Concurrency in weak memory models

Andreas Lööw

Chalmers, 4th year PhD student, formal methods (= mathematical reasoning about software and hardware)

Aim of lecture

Memory-model-related differences between programming in

- ”theoretical” languages/modelling languages like Promela and
- ”practical” languages like Java

Lecture is both Java specific and not Java specific: Java used as an example of a language with a “weak memory model”, but at least (subsets of) C and C++ similar

Outline

- What are memory models?
- Why weak memory models?
- Something about the Java memory model (as an example of a weak memory model)
- Programming in the Java memory model

Outline

- What are memory models?
- Why weak memory models?
- Something about the Java memory model (as an example of a weak memory model)
- Programming in the Java memory model

Java has a ”weak memory model”

- Memory model part of language semantics (what programs mean)
- Different memory models exist
- Promela offers sequential consistency (SC), one of the ”strongest” memory models
- Java offers the Java memory model (JMM), one particular ”weak” memory model

OK… but what is a memory model?

- More or less: Semantics of shared variables (and synchronization)
- Consider the question: What values are variable reads allowed to return?
- ???

Reading variables: Sequential programming

- Obvious answer: The latest value we wrote to the variable

```
int x = 0, y = 0;
x = 1;
y = 1;
print(y); // will obviously print 1
print(x); // again, prints 1
```

Reading variables: Concurrent programming

```
int x = 0, y = 0;

t1 {
  x = 1;
  y = 1;
}

t2 {
  print(y);
  print(x);
}
```

Simple! Just consider the non-deterministic interleavings!

E.g., t1 completes before t2:

```
1
1
```

Or, other interleaving:

```
0
1
```

Reading variables: Concurrent programming

```
int x = 0, y = 0;

t1 {
  x = 1;
  y = 1;
}

t2 {
  print(y);
  print(x);
}
```

But can we print the following?

```
1
0
```

Depends on memory model:

- Sequential consistency: No!
- Weak memory model: Maybe! The program contains a data race.

Reading variables: Sequential consistency (SC)

```
int x = 0, y = 0;

t1 {
  x = 1;
  y = 1;
}

t2 {
  print(y);
  print(x);
}
```

Interleaving-based semantics

Possible execution according to SC:

- `x = 0; y = 0;`
- `x = 1;`
- `print(y);`
- `y = 1;`
- `print(x);`

Reading variables: Sequential consistency (SC)

```
int x = 0, y = 0;

t1 {
  x = 1;
  y = 1;
}

t2 {
  print(y);
  print(x);
}
```

Interleaving-based semantics

Another possible execution according to SC:

- `x = 0; y = 0;`
- `print(y);`
- `print(x);`
- `x = 1;`
- `y = 1;`

Reading variables: Sequential consistency (SC)

```
int x = 0, y = 0;

t1 {
  x = 1;
  y = 1;
}

t2 {
  print(y);
  print(x);
}
```

Notice how ”program order” is always maintained in CS.

E.g., `x = 1` always before `y = 1` in any interleaving

So will not see

```
1
0
```

But not guarateed by some weak memory models!

Reading variables: Weak memory models

```
int x = 0, y = 0;

t1 {
  x = 1;
  y = 1;
}

t2 {
  print(y);
  print(x);
}
```

”Interleaving-based semantics” in some sense the ”obvious” semantics for concurrency

Why make things more difficult?
Why give up program order and other nice things?

Because: SC costs too much

Outline

- What are memory models?
- Why weak memory models?
- Something about the Java memory model (as an example of a weak memory model)
- Programming in the Java memory model

SC cost 1: Prohibits (too many) compiler optimizations

- Aaaaah!!! Messiness! Real-world things! In (e.g.) Promela we do not have to consider ugliness such as compiler ”details” etc.

- For some compiler optimizations we want to reorder writes and reads to variables. (For whatever reason: Might improve register allocation or anything.)

SC cost 1: Prohibits (too many) compiler optimizations

- E.g., the optimization to the right "semantics preserving" in sequential setting if we only consider final state of program
- Not equivalent if we can inspect program under execution, which we can if x and y are shared variables in a concurrent setting
- Breaks illusion of "program order"!

Unoptimized:

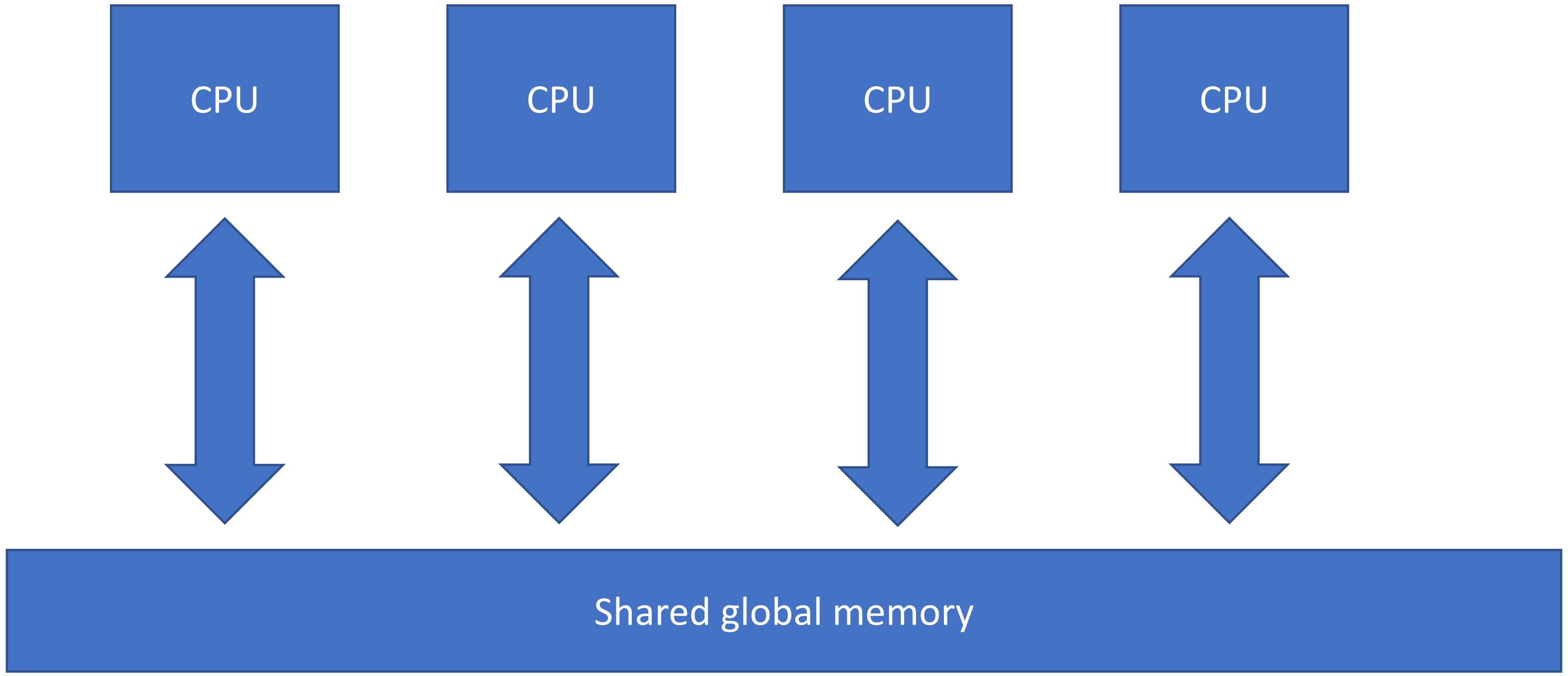
```
x = 1;
y = 2;
z = x + y; // x = 1, y = 2, z = 3
```

"Optimized":

```
y = 2;
x = 1;
z = x + y; // x = 1, y = 2, z = 3
```

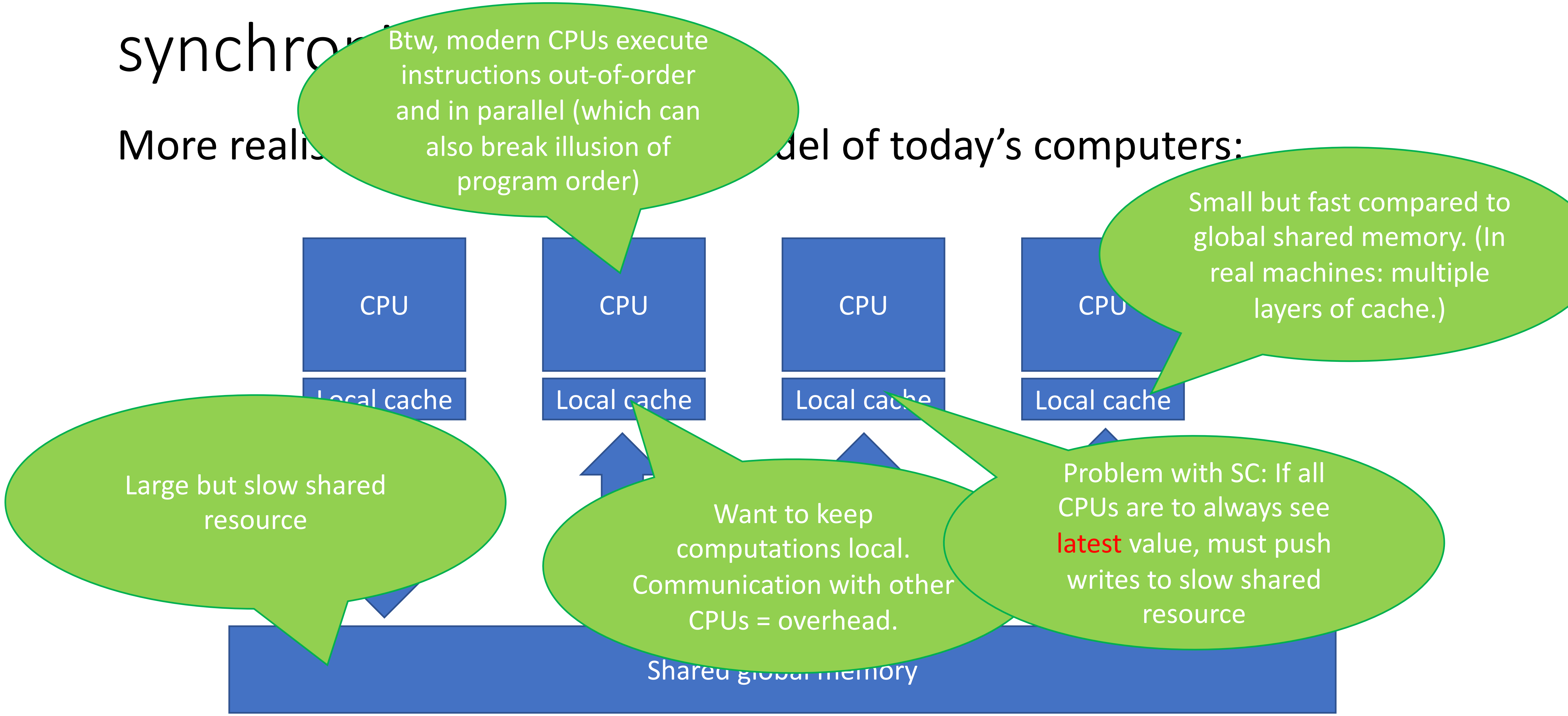
SC cost 2: Causes too much cache synchronization

Cost of SC not obvious with too simplified machine models:



SC cost 2: Causes too much cache synchro

More realis del of today's computers:


Btw, modern CPUs execute instructions out-of-order and in parallel (which can also break illusion of program order)
Small but fast compared to global shared memory. (In real machines: multiple layers of cache.)
CPU
CPU
CPU
CPU
Local cache
Local cache
Local cache
Local cache
Large but slow shared resource
Want to keep computations local. Communication with other CPUs = overhead.
Problem with SC: If all CPUs are to always see latest value, must push writes to slow shared resource
Shared global memory

Why not SC: Summary

Not a complete list of reasons, just two examples!

Anyhow, if all of that was too complicated:
SC too expensive in many situations

Solution to mentioned problems: Relax/remove some guarantees offered by SC → we get weak(er) memory models

Weaker memory models (potentially) more performant, but more difficult to program in

Outline

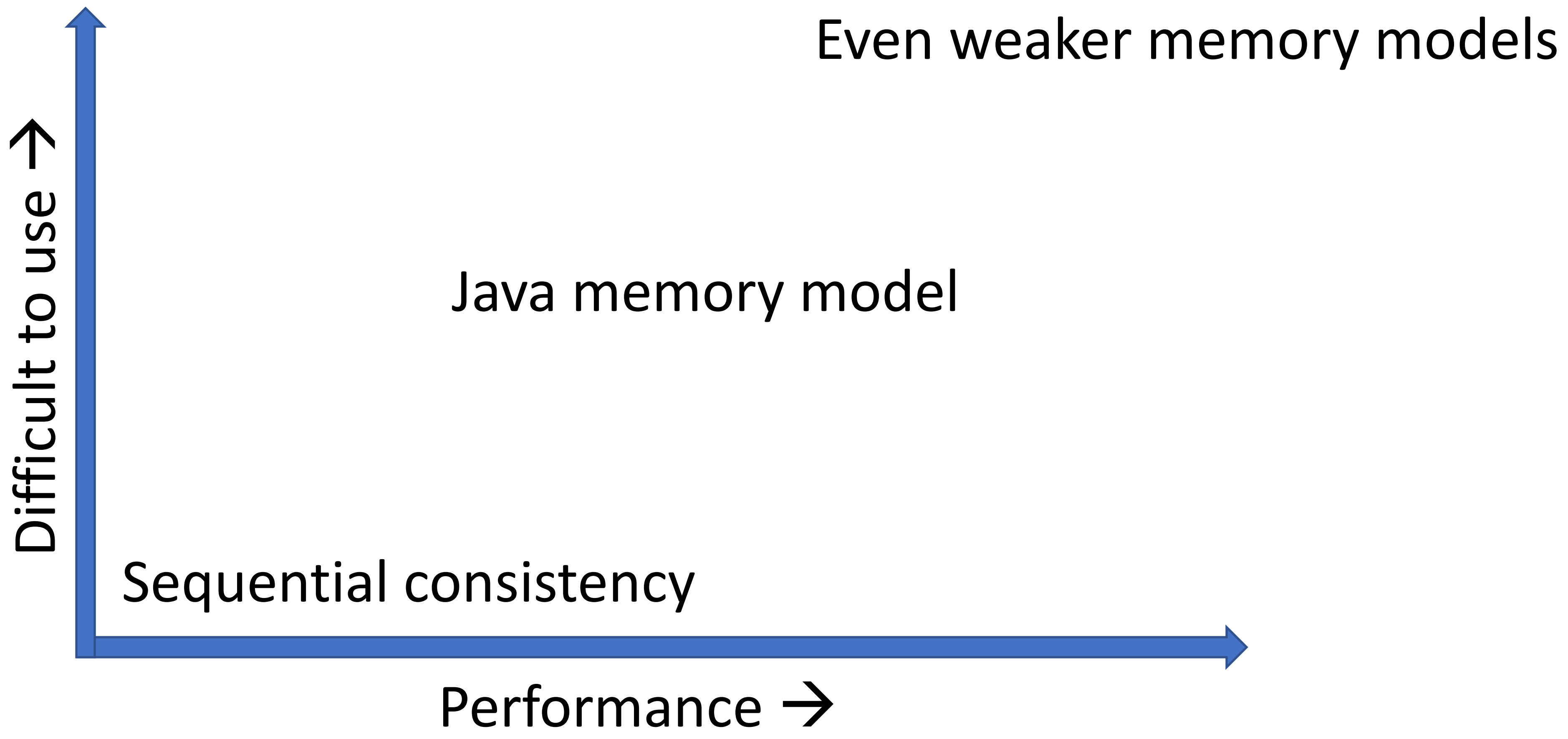
- What are memory models?
- Why weak memory models?
- Something about the Java memory model (as an example of a weak memory model)
- Programming in the Java memory model (as an example of programming in a weak memory model)

The Java memory model

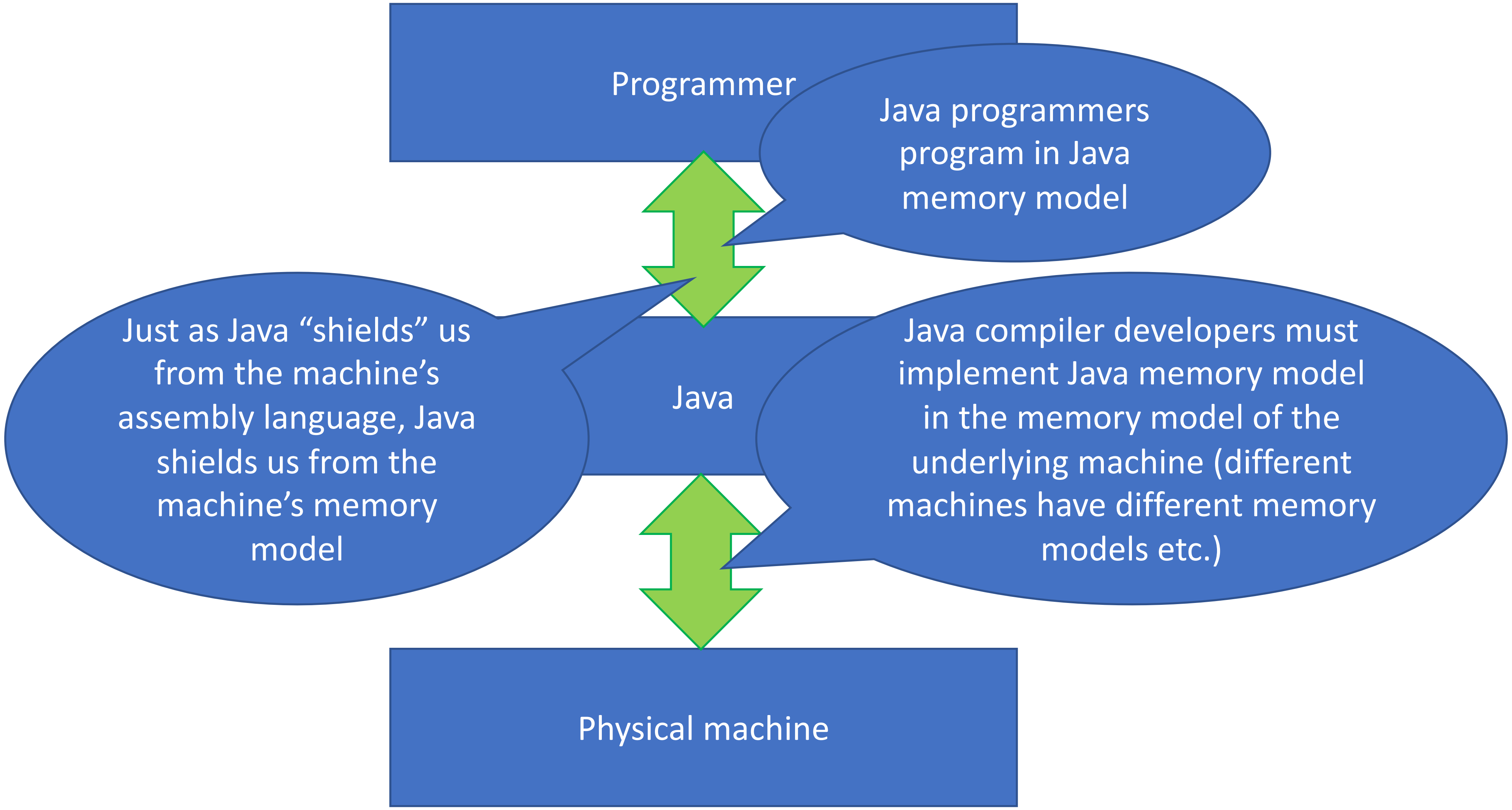
- Less convenient than SC, but implementable on modern machine architectures without too much performance loss

- Opinion: Memory model part of language design, and different coordinates in the design space have different tradeoffs. As with any other language feature: No “right” answer.

Design tradeoff space



More context: Some more machine details


Programmer
Java programmers program in Java memory model
Just as Java “shields” us from the machine’s assembly language, Java shields us from the machine’s memory model
Java
Java compiler developers must implement Java memory model in the memory model of the underlying machine (different machines have different memory models etc.)
Physical machine

SC for data-race-free programs

- A few (C-like) languages have converged at ”sequential consistency for data-race-free programs” memory models. Java included in this family.

- First thought (maybe): Phew!
  - So we have SC after all!
  - We can pretend that all these compiler optimizations (“cost 1”) and hardware implementation details (“cost 2”) do not exist! (At least for correctness, but not for performance…)

- Reasoning principle: If there are no data races (under SC), we can assume SC when reasoning about our program

- Important to remember definition of data race (and difference with race conditions)

Data races

Slight variation of previous definition you seen, to fit Java better:

Def. Two memory accesses are in a data race iff

- they access the same memory location simultaneously (they are interleaved next to each other),
- at least one access is a write,
- insufficient explicit synchronisation used to protect the accesses

Def. A program is data-race-free iff no SC execution of the program contain a data race.

(Why “slight variation”? Note that we quantify over all SC executions in the second definition.)

Note that data-race-freedom is a ”language-level” property!

Definition of data race surprisingly subtle

E.g., does this program contain any data races?

```
bool x = false, y = false;

t1 {
   if (x) y = true;
}

t2 {
   if (y) x = true;
}
```

No!

Race conditions

Definition from course slides:

Def. A *race condition* is a situation where the correctness of a concurrent program depends on the specific execution.

Note that this is an ”application-level” property!

I.e., for a given program p, to answer the question ”is p free from race conditions?” we must have access to the specification of p.

SC for data-race-free programs, again

- For Java programs, we have SC for programs without data races
- Presence of race conditions does not rob us of SC – important to know (the difference between) the two definitions
- What about the semantics of programs with data races?
  - Will not be considered in this course
  - In e.g. C++ data races result in undefined behavior (see C++ specification or https://en.cppreference.com/w/cpp/language/memory_model)
  - Java is supposed to be a ”safe language”, some guarantees (e.g. out-of-thin-air safety)

Outline

- What are memory models?
- Why weak memory models?
- Something about the Java memory model (as an example of a weak memory model)
- Programming in the Java memory model (as an example of programming in a weak memory model)

Practice?

- But what does this mean in practice?
- Question: How does “weak memory models” affect my daily life as a programmer?
- Answer: You have to “annotate” your program more (compared to CS). “Annotations” in the form of variable qualifiers, synchronization mechanisms etc.
- Essentially marking which things are shared and which are not

Simple example

Finally, an example!!!

```
bool done = false;

t1 {
   done = true;
}

t2 {
   if (done) print(33);
}
```

- Does this program contain
  - data races?
  - race conditions?

- Data race = yes, done is accessed without synchronization and one of the accesses is a write

- Race condition = depends on the specification we are to satisfy (what it means for the program to be correct)
- Note: Difficult to reason about correctness because we cannot assume SC because we have data races!

Simple example

Finally, an example!!!

```
bool done = false;

t1 {
   done = true;
}

t2 {
   if (done) print(33);
}
```

- Wait a minute!
- Are you telling me there’s a problem in this program?
- From a SC perspective, everything is fine!
- No atomicity problems or anything like that…
- But for JMM: We must avoid data races to get understandable semantics!

Simple example (fixed)

Finally, an example!!!

```
volatile bool done = false;

t1 {
   done = true;
}

t2 {
   if (done) print(33);
}
```

- Solution: Annotate your program. E.g., in Java `volatile` is considered synchronization.
- Does this program contain
  - data races?
  - race conditions?
- Data race = no, in Java `volatile` accesses are considered synchronized
- Race condition = ???, still depends on specification

Simple example (fixed)

Finally, an example!!!

```
volatile bool done = false;

t1 {
    done = true;
}

t2 {
    if (done) print(33);
}
```

Example specification:

- Spec = “If the program outputs something, it must output 33”
- (In other words: Spec = “Output nothing or 33”)
- Race conditions w.r.t. above specification?
- No race conditions! (As correct output does not depend on specific “execution”/ interleaving.)

Simple example (fixed)

Finally, an example!!!

```
volatile bool done = false;

t1 {
   done = true;
}

t2 {
   if (done) print(33);
}
```

Example specification:

- Spec = "The program outputs 33"
- Race conditions w.r.t. above specification?
- Yes, have race condition. Some interleavings give us correct output, others do not.

Similar example, with locks

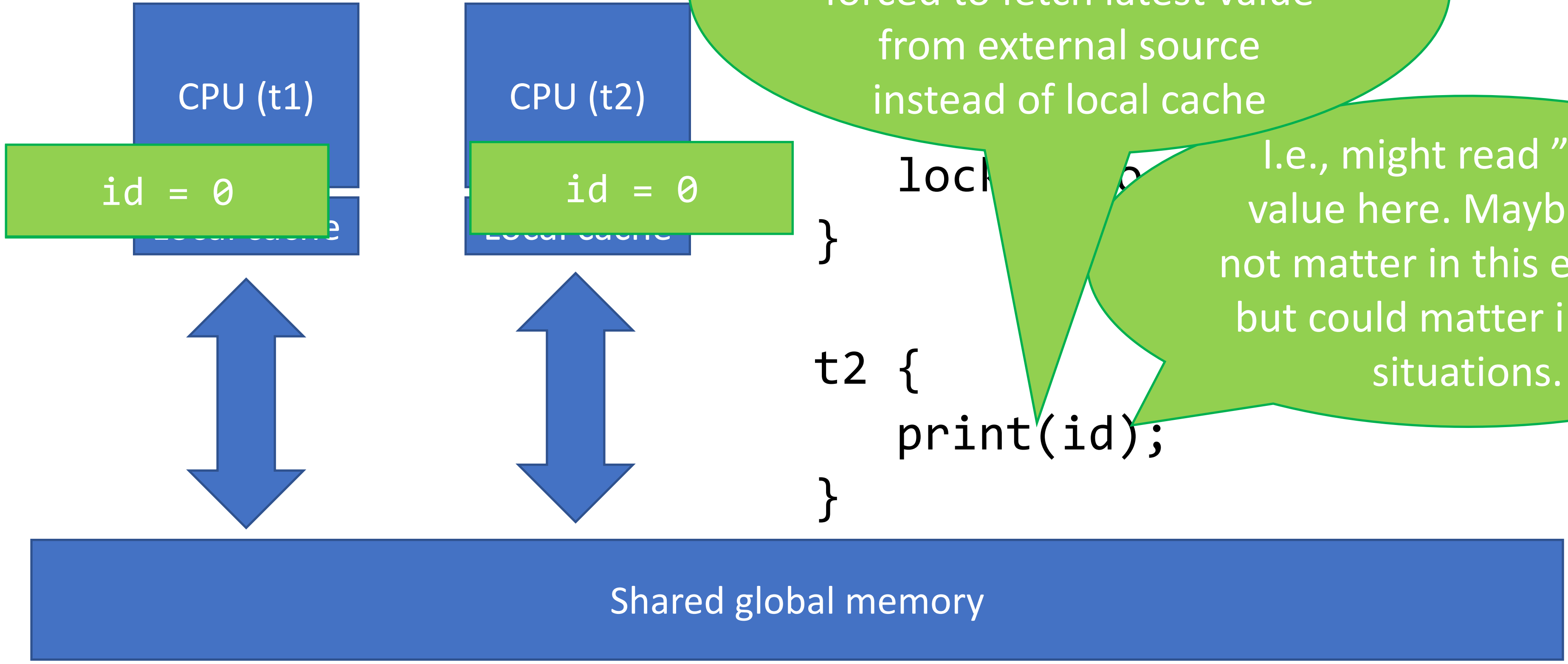
```
lock lock = new lock();
int id = 0;

t1 {
    lock.lock();
    id++;
    lock.unlock();
}

t2 {
    print(id);
}
```

Is there a problem here (race)?

Yes, all accesses to the shared variable done must be synchronized!

Here we have (again) atomicity, but not: visibility

`id` flag might exist as multiple copies...



```
lock lock = new lock();
int

    lock
}

t2 {
    print(id);
}
```

If we would have locked here, we would have been forced to fetch latest value from external source instead of local cache

I.e., might read "stale" value here. Maybe does not matter in this example, but could matter in other situations.

NOTE: Everything on this slide simplified, and makes unsound assumptions about JVM implementation details

Similar example, with locks (fixed)

```
lock lock = new lock();
int id = 0;

t1 {
   lock.lock();
   id++;
   lock.unlock();
}

t2 {
   lock.lock(); // new
   print(id);
   lock.unlock(); // new
}
```

This is how the program would look like with proper annotations/synchronization

No data races in sight!

Data races

Definition from course slides:

A *data race* occurs when two concurrent

- access a shared memory location,
- at least one access is a write,
- the threads use no explicit synchronisation mechanism to protect the shared d

We do not say "why" to synchronize – this is because we tried to formulate a language-agnostic definition (to the extent this is possible)

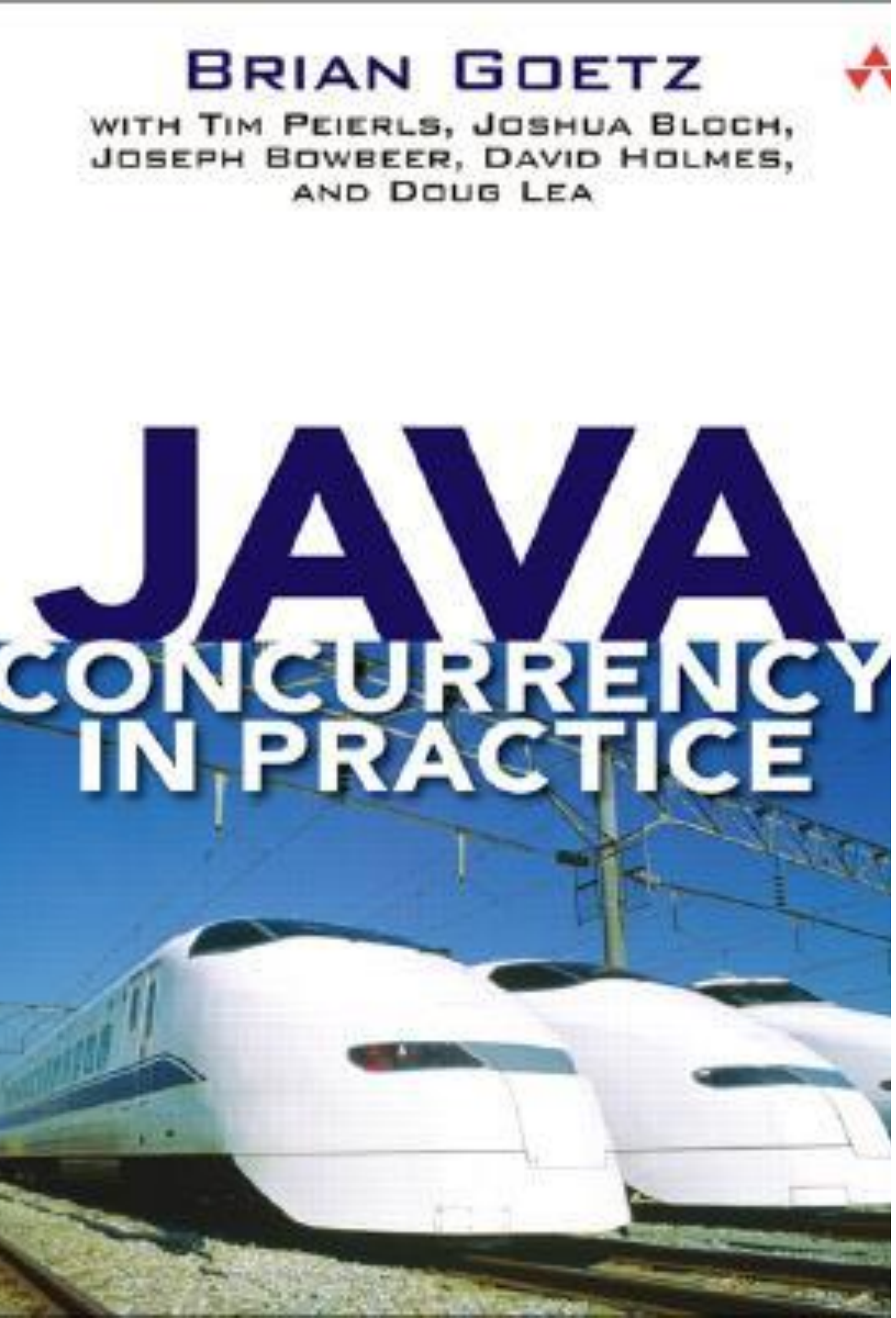
In real-world languages, such as Java, one of the "whys" is visibility (push values out of caches, prohibit compiler optimizations, …)

Summary?

- If you fell asleep and just woke up: Make sure to not have data races in your Java programs

- One way to go about this: Think about all of this in terms of atomicity and visibility

- Visibility aspect new in weak memory models compared to SC!


BRIAN GOETZ
WITH TIM PEIERLS, JOSHUA BLOCH, JOSEPH BOWBEER, DAVID HOLMES, AND DOUG LEA
JAVA CONCURRENCY IN PRACTICE

Reading suggestion

- See *Java Concurrency in Practice* (2006) if you want more of this. The book presents simplified rules you can follow to do concurrent programming in Java instead of having to learn the details of the Java memory model.

- E.g., the book provides useful “safe publication idioms”

If you only will remember one thing, please:

In concurrent programming in Java, not only do we have to consider atomicity, we also have to consider visibility!

visibility visibility

visibility visibility visibility

v i s i b i l i t y

Ack: https://www.hboehm.info/misc_slides/10-pldi-adve-boehm-tutorial.pdf was useful when creating these slides