15-740/18-740
Computer Architecture
Lecture 10: Runahead and MLP

Prof. Onur Mutlu
Carnegie Mellon University

Last Time …

- Issues in Out-of-order execution
  - Buffer decoupling
  - Register alias tables
  - Physical register files
  - Centralized vs. distributed reservation stations
  - Scheduling logic

Readings

- Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," HPCA 2003.
- Mutlu et al., "Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance," IEEE Micro Top Picks 2006.

- Zhou, Dual-Core Execution: "Building a Highly Scalable Single-Thread Instruction Window," PACT 2005.

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

Questions

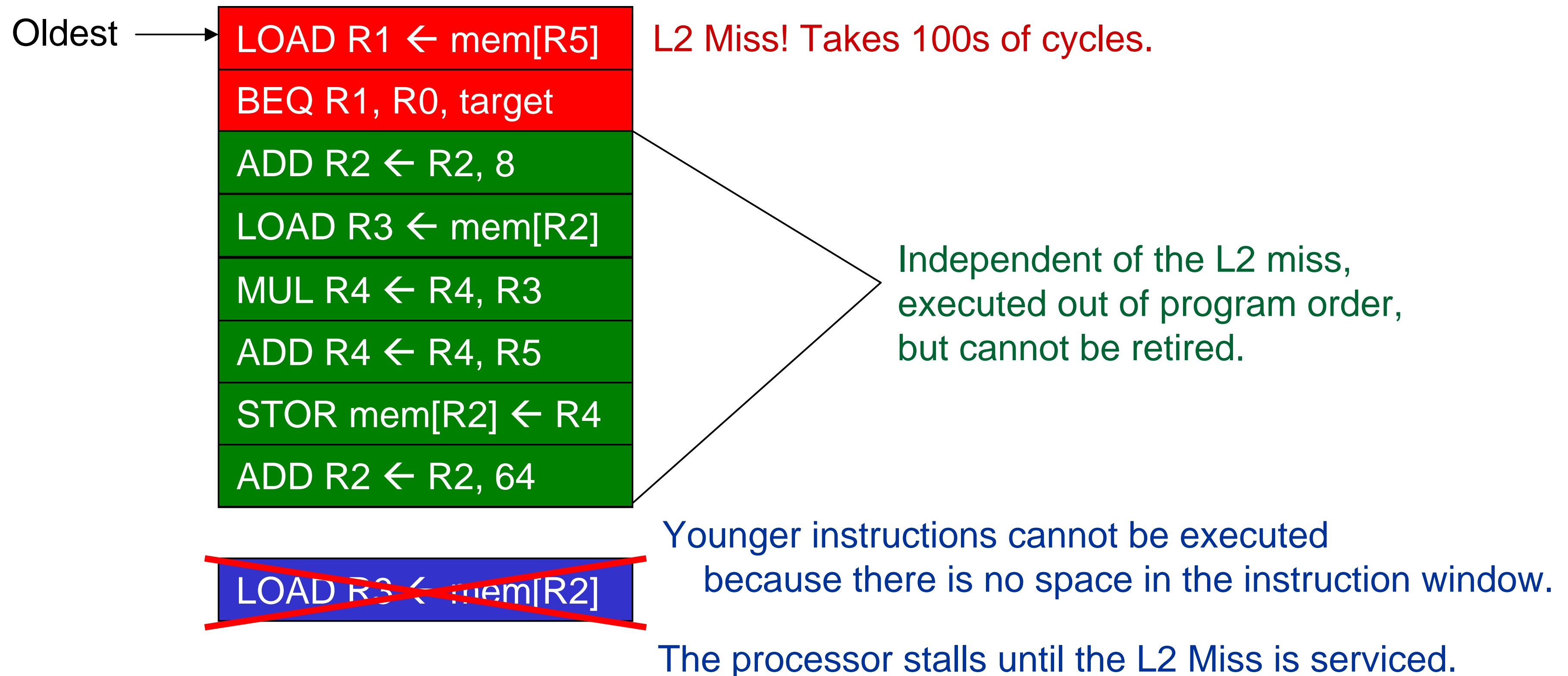
- Why is OoO execution beneficial?
  - What if all operations take single cycle?
  - Latency tolerance: OoO execution tolerates the latency of multi-cycle operations by executing independent operations concurrently

- 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?
  - What limits the latency tolerance scalability of Tomasulo's algorithm?
    - Active/instruction window size: determined by register file, scheduling window, reorder buffer, store buffer, load buffer

Small Windows: Full-window Stalls

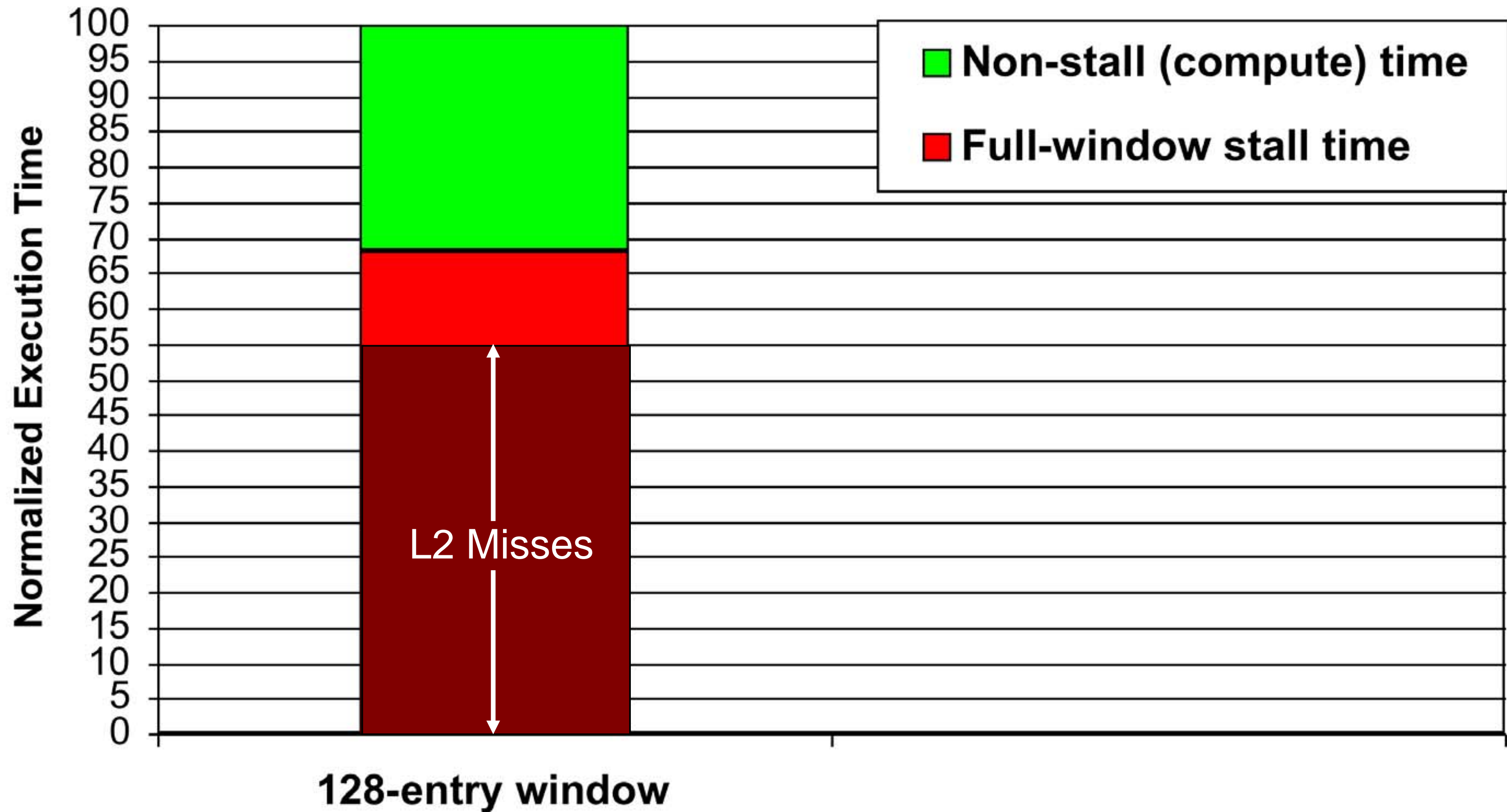
- When a long-latency instruction is not complete, it blocks retirement.
- Incoming instructions fill the instruction window.
- Once the window is full, processor cannot place new instructions into the window.
  - This is called a full-window stall.
- A full-window stall prevents the processor from making progress in the execution of the program.

Small Windows: Full-window Stalls



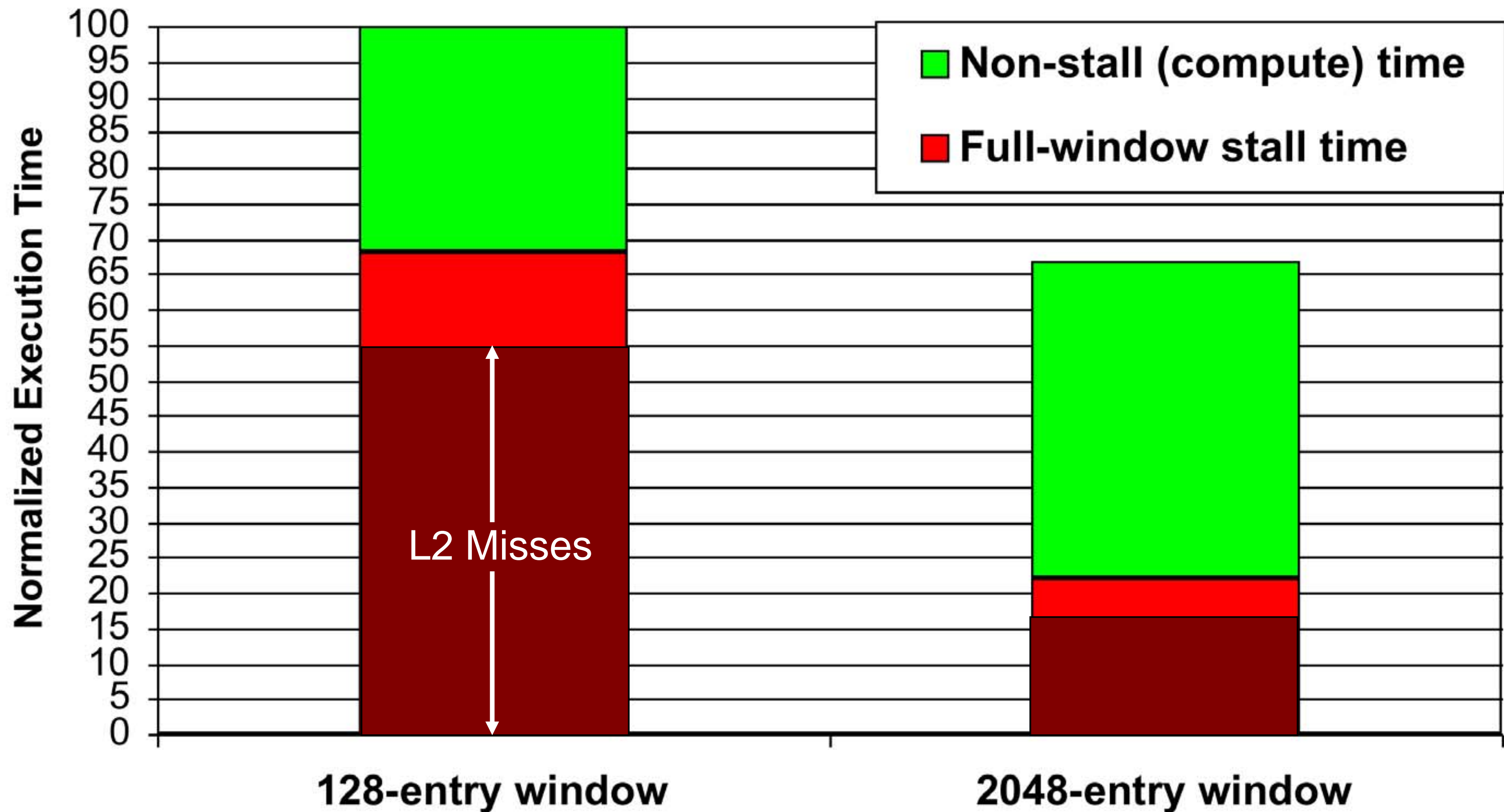
- L2 cache misses are responsible for most full-window stalls.

Impact of L2 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 L2 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)?

  - Runahead execution?
    - Upon L2 miss, checkpoint architectural state, speculatively execute only for prefetching, re-execute when data ready
  - Continual flow pipelines?
    - Upon L2 miss, deallocate everything belonging to an L2 miss dependent, reallocate/re-rename and re-execute upon data ready
  - Dual-core execution?
    - One core runs ahead and does not stall on L2 misses, feeds another core that commits instructions

Runahead Execution (I)

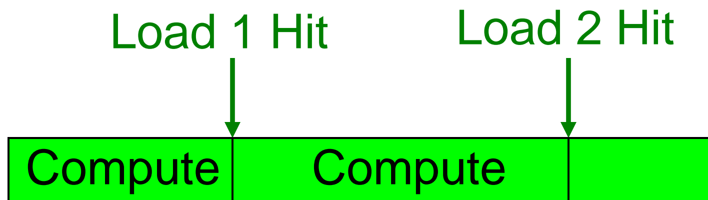
- A technique to obtain the memory-level parallelism benefits of a large instruction window

- 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

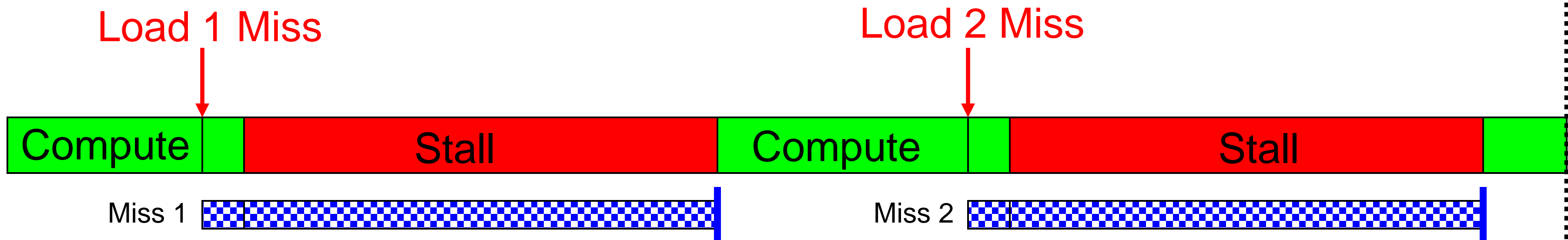
- Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," HPCA 2003.

Runahead Example

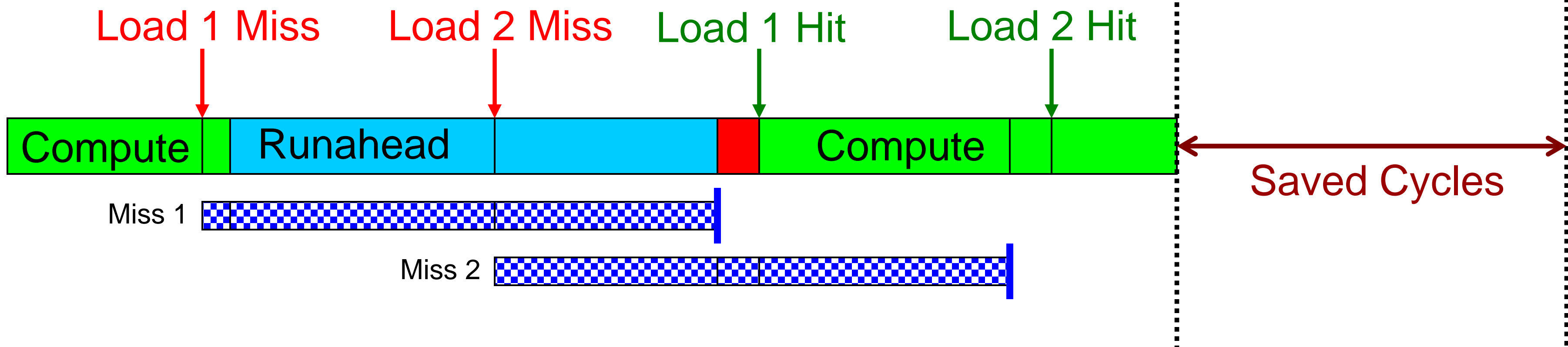
Perfect Caches:



Small Window:



Runahead:



Benefits of Runahead Execution

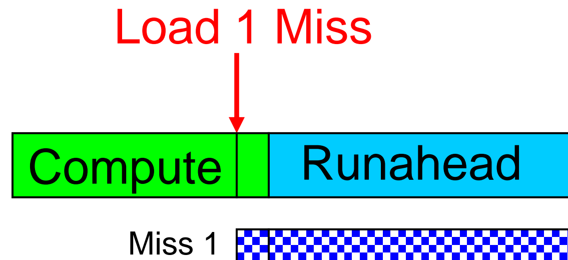
Instead of stalling during an L2 cache miss:

- 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

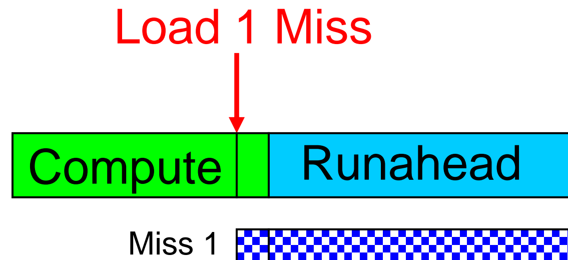
Instruction Processing in Runahead Mode



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

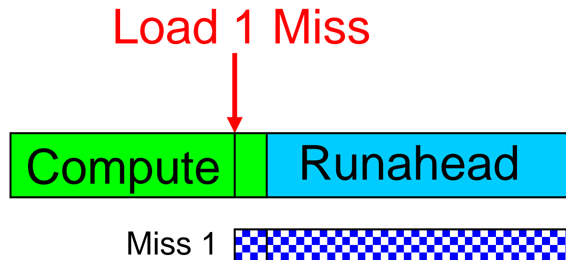
- It is purely speculative: Architectural (software-visible) register/memory state is NOT updated in runahead mode.
- L2-miss dependent instructions are identified and treated specially.
  - They are quickly removed from the instruction window.
  - Their results are not trusted.

L2-Miss Dependent Instructions



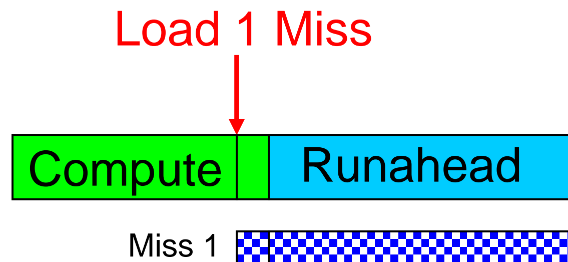
- Two types of results produced: INV and VALID
- INV = Dependent on an L2 miss
- INV results are marked using INV bits in the register file and store buffer.
- INV values are not used for prefetching/branch resolution.

Removal of Instructions from Window



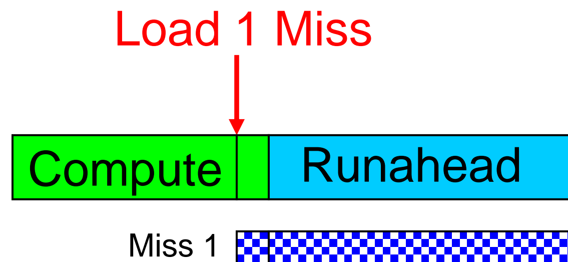
- Oldest instruction is examined for pseudo-retirement
  - An INV instruction is removed from window immediately.
  - A VALID instruction is removed when it completes execution.
- Pseudo-retired instructions free their allocated resources.
  - This allows the processing of later instructions.
- Pseudo-retired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

Runahead Execution (III)

- Advantages:
  - + Very accurate prefetches for data/instructions (all cache levels)
    - + Follows the program path
  - + No need to construct a pre-execution thread
  - + Uses the same thread context as main thread, no waste of context
  - + Simple to implement, most of the hardware is already built in

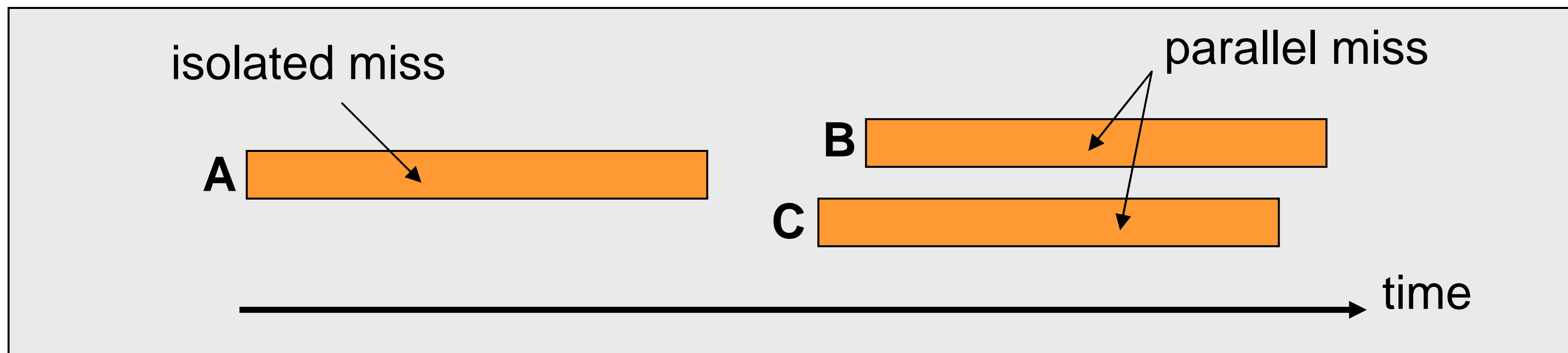
- Disadvantages/Limitations:
  - -- Extra executed instructions
  - -- Limited by branch prediction accuracy
  - -- Cannot prefetch dependent cache misses. Solution?
  - -- Effectiveness limited by available "memory-level parallelism" (MLP)
  - -- Prefetch distance limited by memory latency

- Implemented in IBM POWER6, Sun "Rock"

Memory Level Parallelism (MLP)

- Idea: Find and service multiple cache misses in parallel

- Why generate multiple misses?



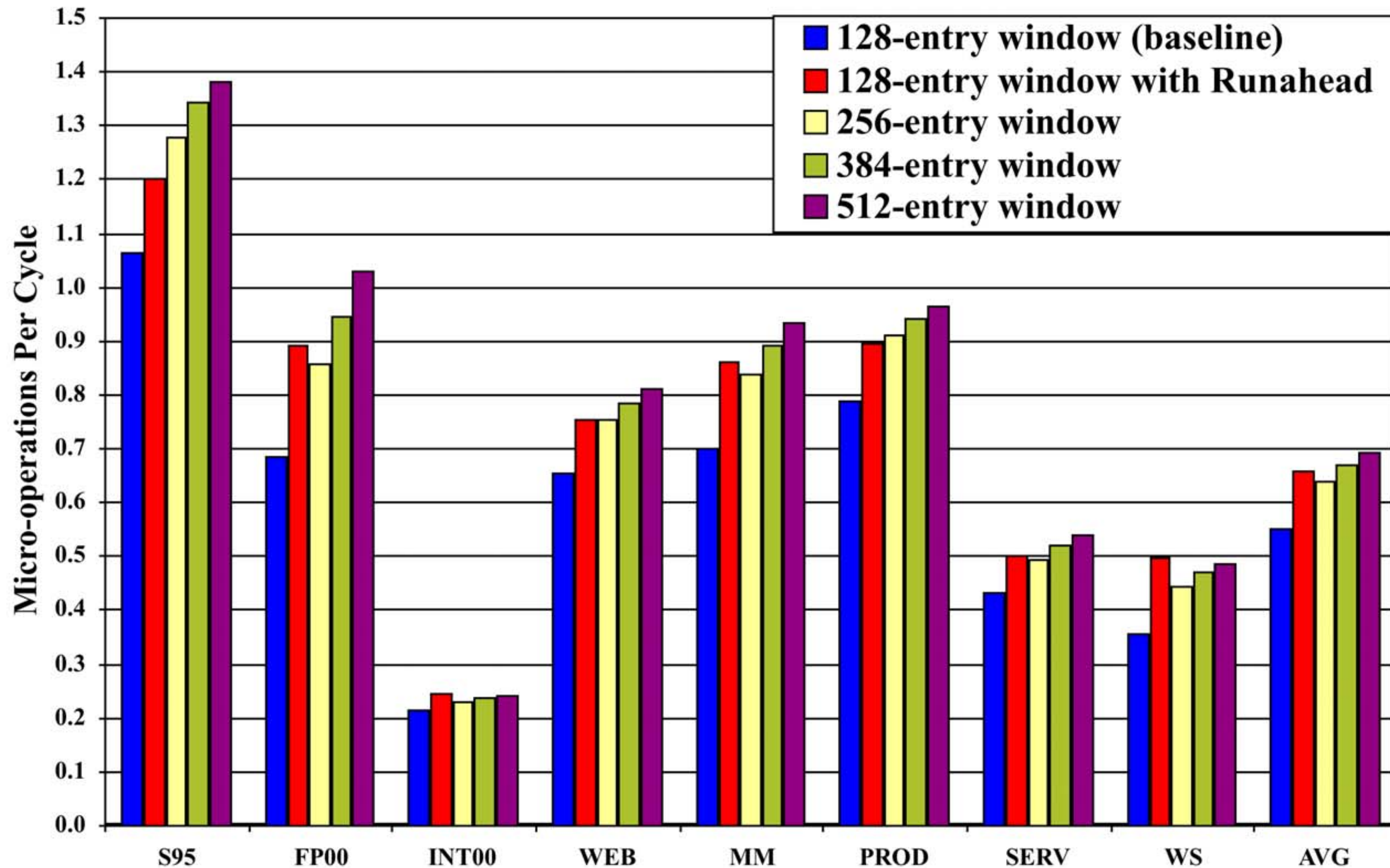
  - Enables latency tolerance: overlaps latency of different misses

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

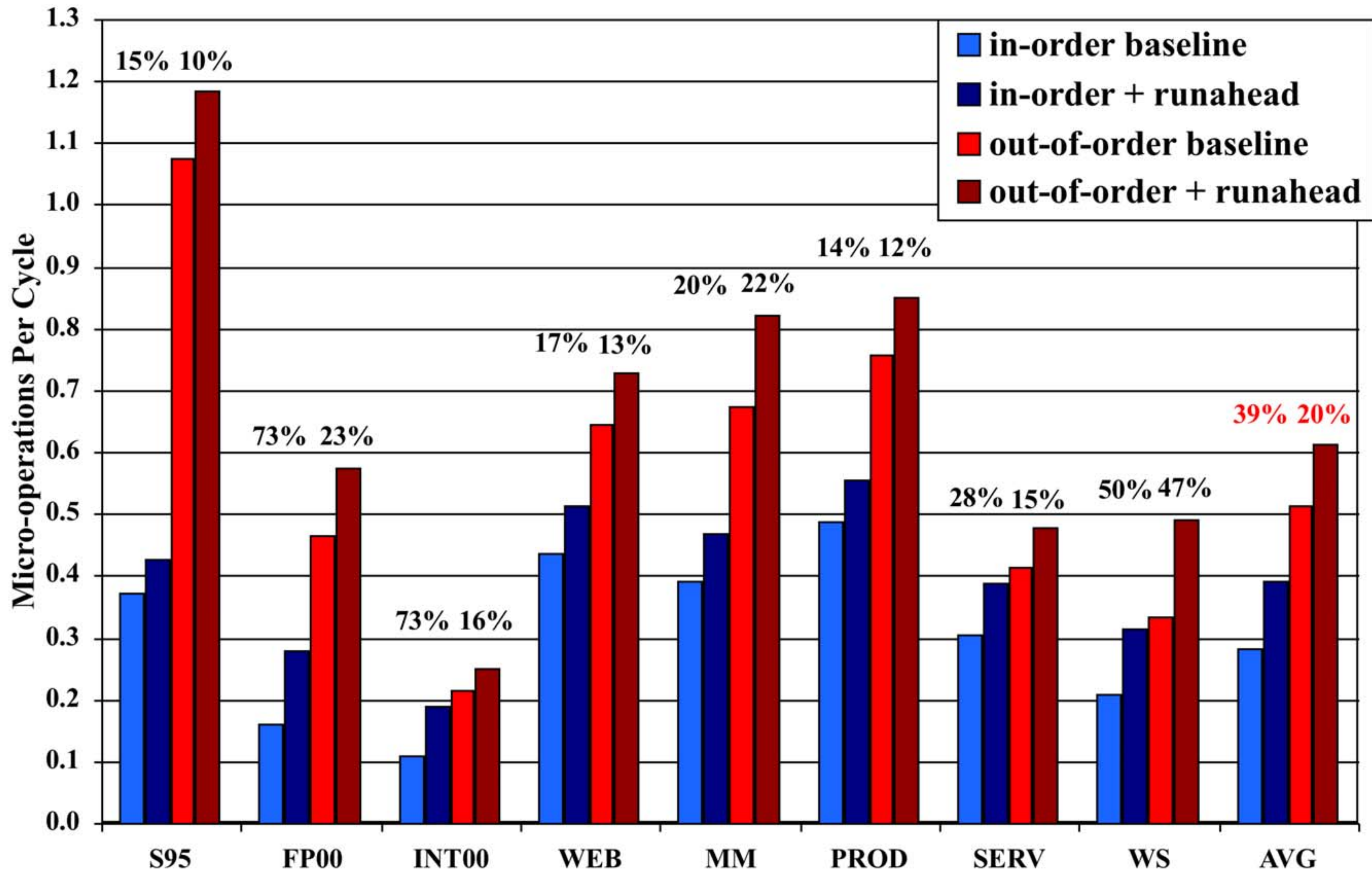
Memory Latency Tolerance Techniques

- 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 cache misses that cannot be prefetched
  - Requires extensive hardware resources for tolerating long latencies

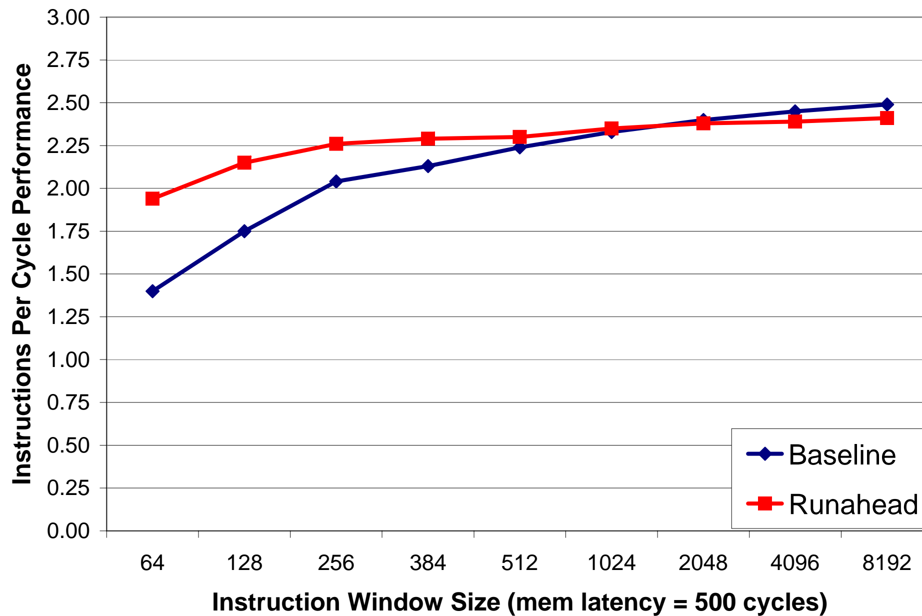
Runahead Execution vs. Large Windows

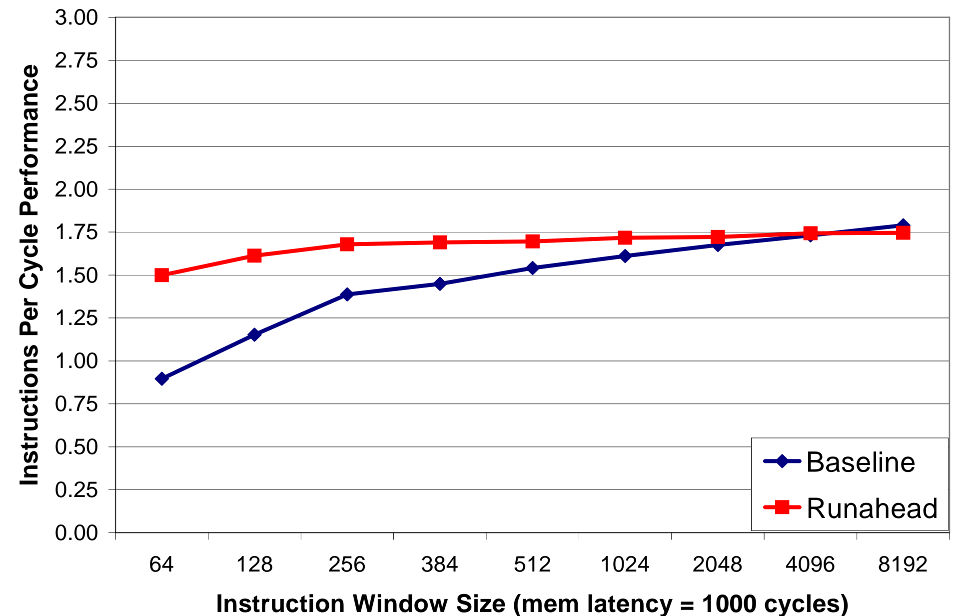


In-order vs. Out-of-order

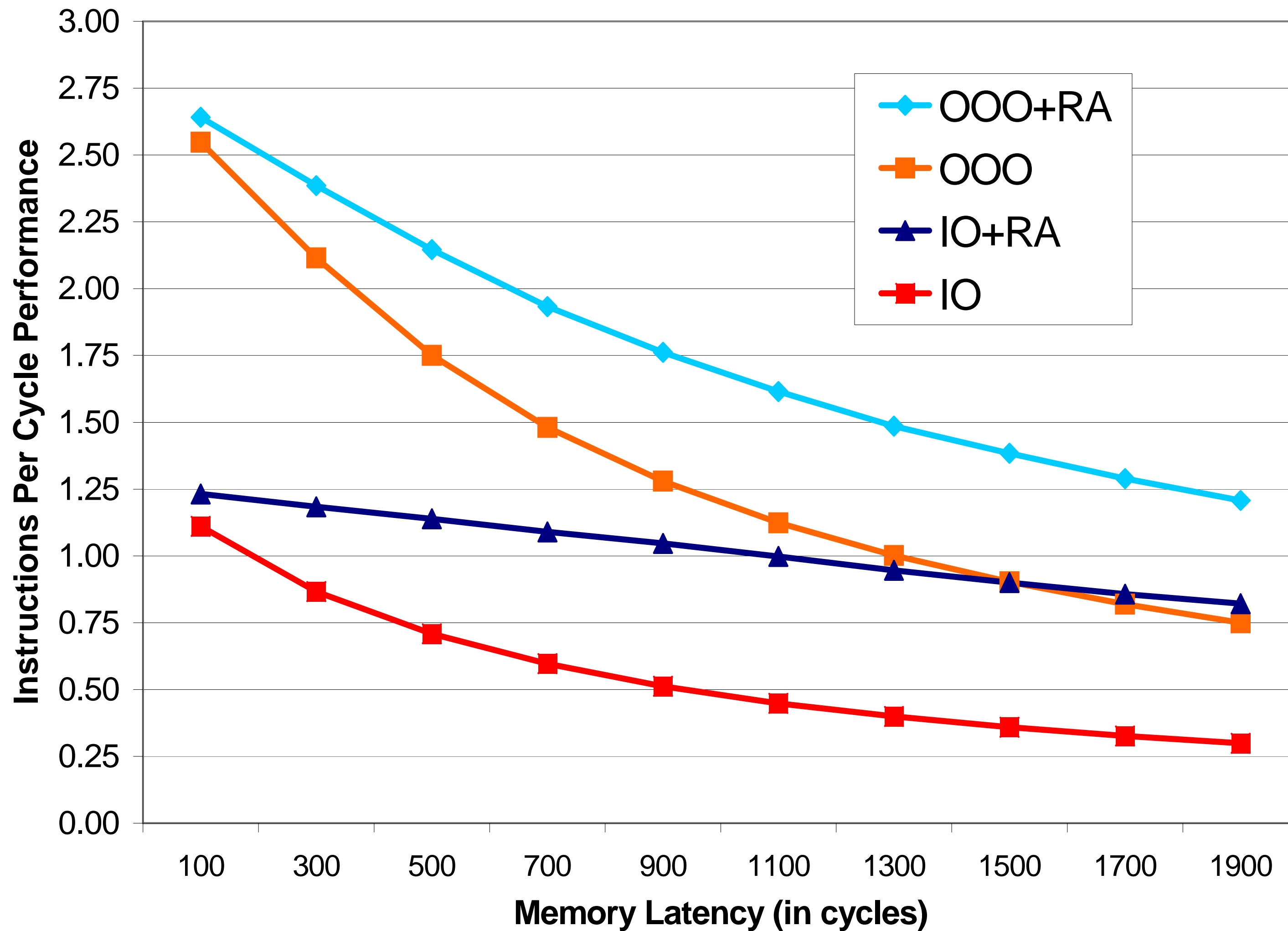


Runahead vs. Large Windows (Alpha)





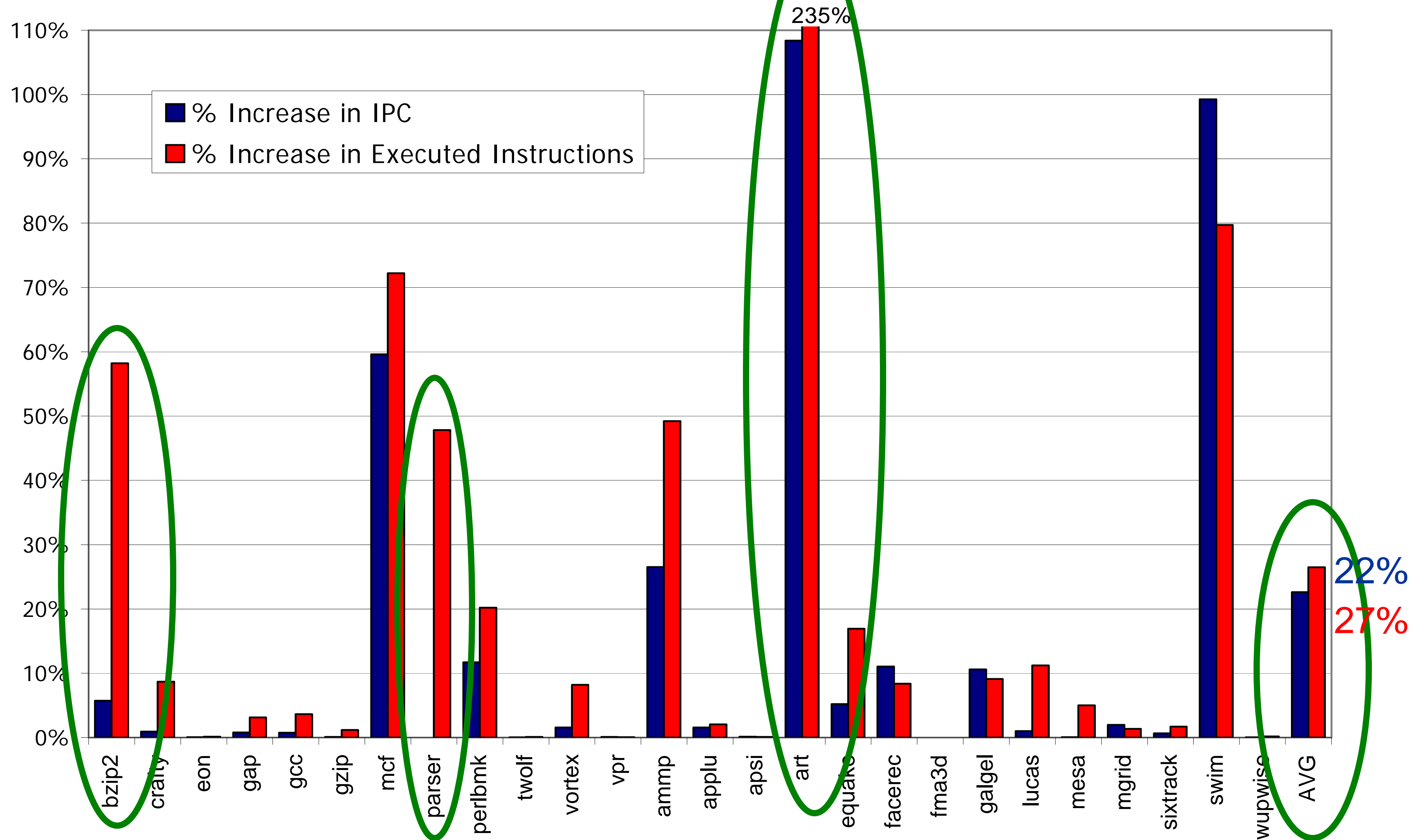
In-order vs. Out-of-order Execution (Alpha)



Limitations of the Baseline Runahead Mechanism

- Energy Inefficiency
  - A large number of instructions are speculatively executed
  - Efficient Runahead Execution [ISCA'05, IEEE Micro Top Picks'06]
- Ineffectiveness for pointer-intensive applications
  - Runahead cannot parallelize dependent L2 cache misses
  - Address-Value Delta (AVD) Prediction [MICRO'05]
- Irresolvable branch mispredictions in runahead mode
  - Cannot recover from a mispredicted L2-miss dependent branch
  - Wrong Path Events [MICRO'04]

The Efficiency Problem

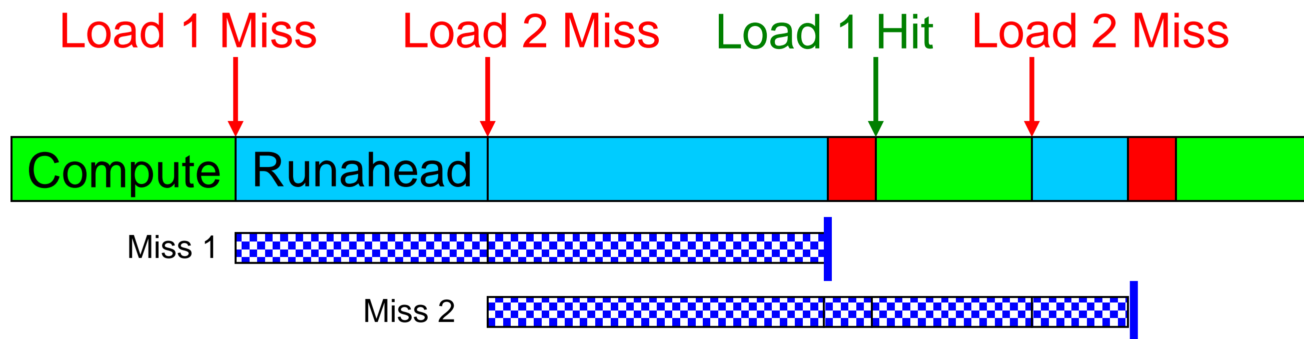


Causes of Inefficiency

- Short runahead periods
- Overlapping runahead periods
- Useless runahead periods
- Mutlu et al., "Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance," IEEE Micro Top Picks 2006.

Short Runahead Periods

- Processor can initiate runahead mode due to an already in-flight L2 miss generated by
  - the prefetcher, wrong-path, or a previous runahead period



- Short periods
  - are less likely to generate useful L2 misses
  - have high overhead due to the flush penalty at runahead exit

Overlapping Runahead Periods

- Two runahead periods that execute the same instructions

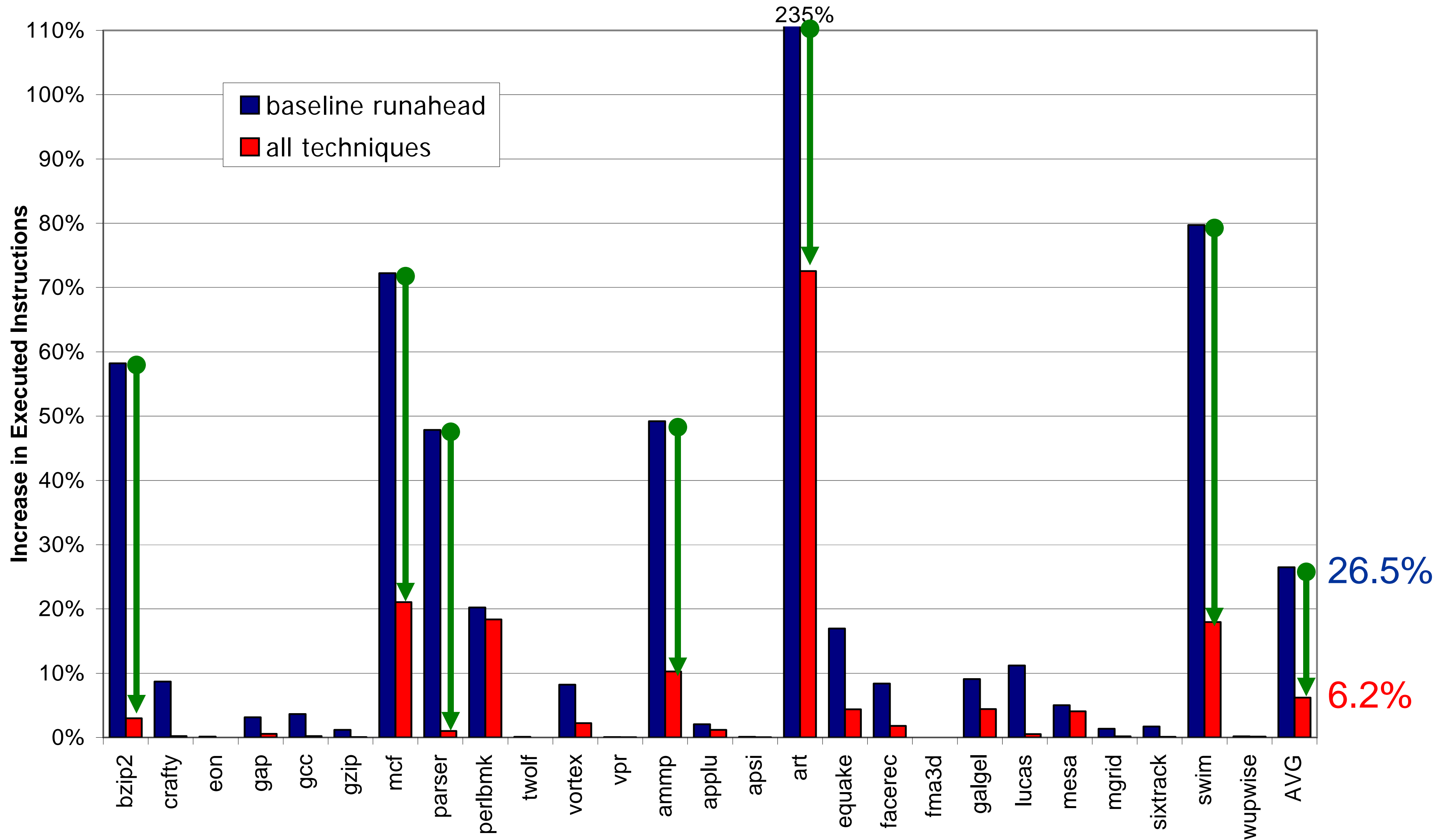
- Second period is inefficient

Useless Runahead Periods

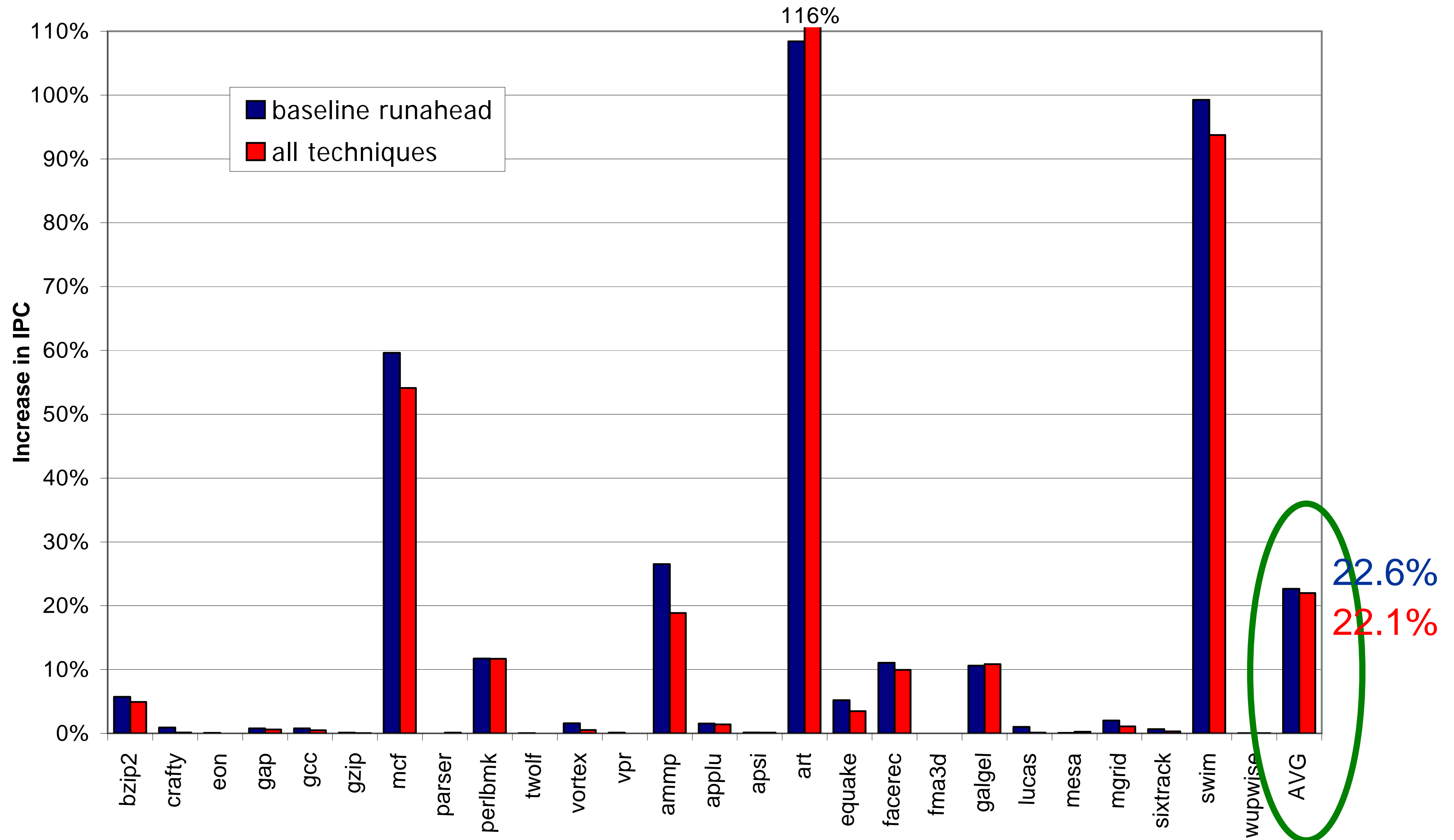
- Periods that do not result in prefetches for normal mode

- They exist due to the lack of memory-level parallelism
- Mechanism to eliminate useless periods:
  - Predict if a period will generate useful L2 misses
  - Estimate a period to be useful if it generated an L2 miss that cannot be captured by the instruction window
    - Useless period predictors are trained based on this estimation

Overall Impact on Executed Instructions



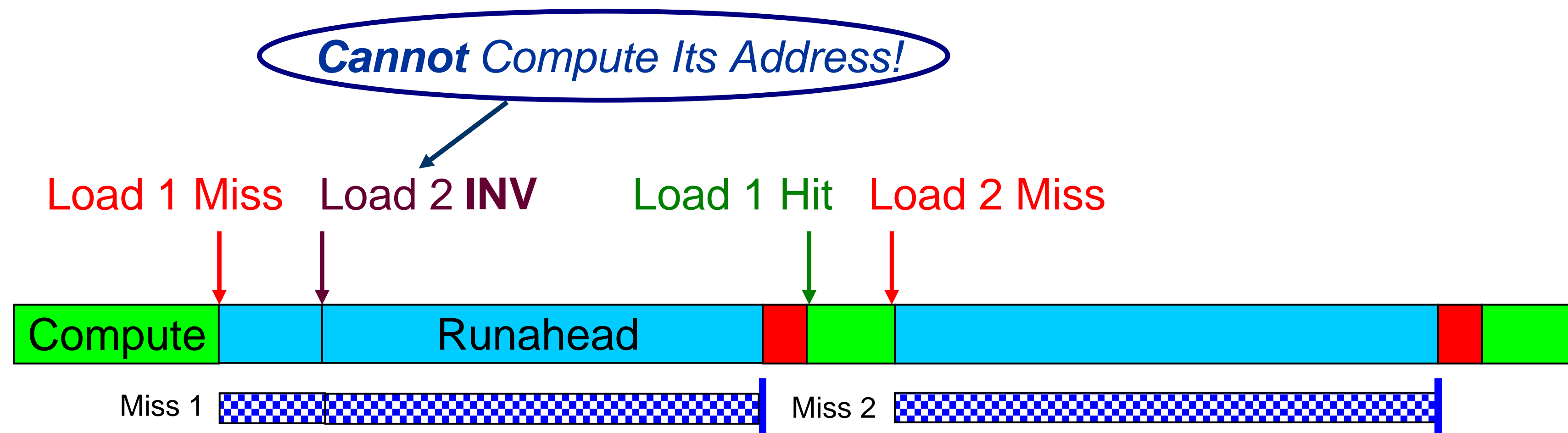
Overall Impact on IPC



Limitations of the Baseline Runahead Mechanism

- Energy Inefficiency
  - A large number of instructions are speculatively executed
  - Efficient Runahead Execution [ISCA'05, IEEE Micro Top Picks'06]
- Ineffectiveness for pointer-intensive applications
  - Runahead cannot parallelize dependent L2 cache misses
  - Address-Value Delta (AVD) Prediction [MICRO'05]
- Irresolvable branch mispredictions in runahead mode
  - Cannot recover from a mispredicted L2-miss dependent branch
  - Wrong Path Events [MICRO'04]

The Problem: Dependent Cache Misses



- Runahead execution cannot parallelize dependent misses
  - wasted opportunity to improve performance
  - wasted energy (useless pre-execution)

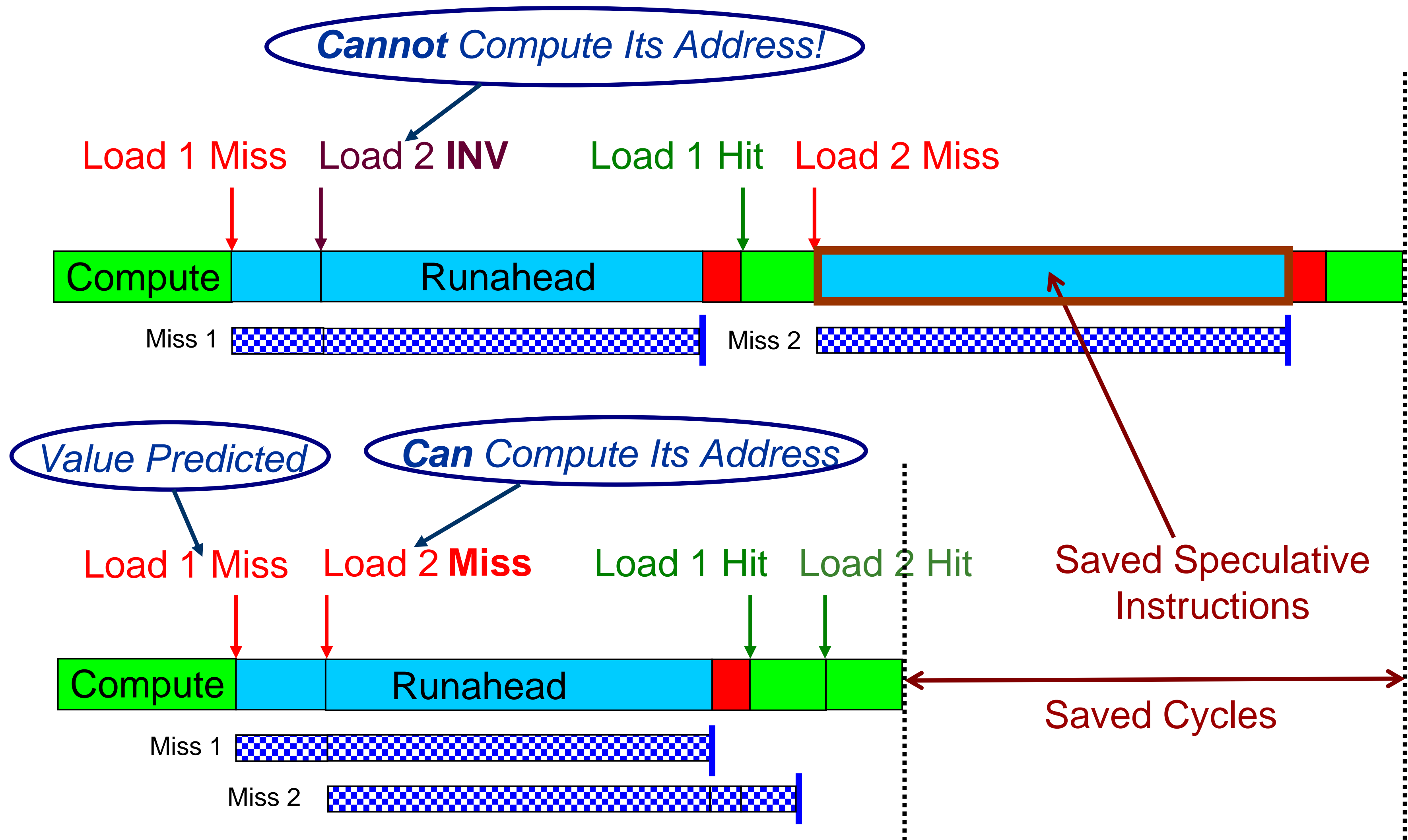
- Runahead performance would improve by 25% if this limitation were ideally overcome

The Goal of AVD Prediction

- Enable the parallelization of dependent L2 cache misses in runahead mode with a low-cost mechanism

- How:
  - Predict the values of L2-miss **address (pointer) loads**
    - **Address load**: loads an address into its destination register, which is later used to calculate the address of another load
    - as opposed to **data load**

Parallelizing Dependent Cache Misses



AVD Prediction [MICRO'05]

- Address-value delta (AVD) of a load instruction defined as:

  AVD = Effective **Address** of Load **–** Data **Value** of Load

- For some address loads, AVD is stable
- An AVD predictor keeps track of the AVDs of address loads
- When a load is an L2 miss in runahead mode, AVD predictor is consulted

- If the predictor returns a stable (confident) AVD for that load, the value of the load is predicted

  Predicted Value = Effective Address **–** Predicted AVD

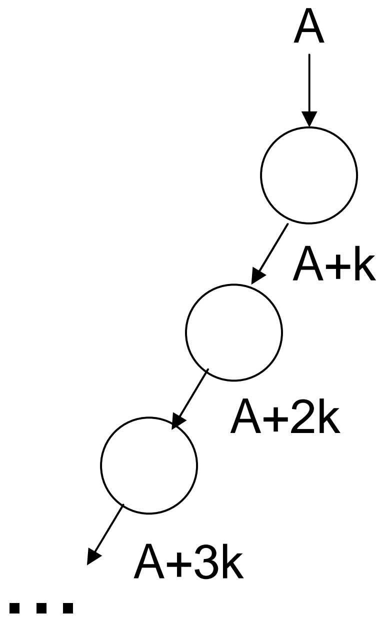
Why Do Stable AVDs Occur?

- Regularity in the way data structures are
  - allocated in memory AND
  - traversed

- Two types of loads can have stable AVDs
  - Traversal address loads
    - Produce addresses consumed by **address loads**
  - Leaf address loads
    - Produce addresses consumed by **data loads**

Traversal Address Loads

Regularly-allocated linked list:



A **traversal address load** loads the pointer to next node:

node = node→next

AVD = Effective Addr – Data Value

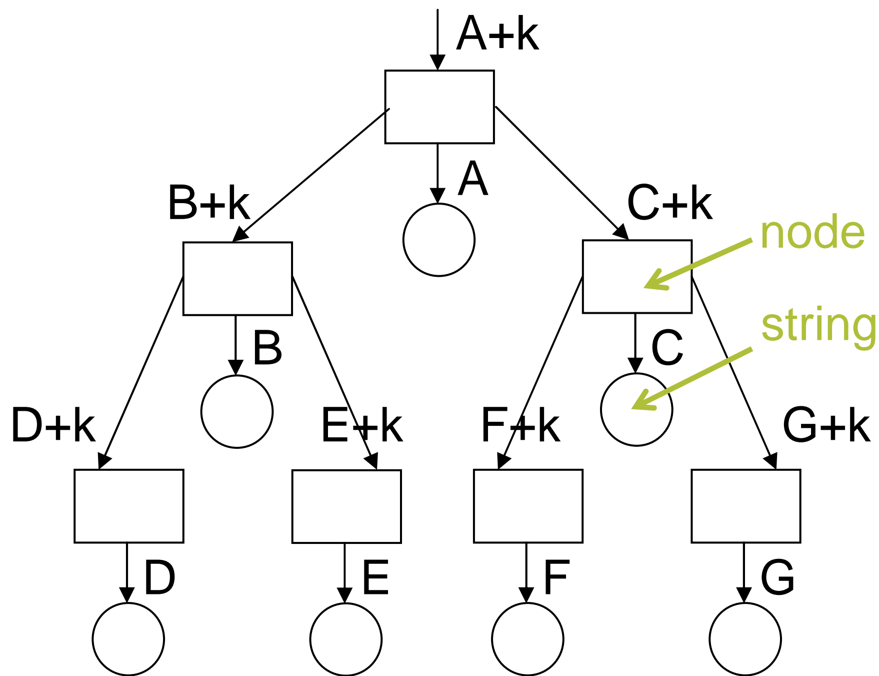
Effective Addr	Data Value	AVD
A	**A+k**	**-k**
A+k	**A+2k**	**-k**
A+2k	**A+3k**	**-k**

Striding data value

Stable AVD

Leaf Address Loads

Sorted dictionary in **parser:**
Nodes point to strings (words)
String and node allocated consecutively



Dictionary looked up for an input word.

A **leaf address load** loads the pointer to the string of each node:

```
lookup (node, input) {     // ...
        ptr_str = node→string;
        m = check_match(ptr_str, input);
        // …
}
```

AVD = Effective Addr – Data Value

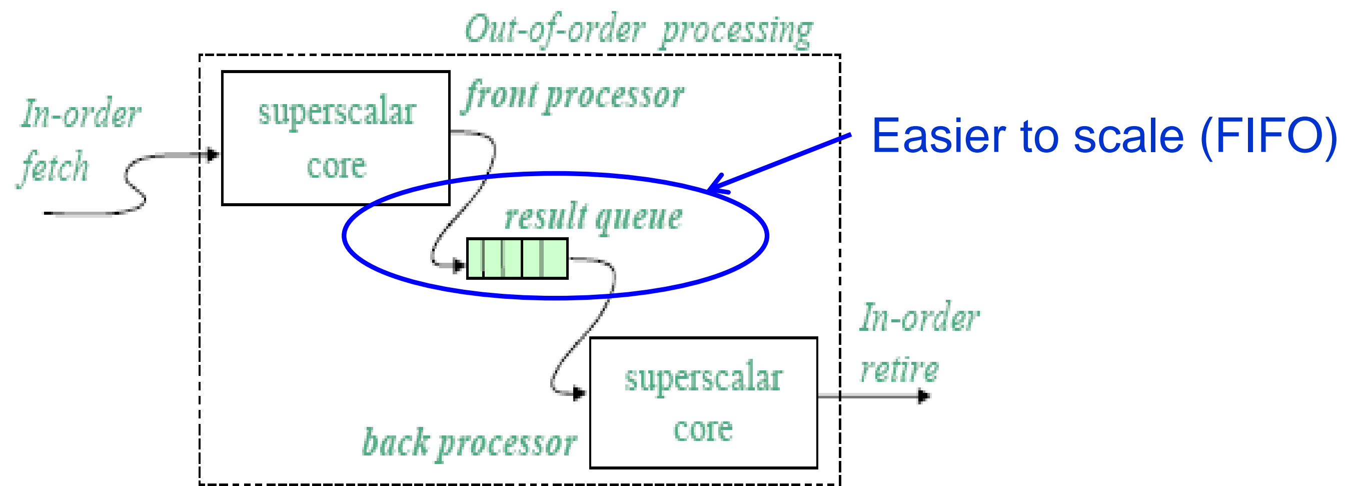
Effective Addr	Data Value	AVD
A+k	**A**	**k**
C+k	**C**	**k**
F+k	**F**	**k**

No stride! Stable AVD

Performance of AVD Prediction

Runahead and Dual Core Execution

- Runahead execution:
  - \+ Approximates the MLP benefits of a large instruction window (no stalling on L2 misses)
  - -- Window size limited by L2 miss latency (runahead ends on miss return)
- Dual-core execution:
  - \+ Window size is not limited by L2 miss latency
  - -- Multiple cores used to execute the application



- Zhou, Dual-Core Execution: "Building a Highly Scalable Single-Thread Instruction Window," PACT 2005.