Memory Consistency Models

Adam Wierman *Daniel Neill*

Adve, Pai, and Ranganathan. *Recent advances in memory consistency models for hardware shared-memory systems,* 1999.

Gniady, Falsafi, and Vijaykumar*. Is SC+ILP=RC?, 1999.*

Hill. *Multiprocessors should support simple memory consistency models*, 1998.

Memory consistency models

- The memory consistency model of a shared-memory system determines the order in which memory operations will appear to execute to the programmer.
  - Processor 1 writes to some memory location…
  - Processor 2 reads from that location…
  - Do I get the result I expect?

- Different models make different guarantees; the processor can reorder/overlap memory operations as long as the guarantees are upheld.

Tradeoff between programmability and performance!

Code example 1

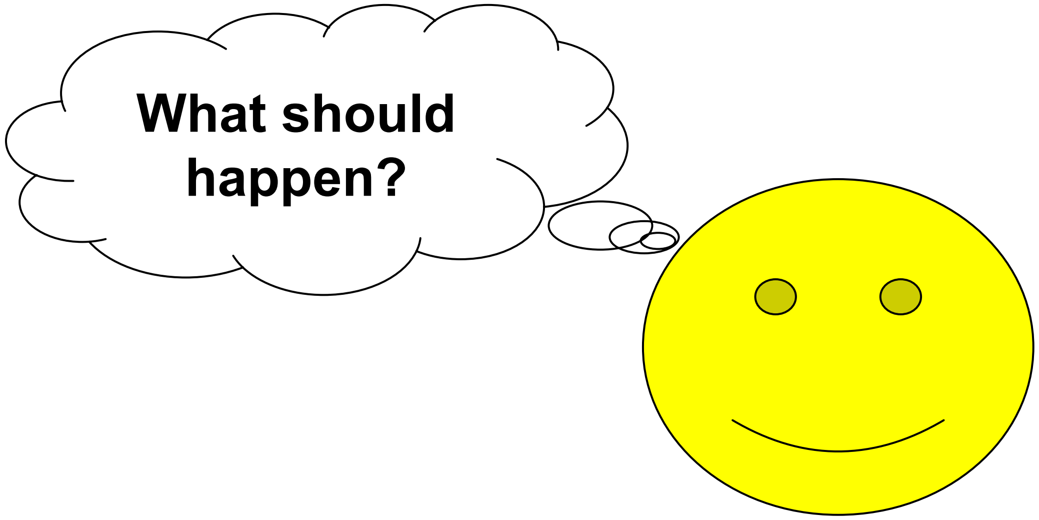
initially Data1 = Data2 = Flag = 0

P1

```
Data1 = 64
Data2 = 55
Flag = 1
```

P2

```
while (Flag != 1) {;}
register1 = Data1
register2 = Data2
```



Code example 1

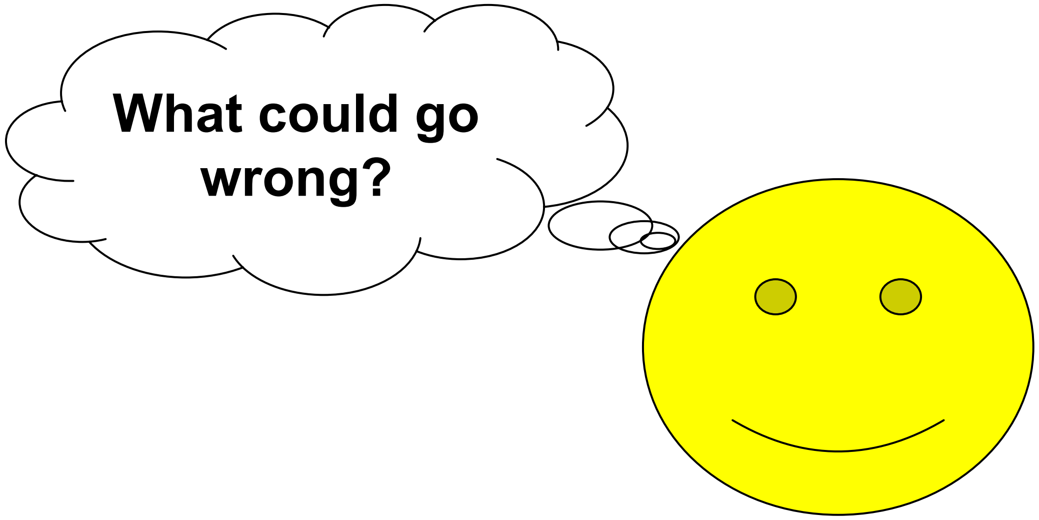
initially Data1 = Data2 = Flag = 0

P1

```
Data1 = 64
Data2 = 55
Flag = 1
```

P2

```
while (Flag != 1) {;}
register1 = Data1
register2 = Data2
```



Three models of memory consistency

- <u>Sequential Consistency</u> (SC):
  - Memory operations *appear* to execute one at a time, in some sequential order.
  - The operations of each individual processor *appear* to execute <u>in program order</u>.
- <u>Processor Consistency</u> (PC):
  - Allows reads following a write to execute out of program order (if they're not reading/writing the *same* address!)
  - Writes may not be immediately visible to other processors, but become visible in program order.
- <u>Release Consistency</u> (RC):
  - All reads and writes (to *different* addresses!) are allowed to operate out of program order.

Code example 1

initially Data1 = Data2 = Flag = 0

P1

```
Data1 = 64
Data2 = 55
Flag = 1
```

P2

```
while (Flag != 1) {;}
register1 = Data1
register2 = Data2
```

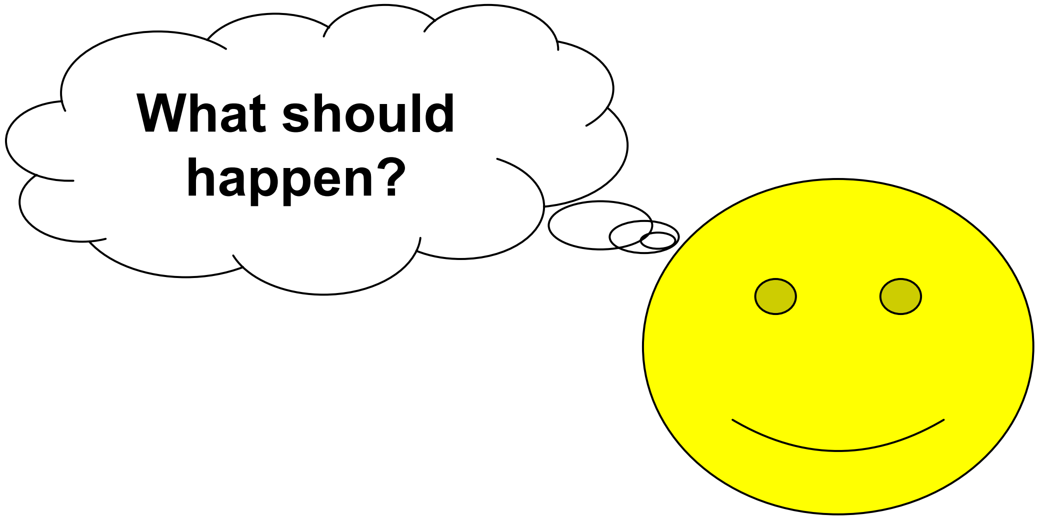
Does it work under:

- SC (no relaxation)?
- PC (Write→Read relaxation)?
- RC (all relaxations)?

Code example 2

initially Flag1 = Flag2 = 0

P1	P2
Flag1 = 1	Flag2 = 1
register1 = Flag2	register2 = Flag1
if (register1 == 0)	if (register2 == 0)
critical section	*critical section*



Code example 2

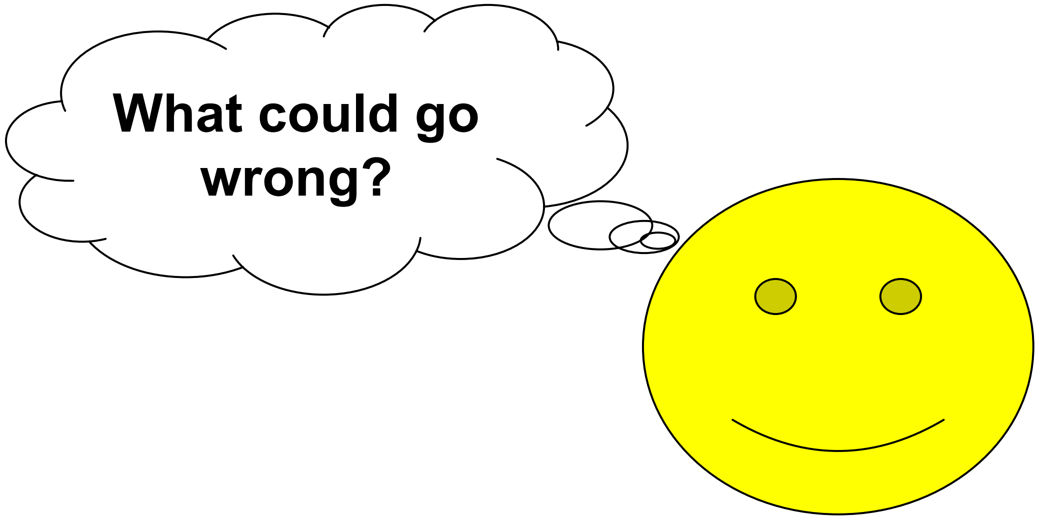
initially Flag1 = Flag2 = 0

P1

```
Flag1 = 1
register1 = Flag2
if (register1 == 0)
    critical section
```

P2

```
Flag2 = 1
register2 = Flag1
if (register2 == 0)
    critical section
```



Code example 2

initially Flag1 = Flag2 = 0

P1

```
Flag1 = 1
register1 = Flag2
if (register1 == 0)
    critical section
```

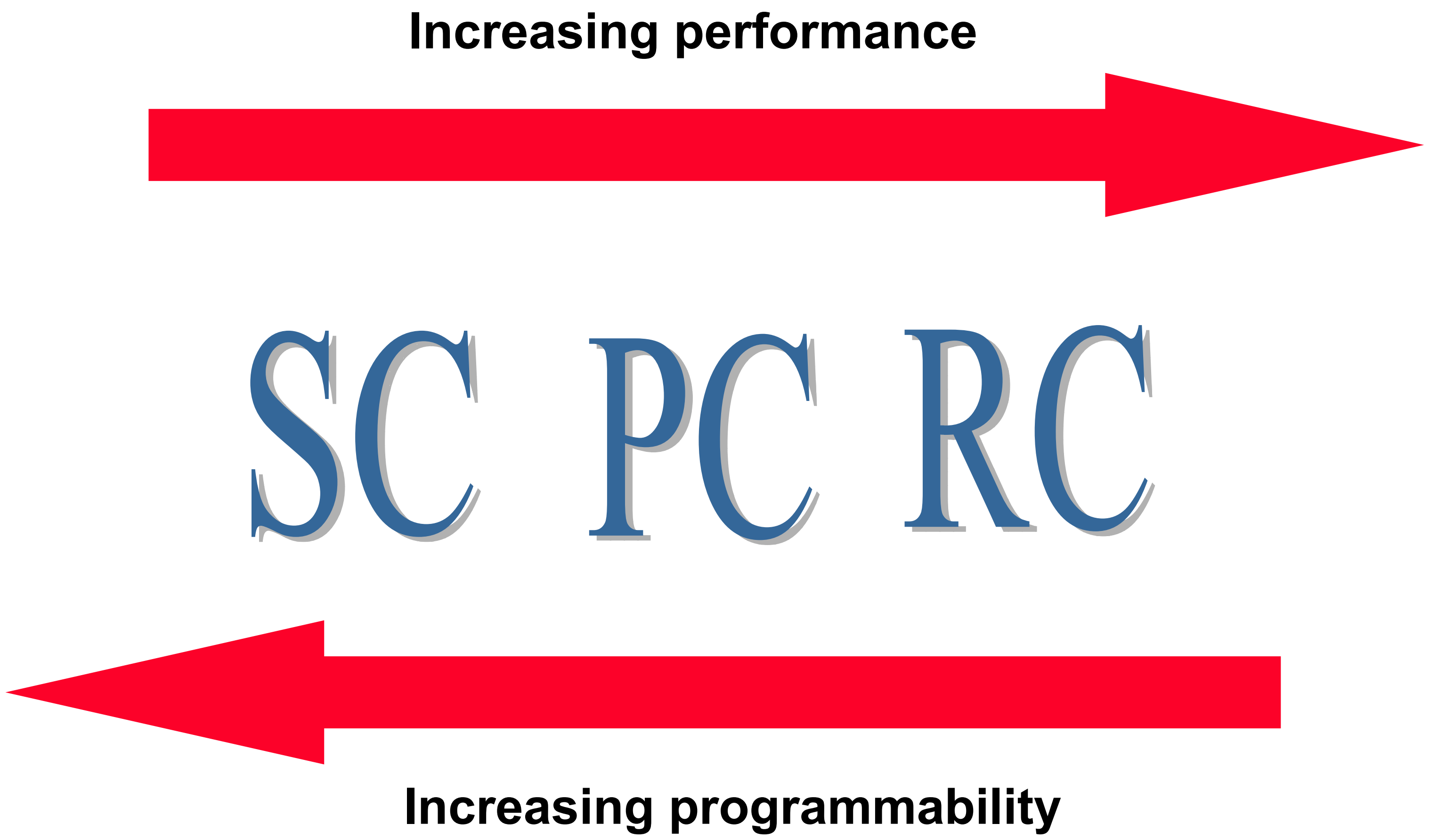
P2

```
Flag2 = 1
register2 = Flag1
if (register2 == 0)
    critical section
```

Does it work under:

- SC (no relaxation)?
- PC (Write→Read relaxation)?
- RC (all relaxations)?

The performance/programmability tradeoff



Programming difficulty

- PC/RC include special synchronization operations to allow specific instructions to execute atomically and in program order.
- The <u>programmer</u> must identify conflicting memory operations, and ensure that they are properly synchronized.
- Missing or incorrect synchronization → program gives unexpected/incorrect results.
- Too many unnecessary synchronizations → performance reduced (no better than SC?)

<u>Idea</u>: normally ensure sequential consistency; allow programmer to specify when relaxation possible?

Code example 1, revisited

initially Data1 = Data2 = Flag = 0

P1

```
Data1 = 64
Data2 = 55
MEMBAR (ST-ST)
Flag = 1
```

P2

```
while (Flag != 1) {;}
MEMBAR (LD-LD)
register1 = Data1
register2 = Data2
```

Programmer adds synchronization commands…
… and now it works as expected!

Performance of memory consistency models

- Relaxed memory models (PC/RC) hide much of memory operations' long latencies by <u>reordering</u> and <u>overlapping</u> some or all memory operations.
  - PC/RC can use write buffering.
  - RC can be aggressively out of order.
- This is particularly important:
  - When cache performance poor, resulting in many memory operations.
  - In distributed shared memory systems, when remote memory accesses may take much longer than local memory accesses.
- Performance results for straightforward implementations: as compared to SC, PC and RC reduce execution time by 23% and 46% respectively (Adve et al).

The big question

How can SC approach the performance of RC?

How can SC approach RC?

2 Techniques

Hardware Optimizations

Compiler Optimizations

What can SC do?

Hardware Optimizations

Can SC have per-processor caches? YES

Can SC have non-binding prefetching? YES

Can SC have multithreading? YES

Can SC use a write buffer? NO

SC cannot reorder memory operations because it might cause inconsistency.

Speculation with SC

Hardware Optimizations

SC only needs to appear to do memory operations in order

1. Speculatively perform all memory operations
2. Roll back to "sequentially consistent" state if constraints are violated

This emulates RC as long as rollbacks are infrequent.

Speculation with SC

Hardware Optimizations

SC only needs to appear to do memory operations in order

1. Speculatively perform all memory operations
2. Roll back to "sequentially consistent" state if constraints are violated

- Must allow both loads and stores to bypass each other
- Needs a very large speculative state
- Don't introduce overhead to the pipeline

Speculation with SC

Hardware Optimizations

SC only needs to appear to do memory operations in order

1. Speculatively perform all memory operations
2. Roll back to "sequentially consistent" state if constraints are violated

- Must detect violations quickly
- Must be able to roll back quickly
- Rollbacks can't happen often

Results

Hardware Optimizations

SC only needs to appear to do memory operations in order

These changes were implemented in SC++ and results showed a narrowing gap as compared to PC and RC

The gap is negligible!

Unlimited SHiQ, BLT

… but SC++ used significantly more hardware.

How can SC approach RC?

2 Techniques

Hardware Optimizations

Compiler Optimizations

Compiler optimizations?

P1

- Data1 = 64
- Data2 = 55
- Flag = 1

P2

```
while (Flag != 1) {;}
register1 = Data1
register2 = Data2
```

If we could figure out ahead of time which operations need to be run in order we wouldn't need speculation

Compiler Optimizations

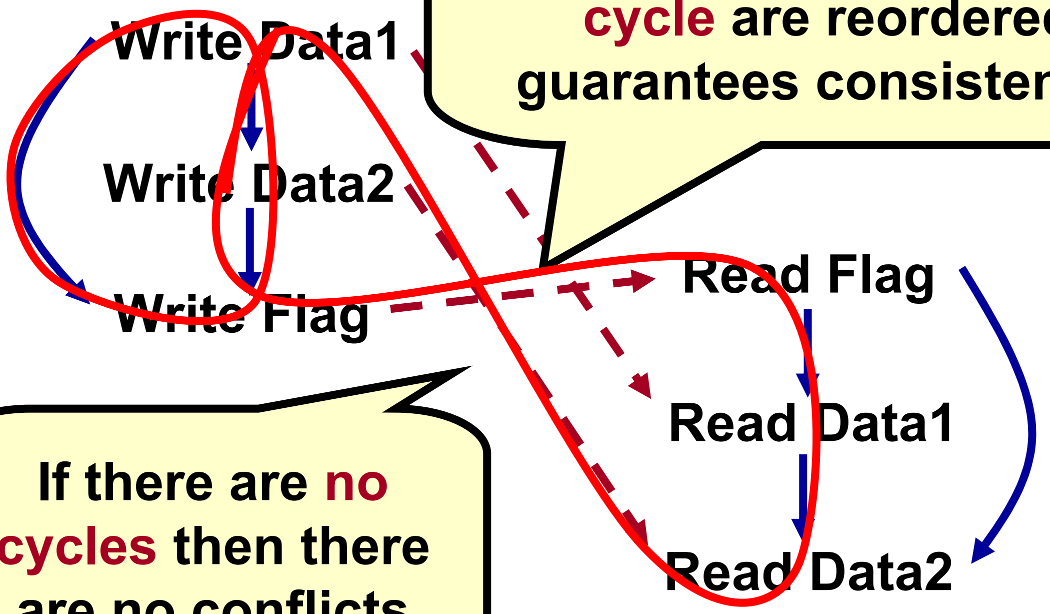
Where are the conflicts?

P1

- Data1 = 64
- Data2 = 55
- Flag = 1

P2

```
while (Flag != 1) {;}
register1 = Data1
register2 = Data2
```



Guaranteeing no operations on an edge in a cycle are reordered guarantees consistency!

If there are no cycles then there are no conflicts

Compiler Optimizations

Architecture

Conclusion

SC approaches RC

Speculation and compiler optimizations allow SC to achieve nearly the same performance as RC

RC approaches SC

Programming constructs allow user to distinguish possible conflicts as synchronization operations and atill obtain the simplicity of SC

Memory Consistency Models

Adam Wierman   *Daniel Neill*

Adve, Pai, and Ranganathan. *Recent advances in memory consistency models for hardware shared-memory systems,* 1999.

Gniady, Falsafi, and Vijaykumar. *Is SC+ILP=RC?, 1999.*

Hill. *Multiprocessors should support simple memory consistency models*, 1998.