



The foundation of C++ atomics: The knowledge you need to correctly use C++ atomics.

Filipe Mulonde



CODE RECKONS

Science to the CORE 1



Filipe Mulonde
@filipe_mulonde

The Foundation of C++ Atomics: The Knowledge You Need to Correctly Use C++ Atomics. 1/N

● Anatomy of C++ atomics

→ Atomic as a class in C++

- Standard Atomic types / typedef (e.g. `atomic_bool`, `atomic_int_least_8_t`)
- How atomic interacts with the other features of the language.(e.g. is `std::atomic<int> x = x + 1` an atomic operation ?)
- Operation available on those types (e.g. `fetch_add`, `is_look_free`)

→ C++ Memory Model

- Mathematical model (axiomatic graph)
- The Model Without Low-Level Atomics
- Sequential Consistency for Data-Race-Free Programs
- Semantics of Data races
- Define what is a memory location in C++.
- The Model Supporting Low-Level Atomics (e.g relaxed)
- Optimizations Allowed by the Model
- Implications of the model for Current Computer architectures. (e.g. need for consistency, coherence, atomicity).

→ Compiler

- Compiler-introduced Data Races (e.g Trace Scheduling)
- Static Instruction Scheduling
- Provide efficient compilation to hardware (How atomic map to their expected machine-instruction implementations on different architecture (e.g x86, Power, ARM, RISK-V).

→ Hardware

- Description of real Hardware
- Dynamic scheduling
- Hardware-introduced Data Races



Onur Mutlu - Supercomputing Frontiers...

Onur Mutlu Lectures
2,3 mil visualizações • há 2 meses

The Story of Rowhammer - Secure Hardware,...

Onur Mutlu Lectures
2,3 mil visualizações • há 8 meses

Frontiers of AI Accelerators: Technologies, Circuits and...

Onur Mutlu Lectures
1,1 mil visualizações • há 1 semana

Arch. Mentoring Workshop @ISCA'21 - Applying to Gra...

Onur Mutlu Lectures
2,2 mil visualizações • há 4 meses

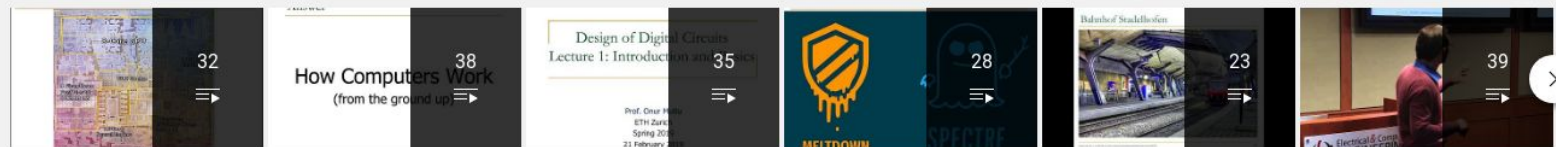
Onur Mutlu - Future Computing Platforms:...

Onur Mutlu Lectures
2,5 mil visualizações • há 8 meses

Accelerating Genome Analysis: A Primer on an...

Onur Mutlu Lectures
991 visualizações • há 8 meses

First Course in Computer Architecture & Digital Design 2021-2013



Livestream - Digital Design and Computer Architecture - ETH...

Onur Mutlu Lectures
VER PLAYLIST COMPLETA

Digital Design & Computer Architecture - ETH Zürich...

Onur Mutlu Lectures
VER PLAYLIST COMPLETA

Design of Digital Circuits - ETH Zürich - Spring 2019

Onur Mutlu Lectures
VER PLAYLIST COMPLETA

Design of Digital Circuits - ETH Zürich - Spring 2018

Onur Mutlu Lectures
VER PLAYLIST COMPLETA

Digital Circuits and Computer Architecture - ETH Zurich - ...

Onur Mutlu Lectures
VER PLAYLIST COMPLETA

Spring 2015 -- Computer Architecture Lectures --...

Carnegie Mellon Computer Architec...
VER PLAYLIST COMPLETA

Advanced Computer Architecture Courses 2020-2012



Computer Architecture - ETH Zürich - Fall 2020

Onur Mutlu Lectures

Computer Architecture - ETH Zürich - Fall 2019

Onur Mutlu Lectures

Computer Architecture - ETH Zürich - Fall 2018

Onur Mutlu Lectures

Computer Architecture - ETH Zürich - Fall 2017

Onur Mutlu Lectures

Fall 2015 - 740 Computer Architecture

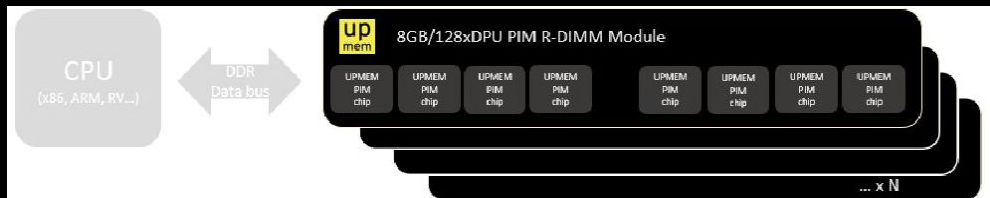
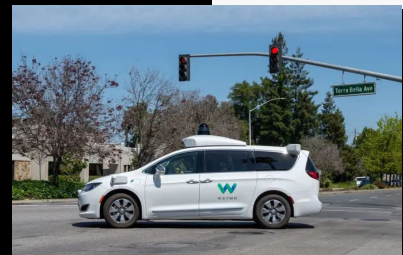
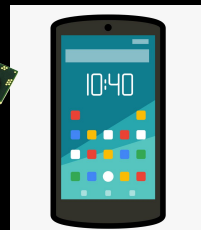
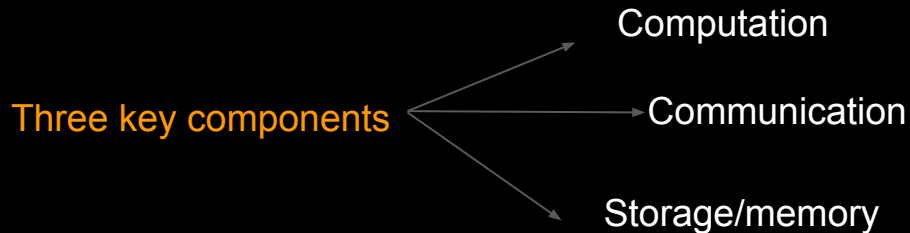
Carnegie Mellon Computer Architec...

Fall 2013 - 740 Computer Architecture - Carnegie Mellon

Carnegie Mellon Computer Architec...

A Computing System

Different Platforms, Different Goals



Why Parallel Computers?

Original Goal

Improve performance (Execution time or task throughput)

Other Goals

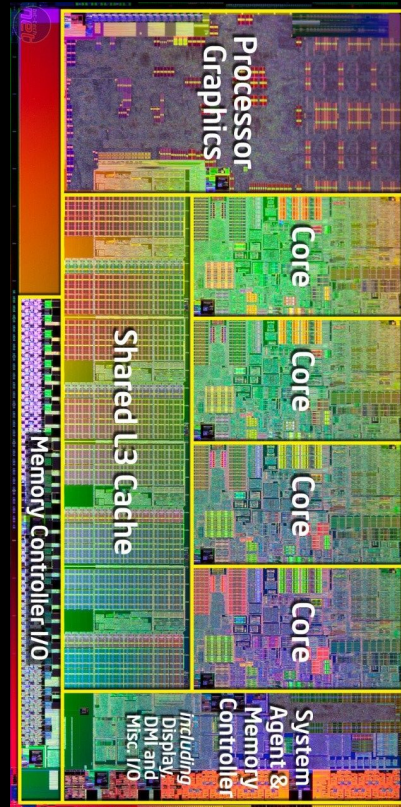
Reduce power consumption

($4N$ units at freq $F/4$) consume less power than (N units at freq F)

Improve cost efficiency and scalability, reduce complexity

Harder to design a single unit that performs as well as N simpler units

Improve dependability: Redundant execution in space



Multiprocessor Types

Loosely coupled multiprocessors

- No shared global memory address space
- Network-based multiprocessors
- Usually programmed via message passing
- Explicit calls (send, receive) for communication

Tightly coupled multiprocessors

- Shared global memory address space
- Multi-core processors, multithreaded processors
- Programming model similar to uniprocessors except
- Operations on shared data require synchronization



C++ standard does not support programming for Loosely coupled multiprocessors

Main Design Issues in Tightly-Coupled MP

Shared memory synchronization

How to handle locks, atomic operations

Cache coherence

How to ensure correct operation in the presence of private caches keeping the same memory address cached

Memory consistency: Ordering of all memory operations

What should the programmer expect the hardware to provide?

Shared resource management

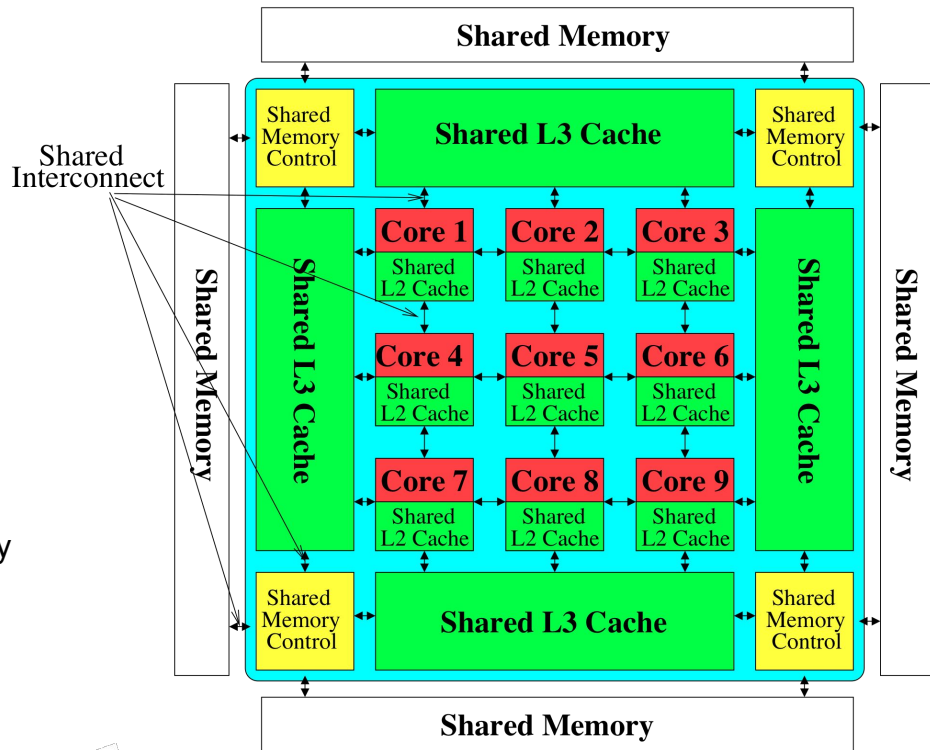
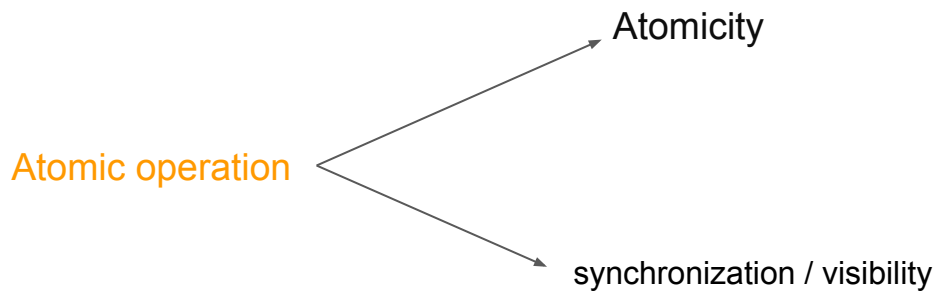
Communication: Interconnects

Load imbalance

- ❑ How to partition a single task into multiple tasks
- ❑ Synchronization
 - How to synchronize (efficiently) between tasks
 - How to communicate between tasks
 - Locks, barriers, pipeline stages, condition variables, semaphores, atomic operations, ...
- ❑ Contention
- ❑ Maximizing parallelism
- ❑ Ensuring correct operation while optimizing for performance

Processor-Centric Design

Why do we need Atomic ?



Most of the system is dedicated to storing and moving data

Memory Consistency vs. Cache Coherence

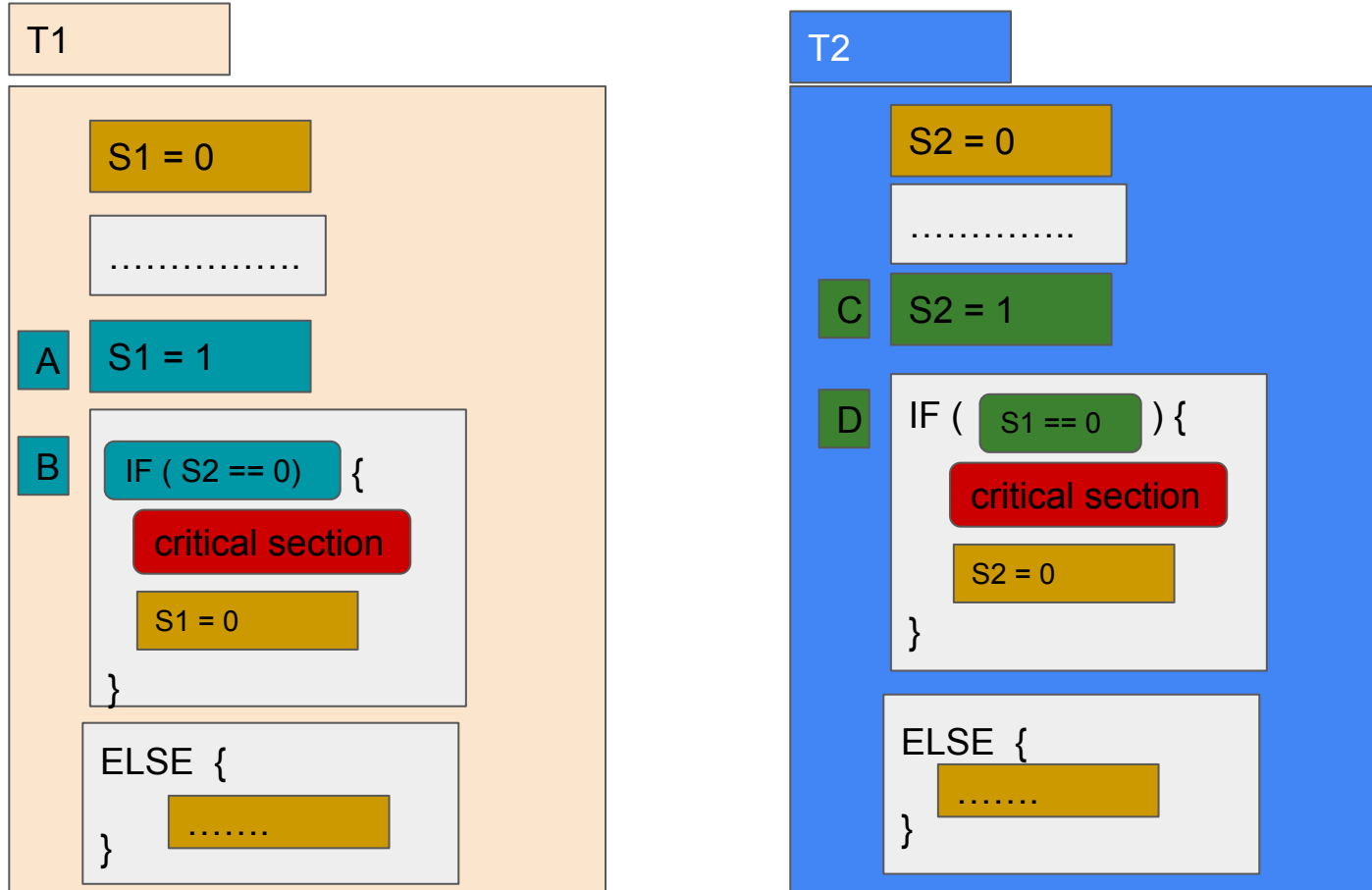
- Consistency is about ordering of all memory operations from different processors (i.e., to different memory locations)
 - Global ordering of accesses to *all* memory *locations*
- Coherence is about ordering of operations from different processors to the same memory location
 - Local ordering of accesses to *each* cache *block*

Memory Ordering in a MIMD Processor

- Each processor's memory operations are in sequential order with respect to the “thread” running on that processor (assume each processor obeys the von Neumann model)
- Multiple processors execute memory operations concurrently
- How does the memory see the order of operations from all processors?
 - In other words, what is the ordering of operations across different processors?

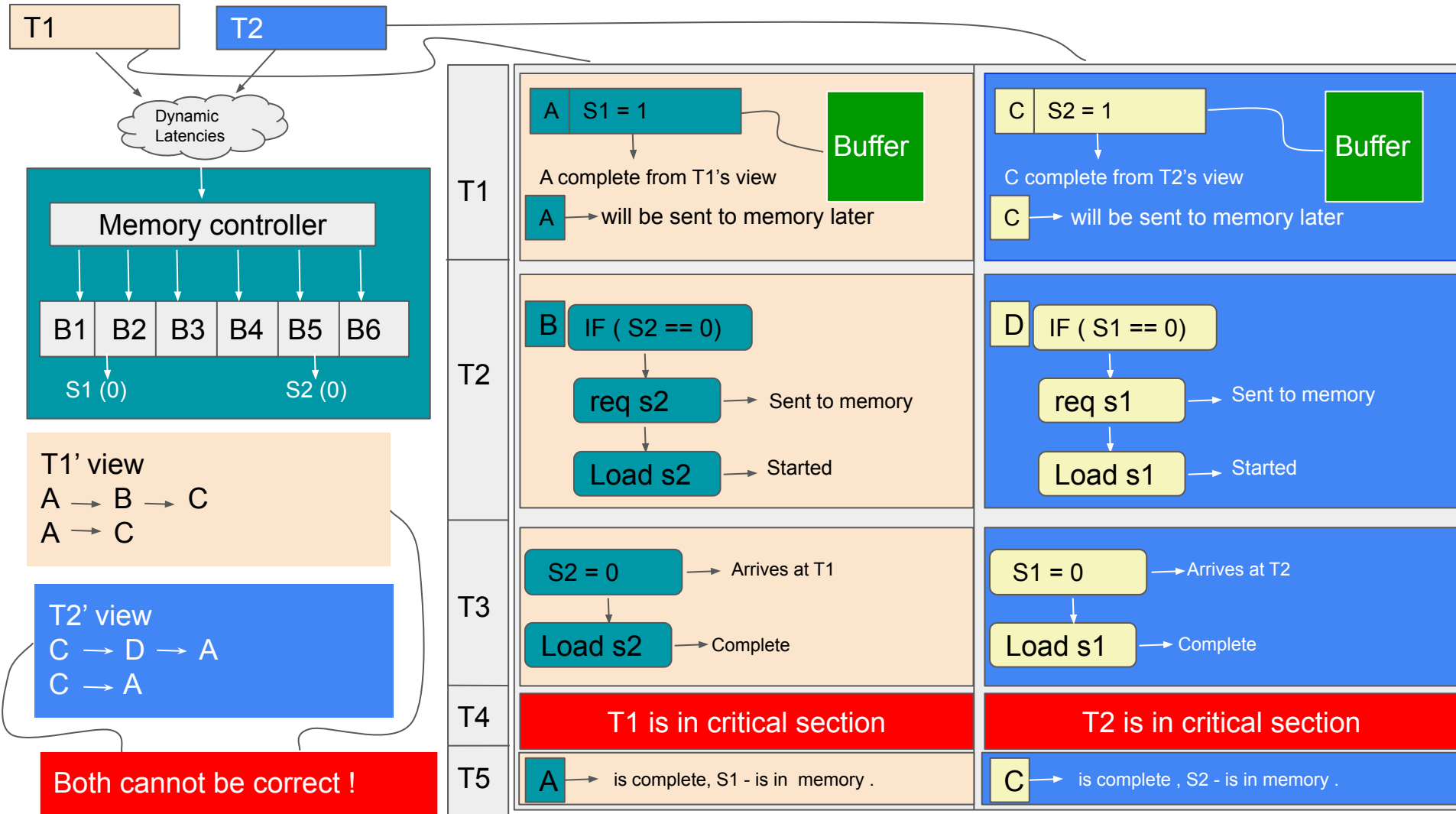
what is the ordering of operations across different processors?

Can the two processors be in the critical section at the same time given that they both obey the von Neumann model? yes.



Protecting Shared Data

- Threads are not allowed to update shared data concurrently
 - For correctness purposes
- Accesses to shared data are encapsulated inside *critical sections* or *protected via synchronization constructs (locks, semaphores, condition variables)*
- Only one thread can execute a critical section at a given time
 - Mutual exclusion principle
- A multiprocessor should provide the *correct* execution of *synchronization primitives* to enable the programmer to protect shared data



The Problem

- The two processors did **NOT** see the same order of operations to memory
- The “happened before” relationship between multiple updates to memory was inconsistent between the two processors’ points of view
- As a result, each processor thought the other was **not** in the critical section

How Can We Solve The Problem?

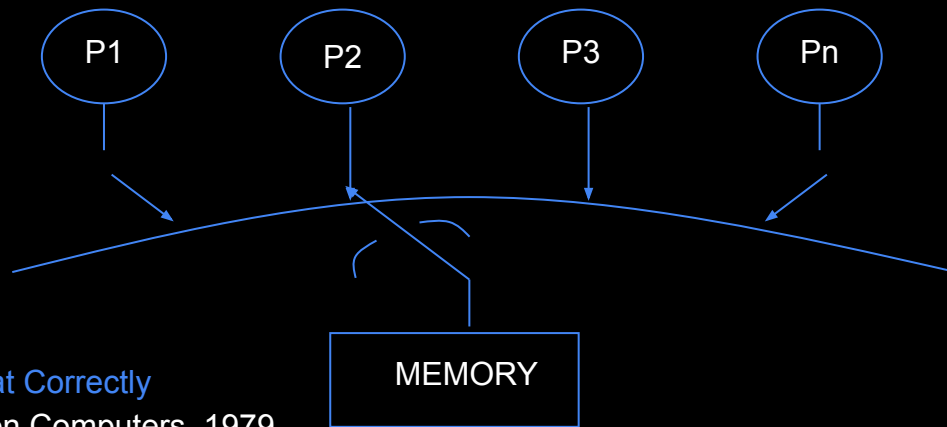
- Idea: Sequential consistency

A multiprocessor system is sequentially consistent if:

- the result 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
- This is a memory ordering model, or memory model
- Specified by the ISA



Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," IEEE Transactions on Computers, 1979

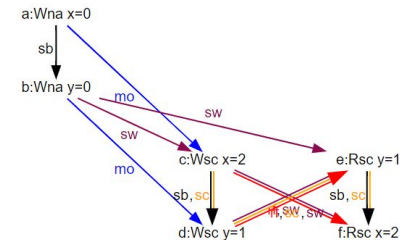
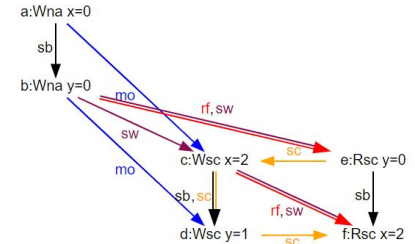
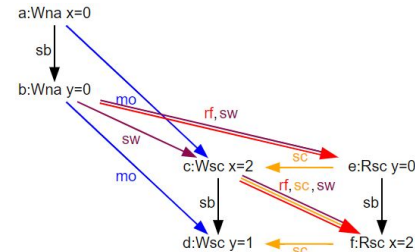
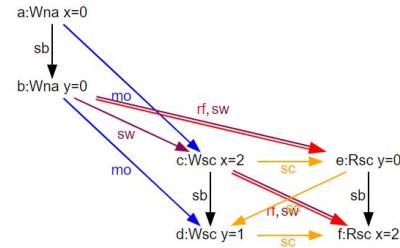
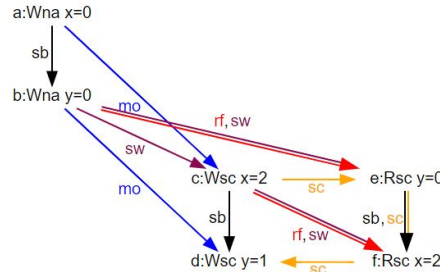
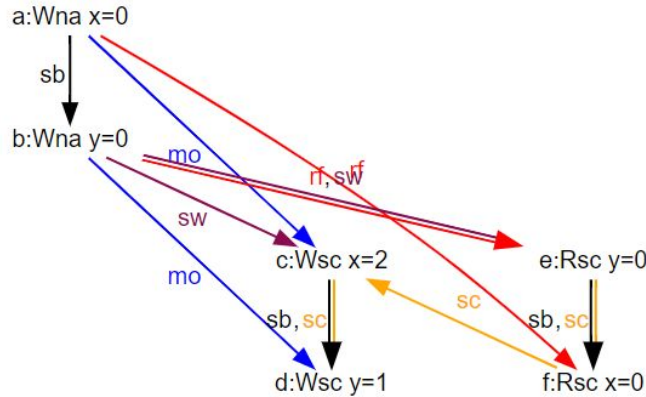
Sequential Consistency Execution graphs

```
#include <atomic>
#include <iostream>
#include <thread>
```

```
std::atomic<int> x{0};
std::atomic<int> y{0};
```

```
void setting(){
    x.store(2);
    y.store(1);
}
```

```
void reading(){
    std::cout << y.load();
    std::cout << x.load();
}
```



Which order (interleaving) is observed depends on implementation and dynamic latencies

Issues with Sequential Consistency?

- Nice abstraction for programming, but two issues:
 - Too conservative ordering requirements
 - Limits the aggressiveness of performance enhancement techniques
- Is the total global order requirement too strong?

The ordering of operations is important when the order affects operations on shared data

□ i.e., when processors need to synchronize to execute a “program region”

- Do we need a global order across all operations and all processors?
- How about a global order only across all stores?
 - Total store order memory model; unique store order model
- How about enforcing a global order only at the boundaries of synchronization?
 - Relaxed memory models
 - Acquire-release consistency model

Techniques to Enhance the Performance of Memory Consistency Models

- Gharachorloo et al., “Two Techniques to Enhance the Performance of Memory Consistency Models,” ICPP 1991.

Abstract

The memory consistency model supported by a multiprocessor directly affects its performance. Thus, several attempts have been made to relax the consistency models to allow for more buffering and pipelining of memory accesses. Unfortunately, the potential increase in performance afforded by relaxing the consistency model is accompanied by a more complex programming model. This paper introduces two general implementation techniques that provide higher performance for all the models. The first technique involves *prefetching* values for accesses that are delayed due to consistency model constraints. The second technique employs *speculative execution* to allow the processor to proceed even though the consistency model requires the memory accesses to be delayed. When combined, the above techniques alleviate the limitations imposed by a consistency model on buffering and pipelining of memory accesses, thus significantly reducing the impact of the memory consistency model on performance.

SC-Preserving optimizations in LLVM Marino et al. 2011

Average slowdown:

- 34% w/ only SC preserving Optimizations

- 5.5% w/ optimizations modified to preserve SC

Drawbacks:

- Hardware still allows weak behaviors, i.e no end-to-end SC

- Requires modifying existing compilers

Examples of Weak Consistency Models

- Gharachorloo et al., “Two Techniques to Enhance the Performance of Memory Consistency Models,”

A more relaxed consistency model can be derived by relating memory request ordering to synchronization points in the program. The weak consistency model (WC) proposed by Dubois et al. [4, 5] is based on the above idea and guarantees a consistent view of memory only at synchronization points. As an example, consider a process updating a data structure within a critical section. Under SC, every access within the critical section is delayed until the previous access completes. But such delays are unnecessary if the programmer has already made sure that no other process can rely on the data structure to be consistent until the critical section is exited. Weak consistency exploits this by allowing accesses within the critical section to be pipelined. Correctness is achieved by guaranteeing that all previous accesses are performed before entering or exiting each critical section.

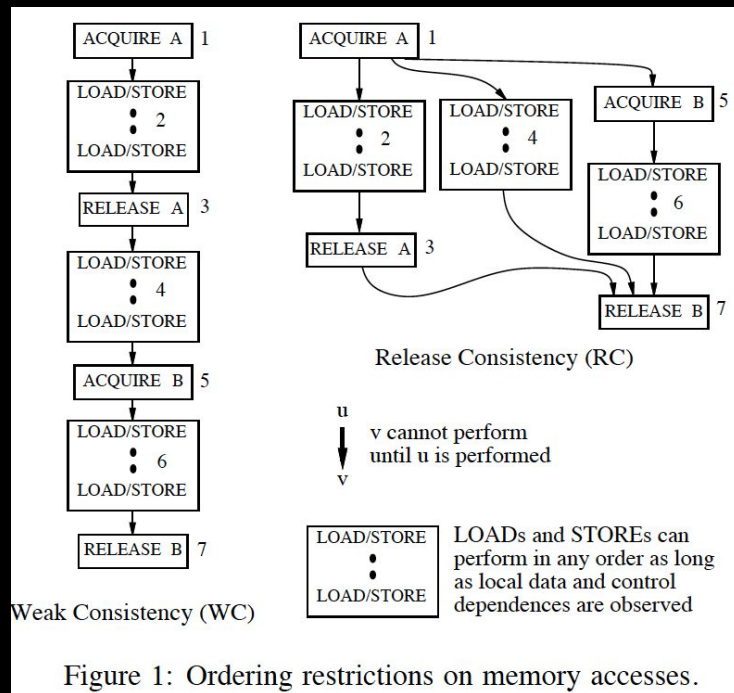


Figure 1: Ordering restrictions on memory accesses.

Memory_order_acquire & Memory_order_release

Execution graphs

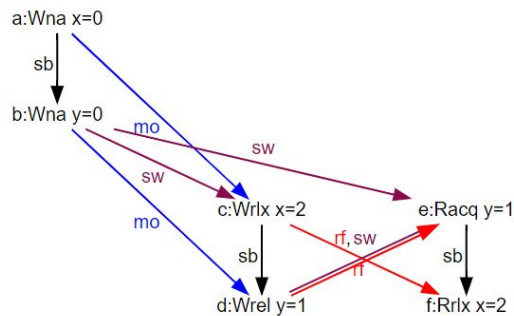
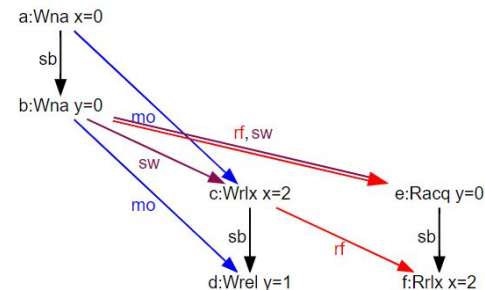
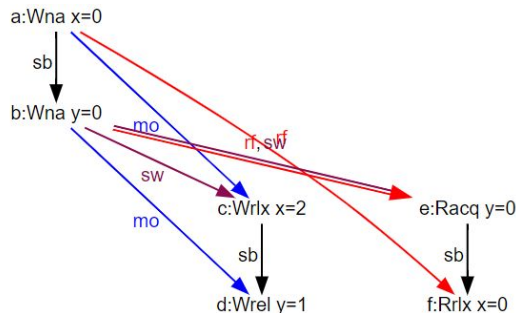
```
std::atomic<int> x{0};
std::atomic<int> y{0};
void setting(){
```

```
    x.store(2,std::memory_order_relaxed);
    y.store(1,std::memory_order_release);
```

```
}
```

```
void reading(){
    std::cout << y.load(std::memory_order_acquire);
    std::cout << x.load(std::memory_order_relaxed);
```

```
}
```



Memory Order Relaxed Execution graphs

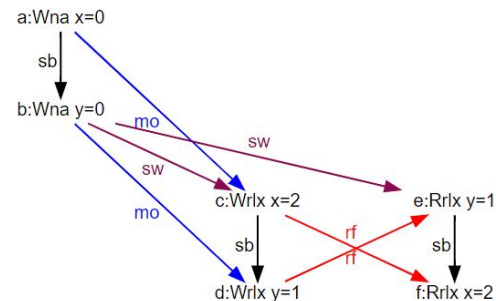
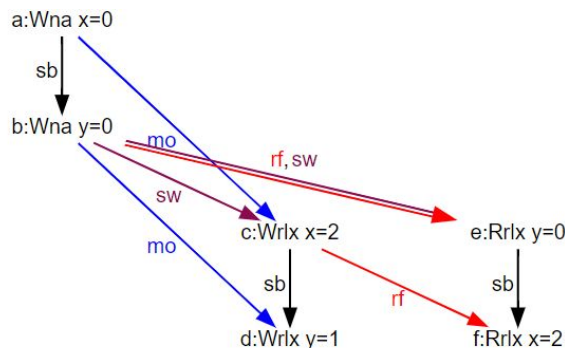
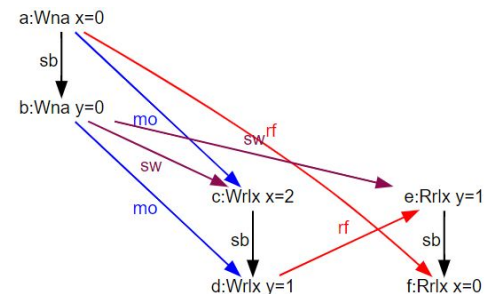
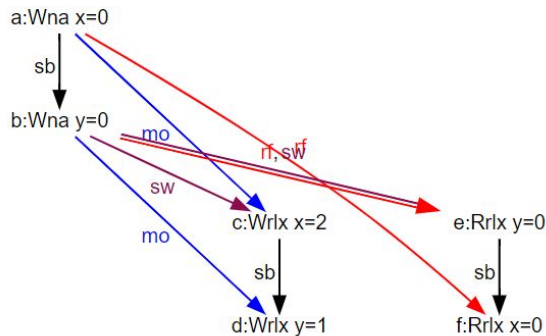
```
std::atomic<int> x{0};  
std::atomic<int> y{0};  
void setting(){
```

```
  x.store(2,std::memory_order_relaxed);  
  y.store(1,std::memory_order_relaxed);
```

```
}  
void reading(){
```

```
  std::cout << y.load(std::memory_order_relaxed);  
  std::cout << x.load(std::memory_order_relaxed);
```

```
}
```



Mathematizing C++ Concurrency

Mathematizing C++ Concurrency

Mark Batty Scott Owens Susmit Sarkar Peter Sewell Tjark Weber

University of Cambridge

Abstract

Shared-memory concurrency in C and C++ is pervasive in systems programming, but has long been poorly defined. This motivated an ongoing shared effort by the standards committees to specify concurrent behaviour in the next versions of both languages. They aim to provide strong guarantees for race-free programs, together with new (but subtle) relaxed-memory atomic primitives for high-performance concurrent code. However, the current draft standards, while the result of careful deliberation, are not yet clear and rigorous definitions, and harbour substantial problems in their details.

In this paper we establish a mathematical (yet readable) semantics for C++ concurrency. We aim to capture the intent of the current ('Final Committee') Draft as closely as possible, but discuss changes that fix many of its problems. We prove that a proposed x86 implementation of the concurrency primitives is correct with respect to the x86-TSO model, and describe our CPPMEM tool for exploring the semantics of examples, using code generated from our Isabelle/HOL definitions.

Having already motivated changes to the draft standard, this work will aid discussion of any further changes, provide a correctness condition for compilers, and give a much-needed basis for analysis and verification of concurrent C and C++ programs.

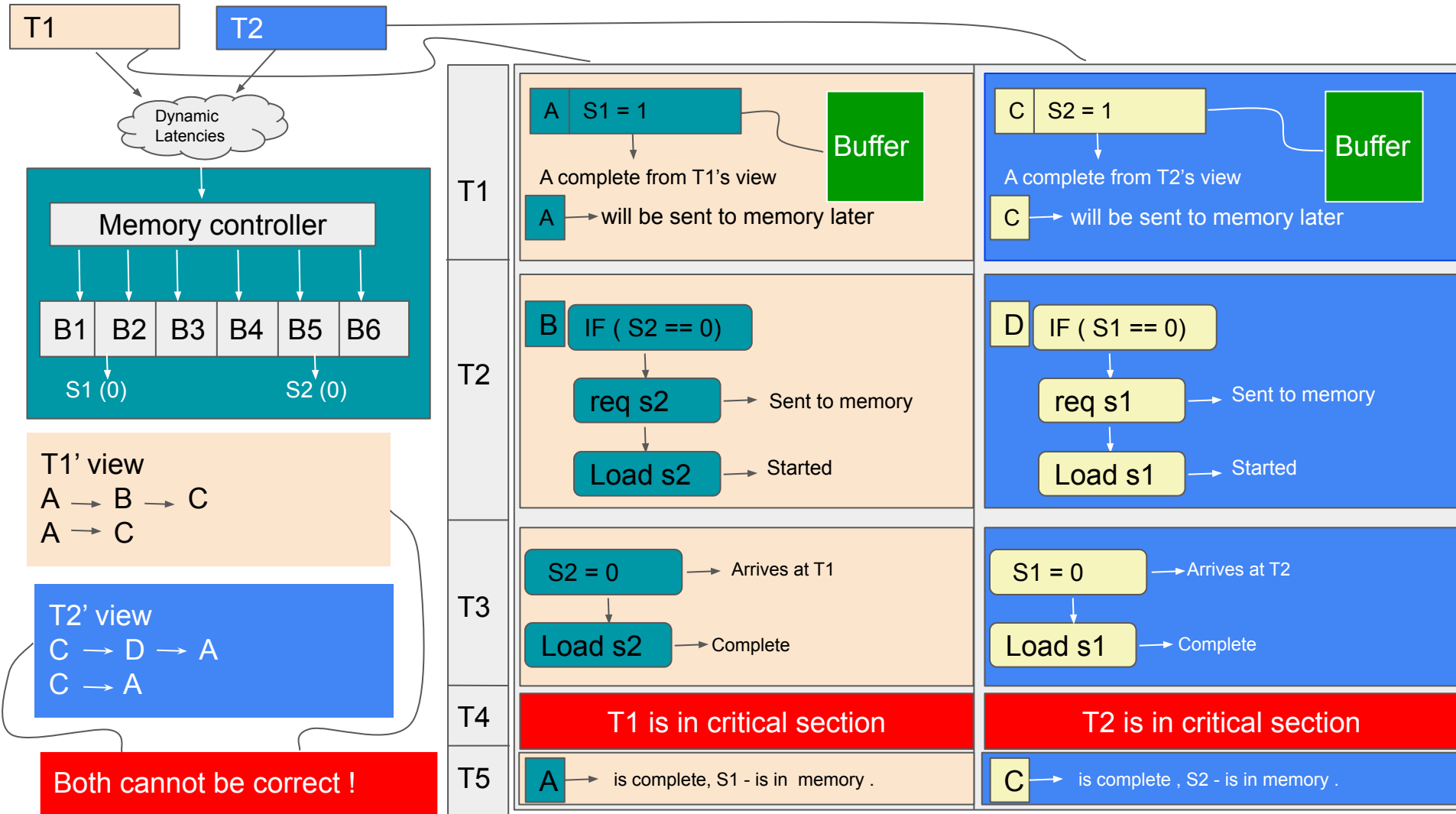
Categories and Subject Descriptors C.1.2 [Multiple Data Stream Architectures (Multiprocessors)]: Parallel processors; D.1.3 [Concurrent Programming]: Parallel programming; F.3.1 [Specifying and Verifying and Reasoning about Programs]

General Terms Documentation, Languages, Reliability, Standardization, Theory, Verification

Keywords Relaxed Memory Models, Semantics

quential consistency (SC) [Lam79], simplifies reasoning about programs but at the cost of invalidating many compiler optimisations, and of requiring expensive hardware synchronisation instructions (e.g. fences). The C++0x design resolves this by providing a relatively strong guarantee for typical application code together with various *atomic* primitives, with weaker semantics, for high-performance concurrent algorithms. Application code that does not use atomics and which is race-free (with shared state properly protected by locks) can rely on sequentially consistent behaviour; in an intermediate regime where one needs concurrent accesses but performance is not critical one can use *SC atomics*; and where performance is critical there are *low-level atomics*. It is expected that only a small fraction of code (and of programmers) will use the latter, but that code —concurrent data structures, OS kernel code, language runtimes, GC algorithms, etc.— may have a large effect on system performance. Low-level atomics provide a common abstraction above widely varying underlying hardware: x86 and Sparc provide relatively strong TSO memory [SSO⁺10, Spa]; Power and ARM provide a weak model with cumulative barriers [Pow09, ARM08, AMSS10]; and Itanium provides a weak model with release/acquire primitives [Int02]. Low-level atomics should be efficiently implementable above all of these, and prototype implementations have been proposed, e.g. [Ter08].

The current draft standard covers all of C++ and is rather large (1357 pages), but the concurrency specification is mostly contained within three chapters [Bec10, Chs.1, 29, 30]. As is usual for industrial specifications, it is a prose document. Mathematical specifications of relaxed memory models are usually either operational (in terms of an abstract machine or operational semantics, typically involving explicit buffers etc.) or axiomatic, defining constraints on the relationships between the memory accesses in a complete candidate execution e.g. with a happens-before relation over them.



TSO/x86 Memory Model

How ROB/SQ within x86 cores influenced the x86
TSO memory model.

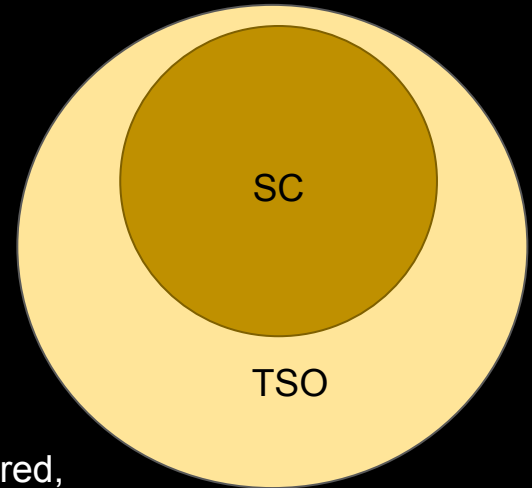
Comparing Memory Consistency Models

SC executions are a proper subset of TSO executions; all SC executions are TSO executions, while some TSO executions are SC executions and some are not.

- Load \rightarrow Load
- Load \rightarrow Store
- Store \rightarrow Store
- Store \rightarrow Load Included for SC but omitted for TSO/x86

Systems that support TSO do not provide ordering between a store and a subsequent (in program order) load, although they do require the load to get the immediately value of the earlier store.

In situations in which the programmer wants those instructions to be ordered, the programmer must explicitly specify that ordering by putting a FENCE instruction between the store and the subsequent load.



● Anatomy of C++ atomics

→ Atomic as a class in C++

- Standard Atomic types / typedef (e.g. `atomic_bool`, `atomic_int_least_8_t`)
- How atomic interacts with the other features of the language.(e.g. is `std::atomic<int> x = x + 1` an atomic operation ?)
- Operation available on those types (e.g. `fetch_add`, `is_lock_free`)

→ C++ Memory Model

- Mathematical model (axiomatic graph)
- The Model Without Low-Level Atomics
- Sequential Consistency for Data-Race-Free Programs
- Semantics of Data races
- Define what is a memory location in C++.
- The Model Supporting Low-Level Atomics (e.g relaxed)
- Optimizations Allowed by the Model
- Implications of the model for Current Computer architectures. (e.g. need for consistency, coherence, atomicity).

→ Compiler

- Compiler-introduced Data Races (e.g Trace Scheduling)
- Provide efficient compilation to hardware (How atomic map to their expected machine-instruction implementations on different architecture (e.g x86, Power, ARM, RISK-V).

→ Hardware

- | |
|--|
| <ul style="list-style-type: none">• Description of real Hardware |
| <ul style="list-style-type: none">• Dynamic scheduling |
- Hardware-introduced Data Races

How ROB/SQ within x86 cores influenced the x86 TSO memory model.

Why is it necessary to describe the hardware?

Knowledge needed to verify if the compiler is providing an efficient compilation to hardware.

Do we need fences to implement the SC memory model on x86 ?

Yes, why ?

Do we need fences to implement the Acquire and Release memory model on x86 ?

No, why ?

Why do the compiler and hardware change the order of execution of instructions?

Performance, Why is it possible to gain performance by changing the order of execution of instructions?

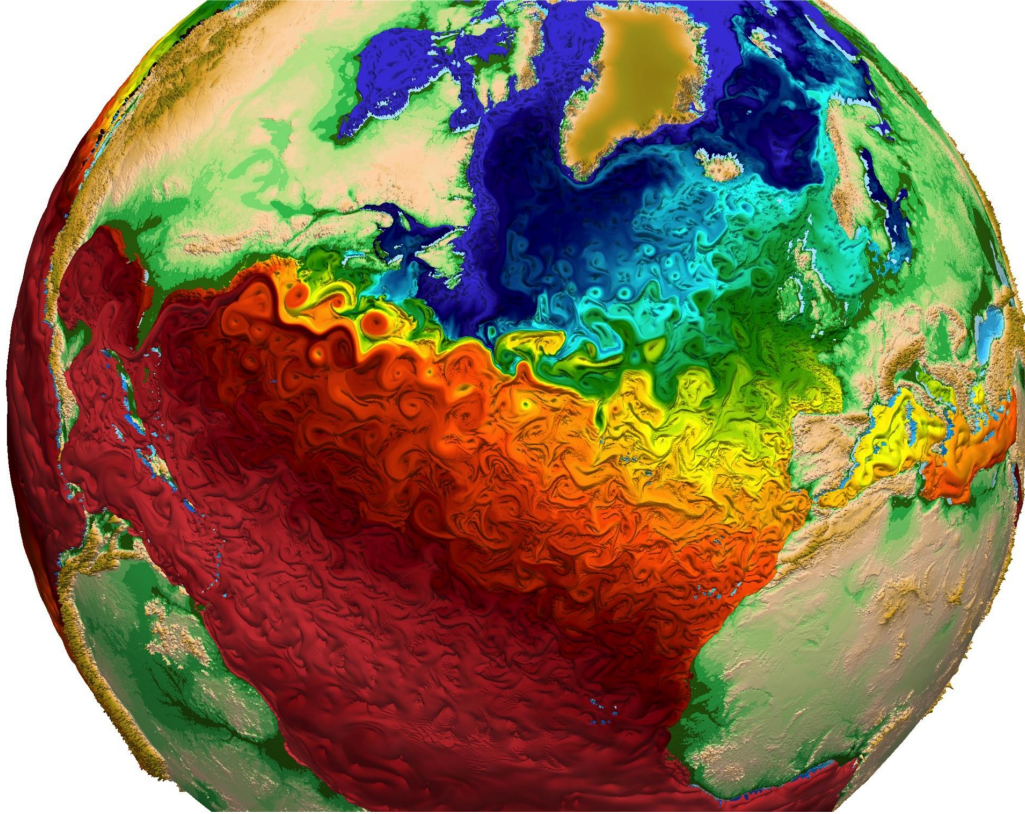
You can only answer these questions if you know your hardware

Why keep a buffer inside a core?

Almost all modern architectures use a write buffer

Precise Exceptions & Memory Disambiguation Problem


Solving the Hardest Problems



Data Dependence Types


Flow dependence (Read-after-Write)

1) $r_3 \leftarrow r_1 \text{ op } r_2$
2) $r_5 \leftarrow r_3 \text{ op } r_4$




Anti dependence (Write-after-Read)

1) $r_3 \leftarrow r_1 \text{ op } r_2$
2) $r_1 \leftarrow r_4 \text{ op } r_5$



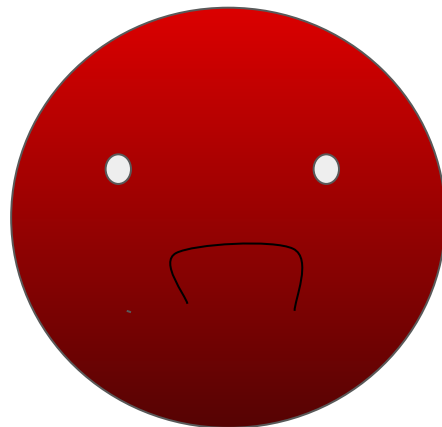
Output-dependence (Write-after-Write)

1) $r_3 \leftarrow r_1 \text{ op } r_2$
2) $r_5 \leftarrow r_3 \text{ op } r_4$
3) $r_3 \leftarrow r_6 \text{ op } r_7$



For all of them, we need to ensure semantics of the program is correct

- ❑ Flow dependences always need to be obeyed because they constitute true dependence on a value
- ❑ Anti and output dependences exist due to limited number of architectural registers
 - They are dependence on a name, not a value



Out-of-Order Execution (Dynamic Instruction Scheduling)

Tomasulo's Algorithm

Retirement RAT

Reg	TAG
R1	2
R2	8
R3	7
R4	
R5	

Frontend RAT

Reg	TAG
R1	2
R2	8
R3	7
R4	
R5	

Mul R3, R4, R5

add R2, R3, R1

	Source 1		Source 2	
	V	Tag	V	Tag
a	1	7	1	2
b				
c				
d				

Reservation station

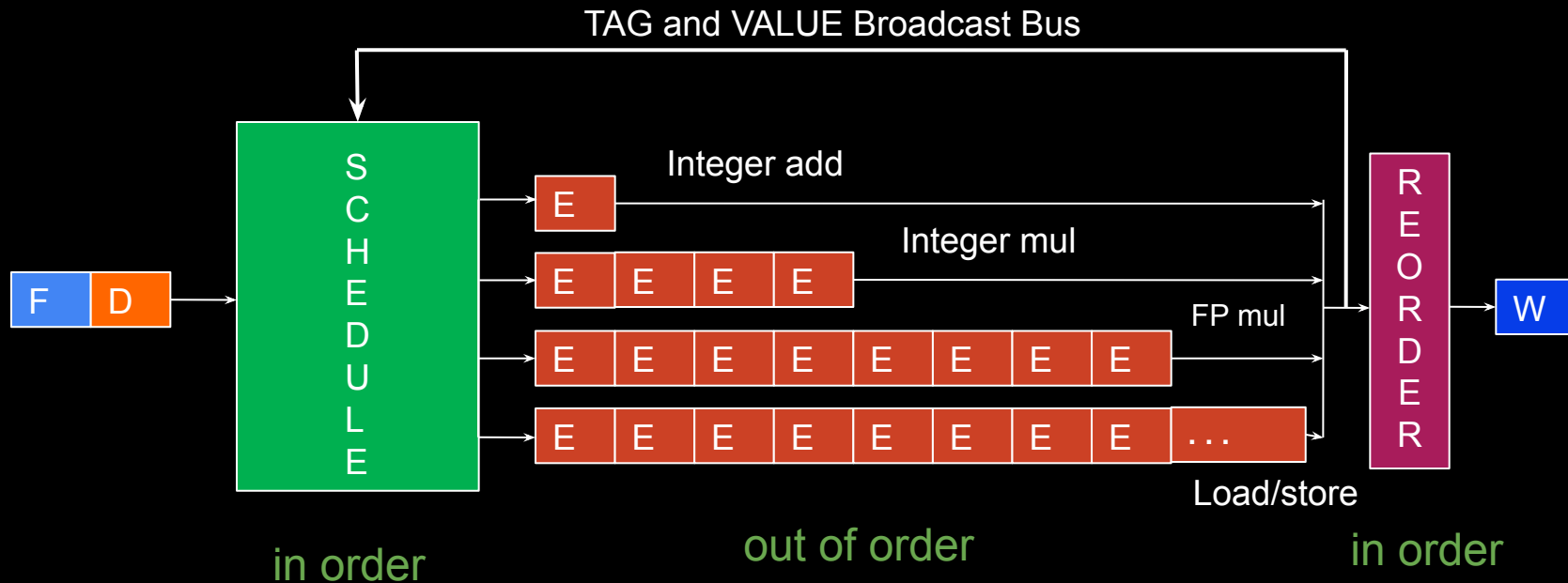
Reorder Buffer

TaG	Reg	V	Value
1	r1	0	x
2	r1	1	10
3	r2	0	x
4	r2	0	x
5	r2	0	x
7	r3	0	x
8	r2	0	x

oldest

youngest

Out-of-Order Execution with Precise Exceptions




- Hump 1: Reservation stations (scheduling window)
- Hump 2: Reordering (reorder buffer, aka instruction window or active window)

Data Dependence Types


Flow dependence (Read-after-Write)

1) $r_3 \leftarrow r_1 \text{ op } r_2$
2) $r_5 \leftarrow r_3 \text{ op } r_4$




Anti dependence (Write-after-Read)

1) $r_3 \leftarrow r_1 \text{ op } r_2$
2) $r_1 \leftarrow r_4 \text{ op } r_5$



Output-dependence (Write-after-Write)

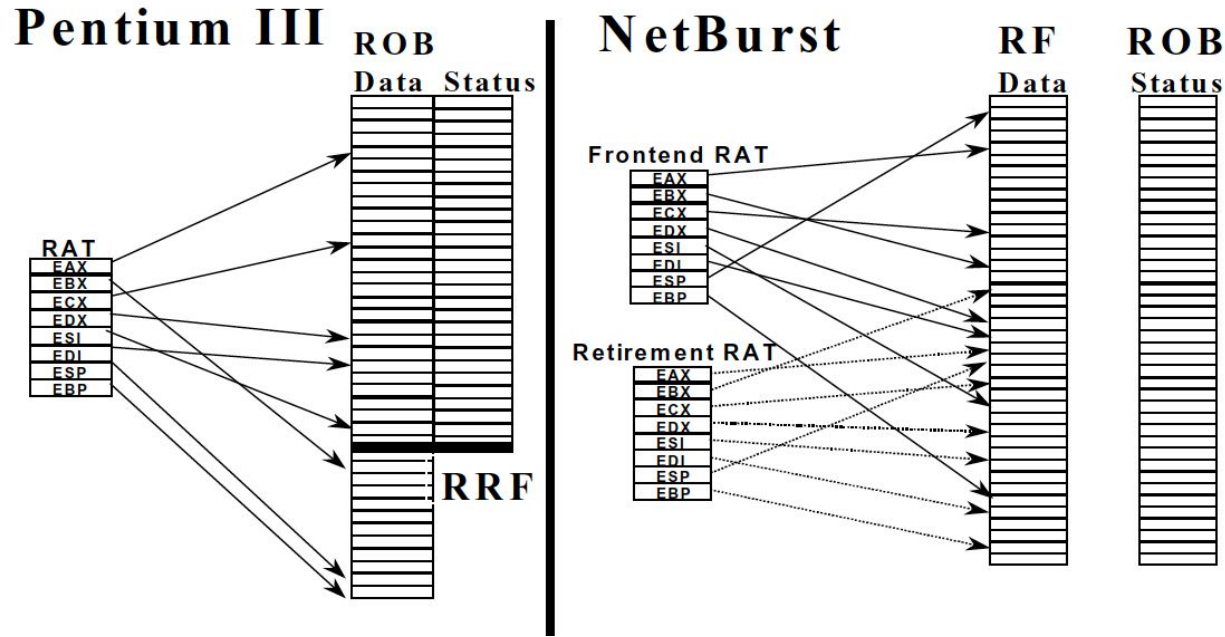
1) $r_3 \leftarrow r_1 \text{ op } r_2$
2) $r_5 \leftarrow r_3 \text{ op } r_4$
3) $r_3 \leftarrow r_6 \text{ op } r_7$



For all of them, we need to ensure semantics of the program is correct

- ❑ Flow dependences always need to be obeyed because they constitute true dependence on a value
- ❑ Anti and output dependences exist due to limited number of architectural registers
 - They are dependence on a name, not a value

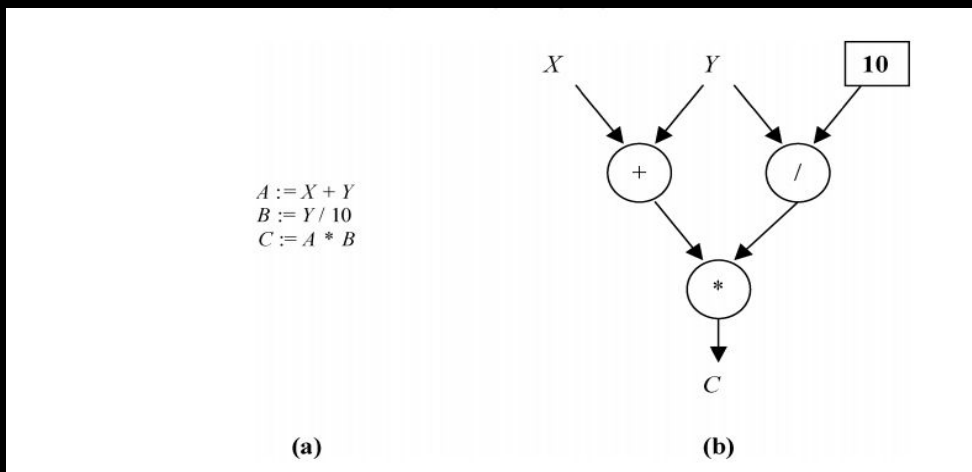
An Example from Modern Processors



Boggs et al., "The Microarchitecture of the Pentium 4 Processor," Intel Technology Journal, 2001.

Out-of-order dispatch

- Benefit:
 - **Latency tolerance:** Allows independent instructions to execute and complete in the presence of a long-latency operation



Enabling OoO Execution

1. Need to link the consumer of a value to the producer
 - Register renaming: Associate a “tag” with each data value
2. Need to buffer instructions until they are ready to execute
 - Insert instruction into reservation stations after renaming
3. Instructions need to keep track of readiness of source values
 - Broadcast the “tag” when the value is produced
 - Instructions compare their “source tags” to the broadcast tag □ if match, source value becomes ready
4. When all source values of an instruction are ready, need to dispatch the instruction to its functional unit (FU)
 - Instruction wakes up if all sources are ready
 - If multiple instructions are awake, need to select one per FU
- OoO variants are used in most high-performance processors
 - Initially in Intel Pentium Pro, AMD K5
 - Alpha 21264, MIPS R10000, IBM POWER5, IBM z196, Oracle UltraSPARC T4, ARM Cortex A15





Handling Out-of-Order Execution of Loads and Stores & TSO memory Model

Registers versus Memory

What are the fundamental differences between registers and memory?

- Register dependences known statically – memory dependences determined dynamically
- Register state is small – memory state is large
- Register state is not visible to other threads/processors – memory state is shared between threads/processors (in a shared memory multiprocessor)

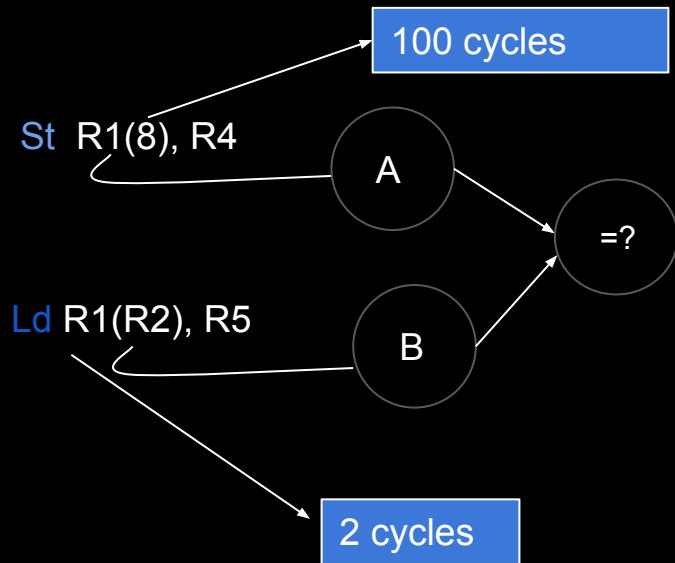
Memory Dependence Handling

- Need to obey memory dependences in an out-of-order machine
 - and need to do so while providing high performance
- Observation and Problem: Memory address is not known until a load/store executes
- Corollary 1: Renaming memory addresses is difficult
- Corollary 2: Determining dependence or independence of loads/stores has to be handled *after* their (partial) execution
- Corollary 3: When a load/store has its address ready, there may be older/younger stores/loads with unknown addresses in the machine

Memory Disambiguation Problem

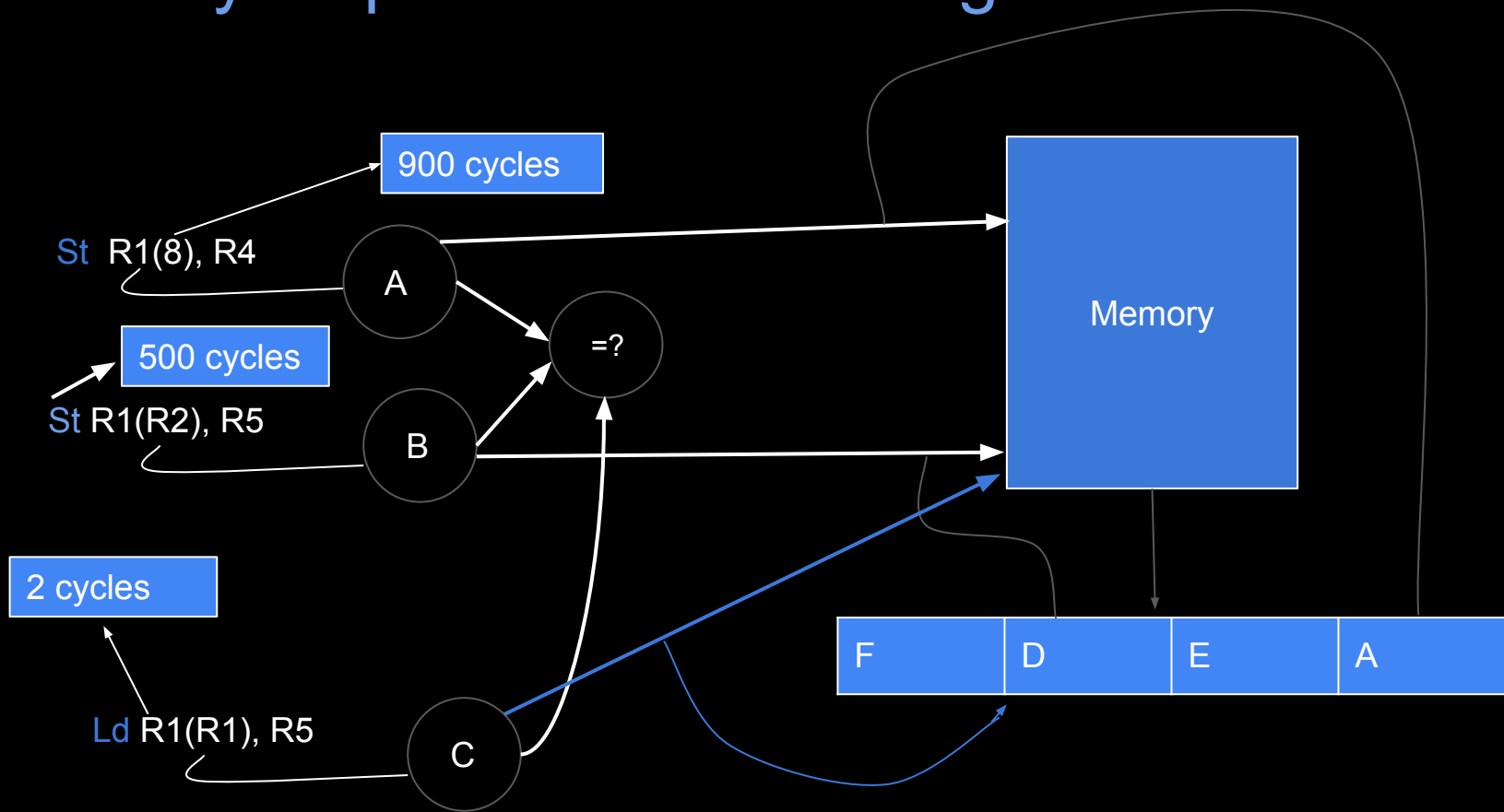
Memory Dependence Handling

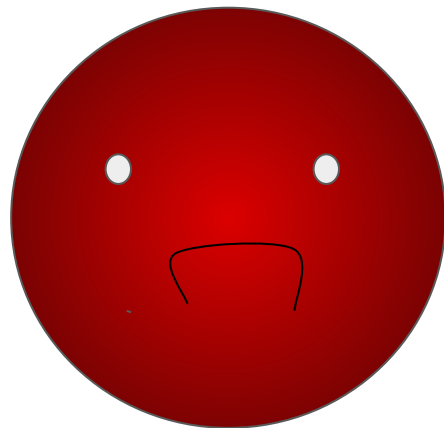
- When to schedule a load instruction in an OOO engine?
 - Problem: A younger load can have its address ready before an older store's address is known
 - Known as the **memory disambiguation** problem or the **unknown address** problem



Load and store have two different types of latencies.
Address calculation and **load/write from/to memory**

Memory Dependence Handling



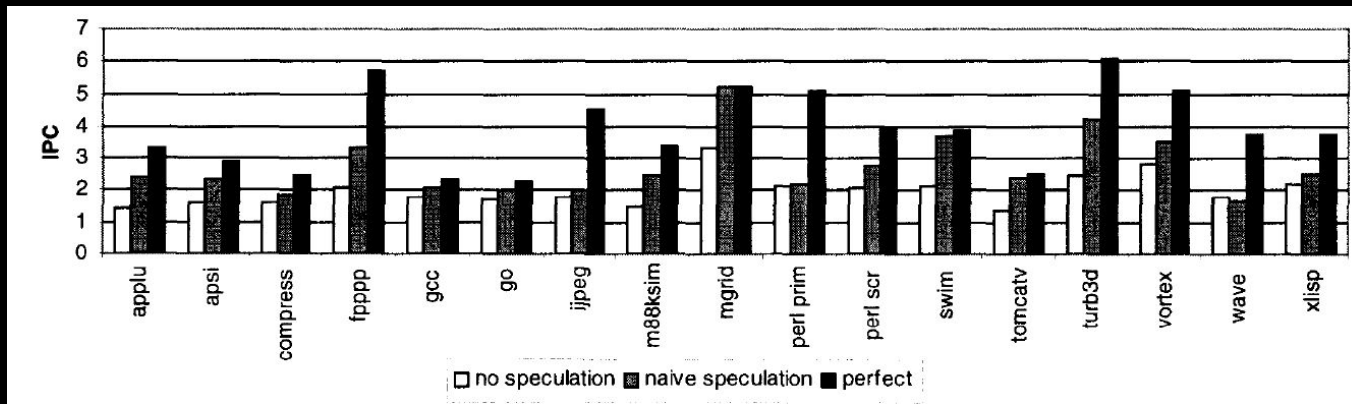


Handling of Store-Load Dependences

- **A load's dependence status is not known** until all previous store addresses are available.
- How does the OOO engine detect dependence of a load instruction on a previous store?
 - Option 1: Wait until all previous stores committed (no need to check for address match)
 - Option 2: Keep a list of pending stores in a store buffer and check whether load address matches a previous store address
- How does the OOO engine treat the scheduling of a load instruction wrt previous stores?
 - **Conservative:** Stall the load until all previous stores have computed their addresses (or even retired from the machine)
 - **Aggressive:** Assume load is independent of unknown-address stores and schedule the load right away
 - **Intelligent:** Predict (with a more sophisticated predictor) if the load is dependent on any unknown address store

Handling of Store-Load Dependences

- Chrysos and Emer, “Memory Dependence Prediction Using Store Sets,” ISCA 1998.



- Predicting store-load dependencies important for performance
- Simple predictors (based on past history) can achieve most of the potential performance

Data Forwarding Between Stores and Loads

- We cannot update memory out of program order
 - Need to buffer all store and load instructions in instruction window

- Even if we **know** all addresses of past stores when we generate the address of a load, two questions still remain:
 1. How do we check whether or not it is dependent on a store
 2. How do we forward data to the load if it is dependent on a store

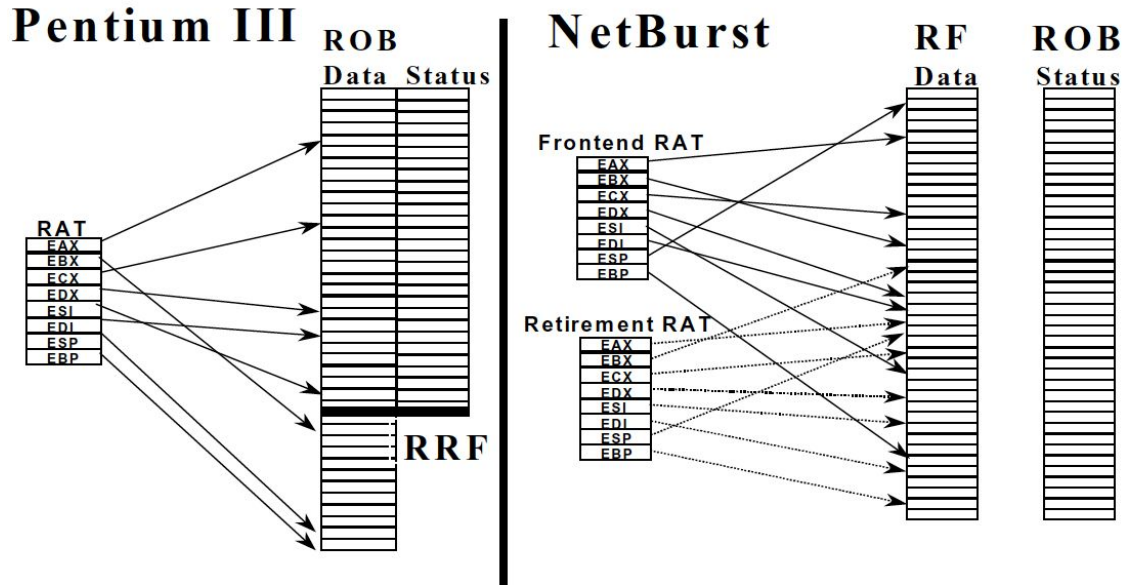
- Modern processors use a LQ (load queue) and a SQ for this
 - Can be combined or separate between loads and stores
 - A load searches the SQ after it computes its address. Why?
 - A store searches the LQ after it computes its address. Why?

Out-of-Order Completion of Memory Ops

- When a store instruction finishes execution, it writes its address and data in its reorder buffer entry (or SQ entry)
- When a later load instruction generates its address, it:
 - searches the SQ with its address
 - accesses memory with its address
 - receives the value from the youngest older instruction that wrote to that address (either from ROB or memory)
- This is a complicated “search logic” implemented as a Content Addressable Memory
 - Content is “memory address” (but also need *size* and *age*)
 - Called **store-to-load forwarding logic**

Renaming memory addresses is difficult

48-bit memory address can directly address every byte of 256 terabytes of storage.



Boggs et al., "The Microarchitecture of the Pentium 4 Processor," Intel Technology Journal, 2001.

Store-Load Forwarding Complexity

- Content Addressable Search (based on Load Address)
- Range Search (based on Address and Size of both the Load and earlier Stores)
- Age-Based Search (for last written values)
- Load data can come from a combination of multiple places
 - One or more stores in the Store Buffer (SQ)
 - Memory/cache

Store-Load Forwarding Complexity

HDD/SSD

DRAM

L2/ L3

L1 & MMU

Load X

=?

=?

=?

=?

=?

=?

=?

=?

=?

Reorder Buffer/SQ

Address	Ad Bit	Va Bit	Value
0x10	1		131
0x12	1		132
0x12	1		676
0x40	1		756
0x40	1		676
0x97	1		25

oldest

youngest

Content Addressable Search

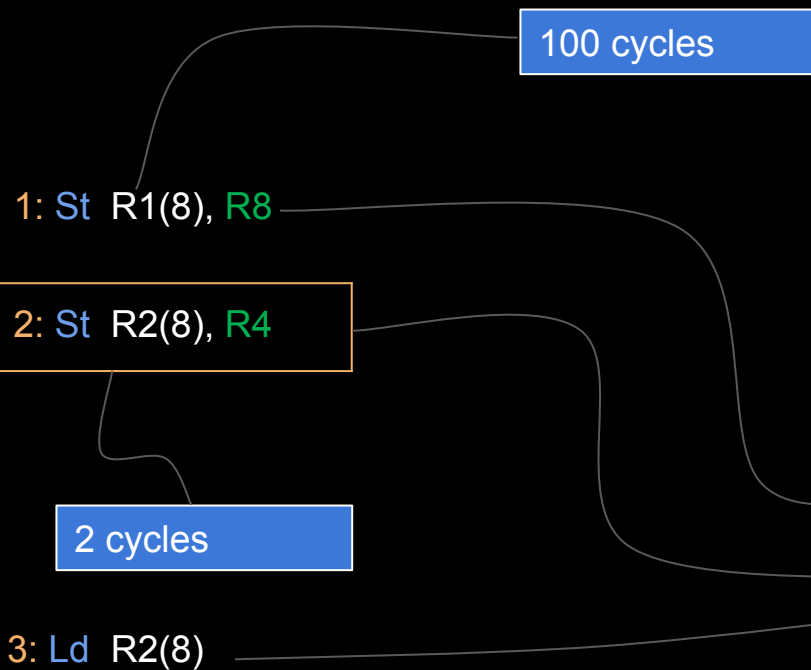
TSO memory Model & ROB

Reorder Buffer/SQ



TSO memory Model & ROB

Reorder Buffer/SQ



No need for fences between store and store.

Address	Ad Bit	Va Bit	Value
0x10	1		131
0x12	1		132
0x12	1		676
0x40	1		756
0x40	0	0	x
0x97	1	1	25

oldest

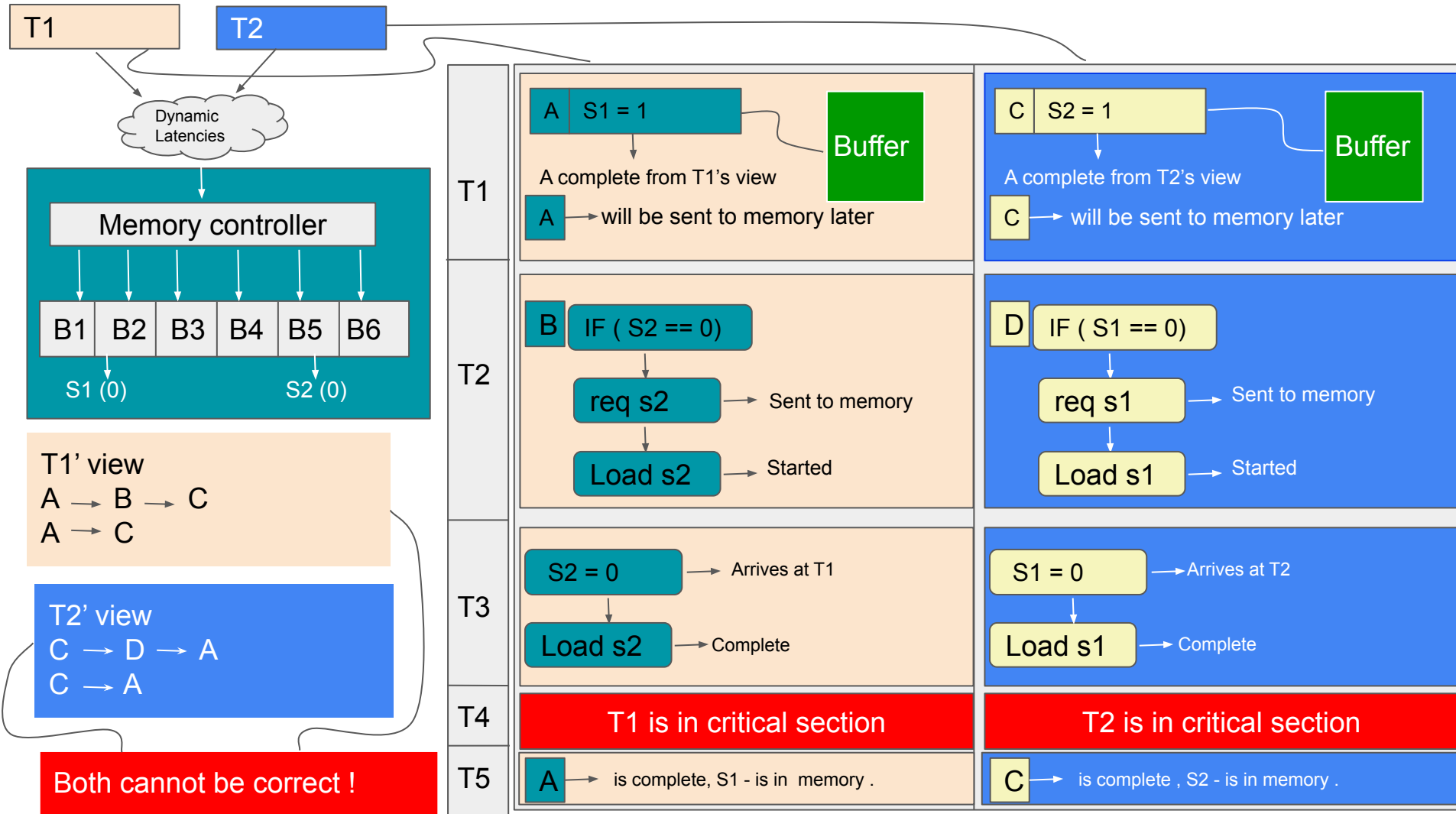
youngest

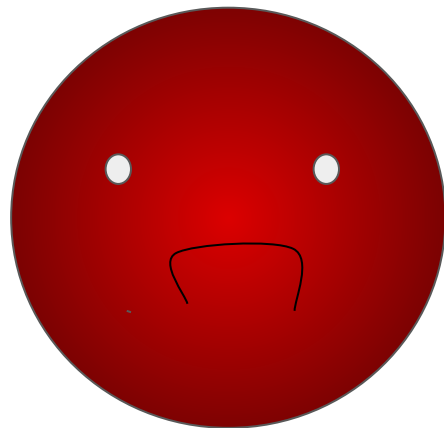
Why keep a buffer inside a core?

Performance.

Precise exception.

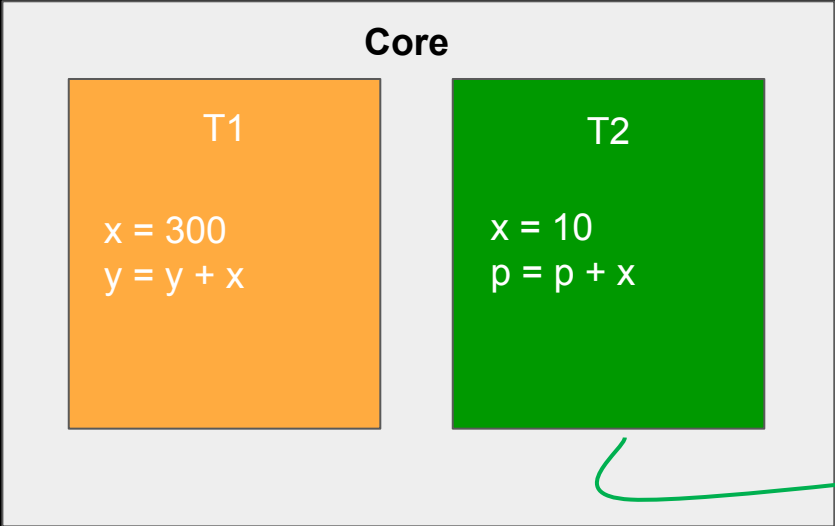
Dependency check.





TSO memory Model and Hyper-threading

x = 0
y = 0
p = 0



Reorder Buffer/SQ

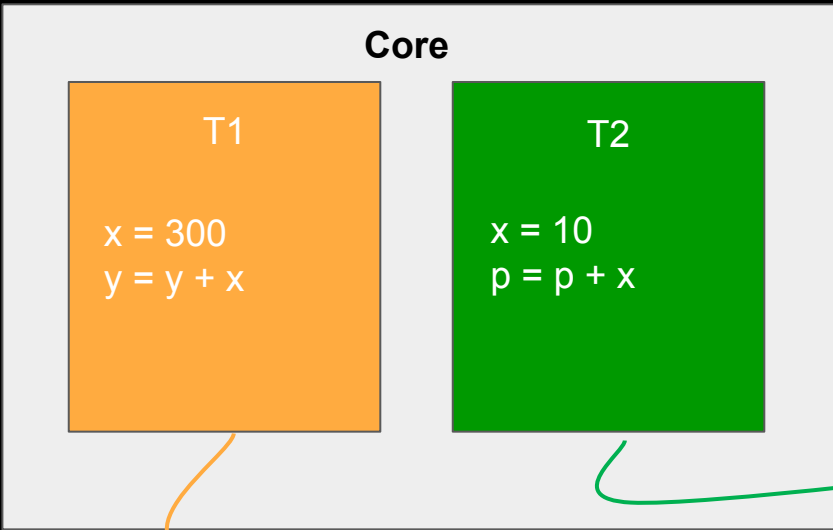
Address	Ad Bit	Va Bit	Value
0x10	1	1	131
0x12	1	1	132
0x12	1	1	676
0x40	1	1	756
0x40	1	1	676
0x97	1	1	25
0x100	1	1	10

oldest

youngest

TSO memory Model and Hyper-threading

x = 0
y = 0
p = 0



2: x = 300

1: x = 10

Reorder Buffer/SQ

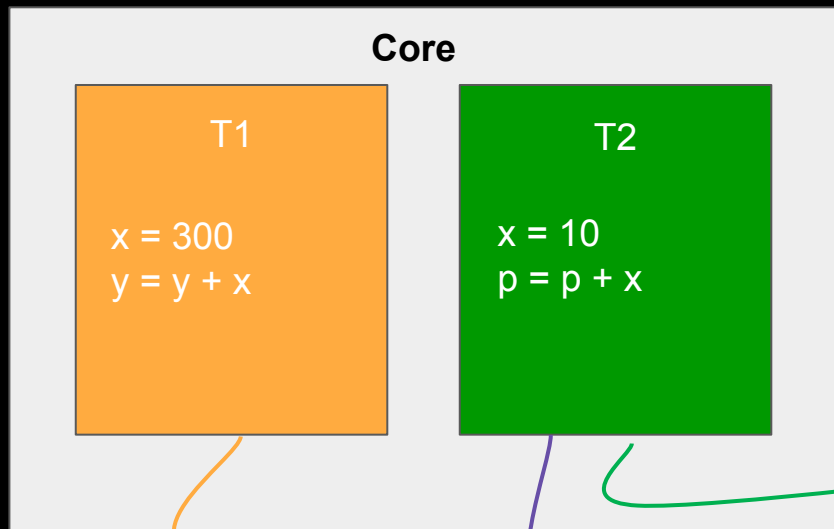
Address	Ad Bit	Va Bit	Value
0x10	1	1	131
0x12	1	1	132
0x12	1	1	676
0x40	1	1	756
0x40	1	1	676
0x97	1	1	25
0x100	1	1	10
0x100	1	1	300

oldest

youngest

TSO memory Model and Hyper-threading

$x = 0$
 $y = 0$
 $p = 0$



2: $x = 300$

1: $x = 10$

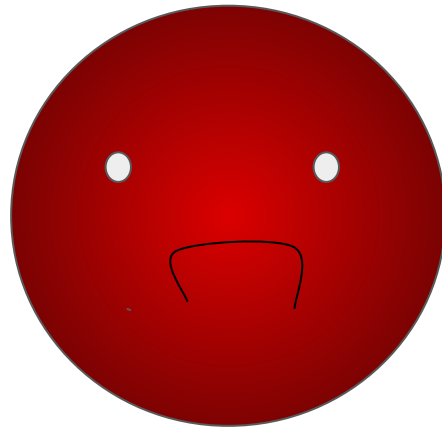
3: $p ?$

Reorder Buffer/SQ

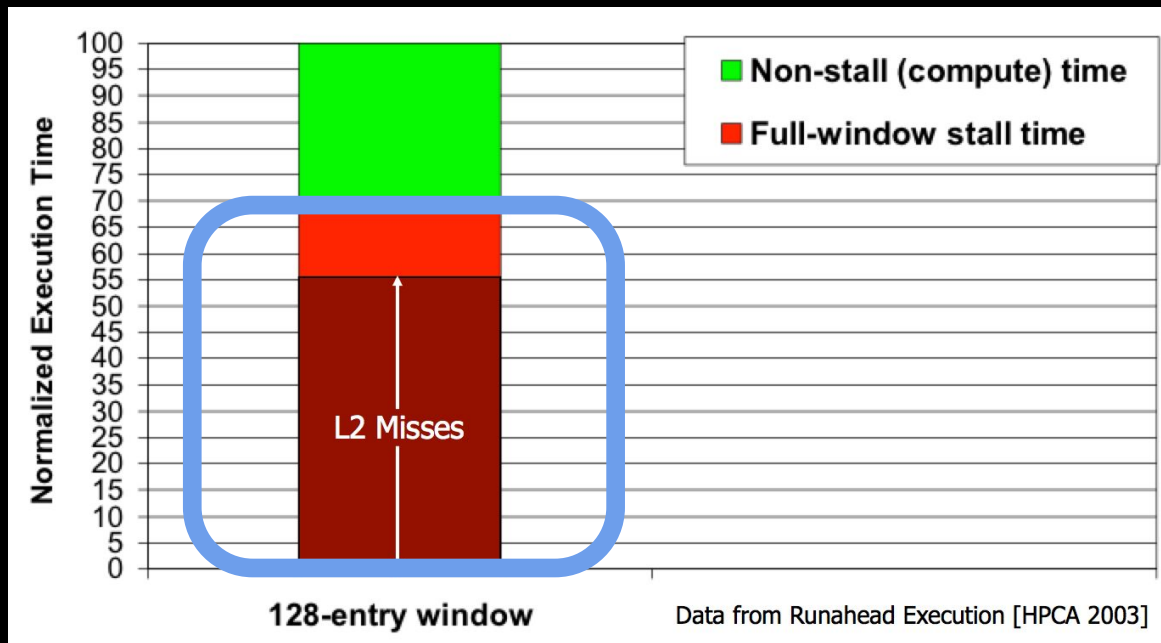
Address	Ad Bit	Va Bit	Value
0x10	1	1	131
0x12	1	1	132
0x12	1	1	676
0x40	1	1	756
0x40	1	1	676
0x97	1	1	25
0x100	1	1	10
0x100	1	1	300

oldest

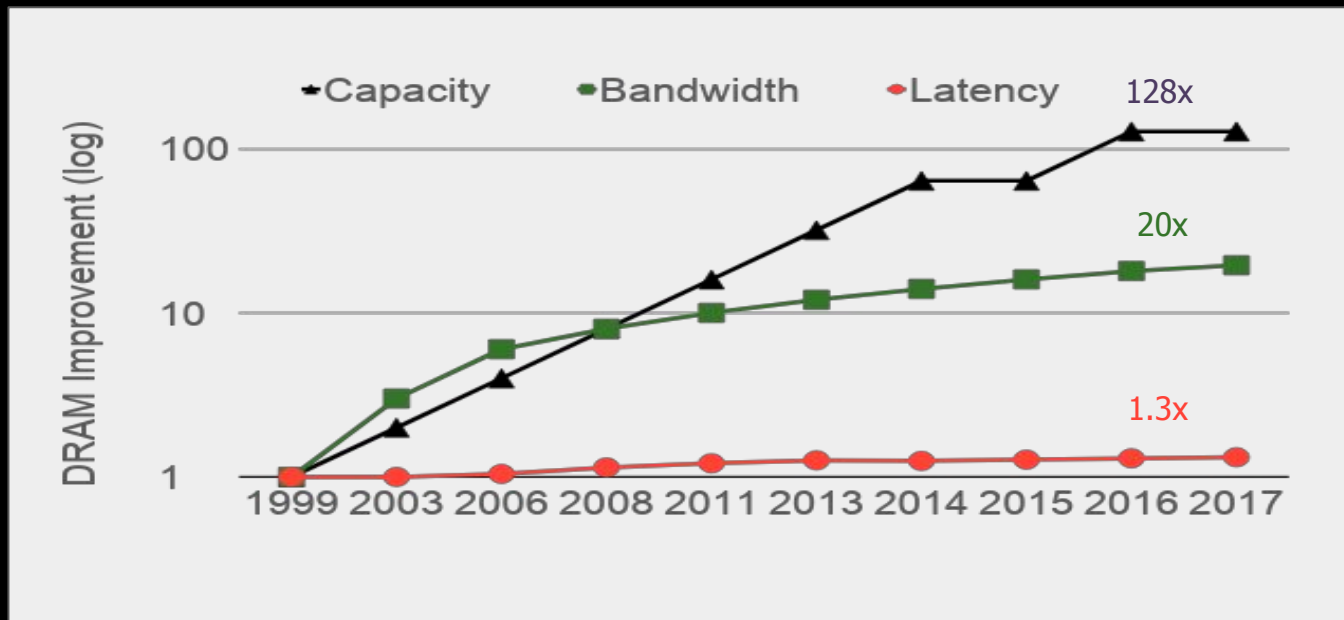
youngest



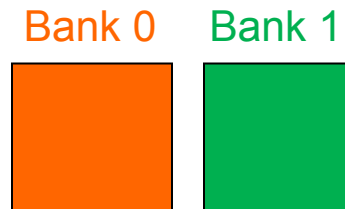
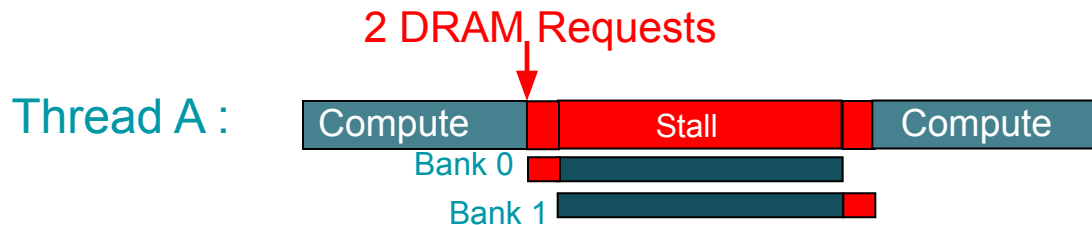
Memory Bottleneck



DRAM Capacity, Bandwidth & Latency



Memory Level Parallelism of a Thread



Thread A: Bank 0, Row 1

Thread A: Bank 1, Row 1

Bank access latencies of the two requests overlapped
Thread stalls for ~ONE bank access latency

“It’s the Memory, Stupid!”

– Richard Sites