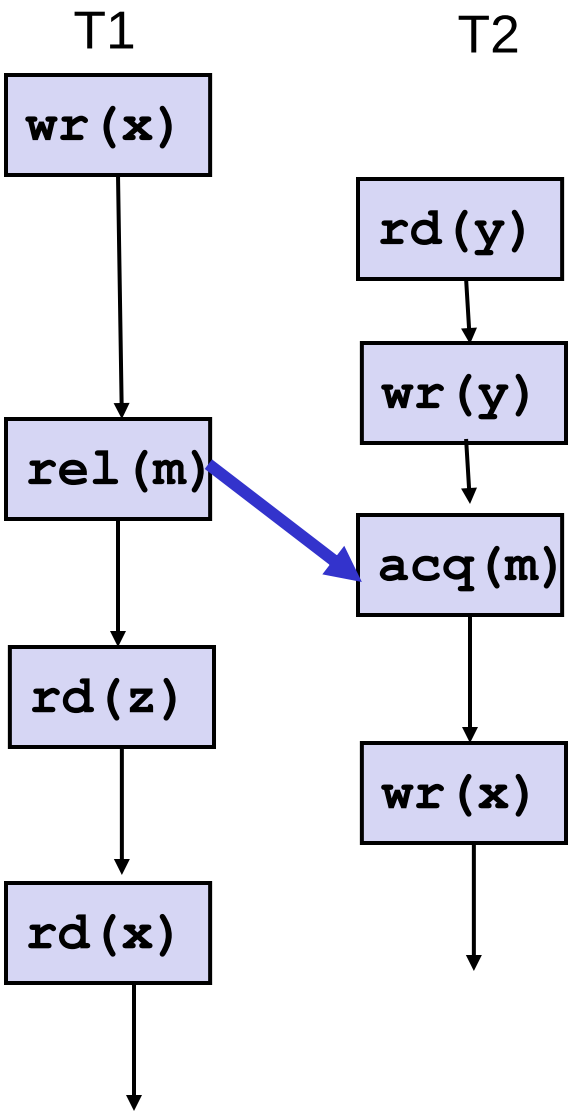
- *Program order:* If Ti performs *a1* before *a2*, then *a1* < *a2*
- *Sync order:* If a2=(Ti acquires m) occurs after a1=(Tj releases m), then *a1* < *a2*
- *Transitivity:* If *a1* < *a2* and *a2* < *a3*, then *a1* < *a3*

Called the *happens-before relation*



Data races, more formally

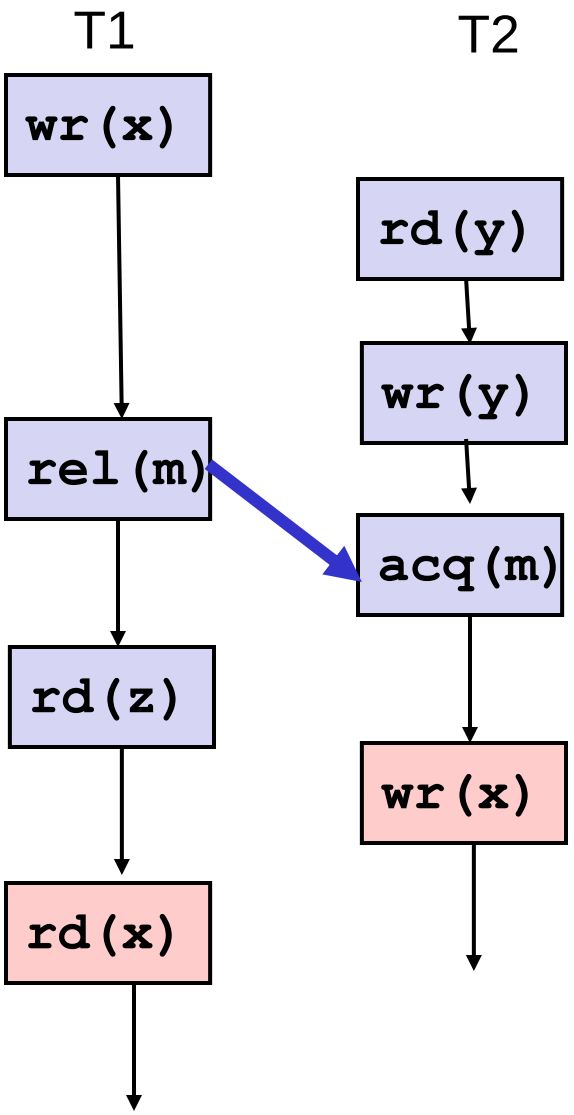
- Two actions conflict if they read/write, write/read, or write/write the same location
  - Different locations not a conflict
  - Read/read not a conflict



Data races, more formally

- Finally, a *data race* is two conflicting actions *a1* and *a2* unordered by the happens-before relation
  - *a1* $\not\leq$ *a2* and *a2* $\not\leq$ *a1*
  - By definition of happens-before, actions will be in different threads
  - By definition of conflicting, will be read/write, write/read, or write/write

- A program is *data-race free* if no trace on any input has a data race



Beyond locks

Notion of data race extends to synchronization other than locks

- Just define happens-before appropriately

Examples:

- Thread fork
- Thread join
- Volatile variables

Outline

- What are data races
- Memory-consistency models: why data races muddy semantics
- Approaches to data-race detection
  - Static vs. dynamic
  - Sound vs. complete vs. both vs. neither
  - Locksets
  - Vector clocks and FastTrack
  - Other recent approaches
- Low-level vs. high-level data races
  - Low-level detection is wrong for detecting high-level data races
  - Abstracting low-level data races to remove this gap

Why care about data races?

Recall not all race conditions are data races…

So why focus on data races?

- One answer: Find some bugs without application-specific knowledge
- More interesting: Semantics for modern languages very *relaxed* for programs with data races
  - Else optimizing compilers and hardware too difficult in practice
    - Our focus: compiler issues
  - Increases importance of writing data-race-free programs

An example

Can the assertion fail?

```
// shared memory
a = 0; b = 0;
```

```
// Thread 1
x = a + b;
y = a;
z = a + b;
assert(z>=y);
```

```
// Thread 2
b = 1;
a = 1;
```

An example

Can the assertion fail?

```
// shared memory
a = 0; b = 0;
```

```
// Thread 1
x = a + b;
y = a;
z = a + b;
assert(z>=y);
```

||

```
// Thread 2
b = 1;
a = 1;
```

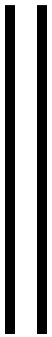
- Argue assertion cannot fail:
  `a` never decreases and `b` is never negative, so `z>=y`
- But argument makes implicit assumptions you *cannot* make in Java, C#, C++, etc. (!)

Common-subexpression elimination

Compilers simplify/optimize code in many ways, e.g.:

```
// shared memory
a = 0; b = 0;
```

```
// Thread 1
x = a + b;
y = a;
z = a + b; x;
assert(z>=y);
```



```
// Thread 2
b = 1;
a = 1;
```

Now assertion can fail

- As though third read of **a** precedes second read of **a**
- Almost every compiler optimization has the effect of reordering/removing/adding memory operations like this

A decision…

```
// shared memory
a = 0; b = 0;
```

```
// Thread 1
x = a + b;
y = a;
z = a + b; x;
assert(z>=y);
```

||

```
// Thread 2
b = 1;
a = 1;
```

Language semantics ***must*** resolve this tension:

- If assertion can fail, the program is wrong
- If assertion cannot fail, the compiler is wrong

Memory-consistency model

- A *memory-consistency model* (or *memory model*) for a shared-memory language specifies *which write a read can see*
  - Essential part of language definition
  - Widely under-appreciated until last several years

- Natural, strong model is *sequential consistency (SC)* [Lamport]
  - Intuitive "interleaving semantics" with a global memory

> "*the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program*"

Considered too strong

- Under SC, compiler is wrong in our example
  - Must disable any optimization that has effect of reordering memory operations [on mutable, thread-shared memory]

- So modern languages do *not* guarantee SC
  - Another reason: Disabling optimization insufficient because the hardware also reorders memory operations unless you use very expensive (10x-100x) instructions

- But still need *some* language semantics to reason about programs…

the grand compromise
великий компромисс
вялікі кампраміс
le grand compromis
die große Kompromiss
il compromesso grande
die groot kompromie

The "grand compromise"

- Basic idea:
  - Guarantee SC only for "*correctly synchronized"* programs [Adve]
  - Rely on programmer to synchronize correctly
  - Correctly synchronized == data-race free (DRF)!

- More precisely:

  If every SC execution of a program *P* has no data races,
  then every execution of *P* is equivalent to an SC execution
  - Notice we use SC to decide if *P* has data races

- Known as "DRF implies SC"

Roles under the compromise

- Programmer: write a DRF program
- Language implementor: provide SC assuming program is DRF

But what if there *is* a data race:

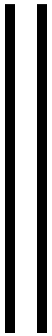
- C++: anything can happen
  - "catch-fire semantics"
  - Just like array-bounds errors, uninitialized data, etc.
- Java/C#: very complicated story
  - Preserve safety/security despite reorderings
  - "DRF implies SC" a theorem about the very-complicated definition

Back to the example

Code has a data race, so program is wrong and compiler is justified

```
// shared memory
a = 0; b = 0;
```

```
// Thread 1
x = a + b;
y = a;
z = a + b; x;
assert(z>=y);
```



```
// Thread 2
b = 1;
a = 1;
```

Back to the example

This version is DRF, so the "optimization" is *illegal*

- Compiler would be wrong: assertion must not fail

```
// shared memory
a = 0; b = 0;
m a lock
```

```
// Thread 1
sync(m){x = a + b;}
sync(m){y = a;}
sync(m){z = a + b;}
assert(z>=y);
```

||

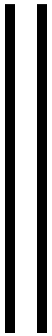
```
// Thread 2
sync(m){b = 1;}
sync(m){a = 1;}
```

Back to the example

This version is DRF, but the optimization is *legal* because it does not affect *observable* behavior: the assertion will not fail

```
// shared memory
a = 0; b = 0;
m a lock
```

```
// Thread 1
sync(m){
 x = a + b;
 y = a;
 z = a + b; x;
}
assert(z>=y);
```



```
// Thread 2
sync(m){
  b = 1;
  a = 1;
}
```

Back to the example

This version is also DRF and the optimization is illegal

- Volatile fields (cf. C++ atomics) exist precisely for writing "clever" code like this (e.g., lock-free data structures)

```
// shared memory
volatile int a, b;
a = 0;
b = 0;
```

```
// Thread 1
x = a + b;
y = a;
z = a + b;
assert(z>=y);
```

||

```
// Thread 2
b = 1;
a = 1;
```

So what is allowed?

How can language implementors know if an optimization obeys "DRF implies SC"? Must be aware of threads! [Boehm]

Basically 2.5 rules suffice, *without* needing inter-thread analysis:

0. Optimization must be legal for single-threaded programs

1. Do not move shared-memory accesses across lock acquires/releases
   - Careful: A callee might do synchronization
   - Can relax this slightly: next slide

2. Never add a memory operation not in the program [Boehm]
   - Seems like it would be strange to do this, but there are some non-strange examples: upcoming slides

Across synchronization operations

- Usual guideline: Do not have effect of moving memory operations across synchronization operations
  - Suffices to understand our example

- In fact, some simple extensions are sound
  - Okay to move memory operations *forward* past *acquires*
  - Okay to move memory operations *backward* past *releases*
  - See work on "interference-free regions"
    - [Effinger-Dean et al, MSPC2010, OOPSLA2012]
    - And older subsumed idea of "roach-motel reordering"

New memory values?

```
// x and y are globals, initially 0
void foo() {  optimized?    void foo() {
  ++x;        ========>       x += 2;
  if(y==1)                    if(y!=1)
    ++x;                        --x;
}                           }
```

What if this code ran concurrently:

```
if(x==2)
  do_evil_things();
```

Transformation is legal in a single-threaded program, but is illegal* because it adds a new write (**2** into **x**) that cannot occur:

* Legal in C++ because there is a data race, but illegal in Java

Trouble with loops

A more subtle example of an illegal* transformation:

- **x** is global memory, **reg** is a (fast) register

```
                        optimized?
for(i=0;i<n;i++)  ========>   reg = x;
  x += a[i];                  for(i=0;i<n;i++)
                                reg += a[i];
                              x = reg;
```

Problematic case: **n==0** and another thread writes to **x** while for-loop executes

- Optimization okay if check **n>0**

*Illegal even in C++ because if **n==0** there are no data races

Outline

- What are data races
- Memory-consistency models: why data races muddy semantics
- Approaches to data-race detection
  - Static vs. dynamic
  - Sound vs. complete vs. both vs. neither
  - Locksets
  - Vector clocks and FastTrack
  - Other recent approaches
- Low-level vs. high-level data races
  - Low-level detection is wrong for detecting high-level data races
  - Abstracting low-level data races to remove this gap

Where we are…

- Done: Technical definition of a data race
- Done: Why programmers must avoid data races

- Now: Trusting programmers to avoid data races is unsatisfying, so would like to detect them
  - ≈ 20 years of research, still ongoing
  - Analogous to research on other programming errors
    - type-checking, array-bounds, …
  - Key design-space dimensions not unique to data-race detection …

Static vs. dynamic

Static data-race detectors analyze a program

- Typically expected to terminate, like type-checking
- Typically do not tune analysis to program inputs: Analyze program for all inputs
- Report pairs of "program points" that can execute data races
  - Line numbers? Plus calling contexts?
- Can purely *infer* results or require programmer *annotations* such as richer types

Static vs. Dynamic

Dynamic data-race detectors observe program executions

- Report data races observed as program runs
- Maintain *metadata* to do detection (time, space)
- Observe only inputs on which detector is run
  - But some detectors attempt *trace generalization*
- When data race occurs, can report as much about program state as desired

Static, considered

- Advantages of static:
  - Results for all inputs, not just inputs tested
  - No run-time overhead
  - Easier to present results in terms of source-code
  - Can adapt well-understood static-analysis techniques
  - Annotations help document (subtle) concurrency invariants
- Disadvantages of static:
  - "Will a data race occur?" is *undecidable*: no terminating procedure can accept any program (in a Turing-complete language) and always correctly identify whether or not there are inputs for which a data race will happen
  - In practice, "false positives" or "false negatives"
  - May be slow and/or require programmer annotations

Dynamic, considered

- Advantages of dynamic:
  - Easier to avoid “false positives”
  - No language extensions or sophisticated static analysis

- Disadvantages of dynamic:
  - Need good test coverage: especially hard with threads because problematic interleavings can be rare
  - Performance overhead: state-of-the-art often 5x-20x
    - Reasonable at most for debugging
  - Harder to interpret results

Design Dimension: Precision

- Sound data-race detector:
  - Every data race is reported (“no false negatives”)
- Complete data-race detector:
  - Every reported data race is a data race (“no false positives”)
- Precise data-race detector is sound and complete
- Some effective detectors are neither sound nor complete
  - For example, use sampling to improve performance
  - Can be precise up to the first data race (if one occurs)

Static vs. dynamic reconsidered

Soundness/completeness mean different things for static and dynamic data-race detectors

- Static: A precise detector would identify all data races possible for any input
  - As noted, this is impossible
  - But sound xor complete is possible: trivial but to be useful try to reduce the *number* of false positives or negatives

- Dynamic: A precise detector identifies all data races that occur on this execution
  - Could miss data races for other inputs or thread schedules
  - Can do some trace generalization

Design dimension: Granularity

- Does the detector distinguish accessing different fields (or array elements) of the same object?
  - If not, then another source of false positives

- For hardware-level detectors, can mean tracking every byte of memory separately
  - Even though most memory not accessed at such *fine-grain*

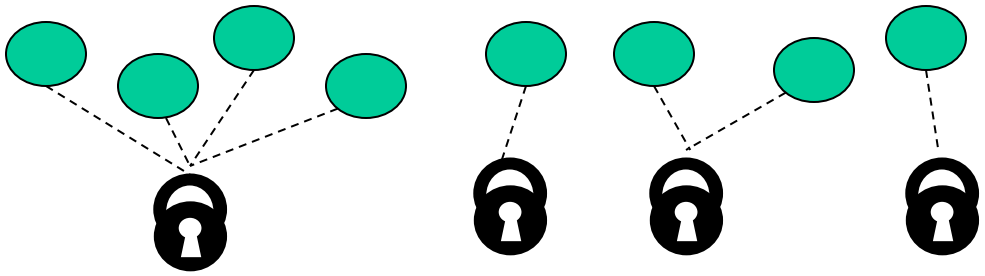
Outline

- What are data races
- Memory-consistency models: why data races muddy semantics
- Approaches to data-race detection
  - Static vs. dynamic
  - Sound vs. complete vs. both vs. neither
  - Locksets
  - Vector clocks and FastTrack
  - Other recent approaches
- Low-level vs. high-level data races
  - Low-level detection is wrong for detecting high-level data races
  - Abstracting low-level data races to remove this gap

Locksets in brief

- A simple approach to data-race detection: Report any violation of consistent locking as a potential data-race source
  - Reports one memory access not two, but still useful

- Recall consistent locking: For all thread-shared mutable memory, there exists a lock that is always held when accessing that memory
  - Partitions shared memory among the locks



Dynamic Lockset detector *[Savage et al]*

- For each shared memory location *x* (e.g., object field), maintain a lockset: LS(*x*)
- For each thread *T*, maintain current locks *T* holds: LS(*T*)
- For each *x*, when memory for *x* is allocated, initialize LS(*x*) to special value “ALL”
- When thread *T* accesses shared memory location *x*:
  - Set LS(*x*) = LS(*x*) ∩ LS(*T*)
    - Where “ALL” ∩ S = S
  - If LS(*x*) = ∅, report error

Sound, incomplete

- Dynamic lockset is sound:
  - If no error reported, then no data race occurred
    - With this schedule on this input
    - Because consistent locking suffices to avoid data races

- Dynamic lockset is incomplete:
  - Because consistent locking is not necessary to avoid data races
  - Example:

```
data  = 0;
ready = false;
```

```
data = 42;
sync(m){
 ready = true;
}
```

||

```
sync(m){
 tmp = ready;
}
if(tmp)
  print(data);
```

Issues and extensions

Many projects improve dynamic lockset approach:

- May not know which memory is thread-local
  - Too many false positives if assume all might be shared
  - Standard shortcut: leave LS(*x*) = "ALL" until second thread accesses *x*, but can introduce (unlikely) false negatives to improve performance
- Recognize immutable data to avoid false positives
  - Recall read/read not a conflict
- Performance (clever set representations)
- Scale to real systems (like OS kernels)
- …

Static lockset *[Abadi, Flanagan, Freund]*

Can analyze source to verify consistent locking

- Most code verifies because complicated concurrency protocols are too difficult to get right
- Forcing type annotations to describe the locking protocol may be "a good thing" for code quality
- Or can try to infer the locking protocol (no annotations)
- Can catch many errors and be "mostly sound," but scaling to full languages (finalizers, reflection, class loading, …) is hard

Pseudocode

```
class SyncedList<T> { // callee locks
  private locked_by<this> Node<T,this> n = null;
  synchronized addToFront(T x) {
    n = new Node<T,this>(x,n);
  }
  synchronized contains(T x) {
    Node<T,this> t = n;
    for(; t!=null; t=t.next)
      if(t.elt.equals(x))
        return true;
    return false;
  }
}
class Node<T,L> { // client locks
  private locked_by<L> T          elt;
  private locked_by<L> Node<T,L> next;
}
```

Key details

- Indicating "content of location A is the lock for location B" is sound only if content of A is not mutated
  - Else can violate consistent locking:

```
locked_by<this> Object foo = new Object();
locked_by<foo>  Bar    x   = new Bar();
…
synchronized(foo) { x.m(); }
foo = new Object();
synchronized(foo) { x.m(); }
```

  - Fix: require fields used in lock-types are **`final`** or **`this`**

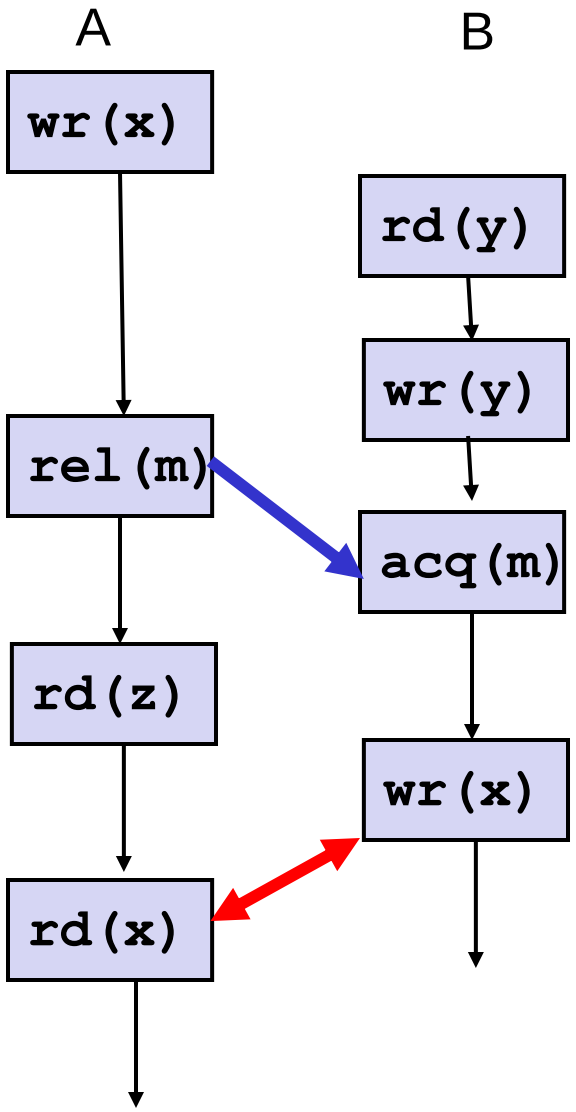
- Methods can indicate locks that must be held by caller

Outline

- What are data races
- Memory-consistency models: why data races muddy semantics
- Approaches to data-race detection
  - Static vs. dynamic
  - Sound vs. complete vs. both vs. neither
  - Locksets
  - Vector clocks and FastTrack
  - Other recent approaches
- Low-level vs. high-level data races
  - Low-level detection is wrong for detecting high-level data races
  - Abstracting low-level data races to remove this gap

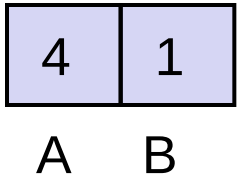
Dynamic happens-before

- For precise (sound and complete) dynamic data-race detection, we could build the full happens-before DAG of a trace and check for data races
  - Program order
  - Sync order
  - Transitivity
- Intractable for long-running programs, but we can use vector clocks to detect all data-race second accesses
  - Won't prove correctness, but will give intuition

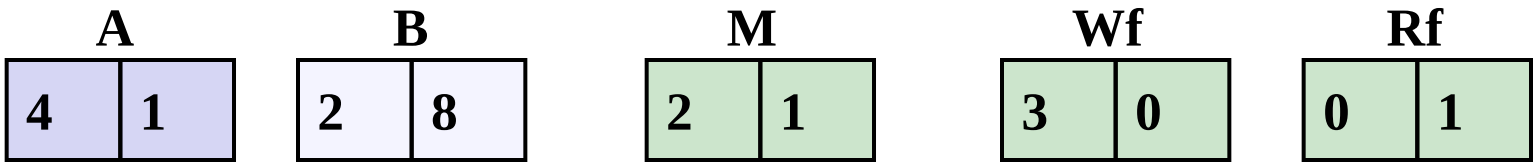


Vector clocks *(as used for data race detection)*

- A thread's "*logical time*" is measured in "*epochs*" where an epoch starts and ends with a lock release
  - Thread X's i^{th} epoch starts with its $(i-1)^{th}$ lock-release and ends with its i^{th} lock-release
- A vector clock records a "logical time" for each thread
  - If there are *N* threads, then each vector clock has *N* entries
  - If the vector-clock entry for thread *X* is *i*, this represents a "time" known to be after or during thread *X*'s i^{th} epoch
- For conciseness, our examples will use just two threads A and B



Lots of vector clocks

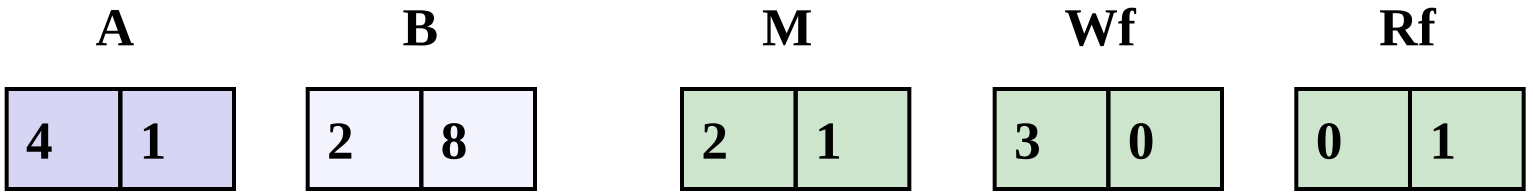


During execution, store a vector clock for:

- Each thread: For thread X's vector clock, if entry Y is i, then Y's i^{th} epoch happens-before X's next instruction
  - Entry *X* is in *X*'s vector clock is *X*'s number of releases – 1 (in odd English, a thread's epoch "happens-before" itself)
- Each lock: Lock M's vector clock holds the logical time when M was most recently released
- Each heap location (read clock): For f's read vector clock, if entry Y is i, then Y's most recent read of f was in Y's i^{th} epoch
- Each heap location (write clock): For f's write vector clock, if entry Y is i, then Y's most recent write of f was in Y's i^{th} epoch

In total

- Total vector-clock space is O(|heap size| * |thread count|)
  - FastTrack will optimize this for common cases later



Now: As program executes, at each step we need to:

1. Check for data races
2. Update vector clocks to avoid future false positives / negatives

Cases for writes, reads, acquires, releases

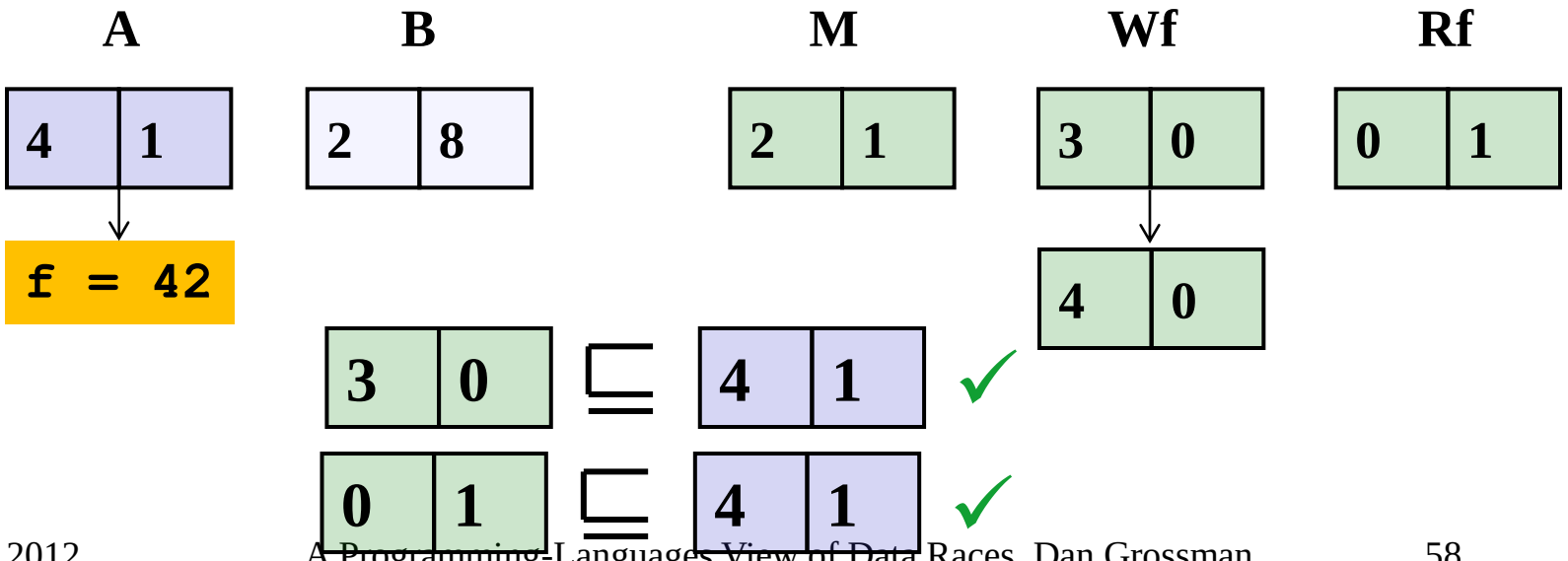
Extra time per program step will be O(|thread count|),

- Again to be optimized

Writes

On write by thread *X* to location *f*:

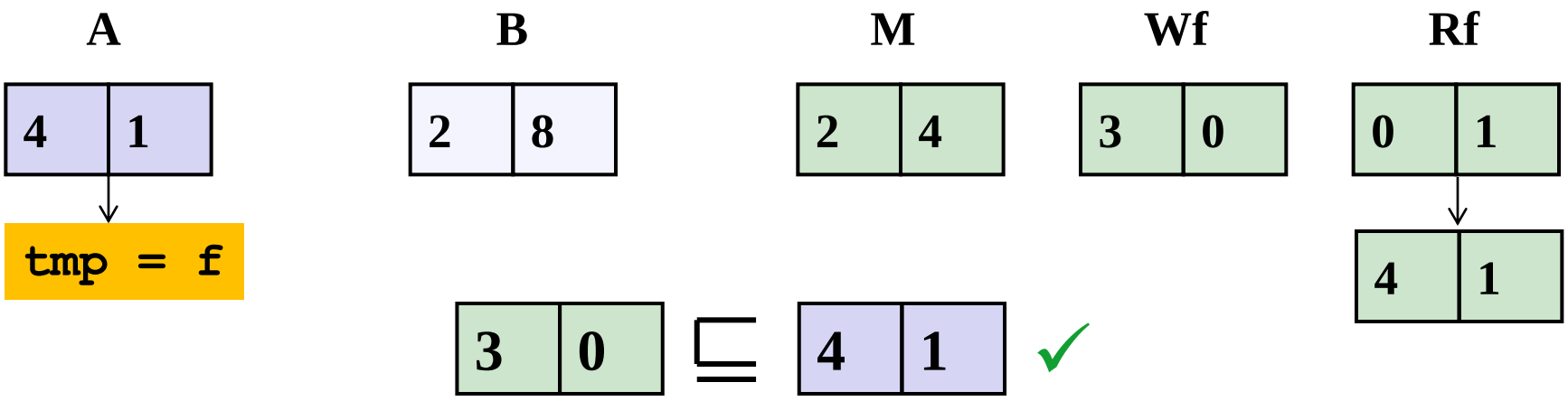
1. Check: most-recent writes *and* reads to location "happen-before" this write
   - vc1 $\sqsubseteq$ vc2 if for all i, vc1[i] $\leq$ vc2[i]
   - If so, all previous writes and reads happen-before, else there is at least one data race
2. Update i^{th} entry of f's write-clock to *X*'s current epoch



Reads

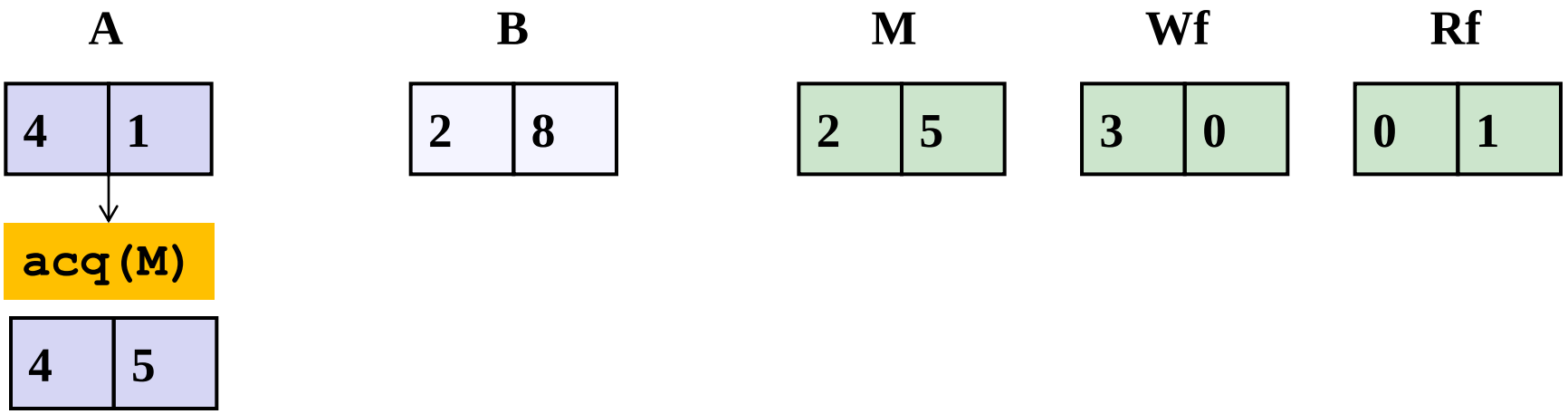
Reads are like writes except:

- Check only against the location's write-clock
  - No read/read conflicts
- Update the read-clock instead of the write-clock



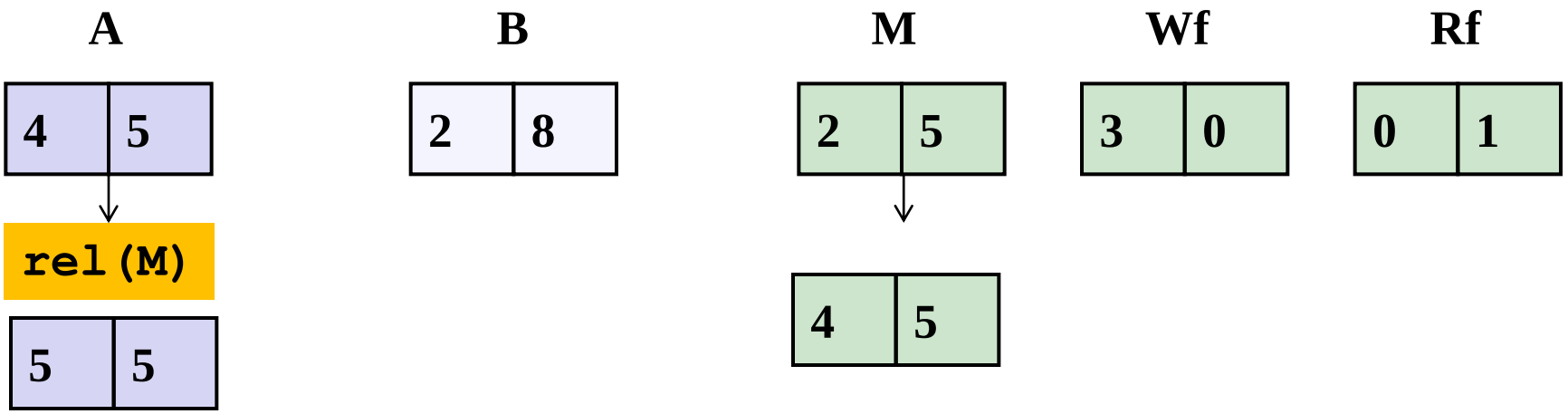
Acquires

- On acquire of *M* by *X*, update *X*'s vector clock to account for what must "happen before" the lock-acquire
  - Set thread *X*'s ith entry to max(*X*[i],*M*[i])

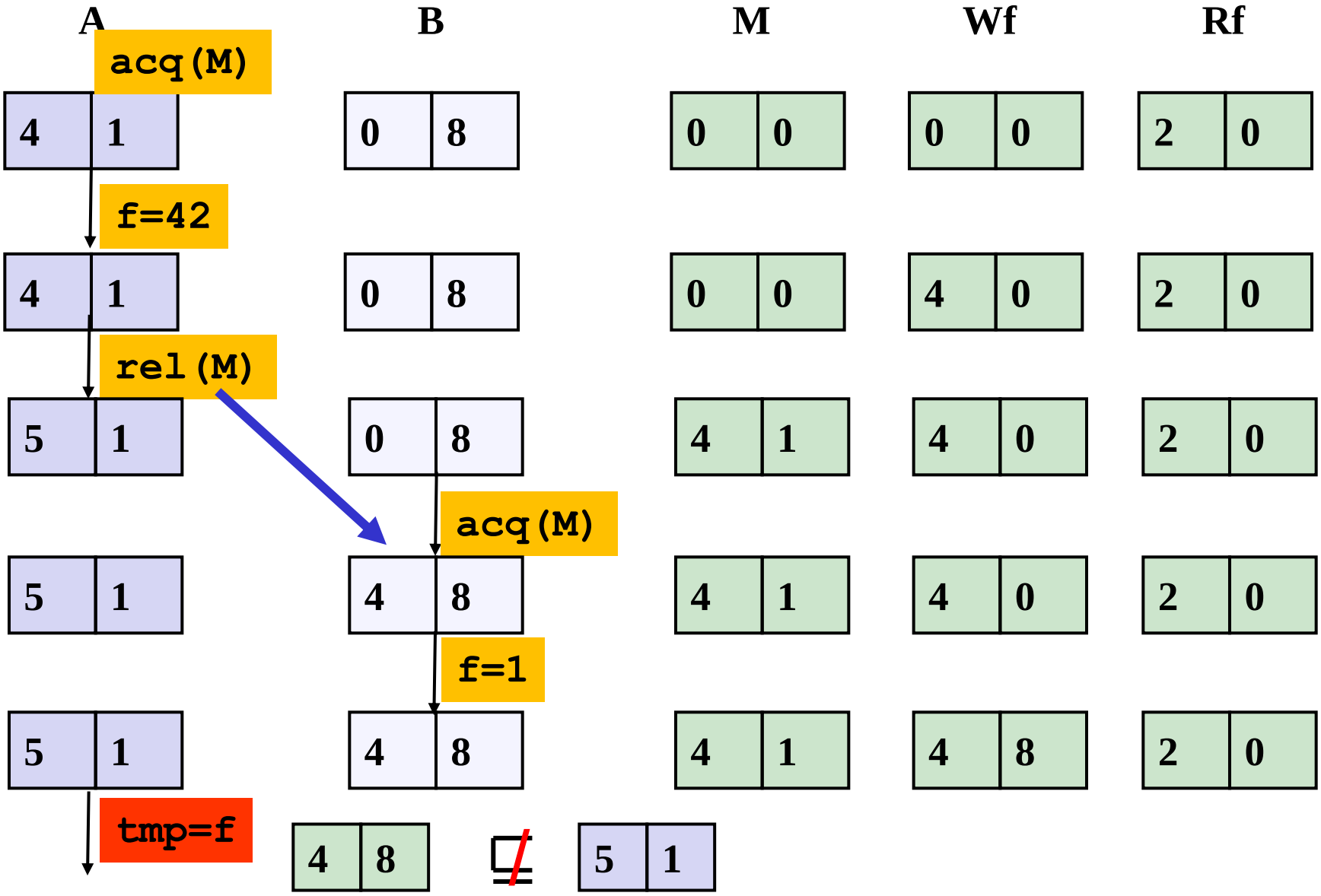


Releases

- On release of *M* by *X*, update vector clock of the lock to account for "*X's* knowledge" and [then] increment *X*'s epoch
  - Copying *all* entries captures transitivity of happens-before
  - Subtle invariant: copying *X*'s clock onto *M*'s clock will not decrease any entry because we updated *X*'s clock when it acquired *M*



Example: Put it all together



It works!

- Complete: If execution is data-race free, every check passes
- Sound: If execution has a data race, a check fails
  - May not report all first-accesses that race with second access
- In theory, slows program by a factor of *O*(thread count)
  - In practice 100x or more
- In theory, extra space of *O*(thread count * heap size)
  - Large in practice too

FastTrack *[Flanagan, Freund, PLDI2010]*

- *FastTrack* lowers time and space overhead by exploiting common cases…
  - Replace most read/write vector clocks with *O*(1) space and most updates/checks in *O*(1) time
  - O(1) improvement happens naturally for all thread-local and consistently locked data
  - Use full vector clocks only as needed

- Same guarantees as full vector clocks up to *first data race on each memory location*
  - More than adequate for testing/debugging

Key idea

- For read/write clocks, *if* all previous reads/writes happen-before the most recent read/write, then store just thread-id and epoch of most recent read/write
  - Number of fields: 2, not |threads|

4@A

- For write clocks, we can *always* do this
  - If there is a previous write that does not happen-before most recent write, report a data race for that location
  - Else latest write comes after all previous: if it does not race with a later access, then neither can any previous one
- For read clocks, we can *usually* do this
  - For thread-local data, all reads in program-order of thread
  - For consistently locked data, all reads in happens-before order

Revised algorithm

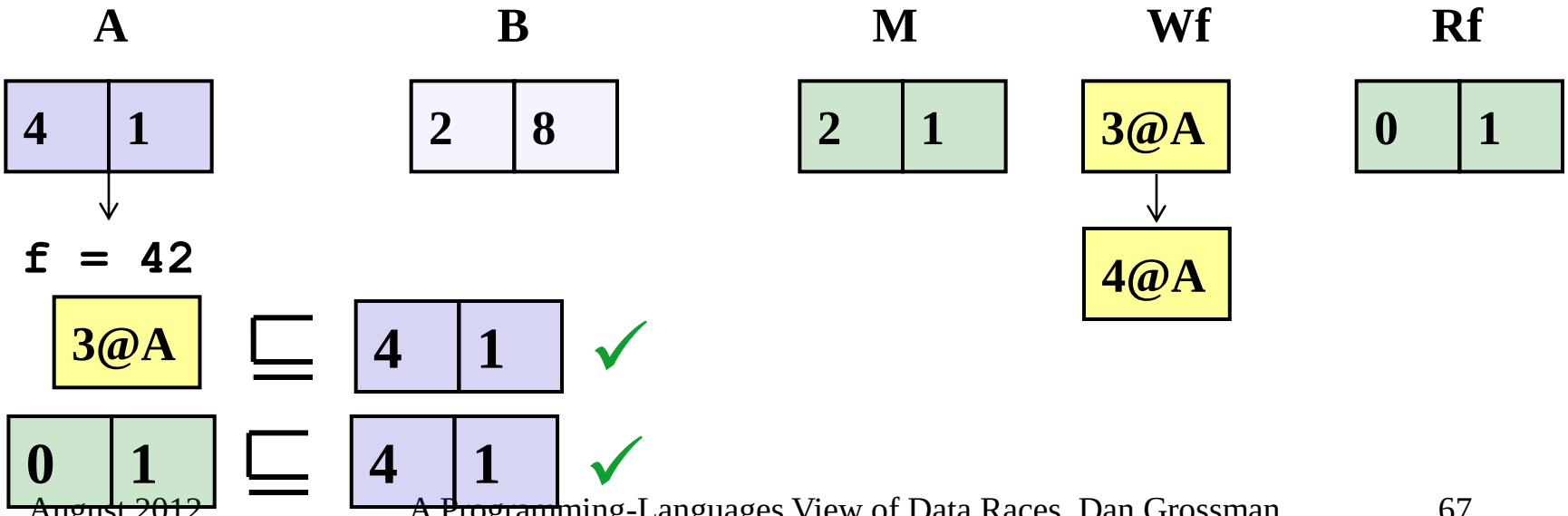
- Acquires and releases unchanged
  - Still maintain full vector clocks for each thread and lock
  - Very few of these compared to size of heap
- For writes, check and update using just thread-and-epoch of last write and the read clock
- For reads, a few cases to switch between compact representation and full vector clock

Writes

On a write by thread *X* to location *f*:

1. Check: most-recent writes *and* reads to location "happen-before" this write
   - vc1 $\sqsubseteq$ vc2 if for all i, vc1[i] <= vc2[i]
   - For compact representation j@T, just check j <= vc1[j]
2. Update of *f*'s write-clock to *X*'s current epoch

Case 1: Rf is a full vector clock

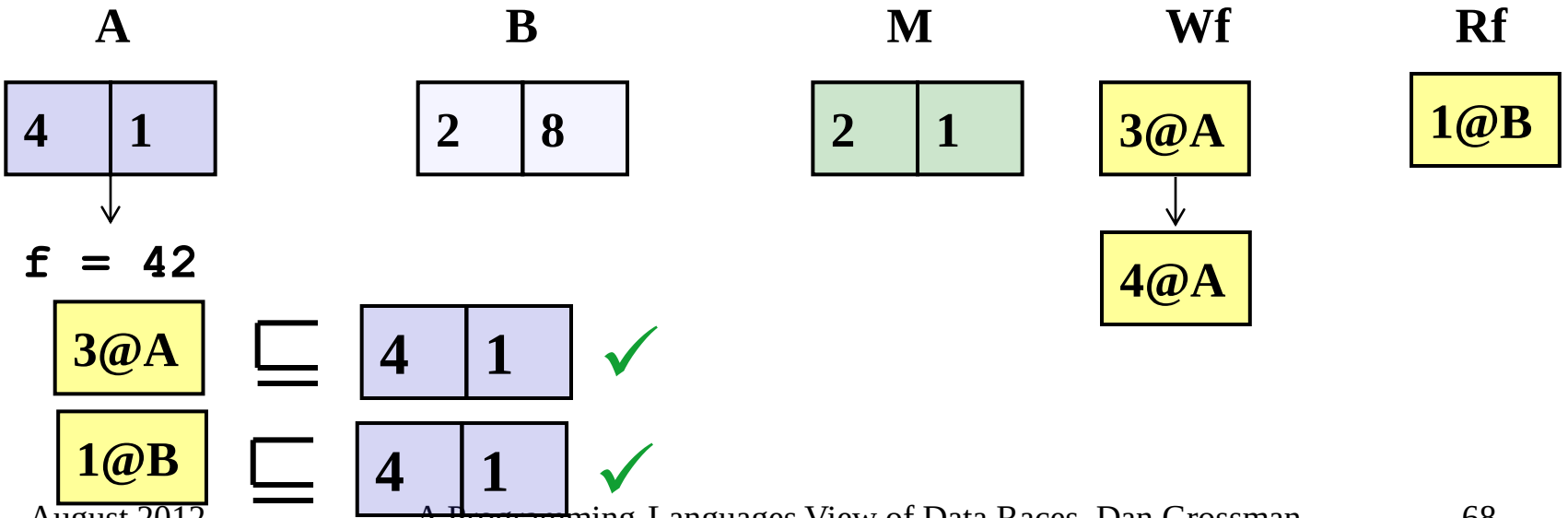


Writes

On a write by thread *X* to location *f*:

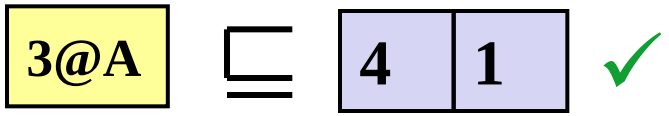
1. Check: most-recent writes *and* reads to location "happen-before" this write
   - vc1 $\sqsubseteq$ vc2 if for all i, vc1[i] <= vc2[i]
   - For compact representation j@T, just check j <= vc1[j]
2. Update of *f*'s write-clock to *X*'s current epoch

Case 2: Rf is compact

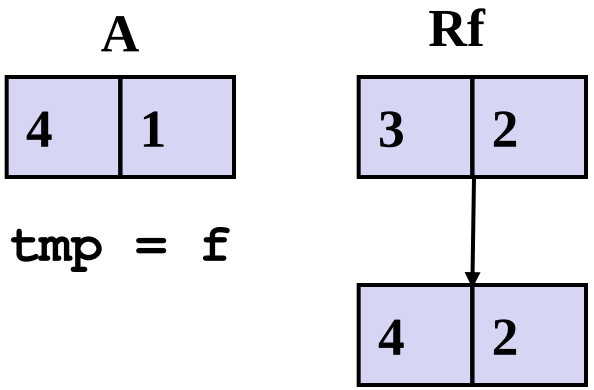


Reads

- As with write case, compare thread's vector clock with last-write:

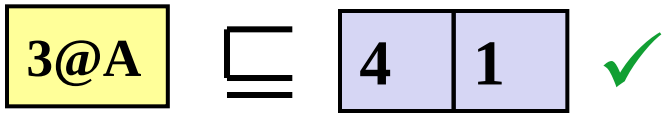


- Now update last-read information (4 cases):
  - Before update, could be compact or full vector clock
  - After update, could be compact or full vector clock

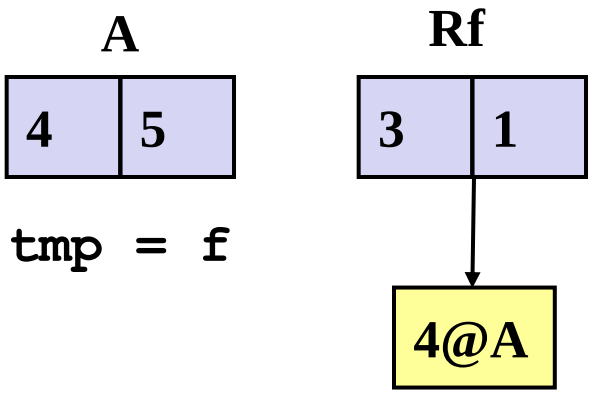


Reads

- As with write case, compare thread's vector clock with last-write:

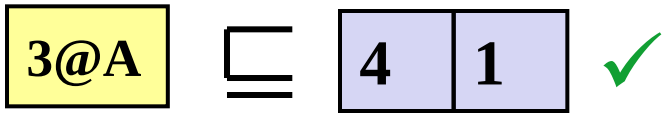


- Now update last-read information (4 cases):
  - Before update, could be compact or full vector clock
  - After update, could be compact or full vector clock

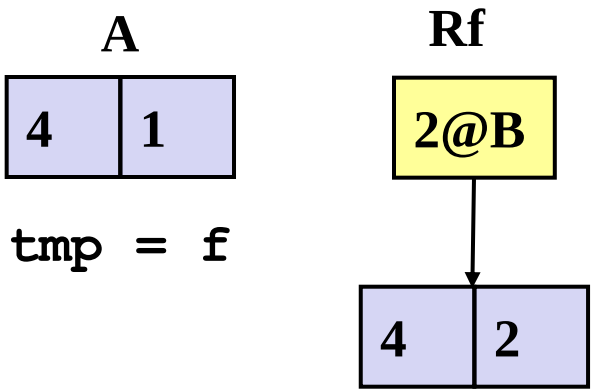


Reads

- As with write case, compare thread's vector clock with last-write:

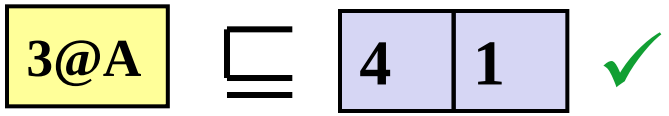


- Now update last-read information (4 cases):
  - Before update, could be compact or full vector clock
  - After update, could be compact or full vector clock

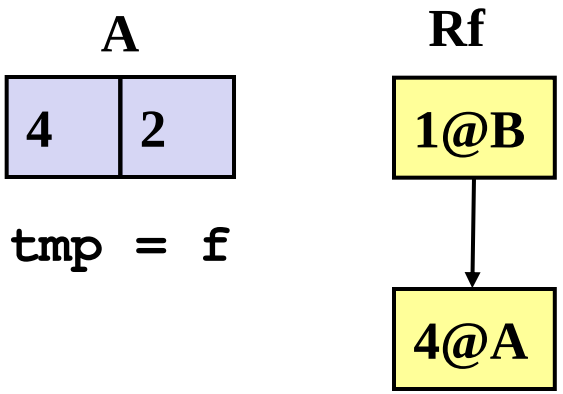


Reads

- As with write case, compare thread's vector clock with last-write:



- Now update last-read information (4 cases):
  - Before update, could be compact or full vector clock
  - After update, could be compact or full vector clock



Common case covers all uses of consistent locking and thread-local data

Summary

- Vector clocks keep enough "recent information" to do precise dynamic data-race detection as the program executes

- FastTrack removes some unnecessary vector-clock elements, getting *O*(1) time and space except when there are reads from multiple threads unordered by happens-before
  - Fundamental improvement in theory and practice

- Much faster but still too slow for deployed software
  - Recent work investigates using hardware support [Devietti et al, ISCA2012] or hypervisor techniques [Olszewski et al, ASPLOS2012] to get within 2x performance in some cases

Outline

- What are data races
- Memory-consistency models: why data races muddy semantics
- Approaches to data-race detection
  - Static vs. dynamic
  - Sound vs. complete vs. both vs. neither
  - Locksets
  - Vector clocks and FastTrack
  - Other recent approaches
- Low-level vs. high-level data races
  - Low-level detection is wrong for detecting high-level data races
  - Abstracting low-level data races to remove this gap

A few other projects

- PACER: proportional sampling:
  - Proportionality: detect any race at rate equal to sampling
  - Inside sampling interval, FastTrack
  - Outside sampling interval, still check “second” accesses, but discard outdated metadata and do not update vector clocks

- LiteRace: more heuristic sampling: neither sound nor complete, but effectively samples “cold” code regions

- More scalable static detectors with fewer false positives:
  - Infer “what locks what” via *conditional must-not aliasing*
  - RELAY: modularize unsoundness and various filters

Outline

- What are data races
- Memory-consistency models: why data races muddy semantics
- Approaches to data-race detection
  - Static vs. dynamic
  - Sound vs. complete vs. both vs. neither
  - Locksets
  - Vector clocks and FastTrack
  - Other recent approaches
- Low-level vs. high-level data races
  - Low-level detection is wrong for detecting high-level data races
  - Abstracting low-level data races to remove this gap

What is memory?

Definition of data race relies on:

- What is a memory location
- What does it mean to read/write a memory location

Definition of memory in Java, C#, etc. is fundamentally different than memory provided to an OS process

- Language-level memory is implemented on top of machine-level memory by the language run-time system

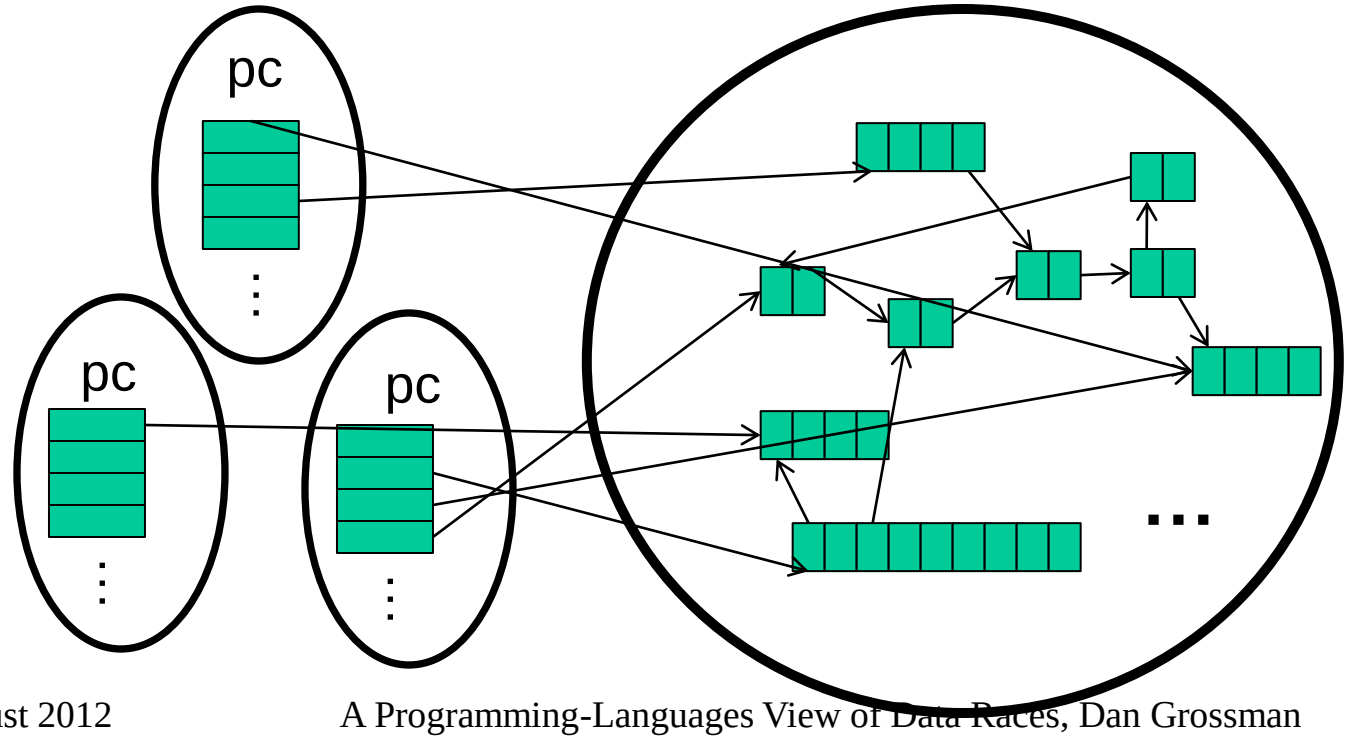
Phenomenon extends to any number of memory levels

- Example: OS has a lower-level view of memory than a running process
- But we will focus on the language-level ("high") and machine-level ("low")

Language-level

Memory in a garbage-collected high-level language:

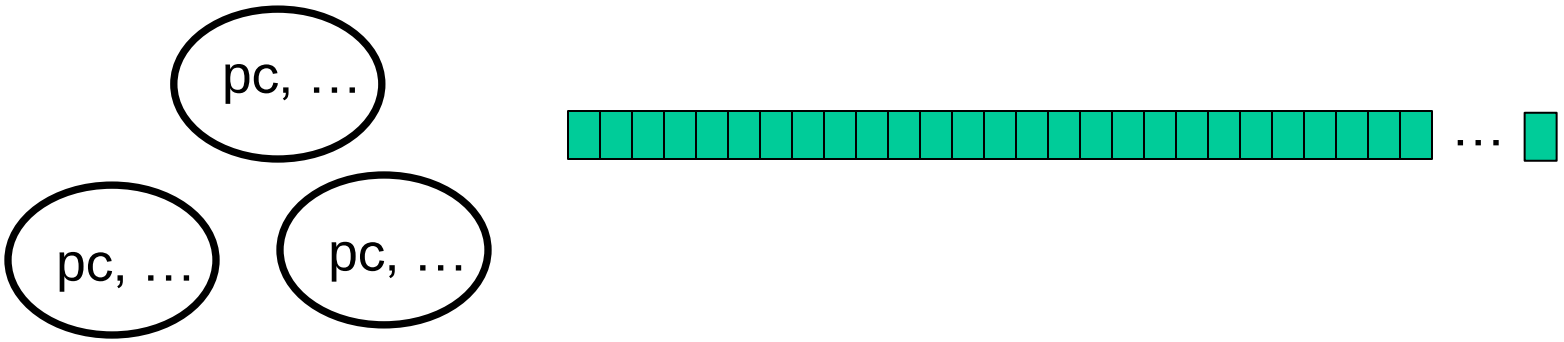
- Logically infinite size (but allocation may fail)
- Memory never observably reused
- Reads/writes are field or array accesses
- Synchronization provided by language primitives



Machine-level

Memory at the binary level of an OS process:

- Logically fixed address space (e.g., 2^{64} bytes)
- No "objects" that get reclaimed, just addresses used for data
- Reads/writes are to elements of "the one global array"
- Synchronization via special instructions like CAS
- Do have notion of distinct thread contexts



The Run-Time System

Language run-time system (garbage collector, systems libraries, etc.), implement the high-level memory on the low-level memory

- Allocate objects
- Reclaim unreachable objects
- Move objects to avoid fragmentation
- Implement synchronization primitives

Compiled code is also relevant

- High-level memory accesses compiled down to low-level ones
  - May include extra memory operations or run-time checks

Connection to data races

- Suppose our goal is a precise dynamic data-race detector for [only] high-level memory (e.g., Java/C# programs)

- Suppose we use a precise dynamic data-race detector for low-level memory
  - Performance advantages
  - Reuse advantages (available for any high-level language)

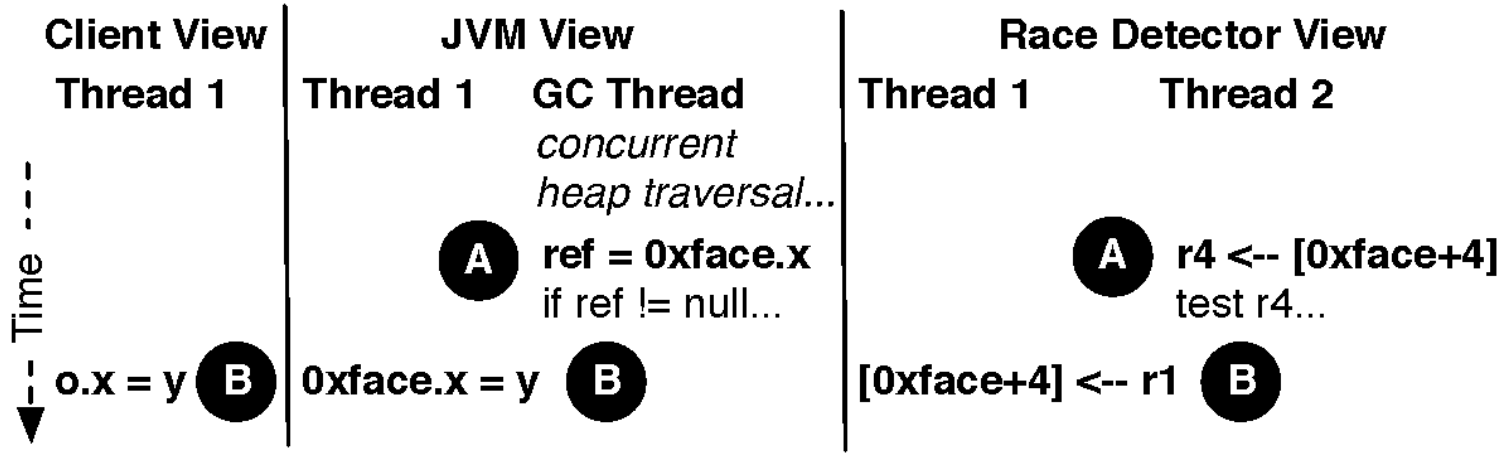
- Unfortunately, it is wrong
  - Will report false positives, in theory and practice
  - Will report false negatives, in theory and practice

  Four fundamental causes of imprecision…

Problem #1: False positives

Data races [by design] in the run-time system or between run-time system and the program are not high-level data races

- Example: Implementation of a lock using spinning
- Example: Clever allocator or parallel garbage collector
- Example: Concurrent garbage collector

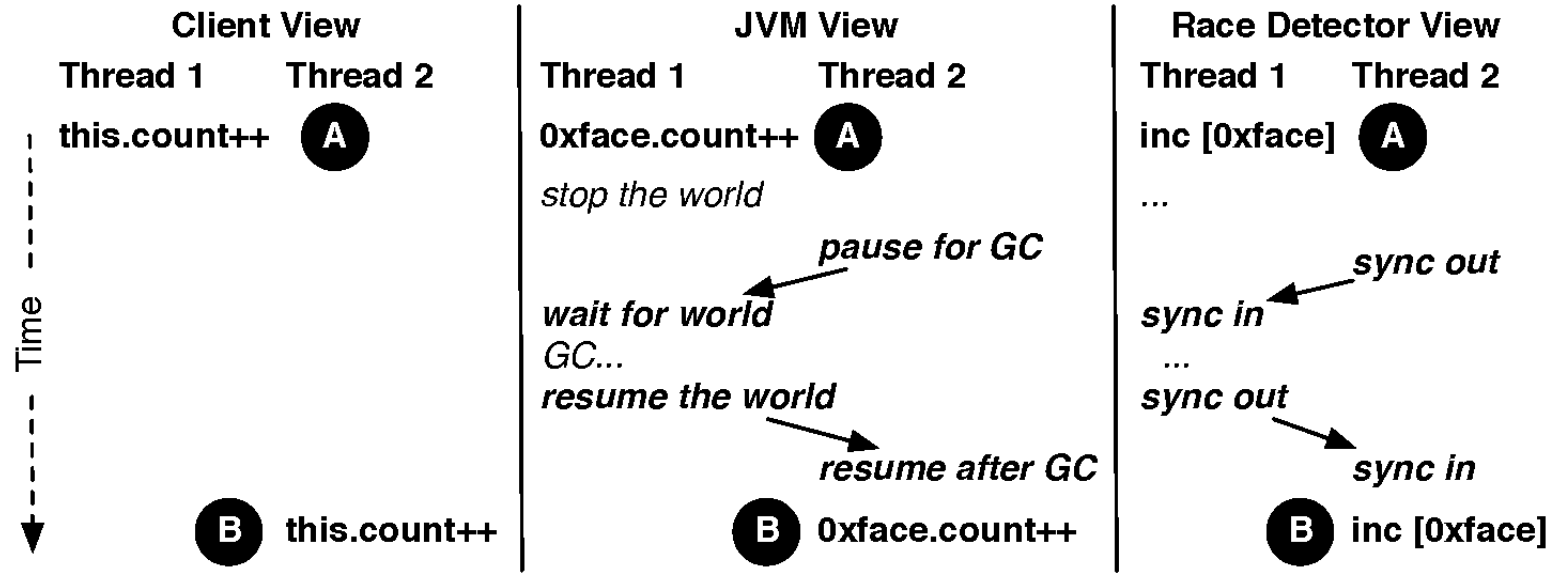


Expert run-time system implementors understand and use low-level memory-consistency model

Problem #2: False negatives

Run-time systems have their own synchronization that should not induce happens-before edges for high-level race detection

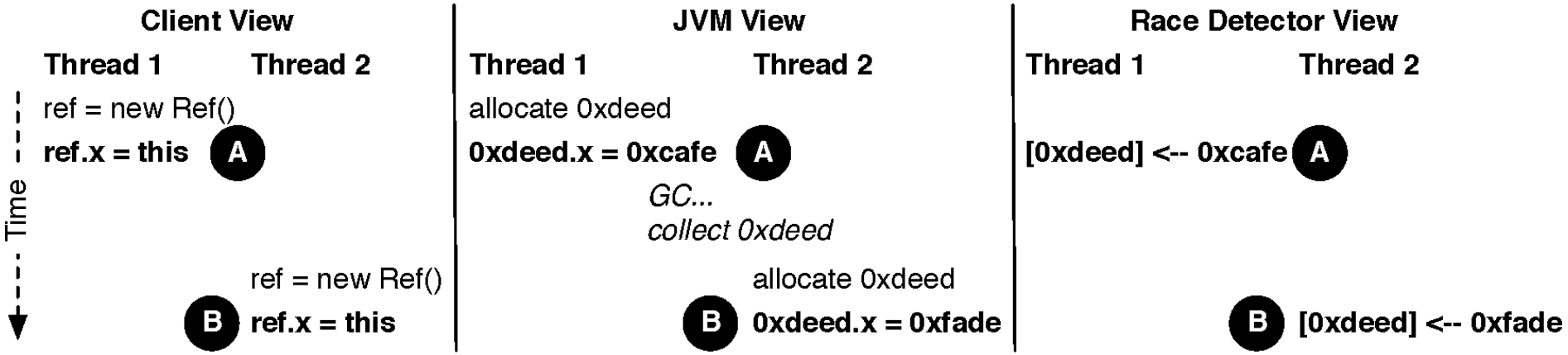
- Example: Allocation from a central memory pool
- Example: "Stop the world" garbage collection



Problem #3: False positives

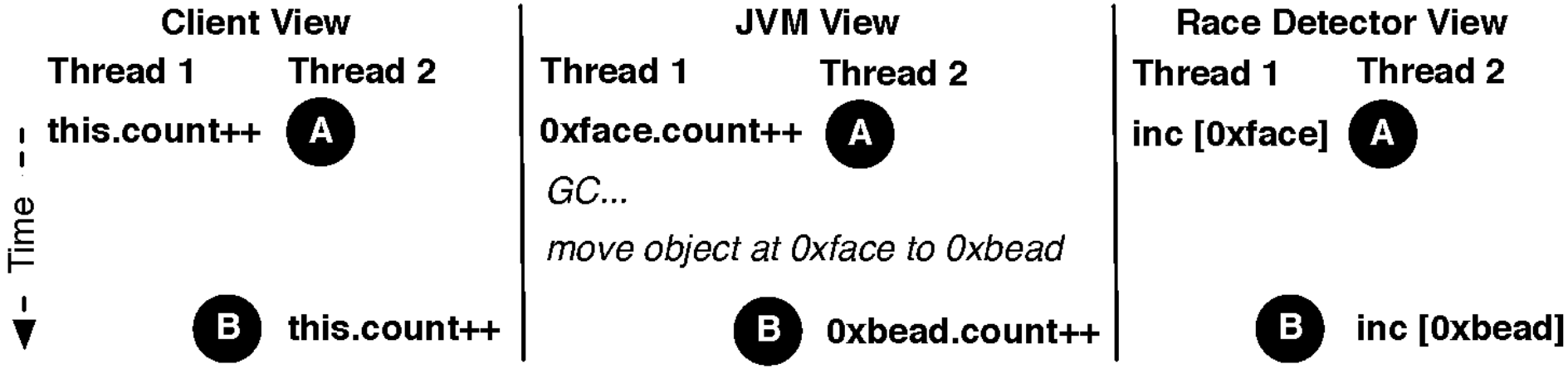
Memory reuse could make accesses to distinct high-level objects appear as unsynchronized races on low-level memory

- Especially after ignoring run-time system internal synchronization to avoid Problem #2



Problem #4: False negatives

Memory movement (for compacting garbage collection) can make high-level conflicting memory accesses go to different low-level addresses



Fixing the problems

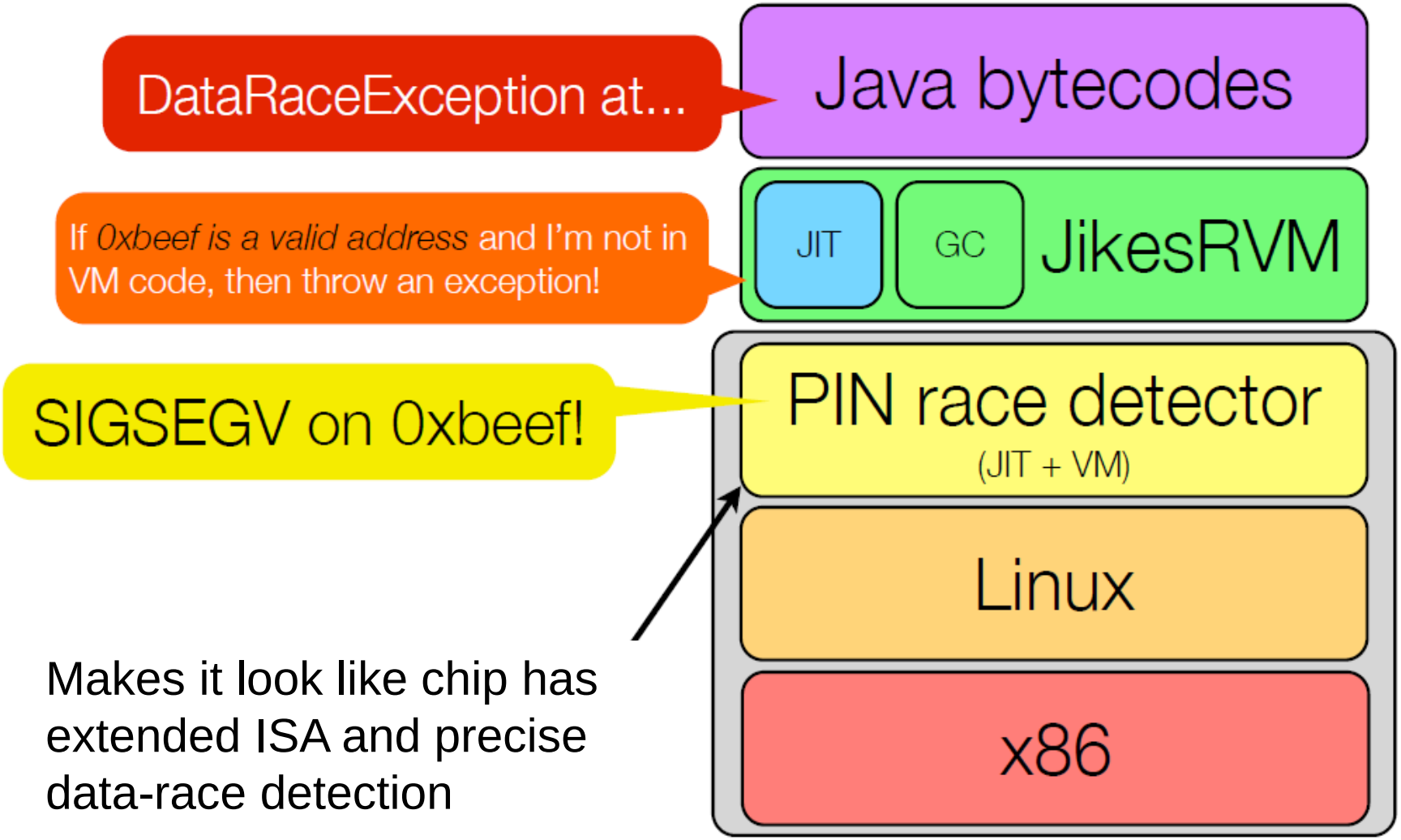
Our solution: Extend low-level data-race detector so the run-time system can control it to perform high-level data-race detection

- But keep low-level performance

Extension points:

1. Versions of load/store that "don't count" as memory accesses for data-race detection
2. Versions of CAS that "don't count" as happens-before edges for data-race detection
3. Instructions to "clear the history" of a memory location (for when object is newly allocated)
4. Instructions to "copy the history" of a memory location to a different location (for moving garbage collectors)

Prototype



Makes it look like chip has extended ISA and precise data-race detection

Reuse / performance

- Changes to JikesRVM to use the low-level race detector:
  - < 1% of lines of code
  - < 0.3% of files

- Prototype data-race detector is slow (simulating hardware), but the overhead from extra JikesRVM work is usually < 50%
  - So *if* we can implement fast low-level data-race detectors, then we can transfer that performance to high-level data-race detectors

But does it work?

Solving the "four problems" sufficient and necessary in practice!

Totals across four DaCapo benchmarks and copying collector:

0. Identifies data races involving 69 program points
1. If don't treat run-time memory accesses as "not counting", false positives involving 1314 program points
2. If don't treat run-time synchronization as "not counting", false negatives involving 67 program points
3. If don't "clear the history" when allocating an object, false positives involving 11 program points
4. If don't "move the history" when moving an object, false negatives involving 12 program points

Outline

- What are data races
- Memory-consistency models: why data races muddy semantics
- Approaches to data-race detection
  - Static vs. dynamic
  - Sound vs. complete vs. both vs. neither
  - Locksets
  - Vector clocks and FastTrack
  - Other recent approaches
- Low-level vs. high-level data races
  - Low-level detection is wrong for detecting high-level data races
  - Abstracting low-level data races to remove this gap

Perspective

I consider the semantics of shared memory in the presence of data races to be "the great failure" of language design and semantics

Options:

- Abandon shared memory (ignore the problem)
- Slow compilers and architectures by 10-100x while providing no benefit to programs that are data-race free
- Detect data races when they occur
- Make data races impossible via static disciplines
- Continue to "trust programmers"
- Other??

On the plus side, much progress in last 20 years, still accelerating

Key References (1/2)

There are > 100 papers – these are just ones I find seminal, excellent background, or especially related to topics focused on in this presentation. Apologies for omissions.

- You Don't Know Jack About Shared Variables or Memory Models, Hans-J. Boehm, Sarita V. Adve, Communications of the ACM, February 2012.
- S. Adve and H. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. Communication of the ACM, Aug 2010.
- Threads Cannot Be Implemented as a Library, Hans-J. Boehm. PLDI, June 2005.
- FastTrack: efficient and precise dynamic race detection Cormac Flanagan, Stephen N. Freund PLDI, June 2009.
- Types for Safe Locking: Static Race Detection for Java. Martin Abadi, Cormac Flanagan, and Stephen N. Freund. ACM Transactions on Programming Languages and Systems, 2006.
- Data-Race Exceptions Have Benefits Beyond the Memory Model Benjamin P. Wood, Luis Ceze, Dan Grossman. ACM SIGPLAN Workshop on Memory Systems Performance and Correctness, San Jose, CA, June, 2011. [Better more recent paper under submission as of August 2012]
- IFRit: Interference-Free Regions for Dynamic Data-Race Detection. Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, Hans-J. Boehm. OOPSLA, October, 2012.

Key References (2/2)

There are > 100 papers – these are just ones I find seminal, excellent background, or especially related to topics focused on in this presentation. Apologies for omissions.

- RADISH: Always-On Sound and Complete Race Detection in Software and Hardware Joseph Devietti, Benjamin P. Wood, Karin Strauss, Luis Ceze, Dan Grossman, Shaz Qadeer. ISCA, June, 2012.
- M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional detection of data races. PLDI, June 2010.
- T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A race and transaction-aware Java runtime. PLDI, June 2007.
- D. R. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. SOSP, 2003.
- M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. PLDI, June 2006.
- R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In PPoPP, 2003.
- S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. Transactions on Computer Systems, 1997.
- J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. FSE, 2007.

"Advertisement"

I have written a "from the beginning" introduction to parallelism and concurrency for second-year undergraduates

- Good background reading for your younger friends

http://www.cs.washington.edu/homes/djg/teachingMaterials/spac/

More basic than this presentation

- Much less on data races
- More on parallel algorithms
- More on synchronization mechanisms

Has been used at roughly 10 universities