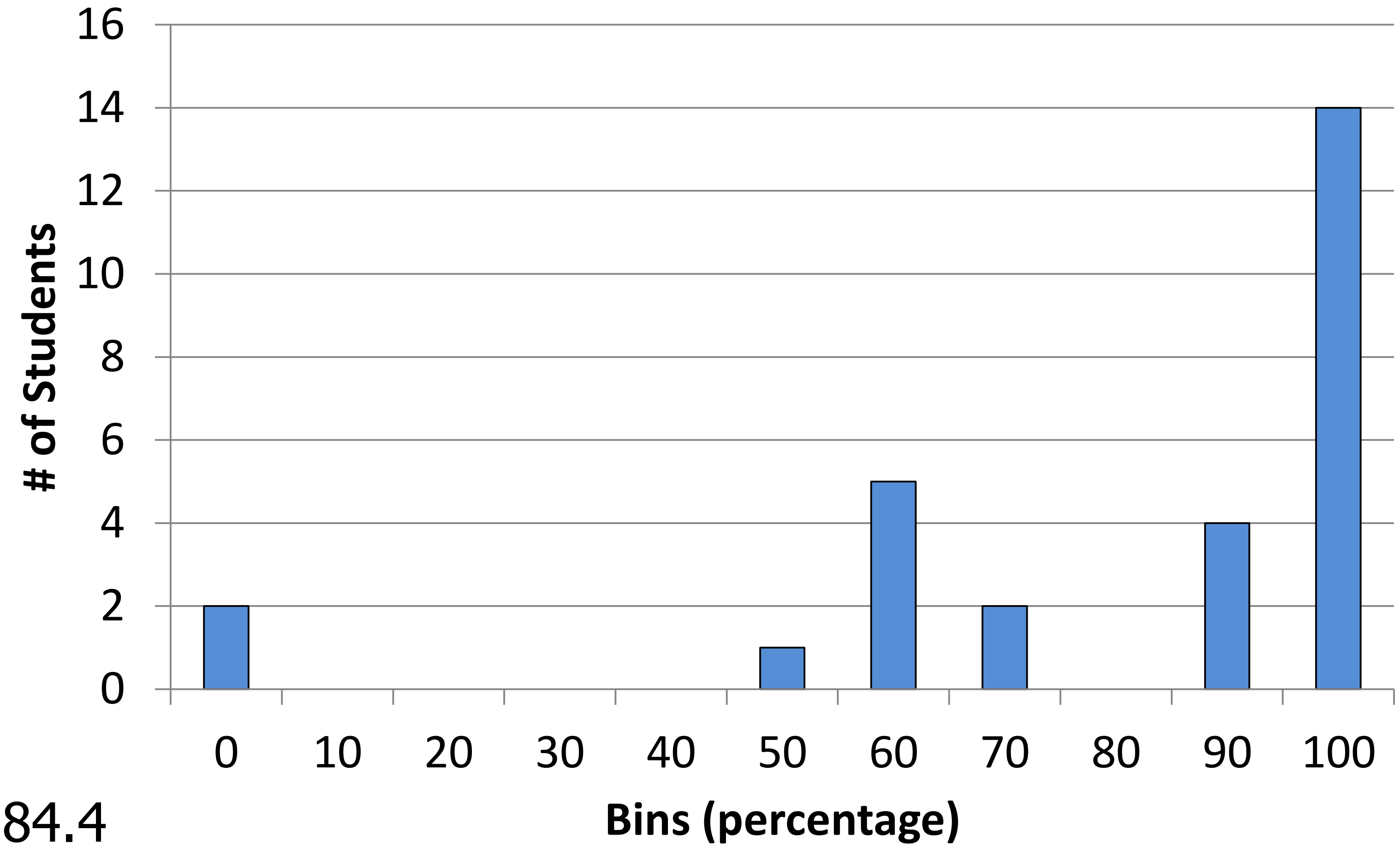
18-447
Computer Architecture
Lecture 24: Simulation and Memory Latency Tolerance

Prof. Onur Mutlu
Carnegie Mellon University

Spring 2015, 3/30/2015

Lab 5 Results



Avg: 84.4

Median: 93.8

Std Dev: 19.2

Reminder on Assignments

- Lab 6 due this Friday (April 3)
  - C-level simulation of data caches and branch prediction

- Homework 6 will be due April 10

- And, we will have a Midterm II

- The course will continue to move quickly… Keep your pace.
- Talk with the TAs and me if you need any help.
  - We cannot do or debug the assignments for you but we can give you suggestions
  - My goal is to enable you learn the material
    - You never know when you will use the principles you learn

Simulation: The Field of Dreams

Dreaming and Reality

- An architect is in part a dreamer, a creator
- Simulation is a key tool of the architect
- Simulation enables
  - The exploration of many dreams
  - A reality check of the dreams
  - Deciding which dream is better
- Simulation also enables
  - The ability to fool yourself with false dreams

Why High-Level Simulation?

- Problem: RTL simulation is intractable for design space exploration → too time consuming to design and evaluate
  - Especially over a large number of workloads
  - Especially if you want to predict the performance of a good chunk of a workload on a particular design
  - Especially if you want to consider many design choices
    - Cache size, associativity, block size, algorithms
    - Memory control and scheduling algorithms
    - In-order vs. out-of-order execution
    - Reservation station sizes, ld/st queue size, register file size, …
    - …

- Goal: Explore design choices quickly to see their impact on the workloads we are designing the platform for

Different Goals in Simulation

- Explore the design space quickly and see what you want to
  - potentially implement in a next-generation platform
  - propose as the next big idea to advance the state of the art
  - the goal is mainly to see relative effects of design decisions
- Match the behavior of an existing system so that you can
  - debug and verify it at cycle-level accuracy
  - propose small tweaks to the design that can make a difference in performance or energy
  - the goal is very high accuracy
- Other goals in-between:
  - Refine the explored design space without going into a full detailed, cycle-accurate design
  - Gain confidence in your design decisions made by higher-level design space exploration

Tradeoffs in Simulation

- Three metrics to evaluate a simulator
  - Speed
  - Flexibility
  - Accuracy

- Speed: How fast the simulator runs (xIPS, xCPS)
- Flexibility: How quickly one can modify the simulator to evaluate different algorithms and design choices?
- Accuracy: How accurate the performance (energy) numbers the simulator generates are vs. a real design (Simulation error)

- The relative importance of these metrics varies depending on where you are in the design process

Trading Off Speed, Flexibility, Accuracy

- Speed & flexibility affect:
  - How quickly you can make design tradeoffs
- Accuracy affects:
  - How good your design tradeoffs may end up being
  - How fast you can build your simulator (simulator design time)
- Flexibility also affects:
  - How much human effort you need to spend modifying the simulator
- You can trade off between the three to achieve design exploration and decision goals

High-Level Simulation

- Key Idea: Raise the abstraction level of modeling to give up some accuracy to enable speed & flexibility (and quick simulator design)

- Advantage

  + Can still make the right tradeoffs, and can do it quickly

  + All you need is modeling the key high-level factors, you can omit corner case conditions

  + All you need is to get the "relative trends" accurately, not exact performance numbers

- Disadvantage

  -- Opens up the possibility of potentially wrong decisions

  -- How do you ensure you get the "relative trends" accurately?

Simulation as Progressive Refinement

- High-level models (Abstract, C)
- …
- Medium-level models (Less abstract)
- …
- Low-level models (RTL with eveything modeled)
- …
- Real design

- As you refine (go down the above list)
  - Abstraction level reduces
  - Accuracy (hopefully) increases (not necessarily, if not careful)
  - Speed and flexibility reduce
  - You can loop back and fix higher-level models

This Course

- A good architect is comfortable at all levels of refinement
  - Including the extremes

- This course, as a result, gives you a flavor of both:
  - High-level, abstract simulation (Labs 6, 7, 8)
  - Low-level, RTL simulation (Labs 2, 3, 4, 5)

Optional Reading on DRAM Simulation

- Kim et al., "Ramulator: A Fast and Extensible DRAM Simulator," IEEE Computer Architecture Letters 2015.

- https://github.com/CMU-SAFARI/ramulator

- http://users.ece.cmu.edu/~omutlu/pub/ramulator_dram_simulator-ieee-cal15.pdf

Where We Are in Lecture Schedule

- The memory hierarchy
- Caches, caches, more caches
- Virtualizing the memory hierarchy: Virtual Memory
- Main memory: DRAM
- Main memory control, scheduling
- Memory latency tolerance techniques
- Non-volatile memory

- Multiprocessors
- Coherence and consistency
- Interconnection networks
- Multi-core issues (e.g., heterogeneous multi-core)

Upcoming Seminar on DRAM (April 3)

- April 3, Friday, 11am-noon, GHC 8201
- Prof. Moinuddin Qureshi, Georgia Tech
  - Lead author of "MLP-Aware Cache Replacement"
- Architecting 3D Memory Systems
  - Die stacked 3D DRAM technology can provide low-energy high-bandwidth memory module by vertically integrating several dies within the same chip. (…) In this talk, I will discuss how memory systems can efficiently architect 3D DRAM either as a cache or as main memory. First, I will show that some of the basic design decisions typically made for conventional caches (such as serialization of tag and data access, large associativity, and update of replacement state) are detrimental to the performance of DRAM caches, as they exacerbate hit latency. (…) Finally, I will present a memory organization that allows 3D DRAM to be a part of the OS-visible memory address space, and yet relieves the OS from data migration duties. (…)"

Required Reading

- Onur Mutlu, Justin Meza, and Lavanya Subramanian, **"The Main Memory System: Challenges and Opportunities"** *Invited Article in Communications of the Korean Institute of Information Scientists and Engineers* ***(KIISE)***, 2015.

http://users.ece.cmu.edu/~omutlu/pub/main-memory-system_kiise15.pdf

Required Readings on DRAM

- DRAM Organization and Operation Basics
  - Sections 1 and 2 of: Lee et al., "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," HPCA 2013.

    http://users.ece.cmu.edu/~omutlu/pub/tldram_hpca13.pdf

  - Sections 1 and 2 of Kim et al., "A Case for Subarray-Level Parallelism (SALP) in DRAM," ISCA 2012.

    http://users.ece.cmu.edu/~omutlu/pub/salp-dram_isca12.pdf

- DRAM Refresh Basics
  - Sections 1 and 2 of Liu et al., "RAIDR: Retention-Aware Intelligent DRAM Refresh," ISCA 2012. http://users.ece.cmu.edu/~omutlu/pub/raidr-dram-refresh_isca12.pdf

Readings on Bloom Filters

- Section 3.1 of
  - Seshadri et al., "The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing," PACT 2012.

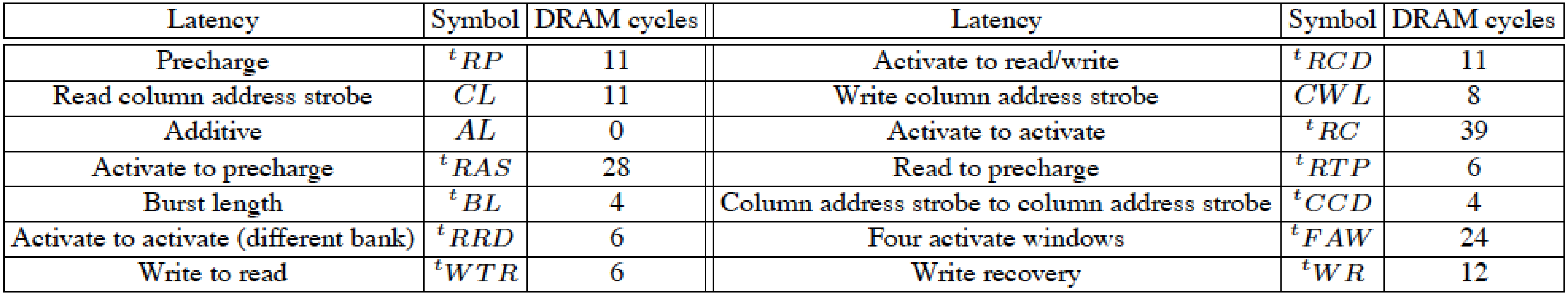
- Section 3.3 of
  - Liu et al., "RAIDR: Retention-Aware Intelligent DRAM Refresh," ISCA 2012.

Difficulty of DRAM Control

Why are DRAM Controllers Difficult to Design?

- Need to obey DRAM timing constraints for correctness
  - There are many (50+) timing constraints in DRAM
  - tWTR: Minimum number of cycles to wait before issuing a read command after a write command is issued
  - tRC: Minimum number of cycles between the issuing of two consecutive activate commands to the same bank
  - …
- Need to keep track of many resources to prevent conflicts
  - Channels, banks, ranks, data bus, address bus, row buffers
- Need to handle DRAM refresh
- Need to manage power consumption
- Need to optimize performance & QoS (in the presence of constraints)
  - Reordering is not simple
  - Fairness and QoS needs complicates the scheduling problem

Many DRAM Timing Constraints



Latency	Symbol	DRAM cycles	Latency	Symbol	DRAM cycles
Precharge	^tRP	11	Activate to read/write	^tRCD	11
Read column address strobe	CL	11	Write column address strobe	CWL	8
Additive	AL	0	Activate to activate	^tRC	39
Activate to precharge	^tRAS	28	Read to precharge	^tRTP	6
Burst length	^tBL	4	Column address strobe to column address strobe	^tCCD	4
Activate to activate (different bank)	^tRRD	6	Four activate windows	^tFAW	24
Write to read	^tWTR	6	Write recovery	^tWR	12

Table 4. DDR3 1600 DRAM timing specifications

- From Lee et al., "DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems," HPS Technical Report, April 2010.

More on DRAM Operation

- Kim et al., "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," ISCA 2012.
- Lee et al., "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," HPCA 2013.

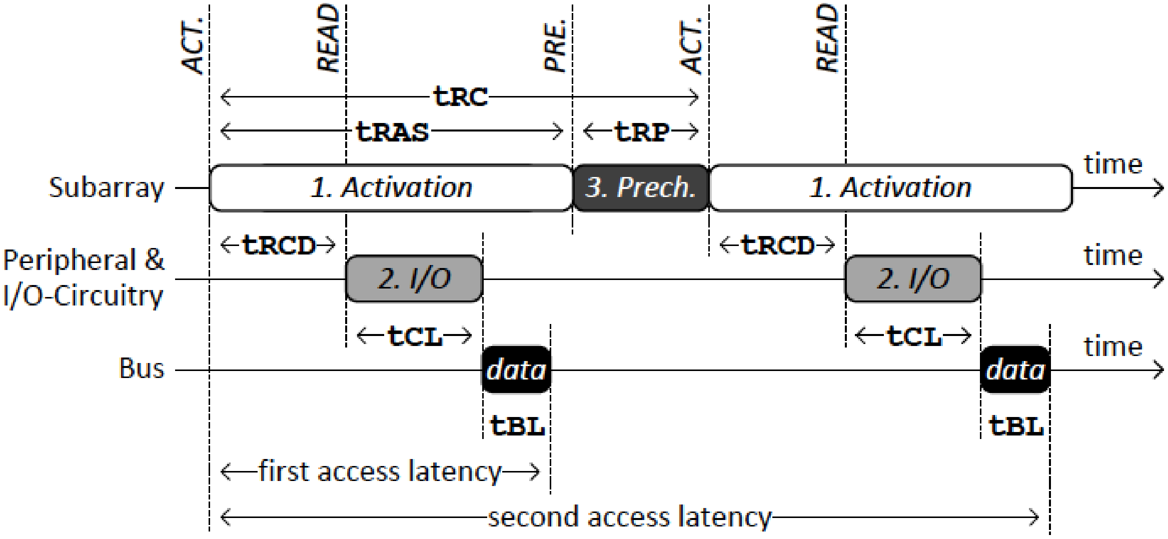
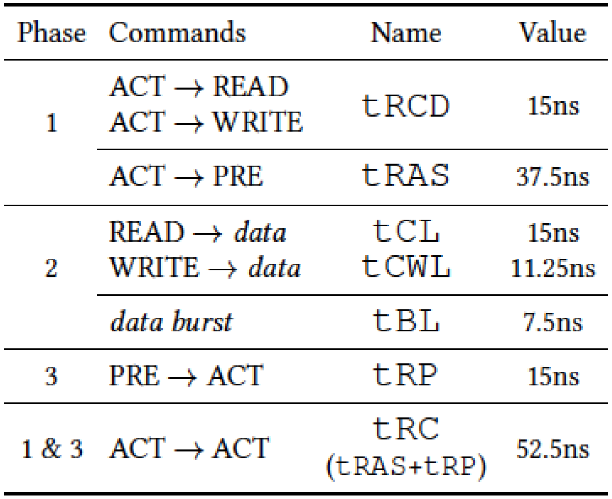


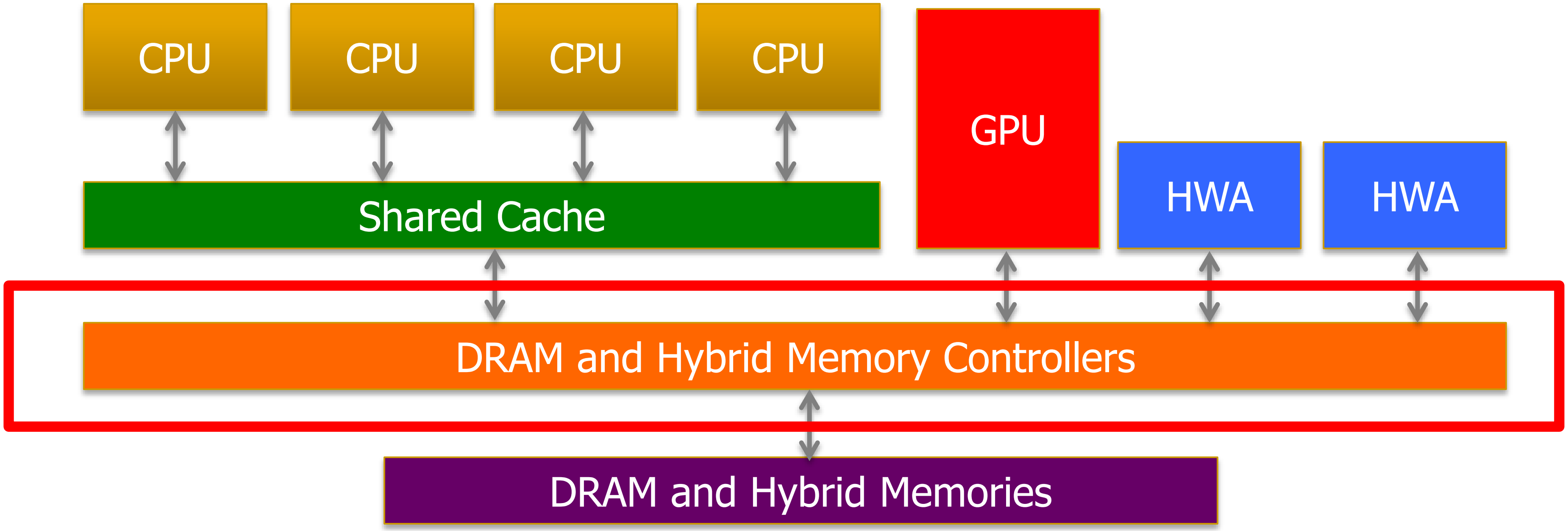
Figure 5. Three Phases of DRAM Access

Table 2. Timing Constraints (DDR3-1066) [43]



Phase	Commands	Name	Value
1	ACT → READ ACT → WRITE	tRCD	15ns
	ACT → PRE	tRAS	37.5ns
2	READ → *data* WRITE → *data*	tCL tCWL	15ns 11.25ns
	data burst	tBL	7.5ns
3	PRE → ACT	tRP	15ns
1 & 3	ACT → ACT	tRC (tRAS+tRP)	52.5ns

DRAM Controller Design Is Becoming More Difficult



- Heterogeneous agents: CPUs, GPUs, and HWAs
- Main memory interference between CPUs, GPUs, HWAs
- Many timing constraints for various memory types
- Many goals at the same time: performance, fairness, QoS, energy efficiency, …

Reality and Dream

- Reality: It difficult to optimize all these different constraints while maximizing performance, QoS, energy-efficiency, …

- Dream: Wouldn't it be nice if the DRAM controller automatically found a good scheduling policy on its own?

Self-Optimizing DRAM Controllers

- Problem: DRAM controllers difficult to design → It is difficult for human designers to design a policy that can adapt itself very well to different workloads and different system conditions

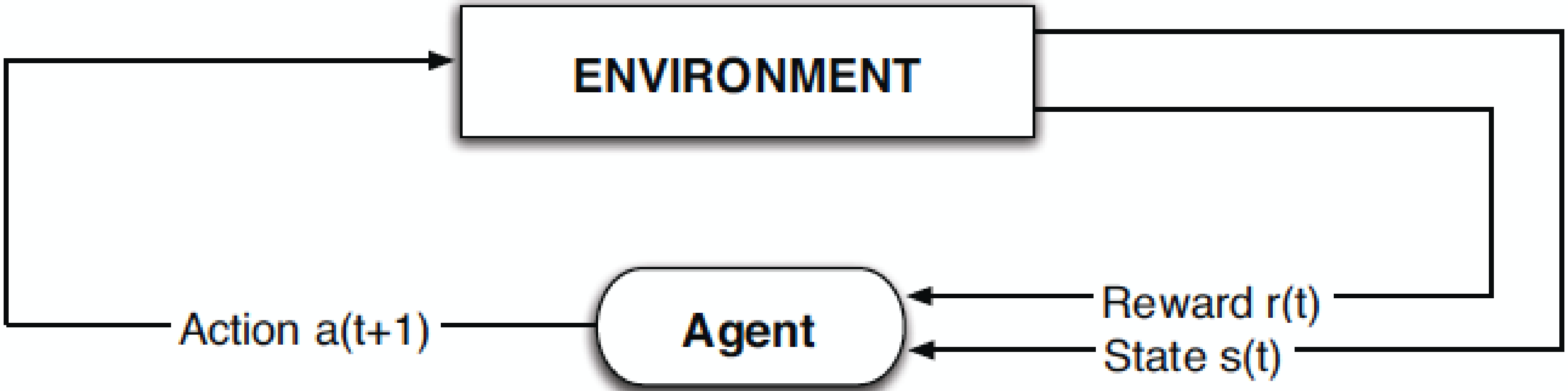
- Idea: Design a memory controller that adapts its scheduling policy decisions to workload behavior and system conditions using machine learning.

- Observation: Reinforcement learning maps nicely to memory control.

- Design: Memory controller is a reinforcement learning agent that dynamically and continuously learns and employs the best scheduling policy.

Ipek+, "Self Optimizing Memory Controllers: A Reinforcement Learning Approach," ISCA 2008.

Self-Optimizing DRAM Controllers

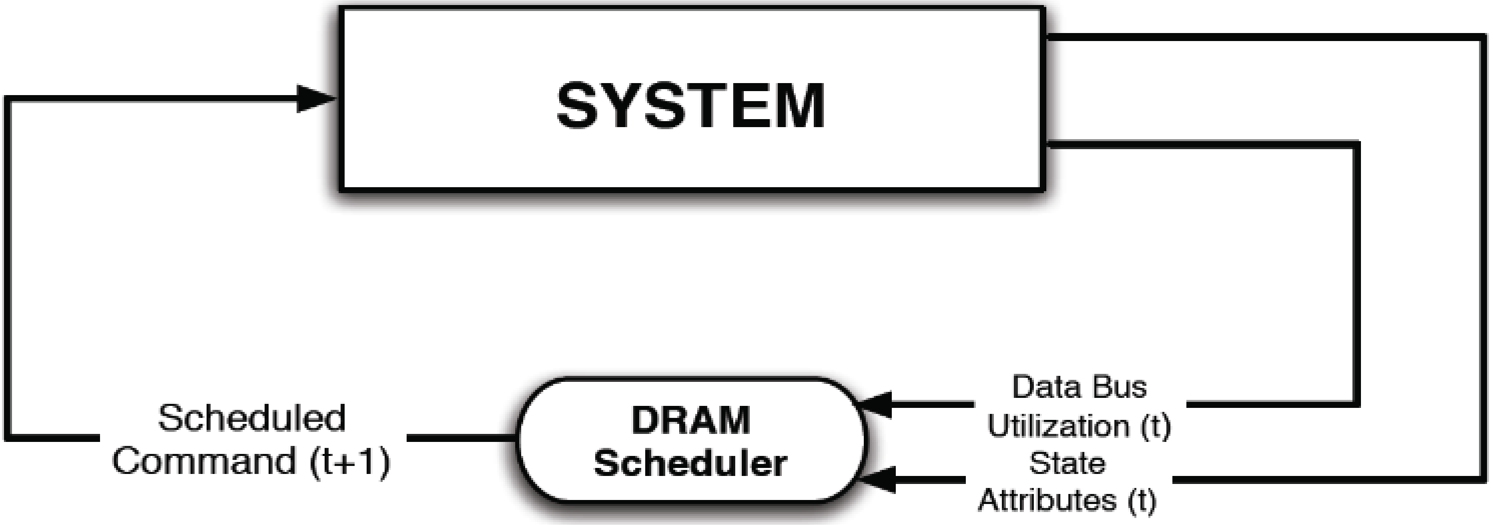


Goal: Learn to choose actions to maximize $r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots$ ($0 \leq \gamma < 1$)

Figure 2: (a) Intelligent agent based on reinforcement learning principles;

Self-Optimizing DRAM Controllers

- Dynamically adapt the memory scheduling policy via interaction with the system at runtime
  - Associate system states and actions (commands) with long term reward values: each action at a given state leads to a learned reward
  - Schedule command with highest estimated long-term reward value in each state
  - Continuously update reward values for <state, action> pairs based on feedback from system



Self-Optimizing DRAM Controllers

- Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana, **"Self Optimizing Memory Controllers: A Reinforcement Learning Approach"** *Proceedings of the 35th International Symposium on Computer Architecture (**ISCA**)*, pages 39-50, Beijing, China, June 2008.

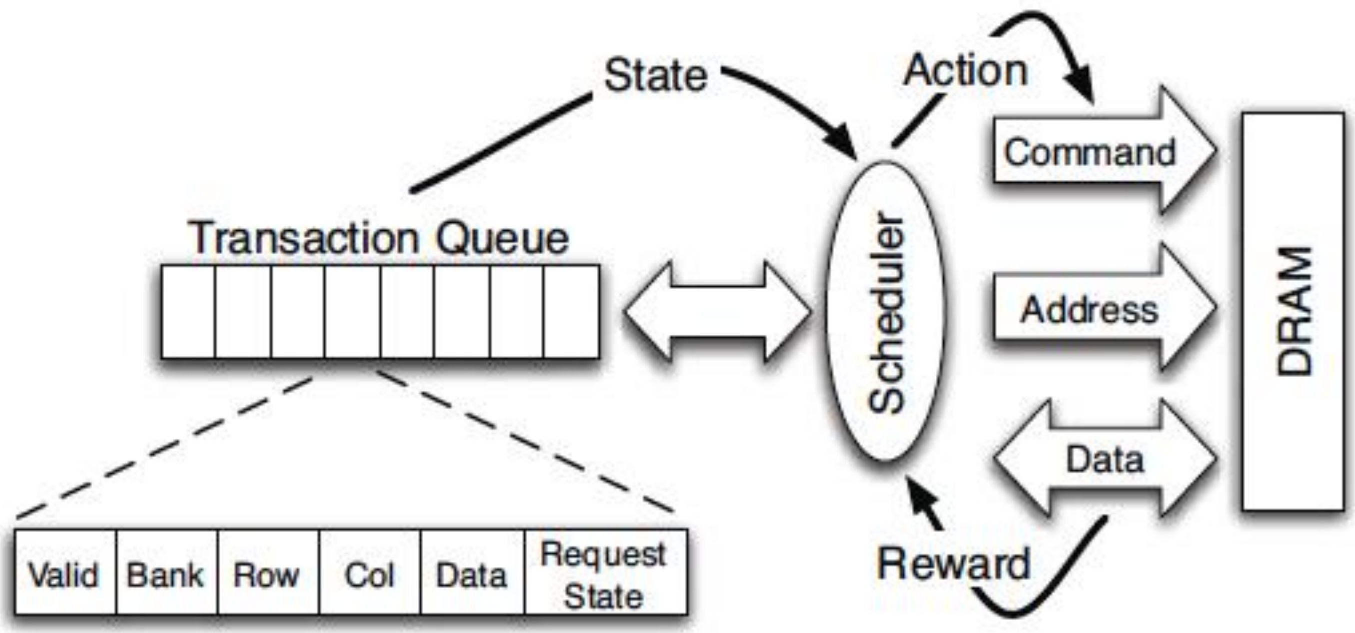


Figure 4: High-level overview of an RL-based scheduler.

States, Actions, Rewards

- Reward function
  - +1 for scheduling Read and Write commands
  - 0 at all other times

  Goal is to maximize data bus utilization

- State attributes
  - Number of reads, writes, and load misses in transaction queue
  - Number of pending writes and ROB heads waiting for referenced row
  - Request's relative ROB order

- Actions
  - Activate
  - Write
  - Read - load miss
  - Read - store miss
  - Precharge - pending
  - Precharge - preemptive
  - NOP

Performance Results

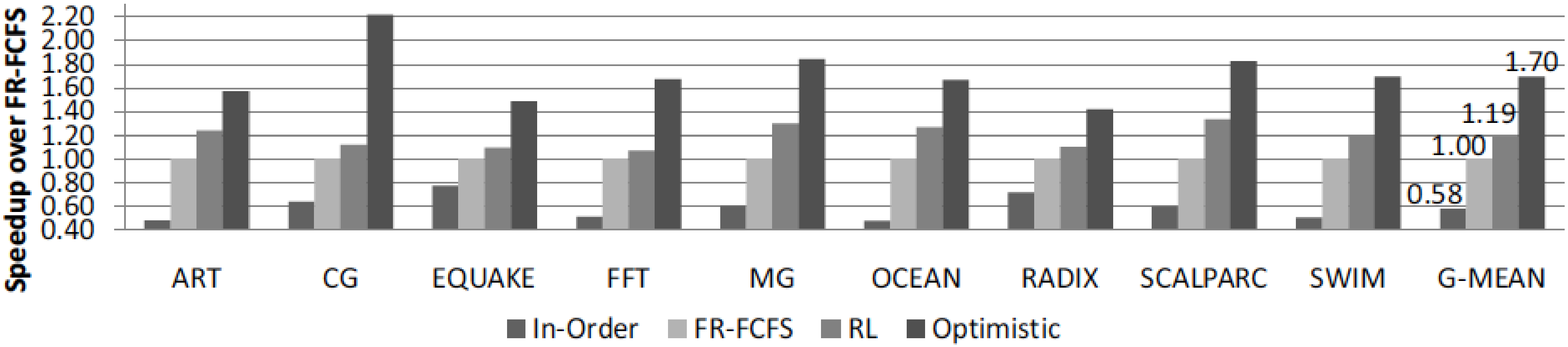


Figure 7: Performance comparison of in-order, FR-FCFS, RL-based, and optimistic memory controllers

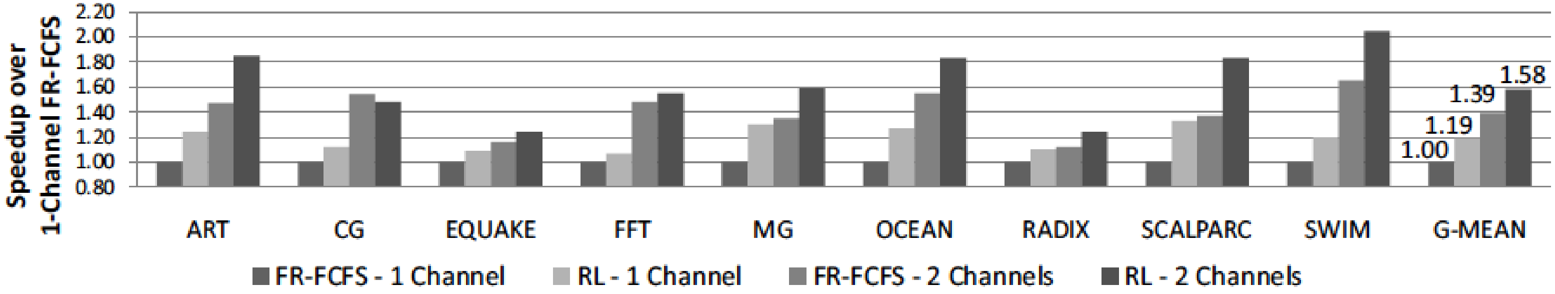


Figure 15: Performance comparison of FR-FCFS and RL-based memory controllers on systems with 6.4GB/s and 12.8GB/s peak DRAM bandwidth

Self Optimizing DRAM Controllers

- Advantages

  + Adapts the scheduling policy dynamically to changing workload behavior and to maximize a long-term target

  + Reduces the designer's burden in finding a good scheduling policy. Designer specifies:

    1) What system variables might be useful

    2) What target to optimize, but not how to optimize it

- Disadvantages and Limitations

  -- Black box: designer much less likely to implement what she cannot easily reason about

  -- How to specify different reward functions that can achieve different objectives? (e.g., fairness, QoS)

  -- Hardware complexity?

Memory Latency Tolerance

Readings on Memory Latency Tolerance

- Required
  - Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," HPCA 2003.
  - Srinath et al., "Feedback directed prefetching", HPCA 2007.

- Optional
  - Mutlu et al., "Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance," ISCA 2005, IEEE Micro Top Picks 2006.
  - Mutlu et al., "Address-Value Delta (AVD) Prediction," MICRO 2005.
  - Armstrong et al., "Wrong Path Events," MICRO 2004.

Remember: Latency Tolerance

- An out-of-order execution processor tolerates latency of multi-cycle operations by executing independent instructions concurrently
  - It does so by buffering instructions in reservation stations and reorder buffer
  - Instruction window: Hardware resources needed to buffer all decoded but not yet retired/committed instructions

- What if an instruction takes 500 cycles?
  - How large of an instruction window do we need to continue decoding?
  - How many cycles of latency can OoO tolerate?

Stalls due to Long-Latency Instructions

- When a long-latency instruction is not complete, it blocks instruction retirement.
  - Because we need to maintain precise exceptions

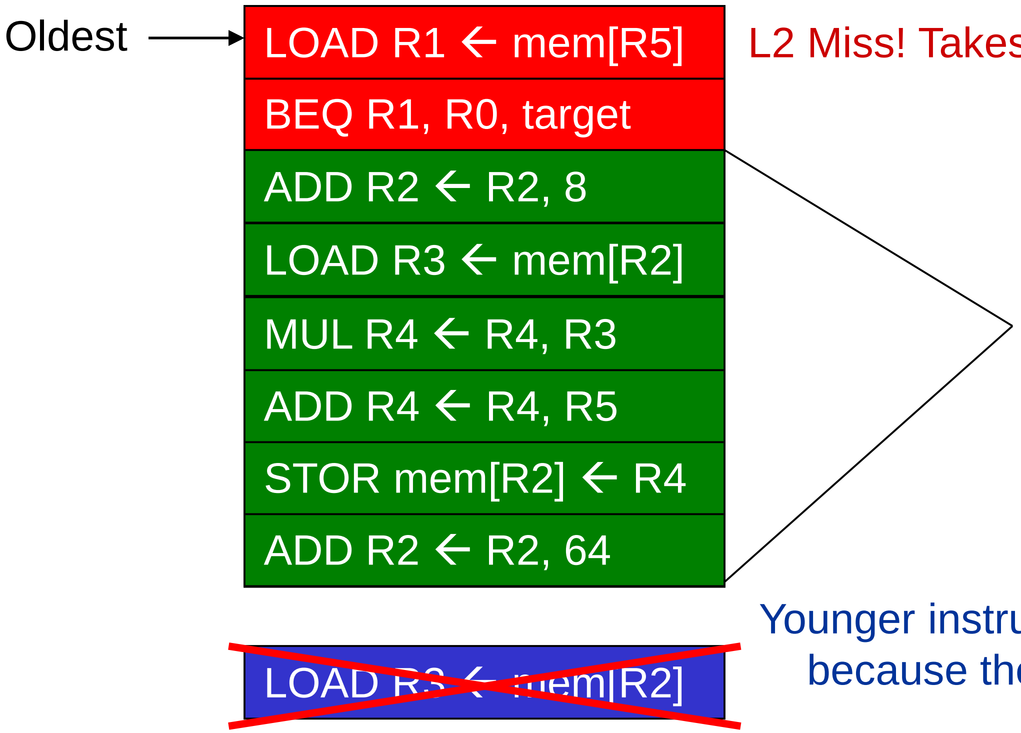
- Incoming instructions fill the instruction window (reorder buffer, reservation stations).

- Once the window is full, processor cannot place new instructions into the window.
  - This is called a full-window stall.

- A full-window stall prevents the processor from making progress in the execution of the program.

Full-window Stall Example

8-entry instruction window:



Oldest

LOAD R1 ← mem[R5]

BEQ R1, R0, target

ADD R2 ← R2, 8

LOAD R3 ← mem[R2]

MUL R4 ← R4, R3

ADD R4 ← R4, R5

STOR mem[R2] ← R4

ADD R2 ← R2, 64

~~LOAD R3 ← mem[R2]~~

L2 Miss! Takes 100s of cycles.

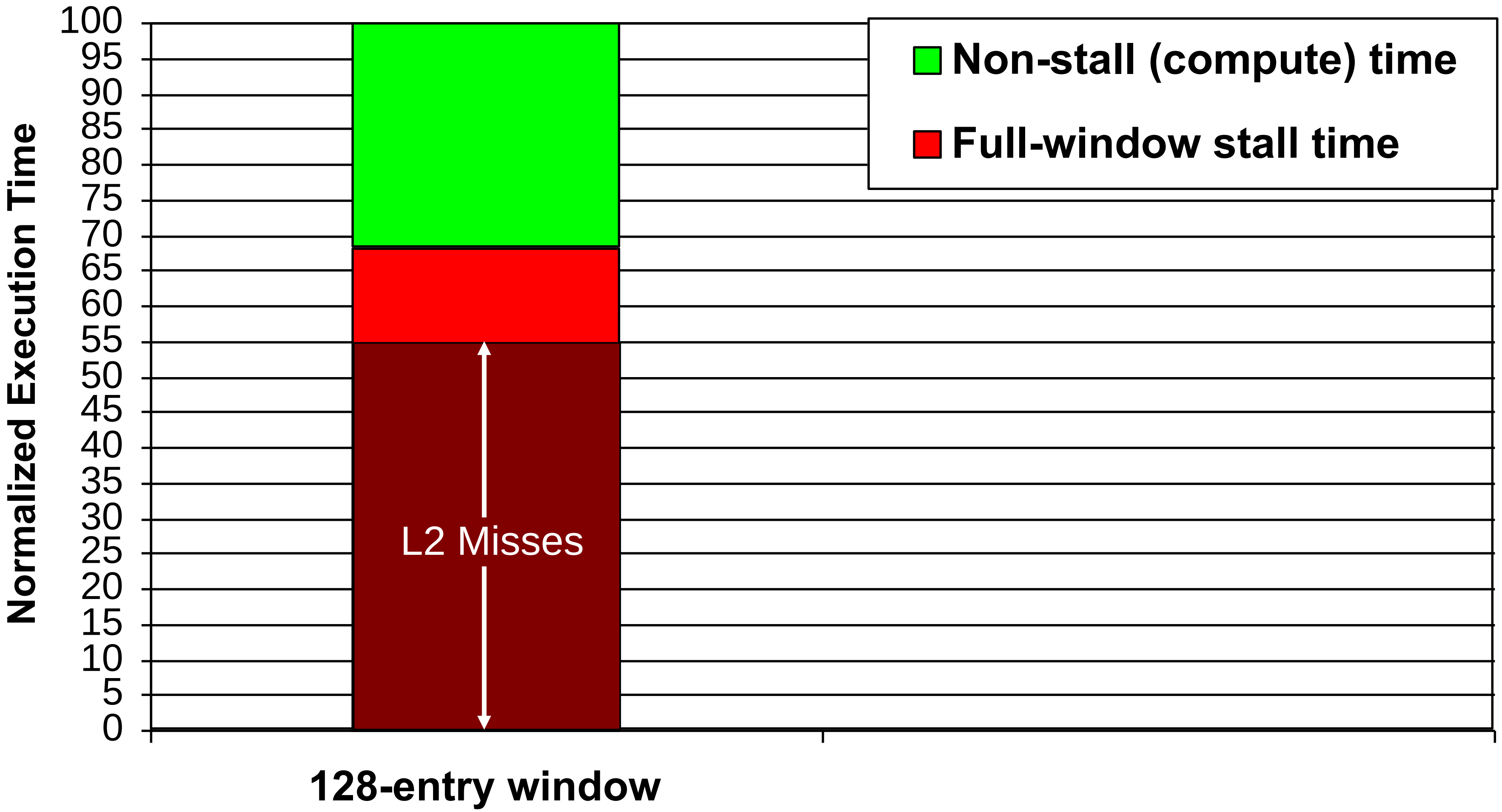
Independent of the L2 miss, executed out of program order, but cannot be retired.

Younger instructions cannot be executed because there is no space in the instruction window.

The processor stalls until the L2 Miss is serviced.

- Long-latency cache misses are responsible for most full-window stalls.

Cache Misses Responsible for Many Stalls



512KB L2 cache, 500-cycle DRAM latency, aggressive stream-based prefetcher
Data averaged over 147 memory-intensive benchmarks on a high-end x86 processor model

The Memory Latency Problem

- Problem: Memory latency is long

- And, it is not easy to reduce it…
  - We will look at methods for reducing DRAM latency in a later lecture
    - Lee et al. "Tiered-Latency DRAM," HPCA 2013.
    - Lee et al., "Adaptive-Latency DRAM," HPCA 2014.

- And, even if we reduce memory latency, it is still long
  - Remember the fundamental capacity-latency tradeoff
  - Contention for memory increases latencies

How Do We Tolerate Stalls Due to Memory?

- Two major approaches
  - Reduce/eliminate stalls
  - Tolerate the effect of a stall when it happens

- Four fundamental techniques to achieve these
  - Caching
  - Prefetching
  - Multithreading
  - Out-of-order execution

- Many techniques have been developed to make these four fundamental techniques more effective in tolerating memory latency

Memory Latency Tolerance Techniques

- Caching [initially by Wilkes, 1965]
    - Widely used, simple, effective, but inefficient, passive
    - Not all applications/phases exhibit temporal or spatial locality

- Prefetching [initially in IBM 360/91, 1967]
    - Works well for regular memory access patterns
    - Prefetching irregular access patterns is difficult, inaccurate, and hardware-intensive

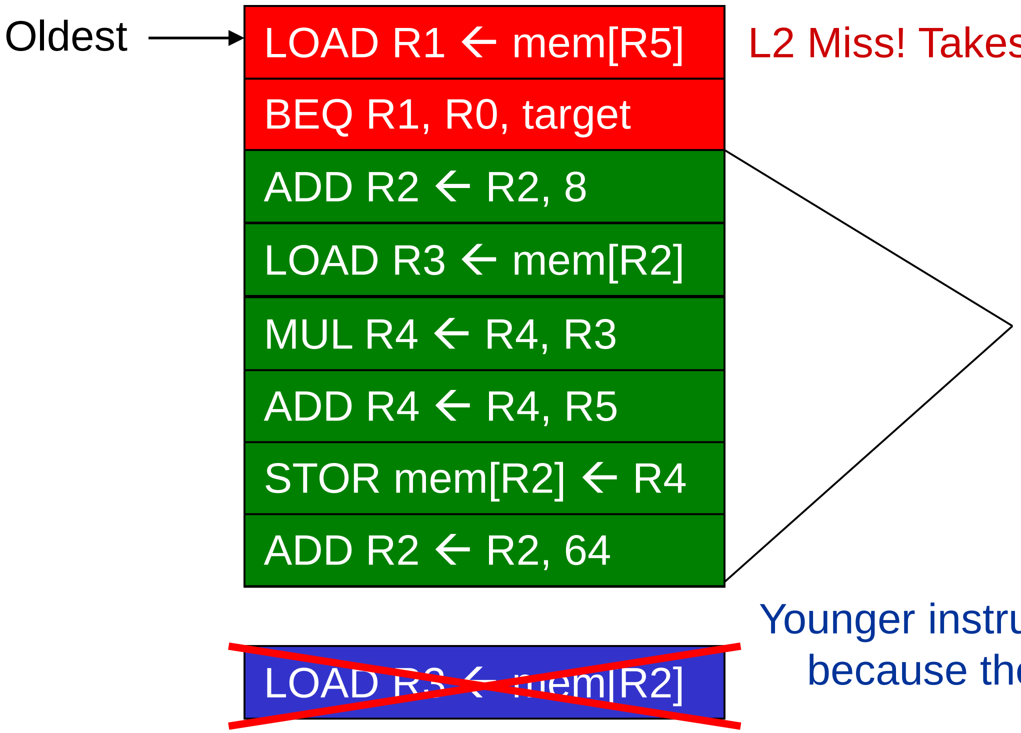
- Multithreading [initially in CDC 6600, 1964]
    - Works well if there are multiple threads
    - Improving single thread performance using multithreading hardware is an ongoing research effort

- Out-of-order execution [initially by Tomasulo, 1967]
    - Tolerates irregular cache misses that cannot be prefetched
    - Requires extensive hardware resources for tolerating long latencies
    - Runahead execution alleviates this problem (as we will see today)

Runahead Execution

Small Windows: Full-window Stalls

8-entry instruction window:



L2 Miss! Takes 100s of cycles.

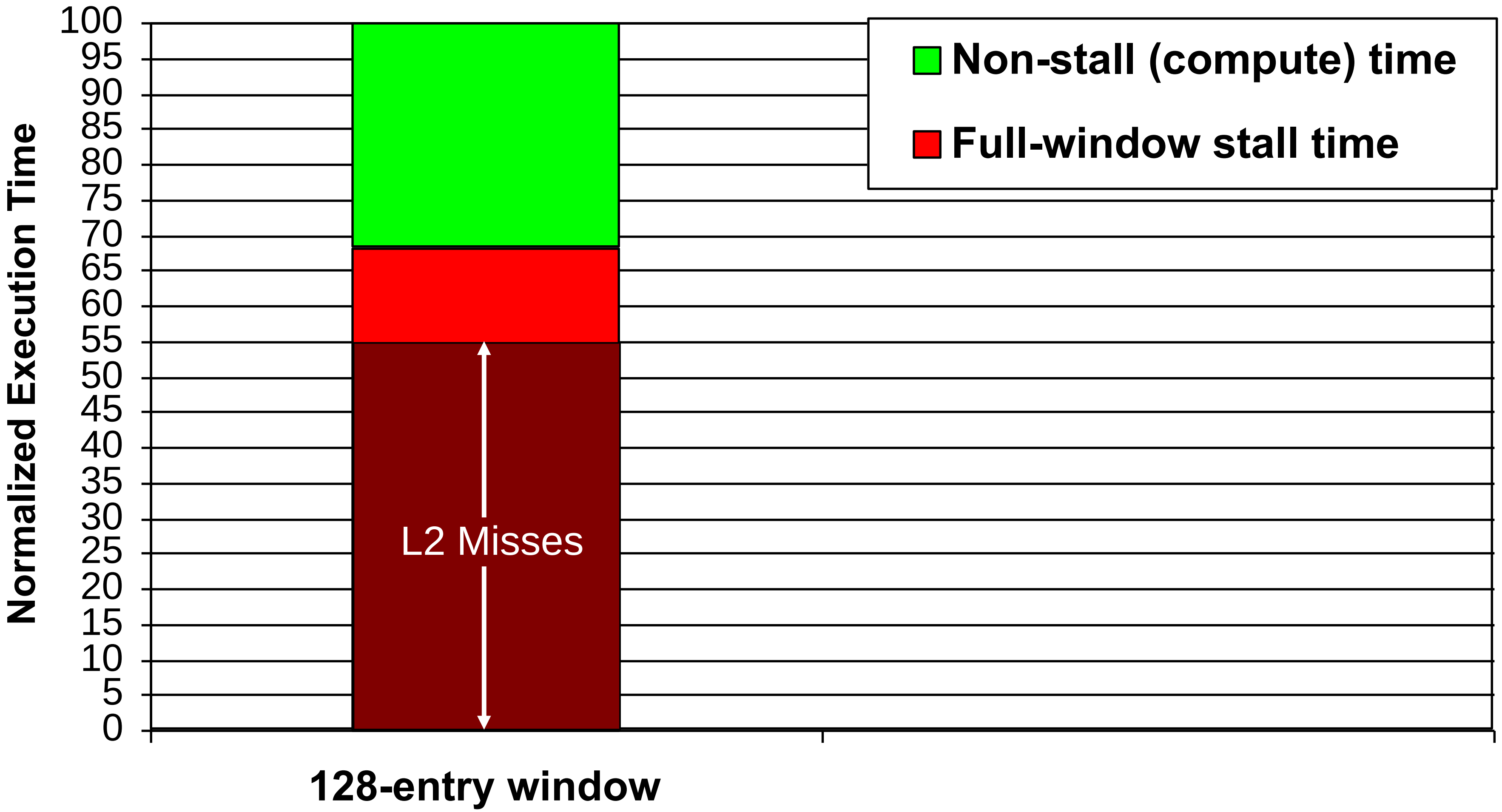
Independent of the L2 miss, executed out of program order, but cannot be retired.

Younger instructions cannot be executed because there is no space in the instruction window.

The processor stalls until the L2 Miss is serviced.

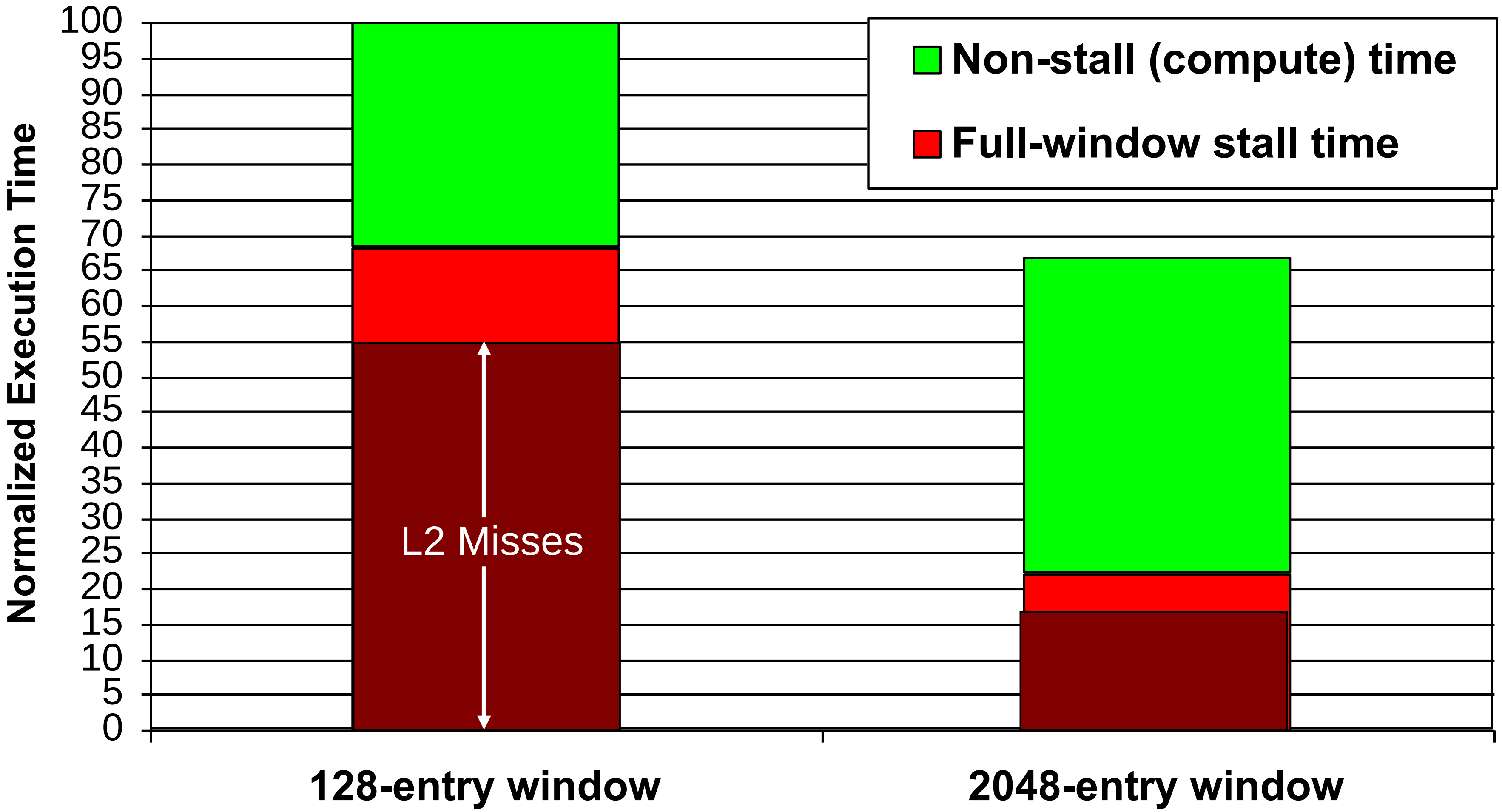
- Long-latency cache misses are responsible for most full-window stalls.

Impact of Long-Latency Cache Misses



512KB L2 cache, 500-cycle DRAM latency, aggressive stream-based prefetcher
Data averaged over 147 memory-intensive benchmarks on a high-end x86 processor model

Impact of Long-Latency Cache Misses



500-cycle DRAM latency, aggressive stream-based prefetcher

Data averaged over 147 memory-intensive benchmarks on a high-end x86 processor model

The Problem

- Out-of-order execution requires large instruction windows to tolerate today’s main memory latencies.

- As main memory latency increases, instruction window size should also increase to fully tolerate the memory latency.

- Building a large instruction window is a challenging task if we would like to achieve
  - Low power/energy consumption (tag matching logic, ld/st buffers)
  - Short cycle time (access, wakeup/select latencies)
  - Low design and verification complexity

Efficient Scaling of Instruction Window Size

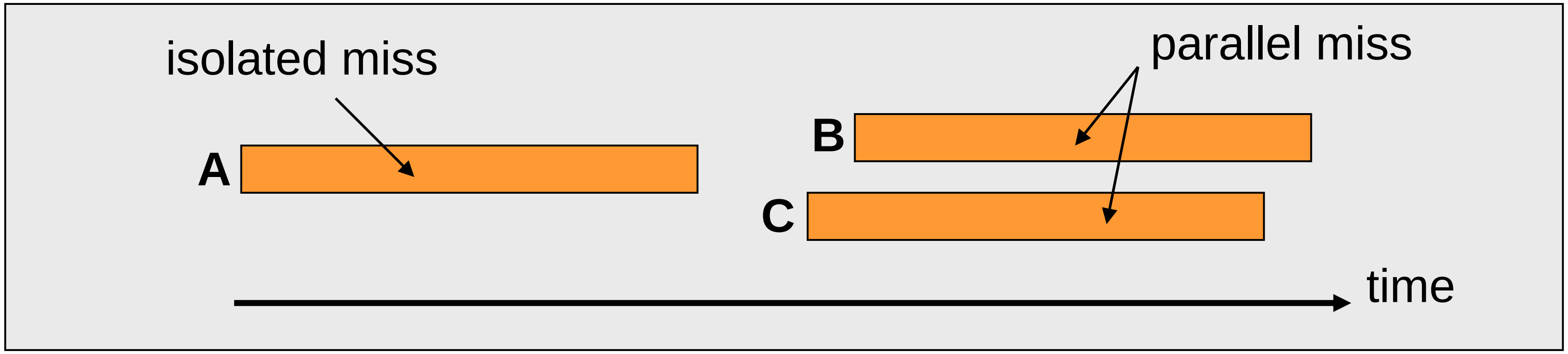
- One of the major research issues in out of order execution

- How to achieve the benefits of a large window with a small one (or in a simpler way)?

- How do we efficiently tolerate memory latency with the machinery of out-of-order execution (and a small instruction window)?

Memory Level Parallelism (MLP)

- Idea: Find and service multiple cache misses in parallel so that the processor stalls only once for all misses



  - Enables latency tolerance: overlaps latency of different misses

- How to generate multiple misses?
  - Out-of-order execution, multithreading, prefetching, runahead

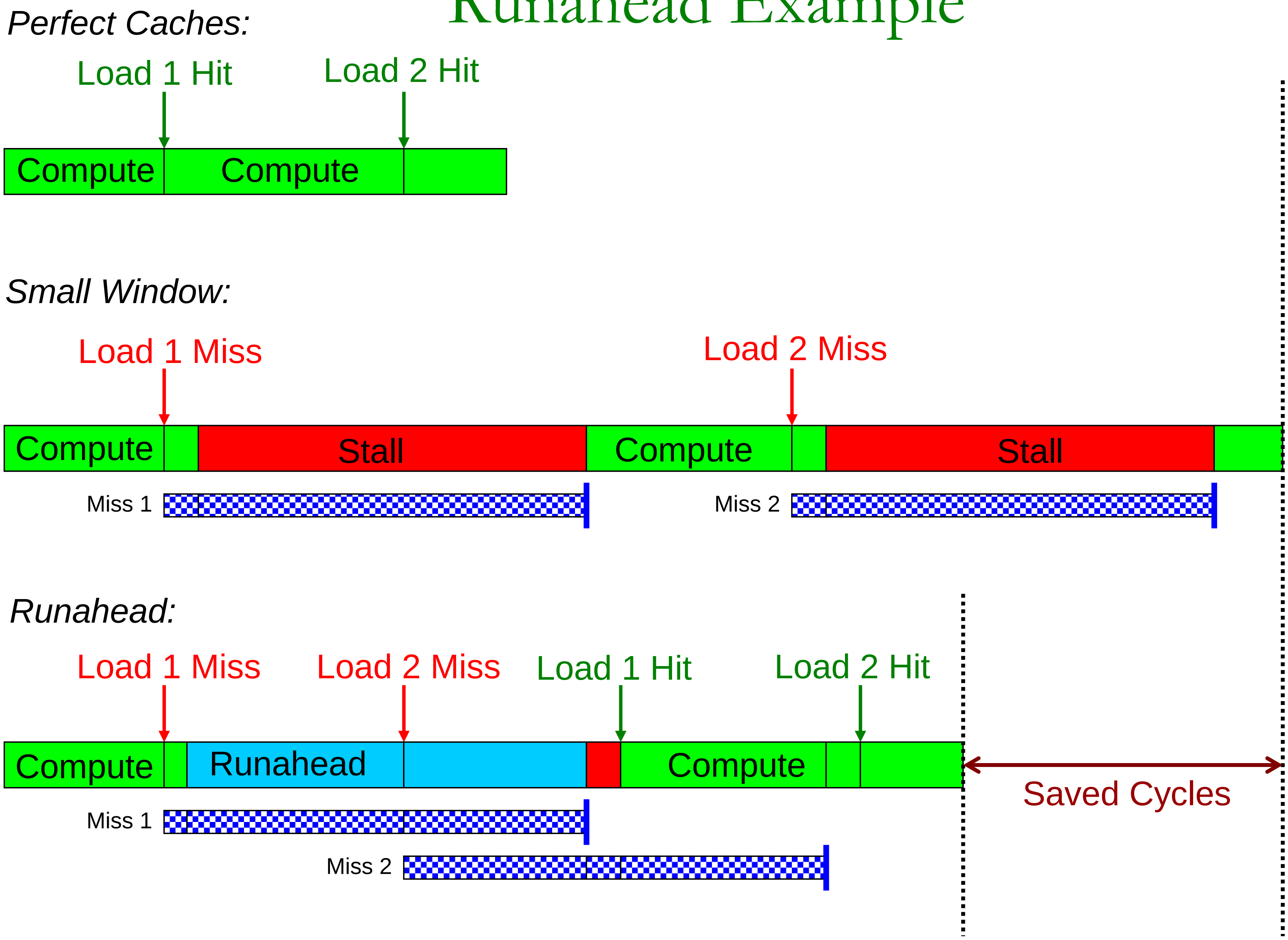
Runahead Execution (I)

- A technique to obtain the memory-level parallelism benefits of a large instruction window

- When the oldest instruction is a long-latency cache miss:
  - Checkpoint architectural state and enter runahead mode
- In runahead mode:
  - Speculatively pre-execute instructions
  - The purpose of pre-execution is to generate prefetches
  - L2-miss dependent instructions are marked INV and dropped
- Runahead mode ends when the original miss returns
  - Checkpoint is restored and normal execution resumes

- Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," HPCA 2003.

Runahead Example



Benefits of Runahead Execution

Instead of stalling during an L2 cache miss:

- Pre-executed loads and stores independent of L2-miss instructions generate very accurate data prefetches:
  - For both regular and irregular access patterns
- Instructions on the predicted program path are prefetched into the instruction/trace cache and L2.
- Hardware prefetcher and branch predictor tables are trained using future access information.

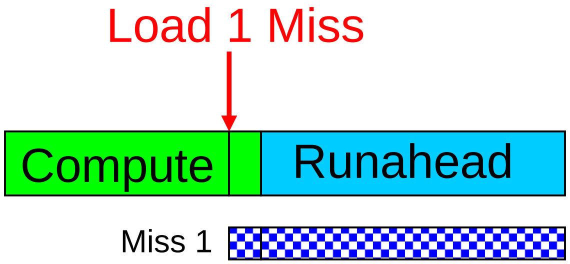
Runahead Execution Mechanism

- Entry into runahead mode
  - Checkpoint architectural register state

- Instruction processing in runahead mode

- Exit from runahead mode
  - Restore architectural register state from checkpoint

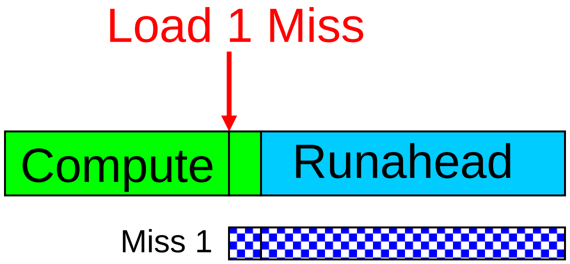
Instruction Processing in Runahead Mode



Runahead mode processing is the same as normal instruction processing, EXCEPT:

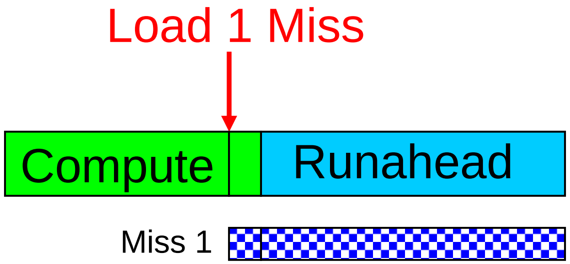
- It is purely speculative: Architectural (software-visible) register/memory state is NOT updated in runahead mode.
- L2-miss dependent instructions are identified and treated specially.
    - They are quickly removed from the instruction window.
    - Their results are not trusted.

L2-Miss Dependent Instructions



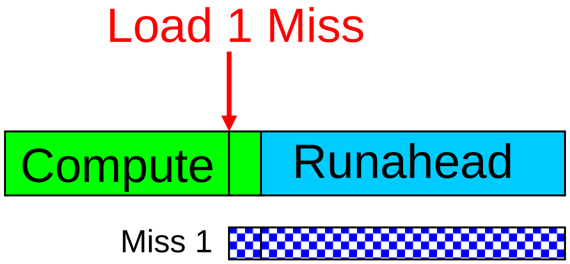
- Two types of results produced: INV and VALID
- INV = Dependent on an L2 miss
- INV results are marked using INV bits in the register file and store buffer.
- INV values are not used for prefetching/branch resolution.

Removal of Instructions from Window



- Oldest instruction is examined for pseudo-retirement
  - An INV instruction is removed from window immediately.
  - A VALID instruction is removed when it completes execution.
- Pseudo-retired instructions free their allocated resources.
  - This allows the processing of later instructions.
- Pseudo-retired stores communicate their data to dependent loads.

Store/Load Handling in Runahead Mode



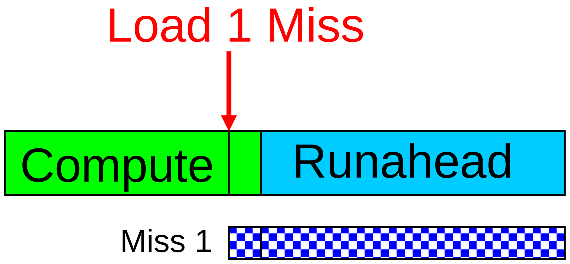
- A pseudo-retired store writes its data and INV status to a dedicated memory, called runahead cache.

- Purpose: Data communication through memory in runahead mode.

- A dependent load reads its data from the runahead cache.

- Does not need to be always correct → Size of runahead cache is very small.

Branch Handling in Runahead Mode

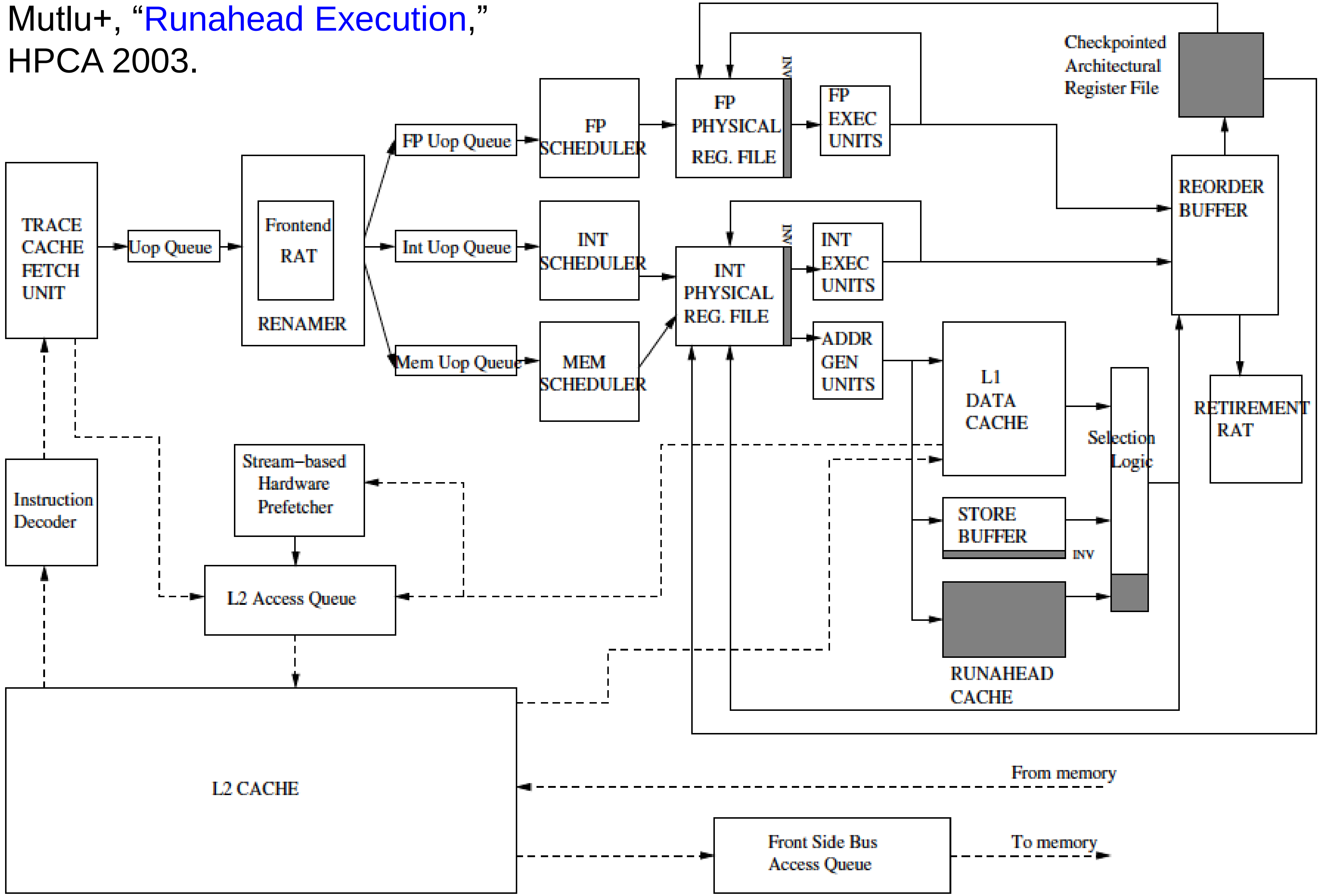


- INV branches cannot be resolved.
  - A mispredicted INV branch causes the processor to stay on the wrong program path until the end of runahead execution.

- VALID branches are resolved and initiate recovery if mispredicted.

A Runahead Processor Diagram

Mutlu+, “Runahead Execution,” HPCA 2003.



Runahead Execution Pros and Cons

- Advantages:

  + Very accurate prefetches for data/instructions (all cache levels)

    + Follows the program path

  + Simple to implement, most of the hardware is already built in

  + Versus other pre-execution based prefetching mechanisms (as we will see):

    + Uses the same thread context as main thread, no waste of context

    + No need to construct a pre-execution thread

- Disadvantages/Limitations:

  -- Extra executed instructions

  -- Limited by branch prediction accuracy

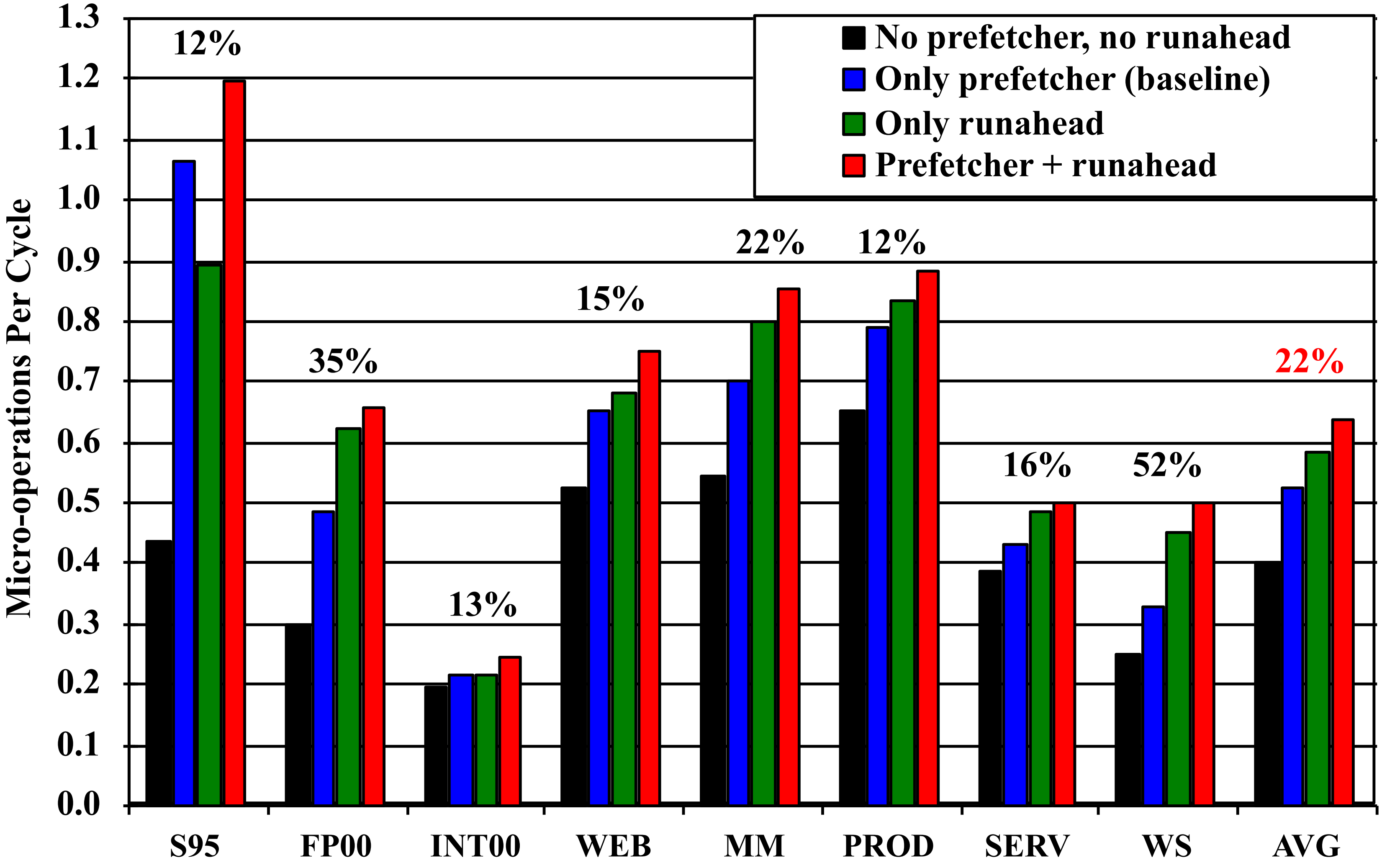
  -- Cannot prefetch dependent cache misses

  -- Effectiveness limited by available "memory-level parallelism" (MLP)

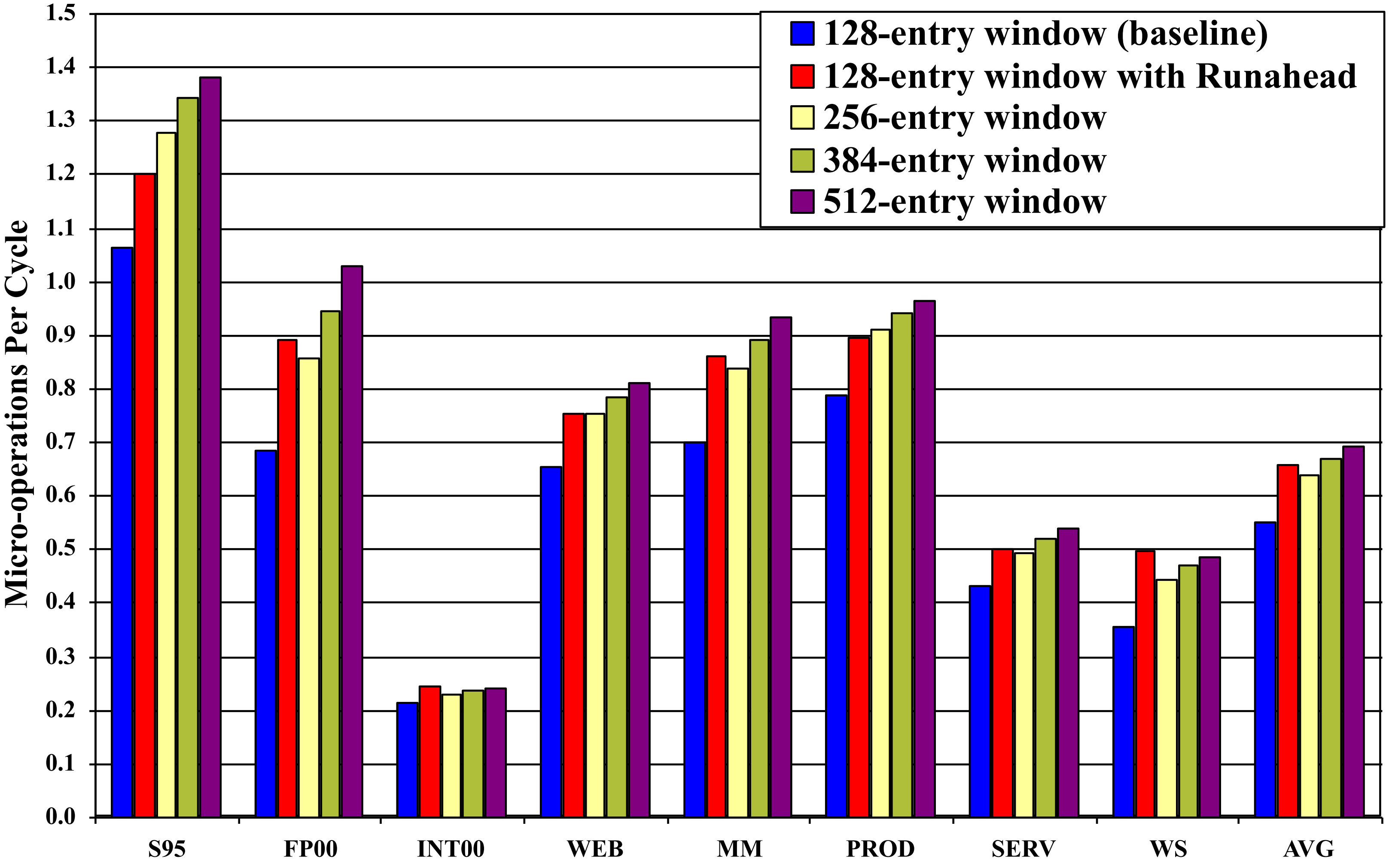
  -- Prefetch distance limited by memory latency

- Implemented in IBM POWER6, Sun "Rock"

Performance of Runahead Execution



Runahead Execution vs. Large Windows

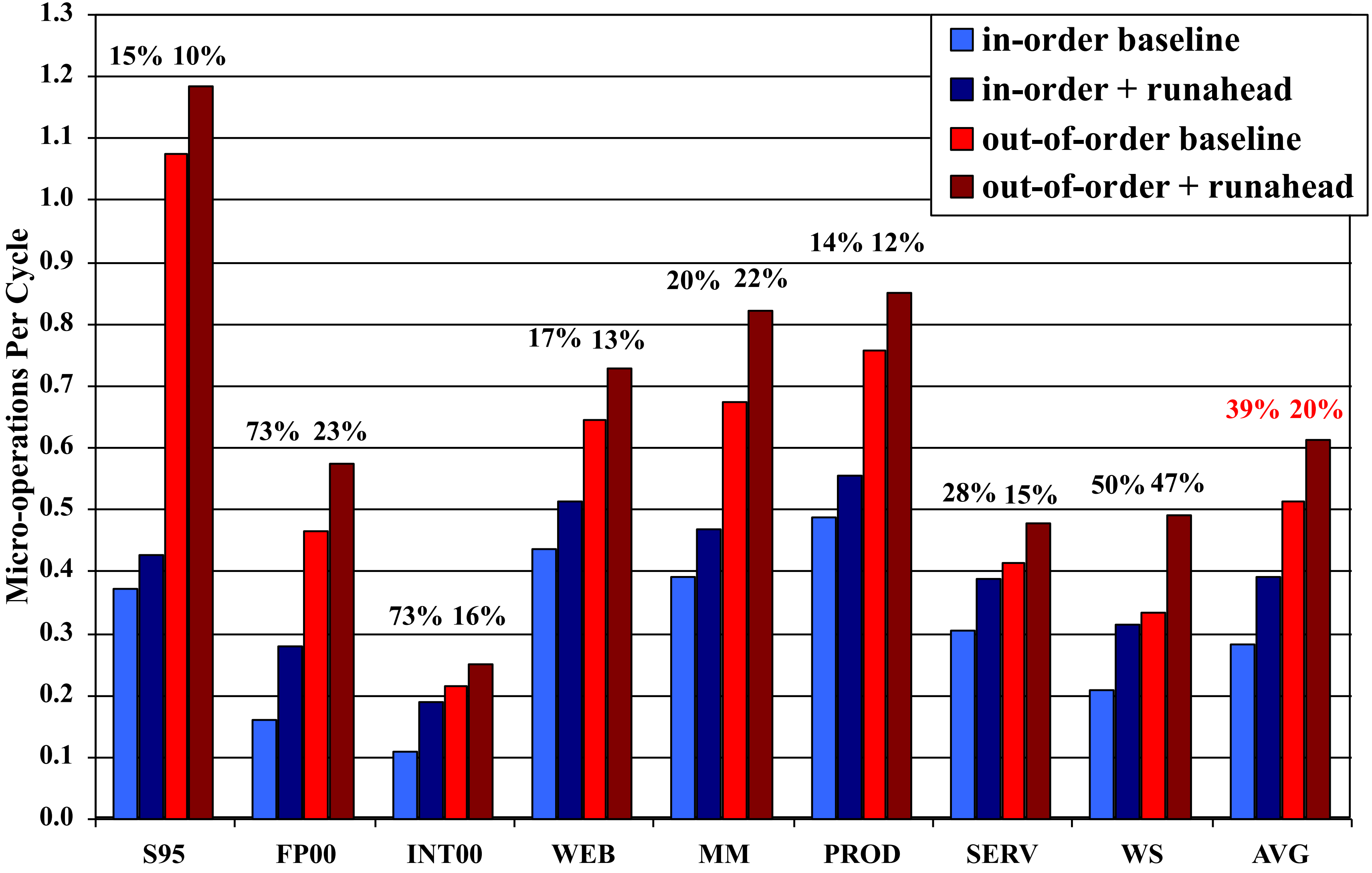


Runahead vs. A (Real) Large Window

- When is one beneficial, when is the other?
- Pros and cons of each

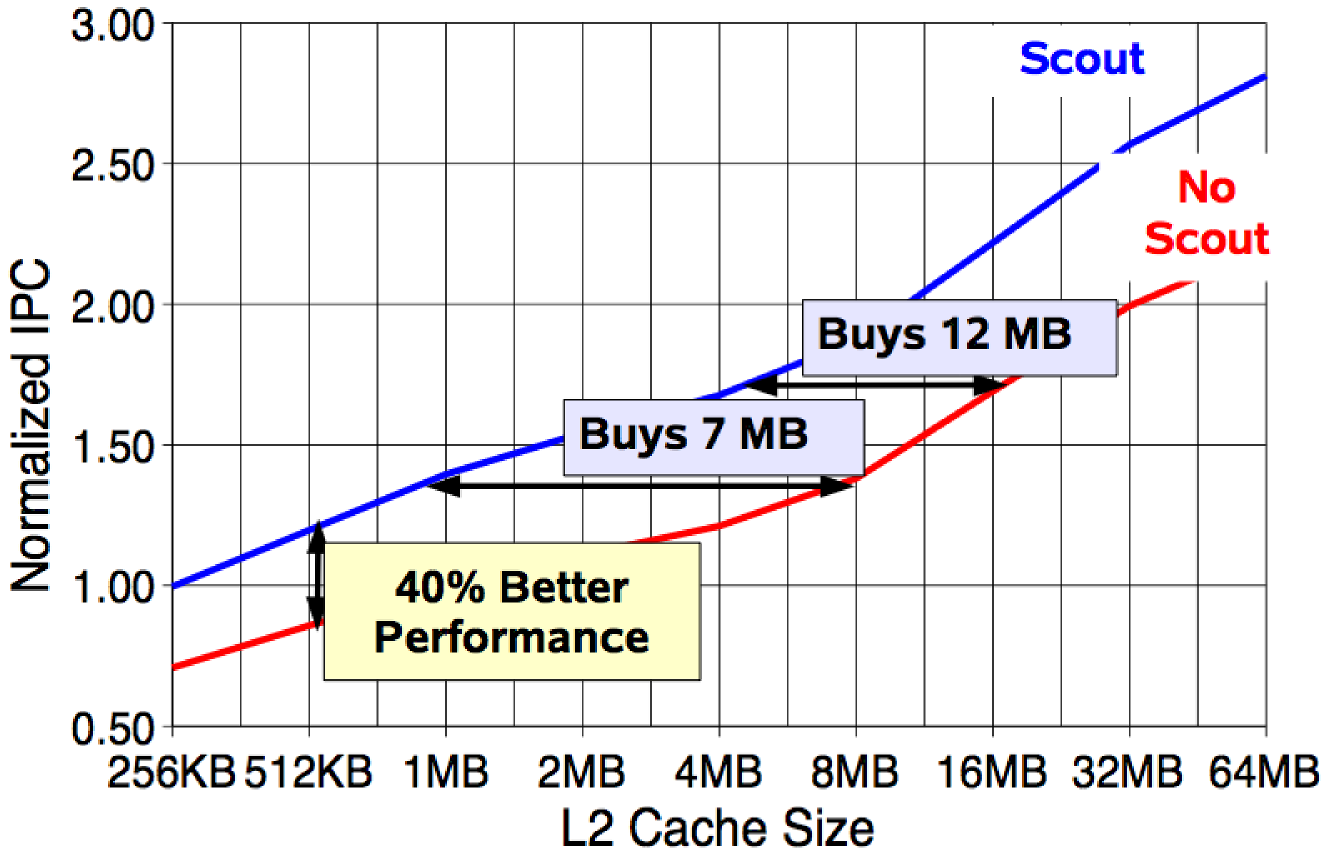
- Which can tolerate FP operation latencies better?
- Which leads to less wasted execution?

Runahead on In-order vs. Out-of-order



Effect of Runahead in Sun ROCK

- Shailender Chaudhry talk, Aug 2008.


Normalized IPC
256KB
512KB
1MB
2MB
4MB
8MB
16MB
32MB
64MB
L2 Cache Size
Scout
No Scout
Buys 12 MB
Buys 7 MB
40% Better Performance