18-447: Computer Architecture
Lecture 28: Runahead Execution

Prof. Onur Mutlu
Carnegie Mellon University

Spring 2013, 4/12/2013

Homework 6

- Due April 19 (Friday)
- Topics: Virtual memory and cache interaction, main memory, memory scheduling

- Strong suggestion:
  - Please complete this before the exam to prepare for the exam

- Reminder:
  - Homeworks are mainly for your benefit and learning (and preparation for the exam).
  - They are not meant to be a large part of your grade

Lab 6: Memory Hierarchy

- Due April 22 (Monday)
- Cycle-level modeling of L2 cache and DRAM-based main memory

- Extra credit: Prefetching
  - Design your own hardware prefetcher to improve system performance

- HW 6 and Lab 6 are synergistic – work on them together

Midterm II Next Week

- April 17

- Similar format as Midterm I
- Suggestion: Do Homework 6 to prepare for the Midterm

Last Lecture

- Memory Interference (and Techniques to Manage It)
  - With a focus on Memory Request Scheduling
  - Memory Performance Hogs
  - QoS-Aware Memory Scheduling
    - STFM
    - PAR-BS
    - TCM
    - ATLAS
  - Memory Channel Partitioning
  - Source Throttling
  - Thread Co-scheduling
  - Memory Scheduling for Parallel Applications

Today

- Start Memory Latency Tolerance
- Runahead Execution
- Prefetching

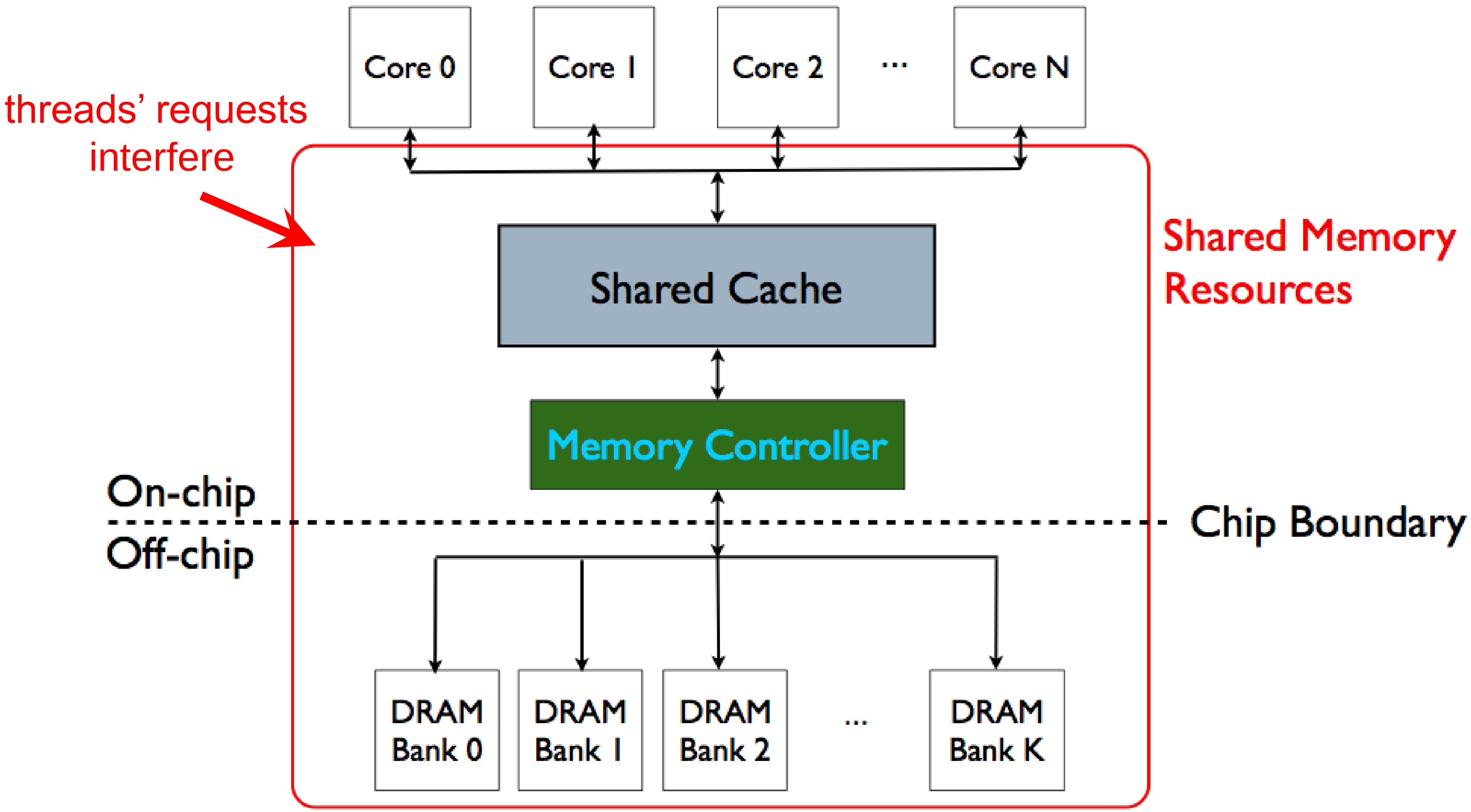
Readings

- Required
  - Mutlu et al., "Runahead execution", 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.

Handling Interference in the Entire Memory System

Memory System is the Major Shared Resource



Inter-Thread/Application Interference

- Problem: Threads share the memory system, but memory system does not distinguish between threads' requests

- Existing memory systems
  - Free-for-all, shared based on demand
  - Control algorithms thread-unaware and thread-unfair
  - Aggressive threads can deny service to others
  - Do not try to reduce or control inter-thread interference

How Do We Solve The Problem?

- Inter-thread interference is uncontrolled in all memory resources
  - Memory controller
  - Interconnect
  - Caches

- We need to control it
  - i.e., design an interference-aware (QoS-aware) memory system

Designing QoS-Aware Memory Systems: Approaches

- Smart resources: Design each shared resource to have a configurable interference control/reduction mechanism
  - QoS-aware memory controllers [Mutlu+ MICRO'07] [Moscibroda+, Usenix Security'07] [Mutlu+ ISCA'08, Top Picks'09] [Kim+ HPCA'10] [Kim+ MICRO'10, Top Picks'11] [Ebrahimi+ ISCA'11, MICRO'11] [Ausavarungnirun+, ISCA'12] [Subramanian+, HPCA'13]
  - QoS-aware interconnects [Das+ MICRO'09, ISCA'10, Top Picks '11, HPCA'13] [Grot+ MICRO'09, ISCA'11, Top Picks '12]
  - QoS-aware caches

- Dumb resources: Keep each resource free-for-all, but reduce/control interference by injection control or data mapping
  - Source throttling to control access to memory system [Ebrahimi+ ASPLOS'10, ISCA'11, TOCS'12] [Ebrahimi+ MICRO'09] [Nychis+ HotNets'10] [Nychis+ SIGCOMM'12]
  - QoS-aware data mapping to memory controllers [Muralidhara+ MICRO'11]
  - QoS-aware thread scheduling to cores [Das+ HPCA'13]

Summary: Memory QoS Approaches and Techniques

- Approaches: Smart vs. dumb resources
  - Smart resources: QoS-aware memory scheduling
  - Dumb resources: Source throttling; channel partitioning
  - Both approaches are effective in reducing interference
  - No single best approach for all workloads

- Techniques: Request scheduling, source throttling, memory partitioning
  - All approaches are effective in reducing interference
  - No single best technique for all workloads

- Combined approaches and techniques are the most powerful
  - Integrated Memory Channel Partitioning and Scheduling [MICRO’11]

Tolerating Memory Latency

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

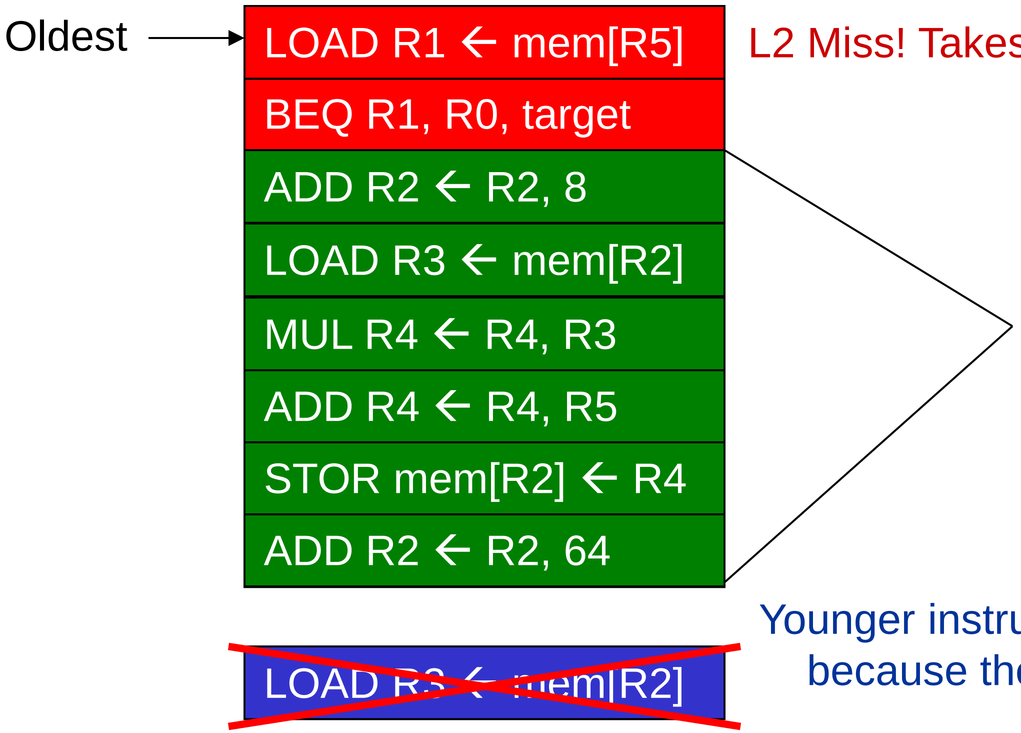
- Incoming instructions fill the instruction window (reorder buffer, reservation stations).

- 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.

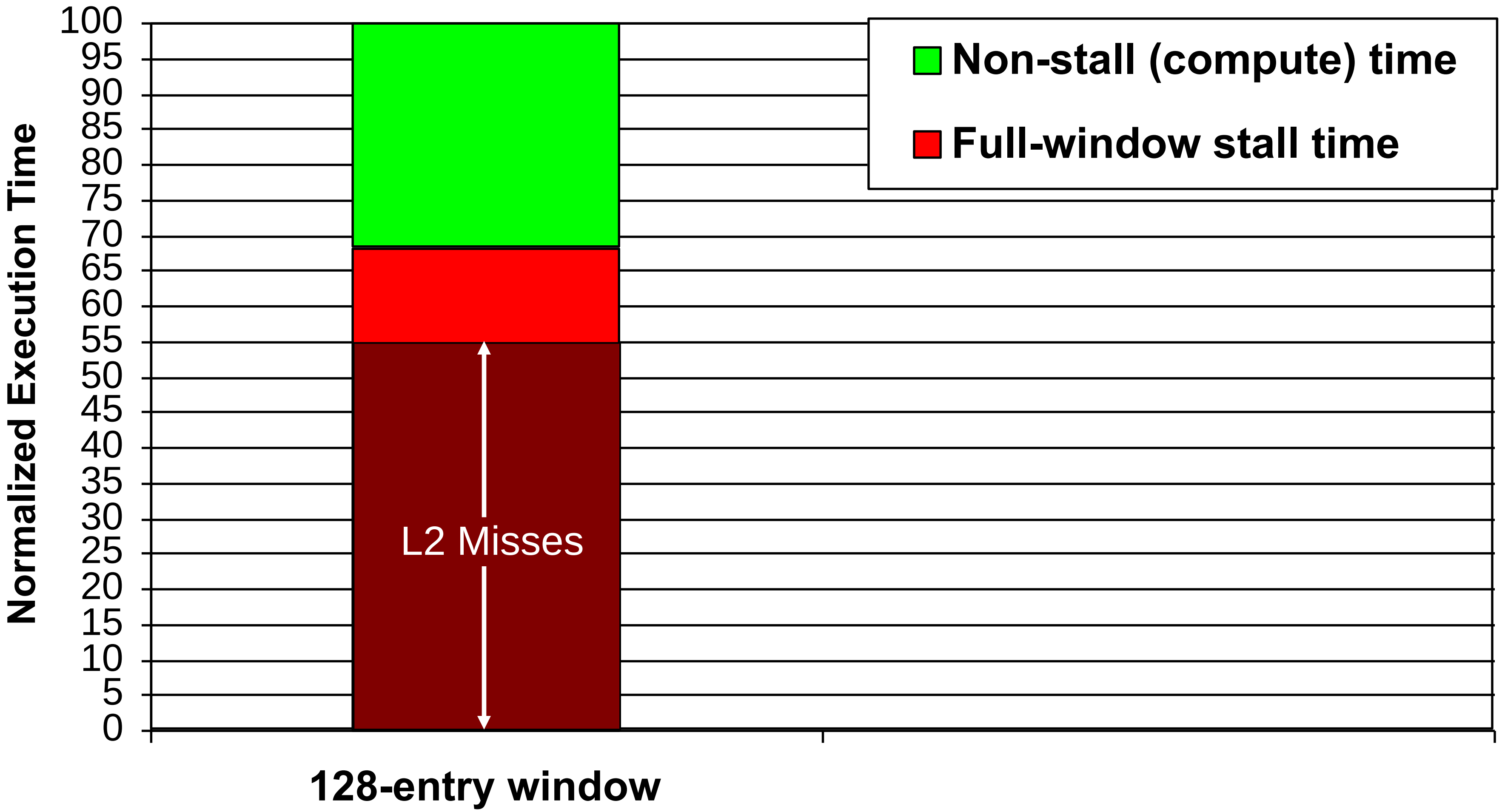
Independent of the L2 miss, executed out of program order, but cannot be retired.

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

The processor stalls until the L2 Miss is serviced.

- L2 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

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

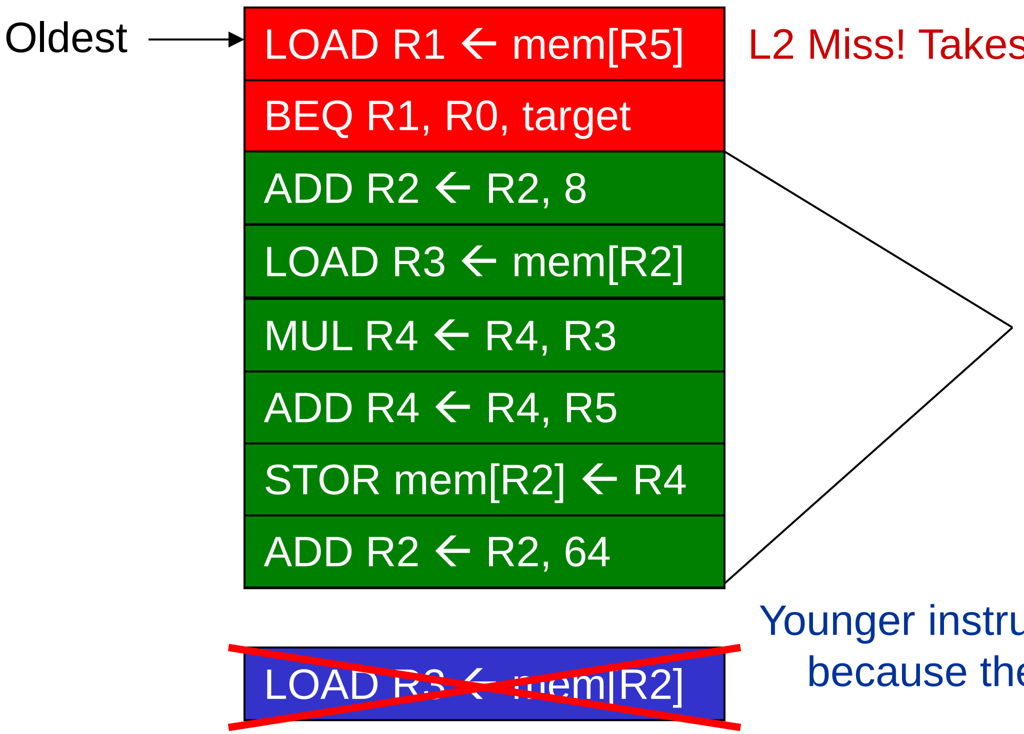
- Multithreading [initially in CDC 6600, 1964]
  - Works well if there are multiple threads
  - Improving single thread performance using multithreading hardware is an ongoing research effort

- 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 in a later lecture)

Runahead Execution

Small Windows: Full-window Stalls

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.

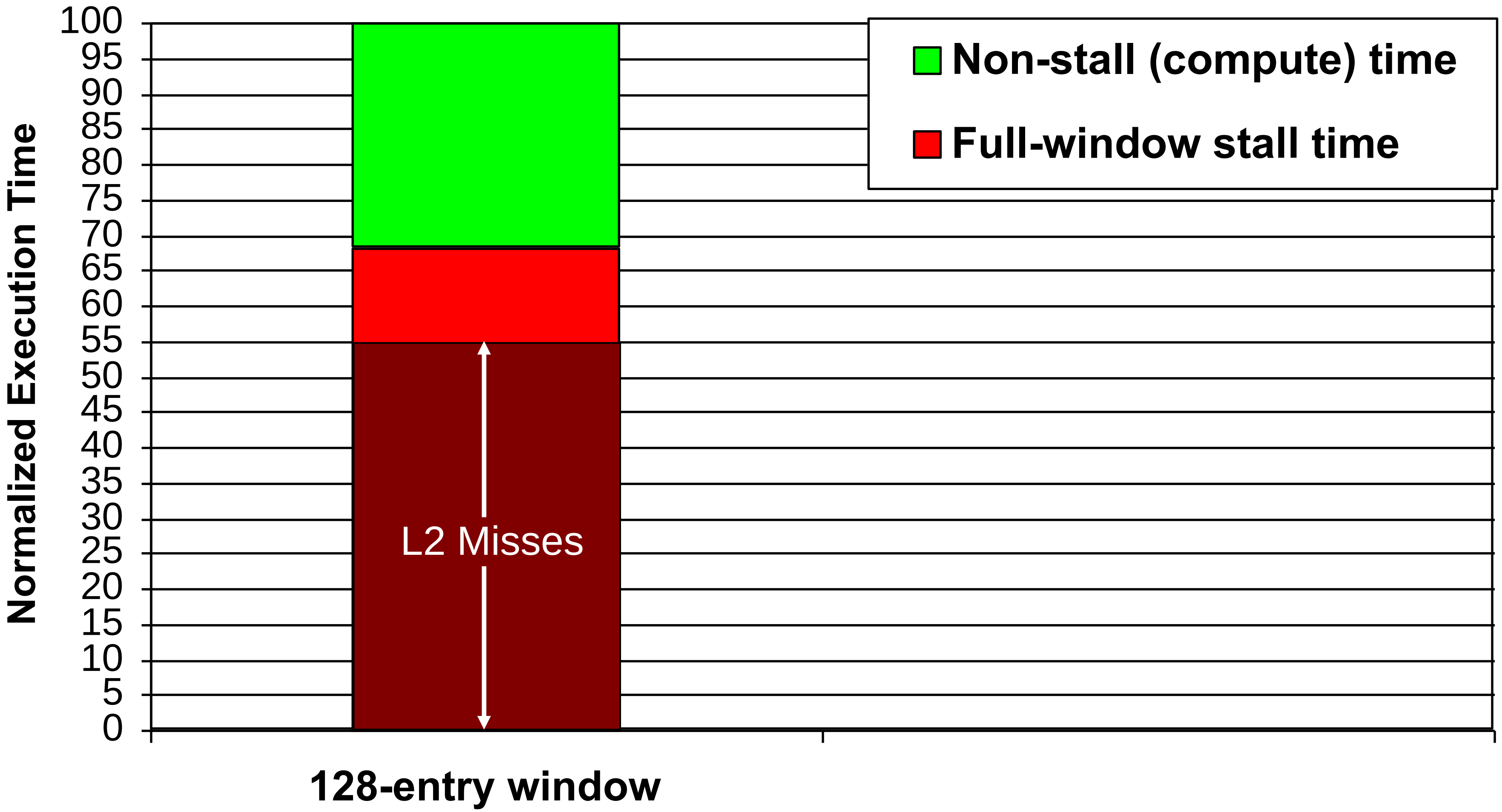
Independent of the L2 miss, executed out of program order, but cannot be retired.

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

The processor stalls until the L2 Miss is serviced.

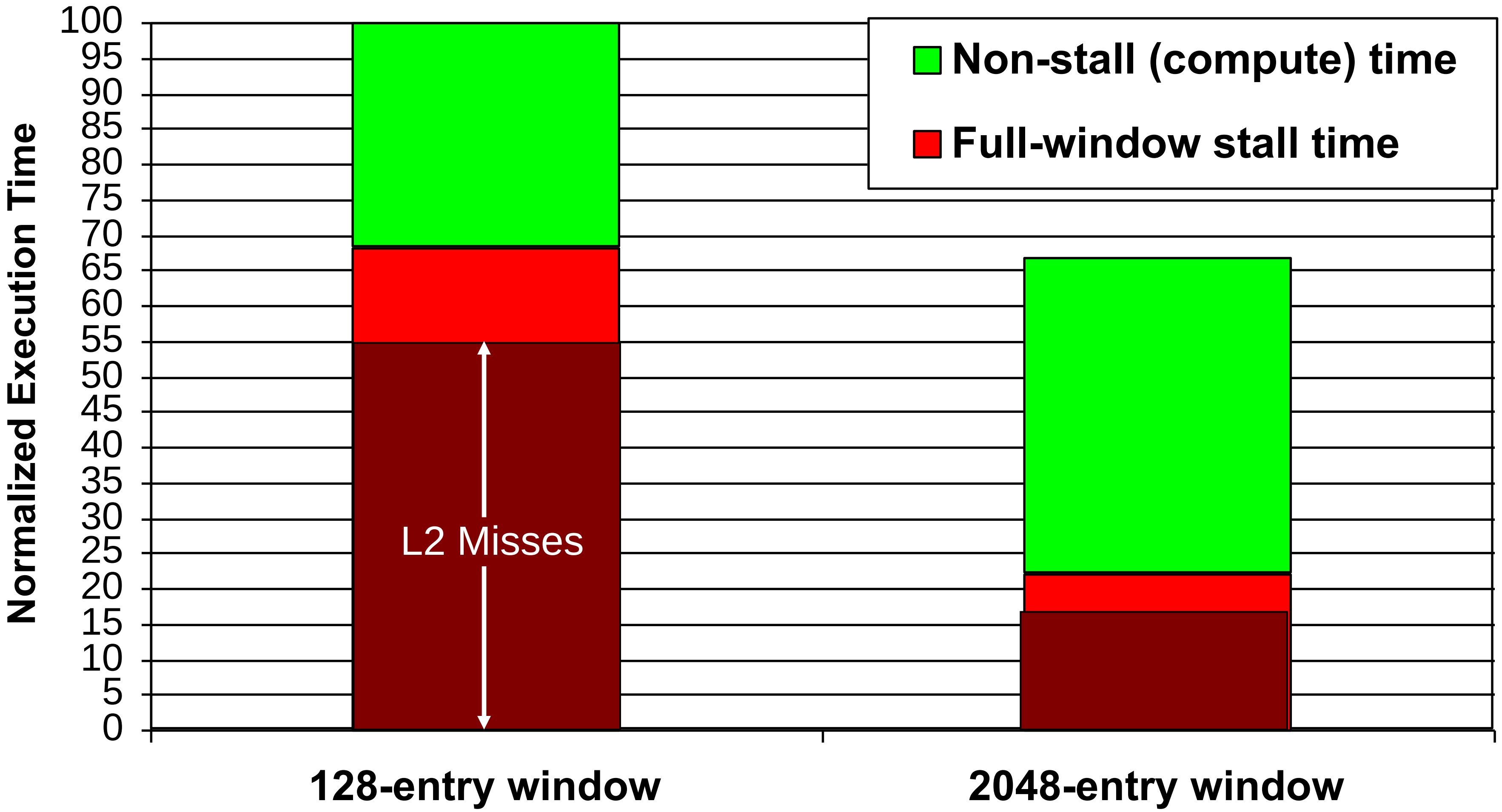
- L2 cache misses are responsible for most full-window stalls.

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

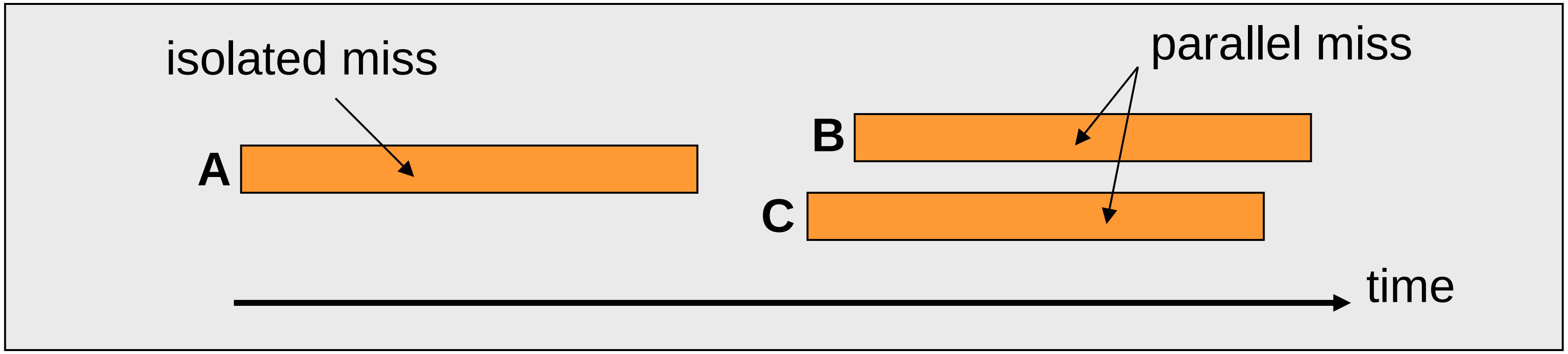
- One of the major research issues in out of order execution

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

- 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, runahead, prefetching

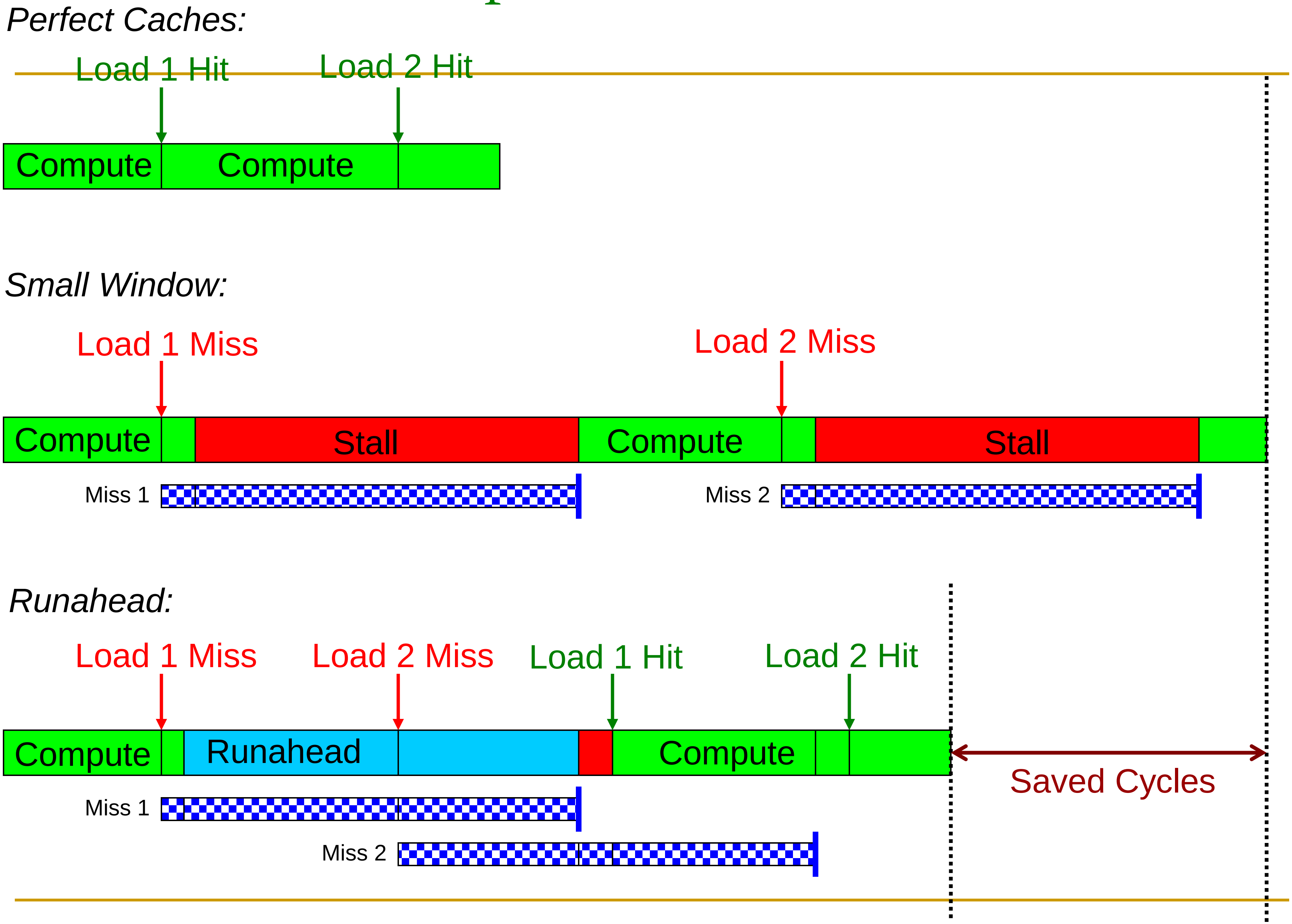
Runahead Execution (I)

- 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.

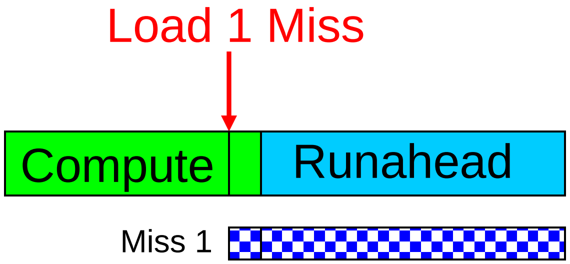
Runahead Execution Mechanism

- Entry into runahead mode
  - Checkpoint architectural register state

- Instruction processing in runahead mode

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

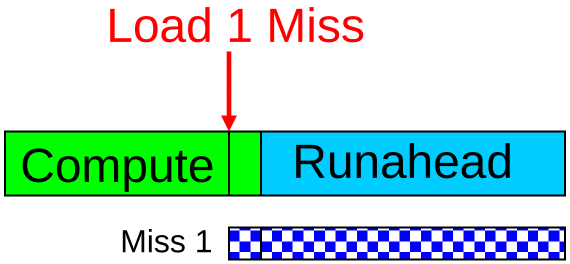
Instruction Processing in Runahead Mode



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

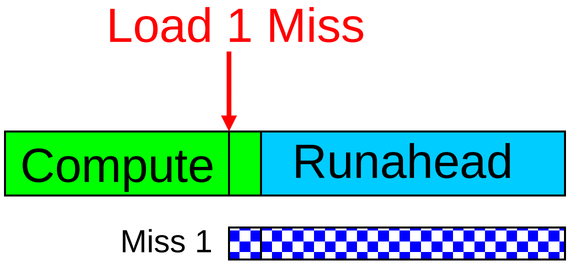
- It is purely speculative: Architectural (software-visible) register/memory state is NOT updated in runahead mode.
- L2-miss dependent instructions are identified and treated specially.
  - They are quickly removed from the instruction window.
  - Their results are not trusted.

L2-Miss Dependent Instructions



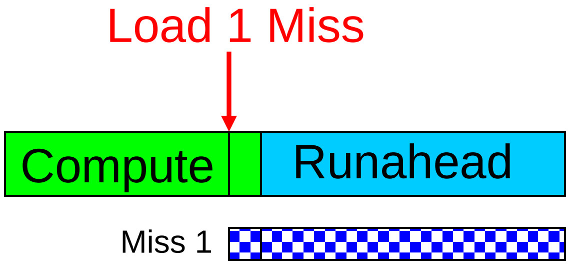
- Two types of results produced: INV and VALID
- INV = Dependent on an L2 miss
- INV results are marked using INV bits in the register file and store buffer.
- INV values are not used for prefetching/branch resolution.

Removal of Instructions from Window



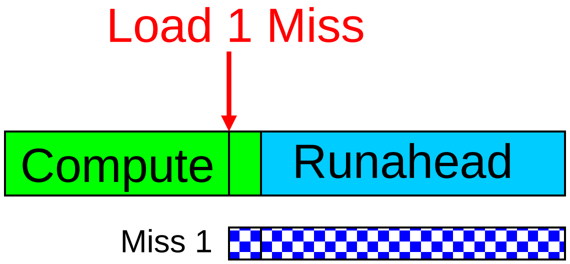
- Oldest instruction is examined for pseudo-retirement
  - An INV instruction is removed from window immediately.
  - A VALID instruction is removed when it completes execution.
- Pseudo-retired instructions free their allocated resources.
  - This allows the processing of later instructions.
- Pseudo-retired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

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:

    + 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

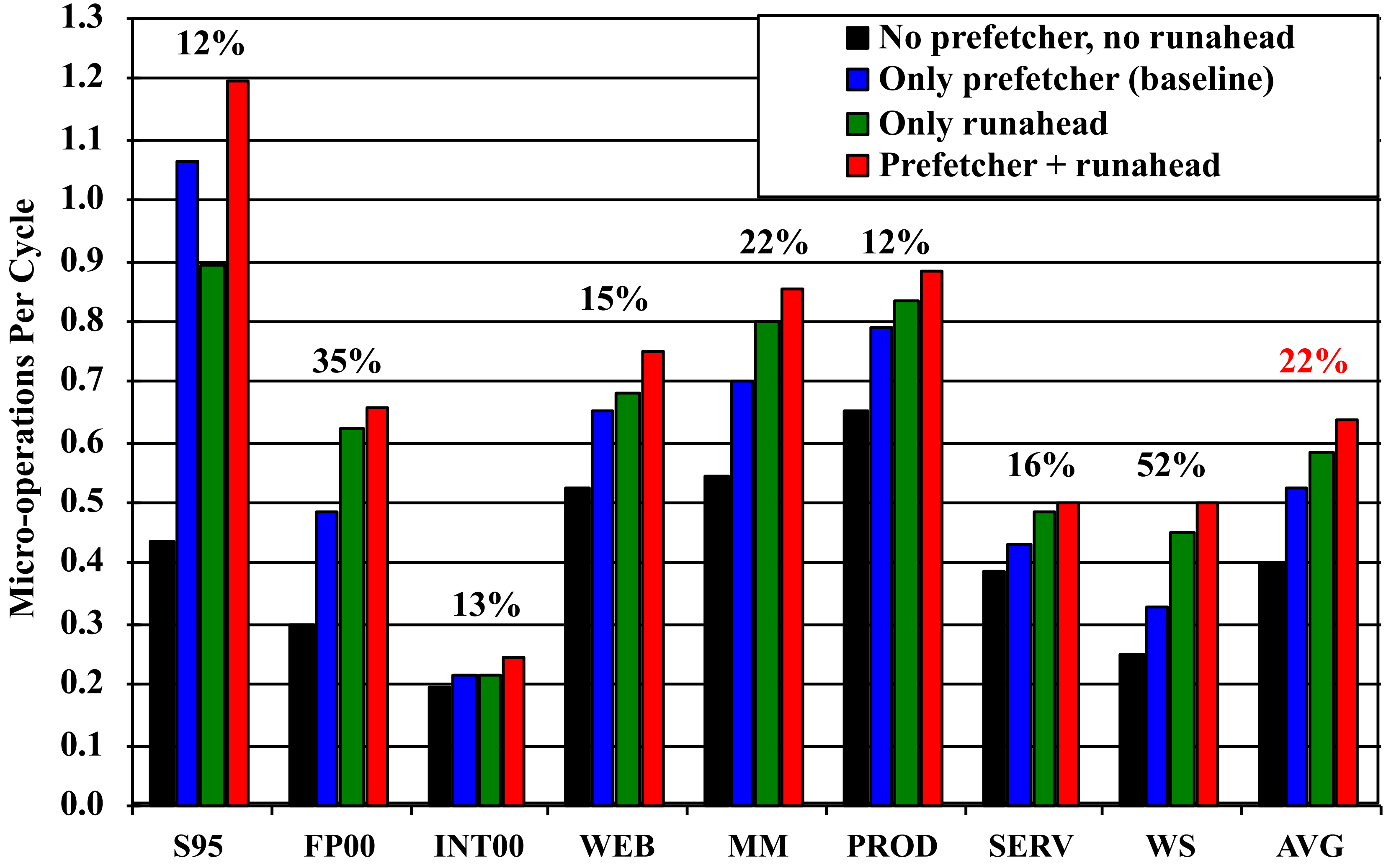
  -- Cannot prefetch dependent cache misses. Solution?

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

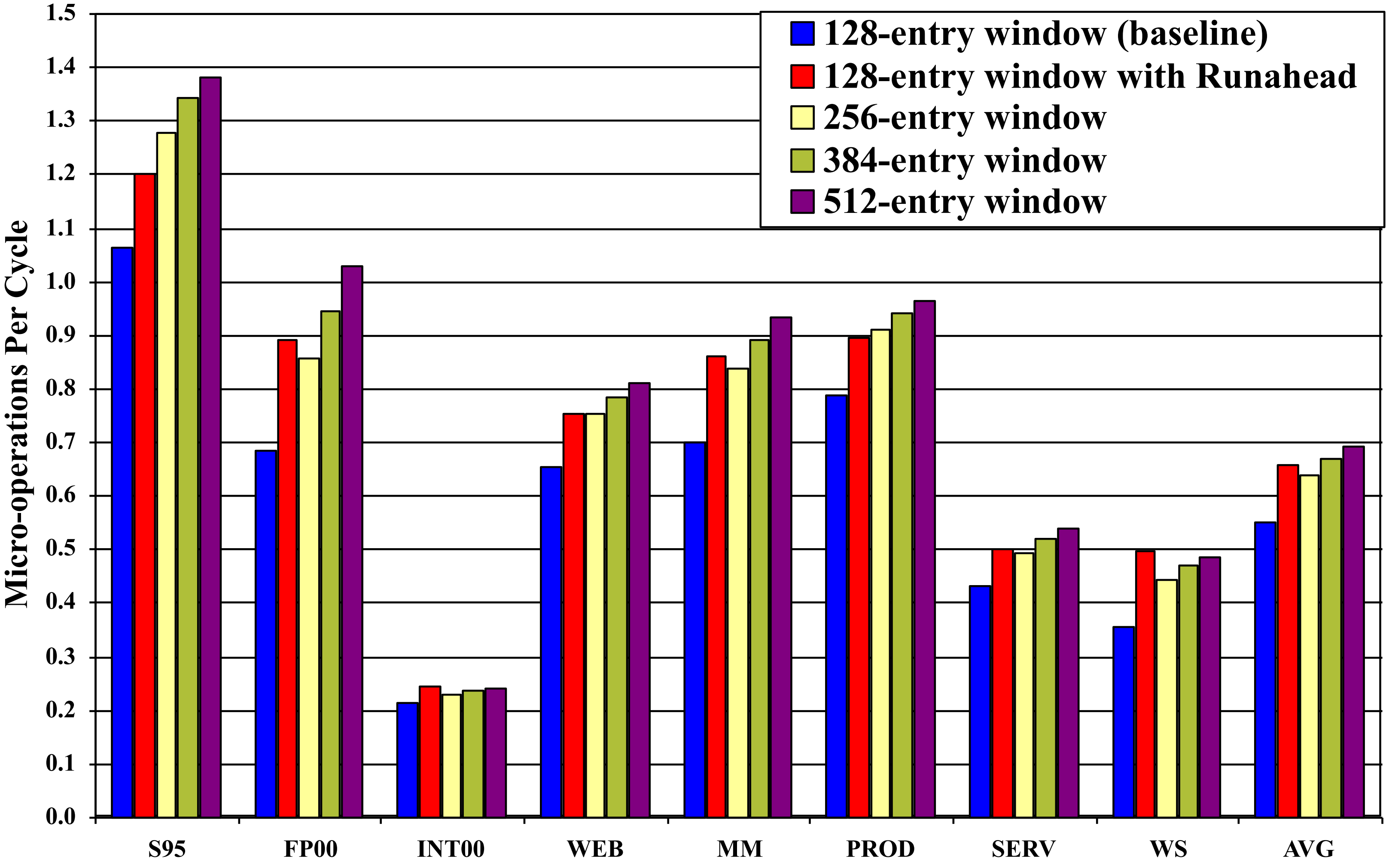
  -- Prefetch distance limited by memory latency

- Implemented in IBM POWER6, Sun "Rock"

Performance of Runahead Execution



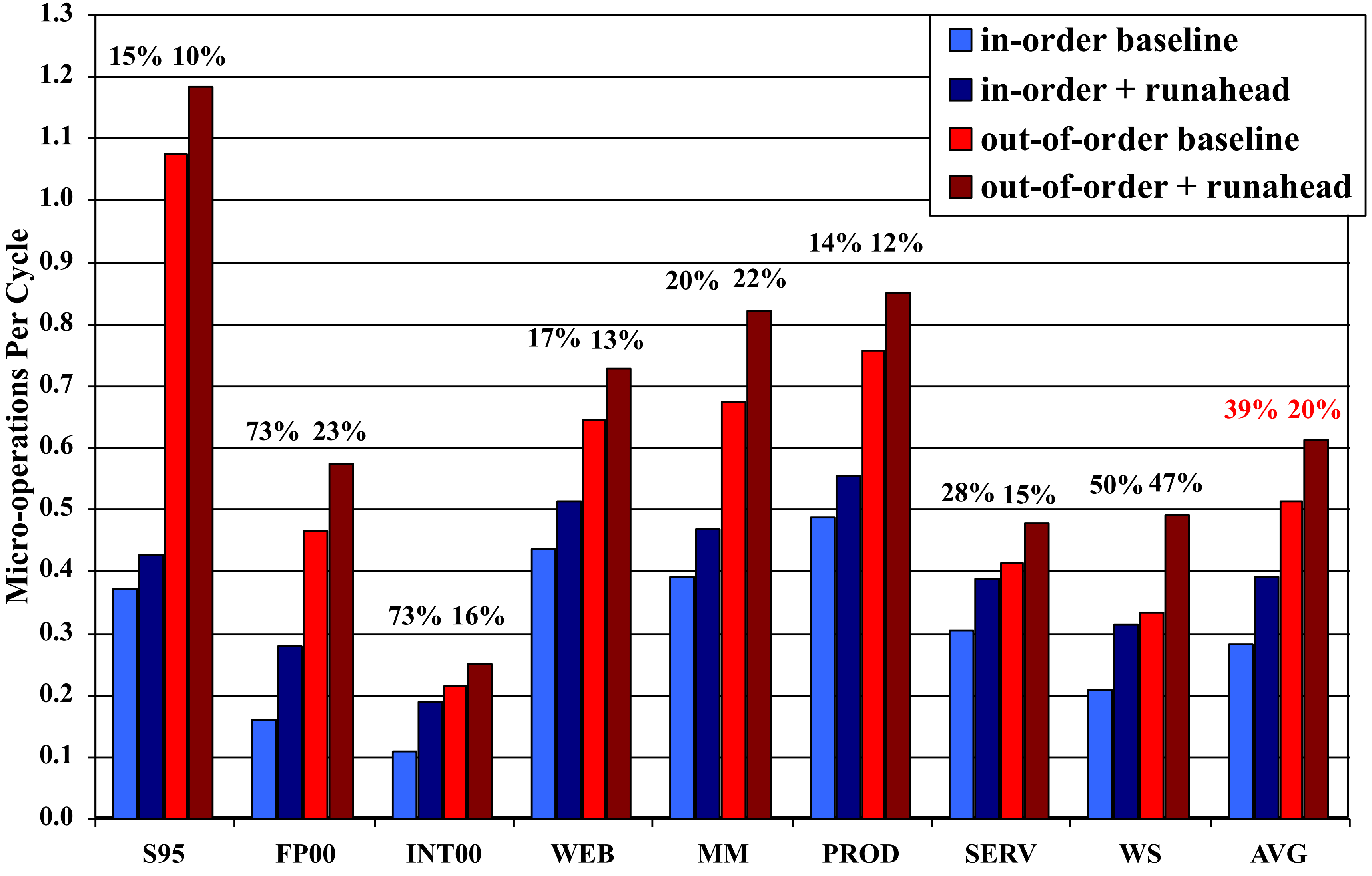
Runahead Execution vs. Large Windows



Runahead vs. A Real Large Window

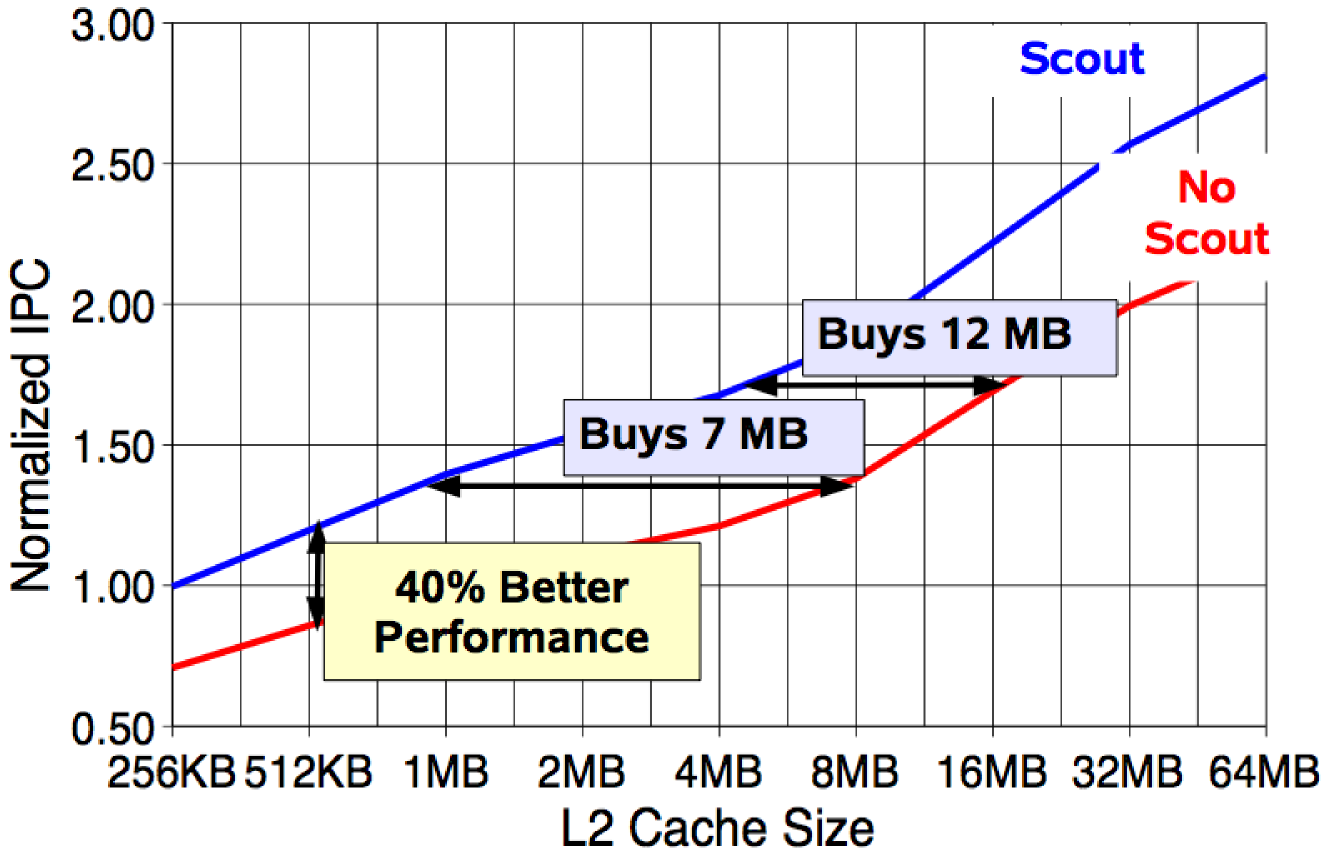
- When is one beneficial, when is the other?
- Pros and cons of each

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



Effect of Runahead in Sun ROCK

- Shailender Chaudhry talk, Aug 2008.



Runahead Enhancements

Readings

- Required
  - Mutlu et al., "Runahead Execution", HPCA 2003, Top Picks 2003.
- Recommended
  - 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.

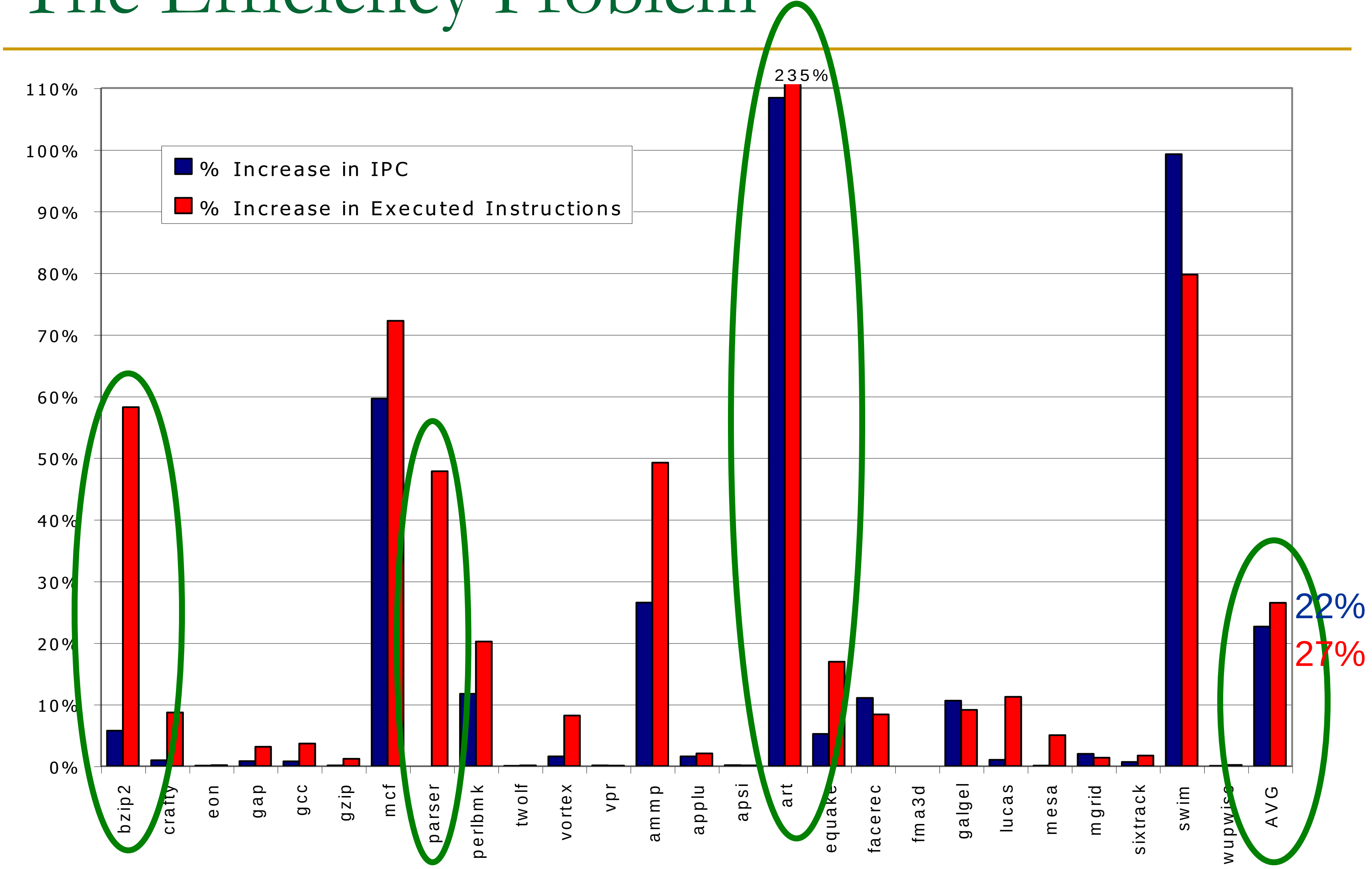
Limitations of the Baseline Runahead Mechanism

- 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

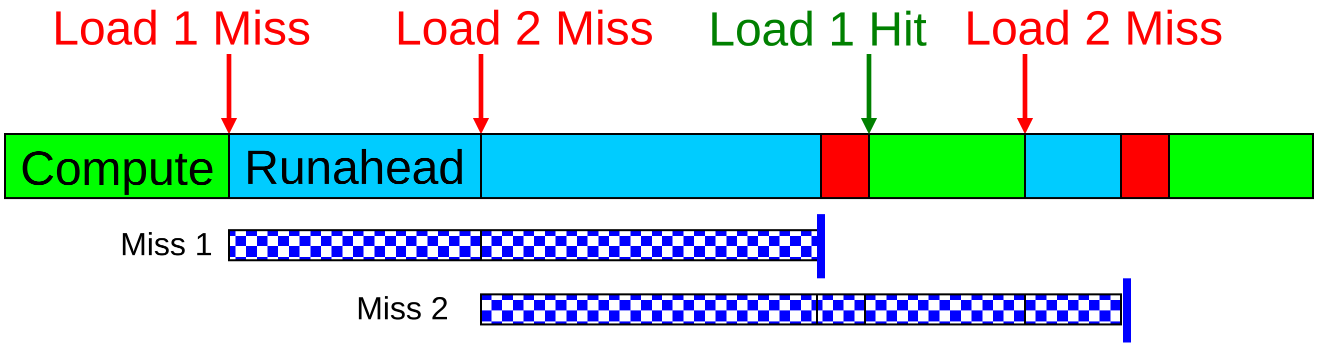
- Overlapping runahead periods

- Useless runahead periods

- Mutlu et al., "Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance," ISCA 2005, IEEE Micro Top Picks 2006.

Short Runahead Periods

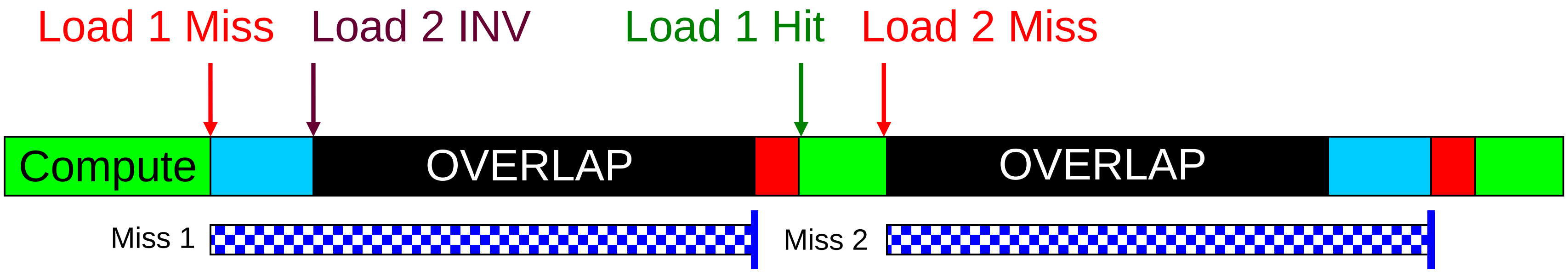
- Processor can initiate runahead mode due to an already in-flight L2 miss generated by
  - the prefetcher, wrong-path, or a previous runahead period



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

Overlapping Runahead Periods

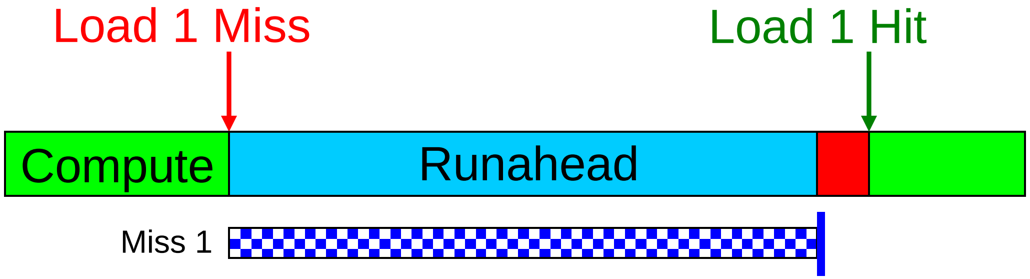
- Two runahead periods that execute the same instructions



- Second period is inefficient

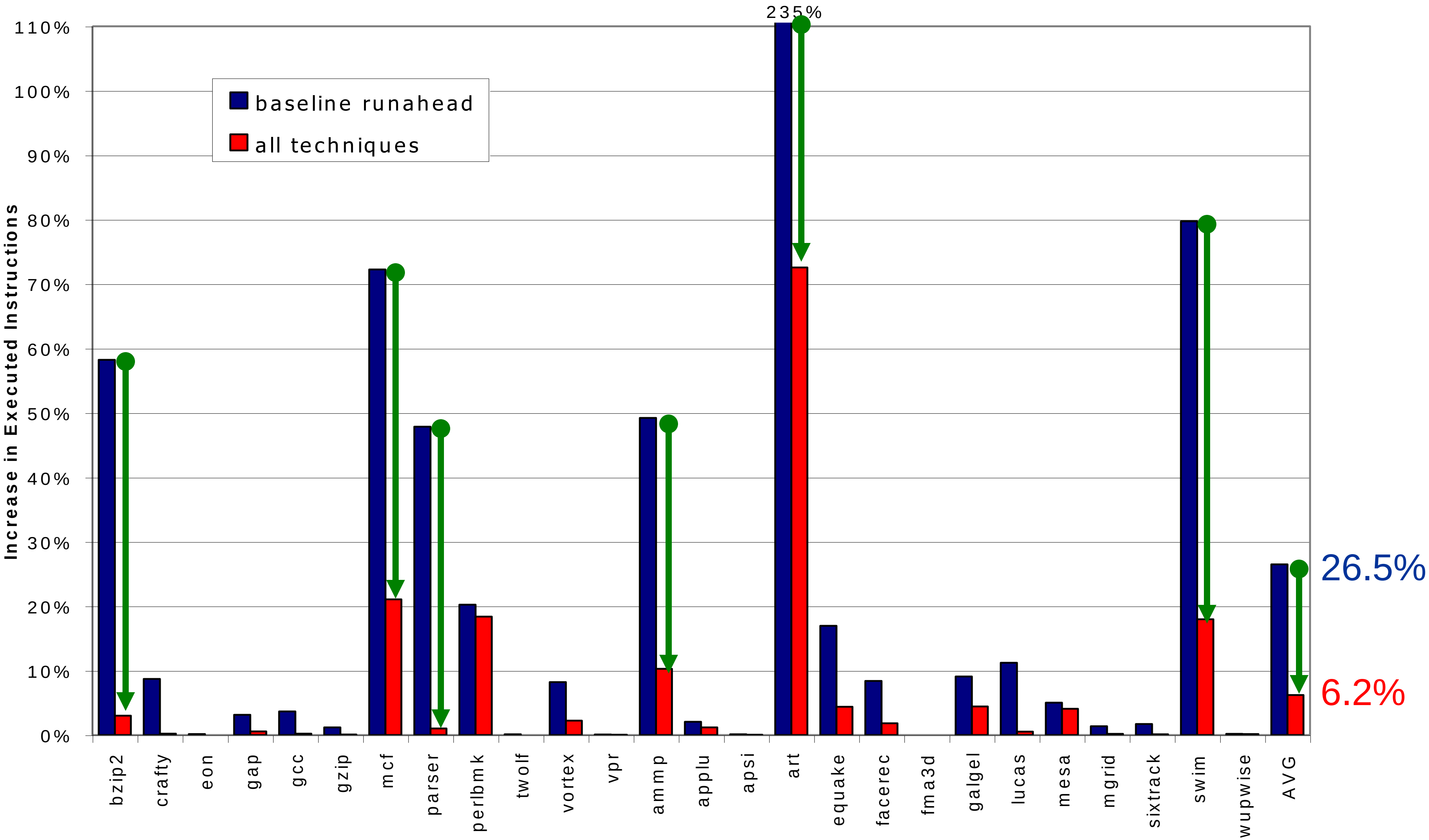
Useless Runahead Periods

- Periods that do not result in prefetches for normal mode

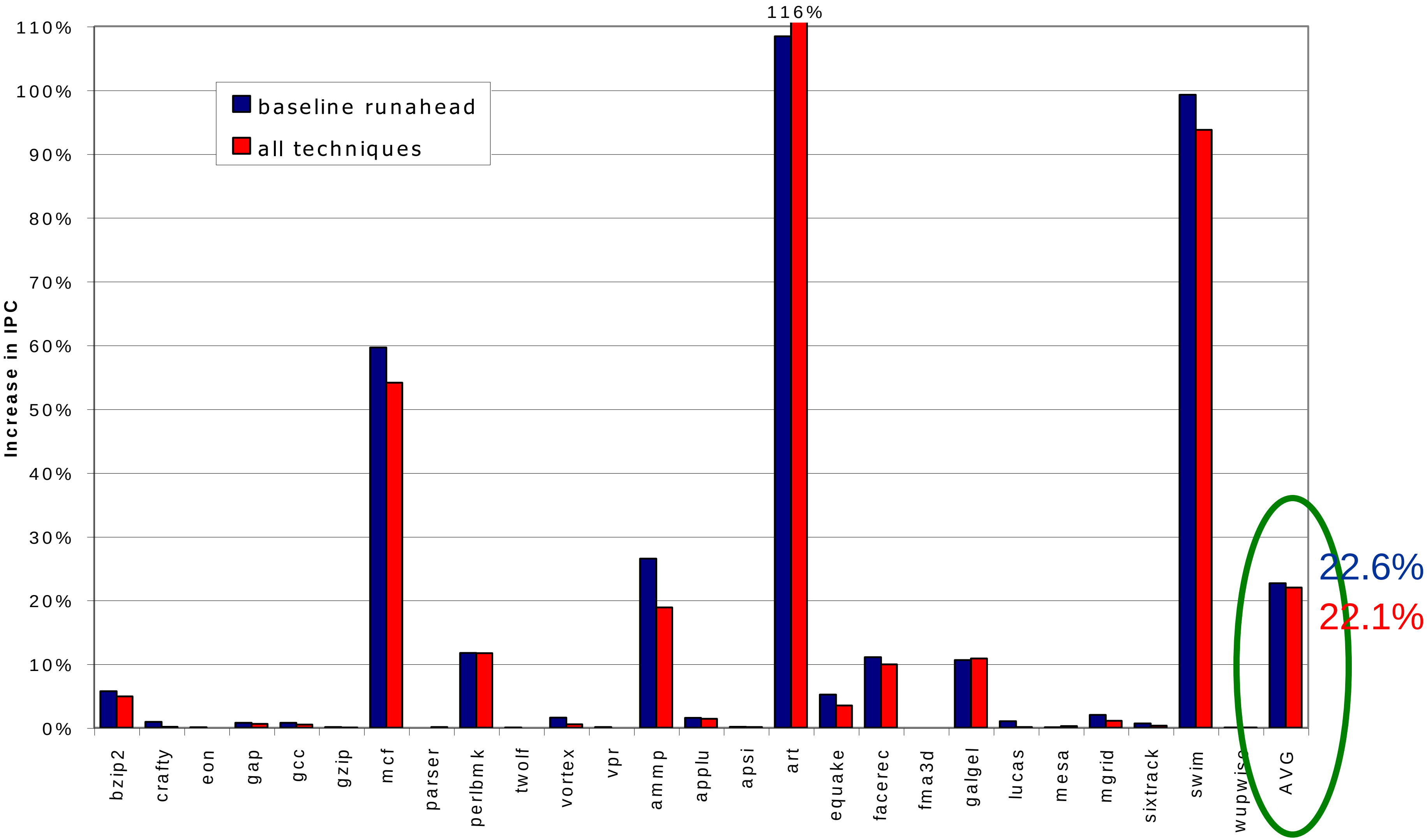


- They exist due to the lack of memory-level parallelism
- Mechanism to eliminate useless periods:
  - Predict if a period will generate useful L2 misses
  - Estimate a period to be useful if it generated an L2 miss that cannot be captured by the instruction window
    - Useless period predictors are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

Overall Impact on Executed Instructions



Overall Impact on IPC



Taking Advantage of Pure Speculation

- Runahead mode is purely speculative

- The goal is to find and generate cache misses that would otherwise stall execution later on

- How do we achieve this goal most efficiently and with the highest benefit?

- Idea: Find and execute only those instructions that will lead to cache misses (that cannot already be captured by the instruction window)

- How?

Limitations of the Baseline Runahead Mechanism

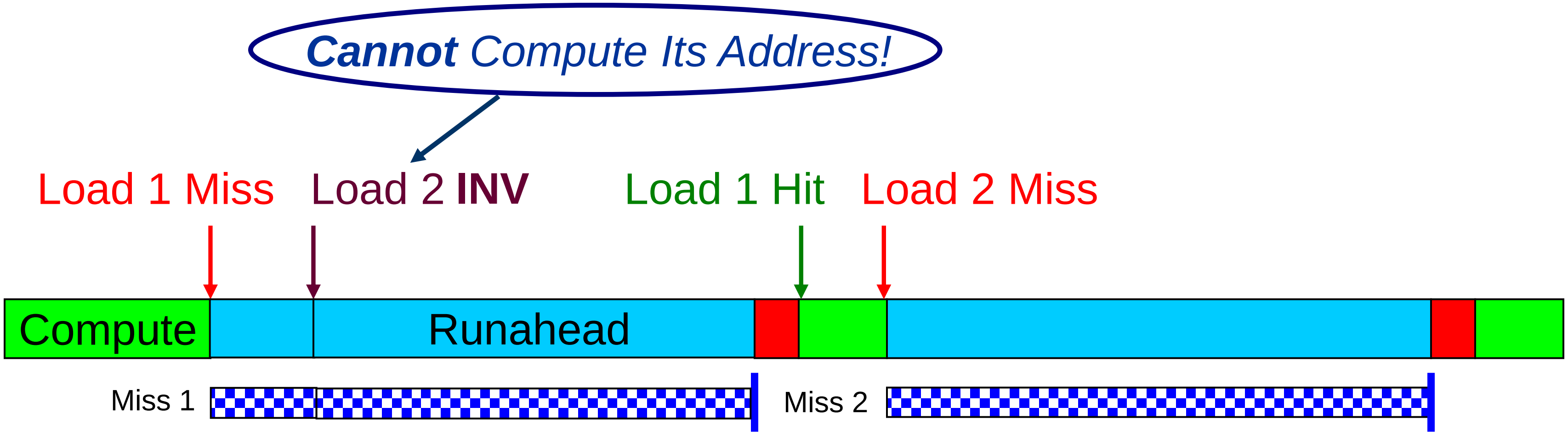
- Energy Inefficiency
  - A large number of instructions are speculatively executed
  - Efficient Runahead Execution [ISCA' 05, IEEE Micro Top Picks' 06]

- 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: **Load 2 is dependent on Load 1***



- 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

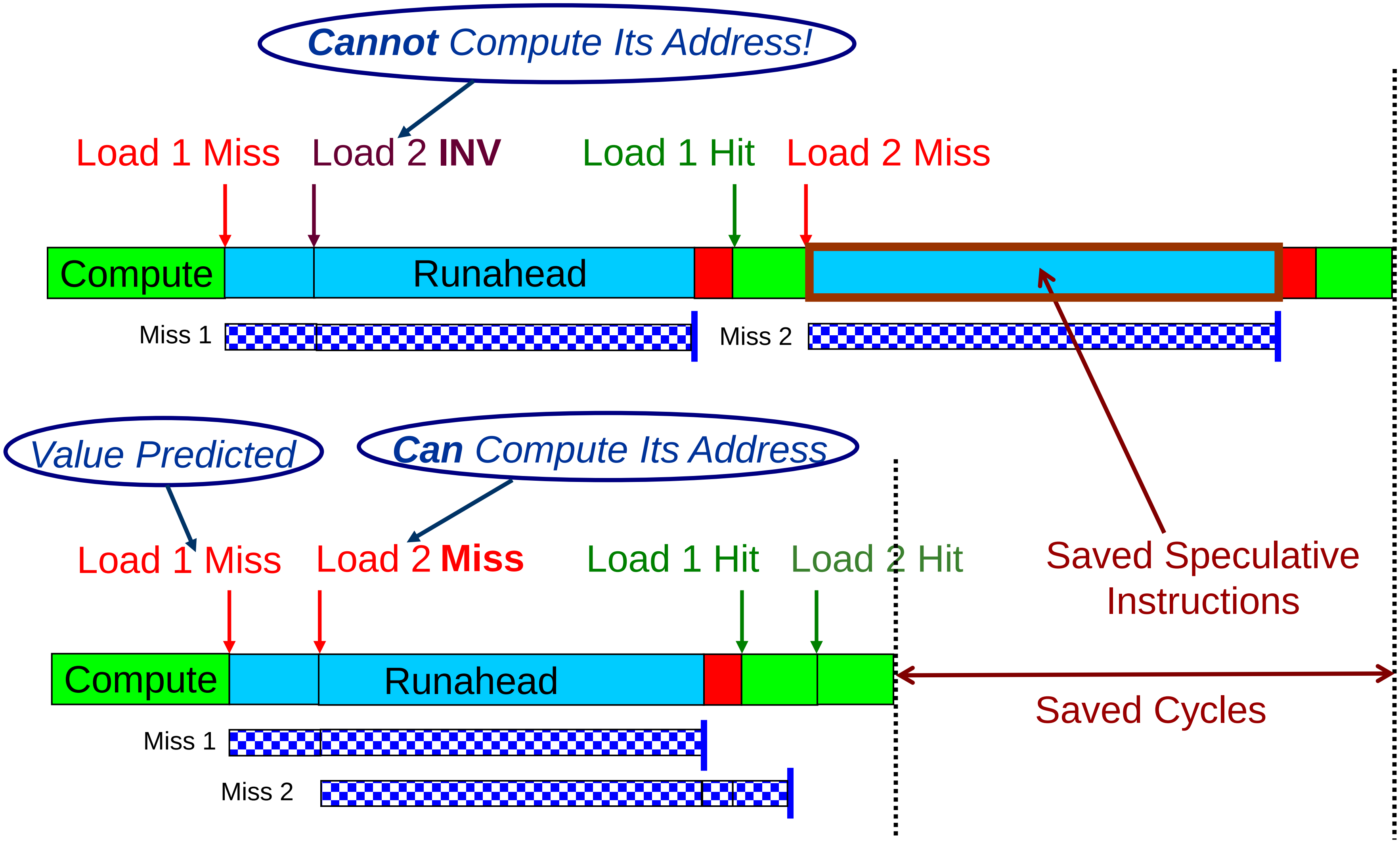
Parallelizing Dependent Cache Misses

- Idea: 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**

- Read:
  - Mutlu et al., "Address-Value Delta (AVD) Prediction," MICRO 2005.

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

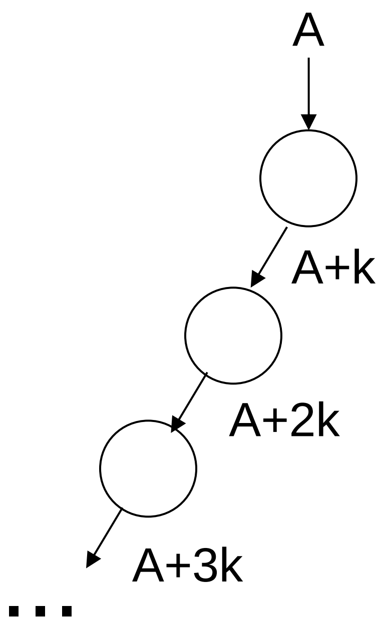
Why Do Stable AVDs Occur?

- 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

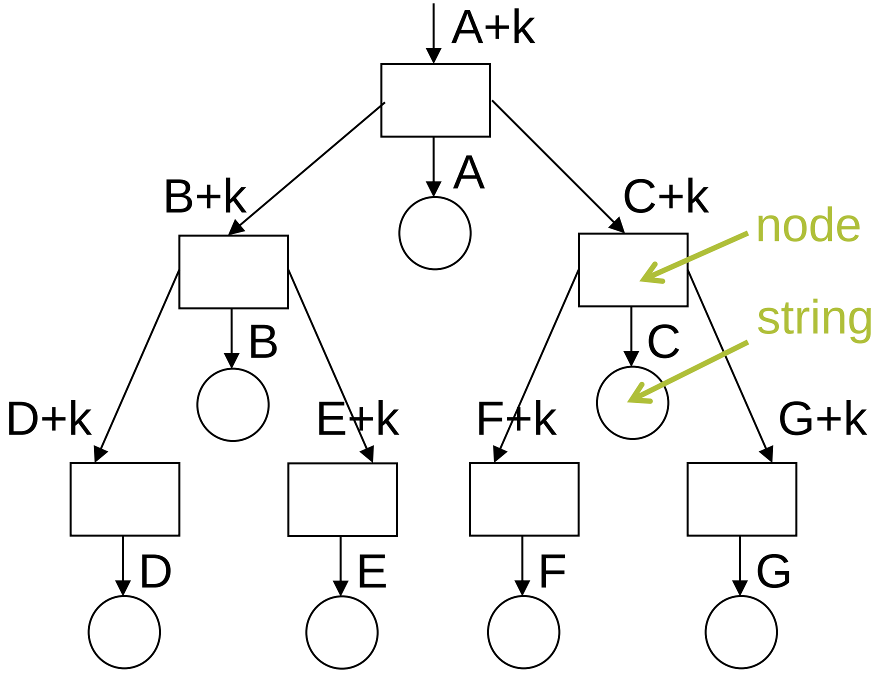
Effective Addr	Data Value	AVD
A	**A+k**	**-k**
A+k	**A+2k**	**-k**
A+2k	**A+3k**	**-k**

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

Identifying Address Loads in Hardware

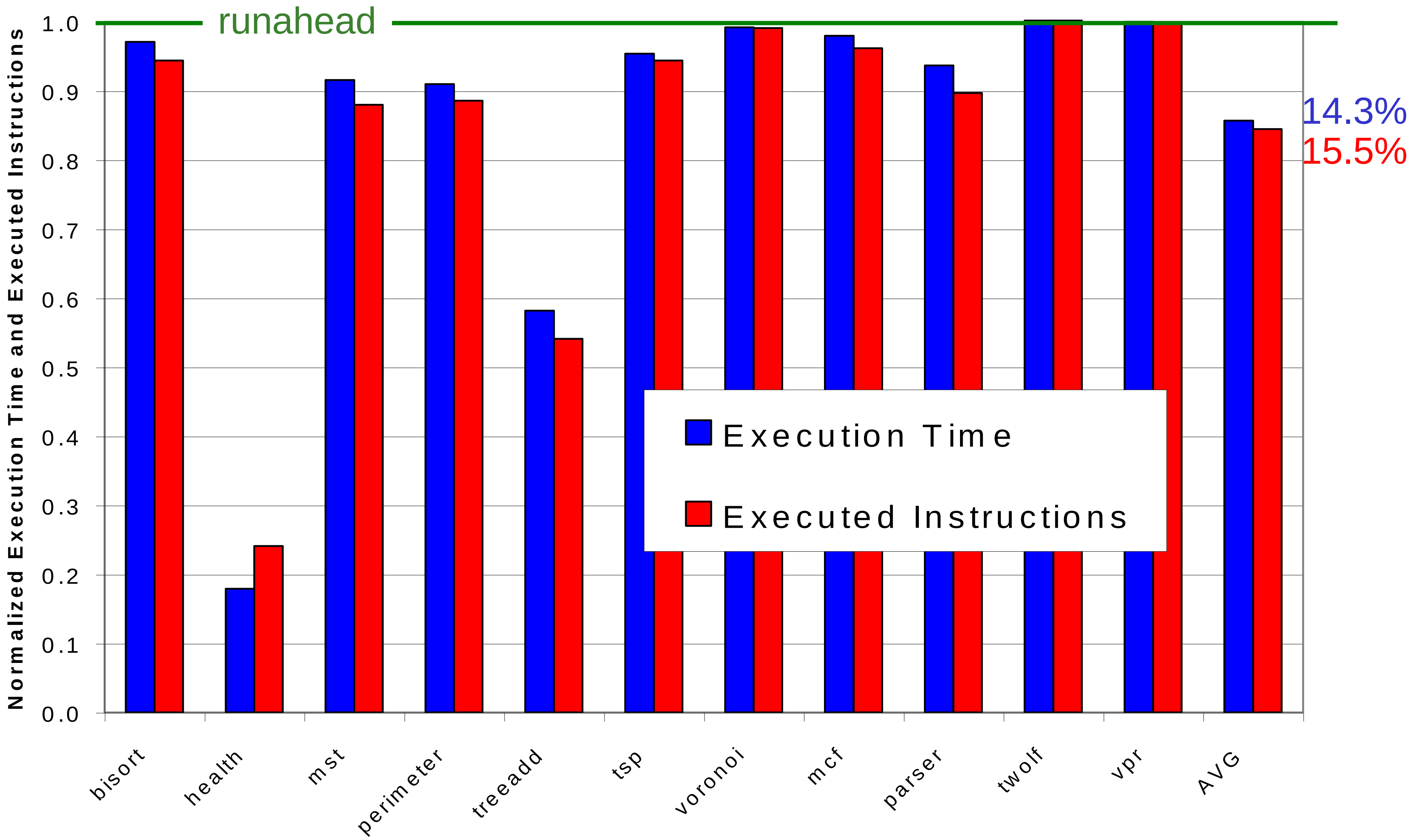
- Insight:
  - If the AVD is too large, the value that is loaded is likely **not** an address

- Only keep track of loads that satisfy:

  $-\text{MaxAVD} \leq \text{AVD} \leq +\text{MaxAVD}$

- This identification mechanism eliminates many loads from consideration
  - Enables the AVD predictor to be small

Performance of AVD Prediction



Readings

- Required
  - Mutlu et al., "Runahead Execution", HPCA 2003, Top Picks 2003.
- Recommended
  - 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.

We did not cover the following slides in lecture. They are for your benefit.

More on Runahead Enhancements

Eliminating Short Periods

- Mechanism to eliminate short periods:
  - Record the number of cycles C an L2-miss has been in flight
  - If C is greater than a threshold T for an L2 miss, disable entry into runahead mode due to that miss
  - T can be determined statically (at design time) or dynamically

- T=400 for a minimum main memory latency of 500 cycles works well

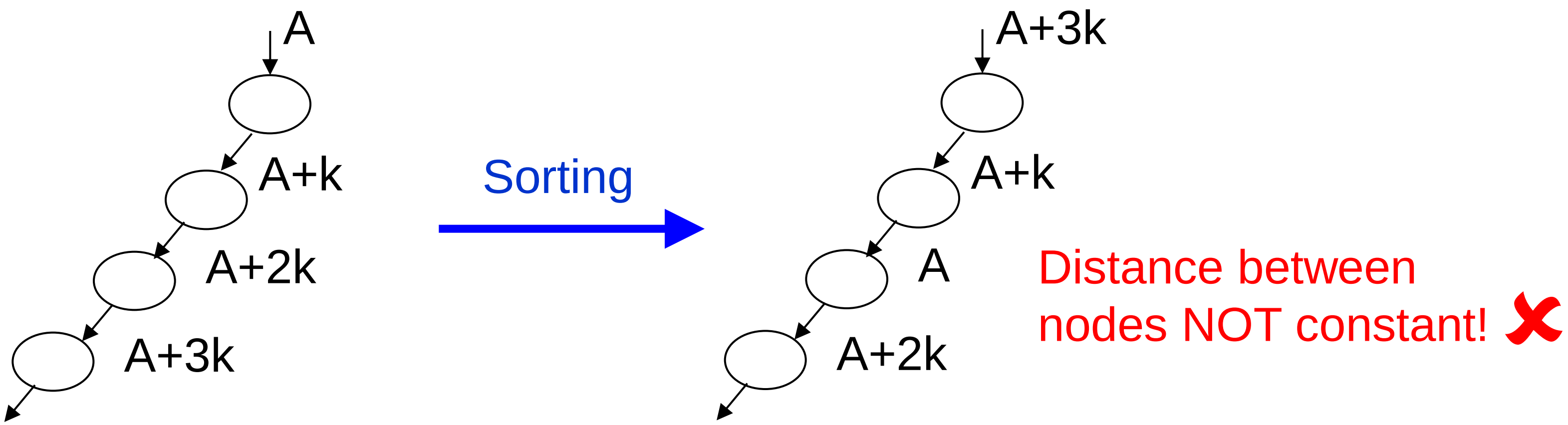
Eliminating Overlapping Periods

- Overlapping periods are not necessarily useless
  - The availability of a new data value can result in the generation of useful L2 misses
- But, this does not happen often enough

- Mechanism to eliminate overlapping periods:
  - Keep track of the number of pseudo-retired instructions *R* during a runahead period
  - Keep track of the number of fetched instructions *N* since the exit from last runahead period
  - If $N < R$, do not enter runahead mode

Properties of Traversal-based AVDs

- Stable AVDs can be captured with a stride value predictor
- Stable AVDs disappear with the re-organization of the data structure (e.g., sorting)

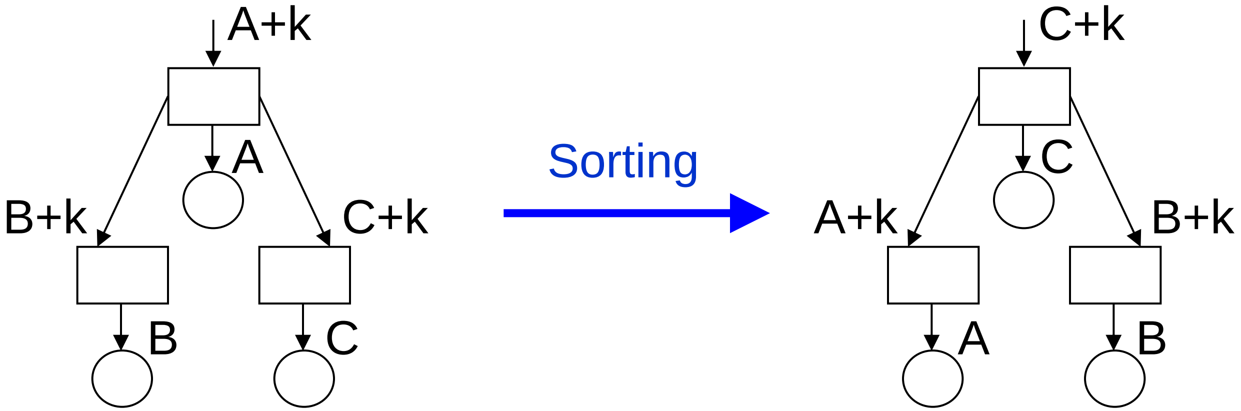


Distance between nodes NOT constant! ✘

- Stability of AVDs is dependent on the behavior of the memory allocator
  - Allocation of contiguous, fixed-size chunks is useful

Properties of Leaf-based AVDs

- Stable AVDs **cannot** be captured with a stride value predictor
- Stable AVDs **do not** disappear with the re-organization of the data structure (e.g., sorting)



Distance between **node** and **string** still constant! ✓

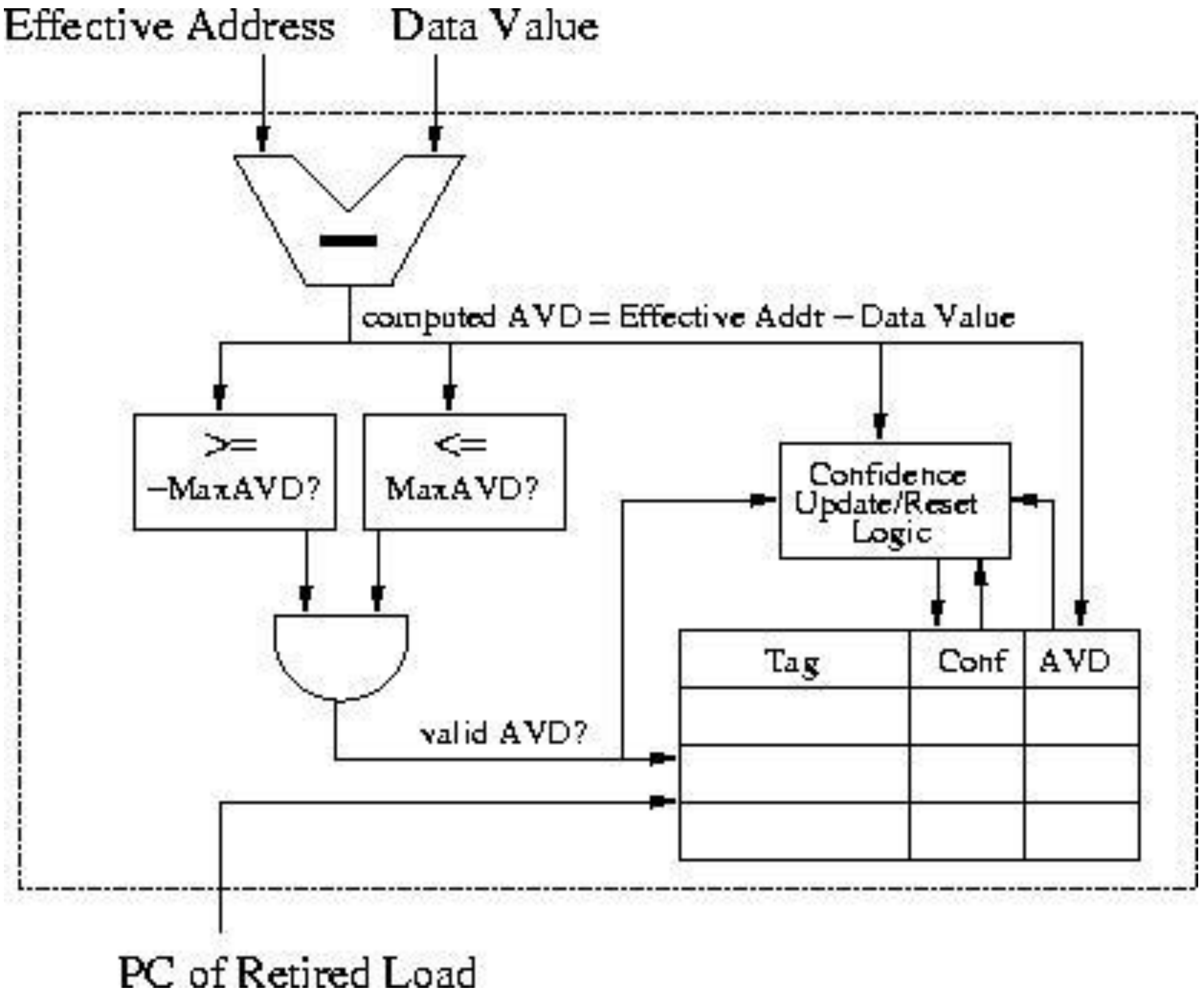
- Stability of AVDs is dependent on the behavior of the memory allocator

An Implementable AVD Predictor

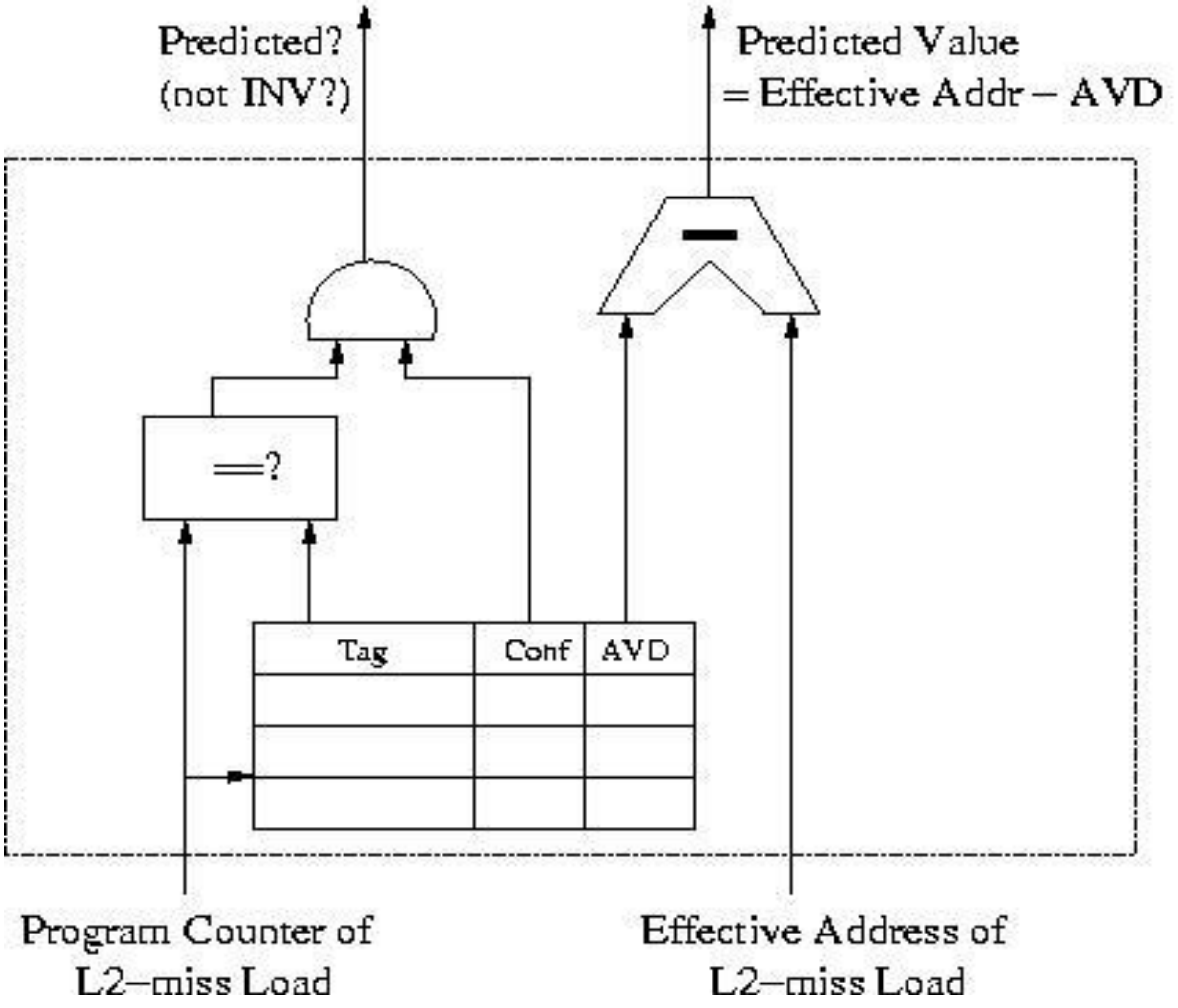
- Set-associative prediction table
- Prediction table entry consists of
  - Tag (Program Counter of the load)
  - Last AVD seen for the load
  - Confidence counter for the recorded AVD

- Updated when an address load is retired in normal mode
- Accessed when a load misses in L2 cache in runahead mode
- Recovery-free: No need to recover the state of the processor or the predictor on misprediction
  - Runahead mode is purely speculative

AVD Update Logic



AVD Prediction Logic

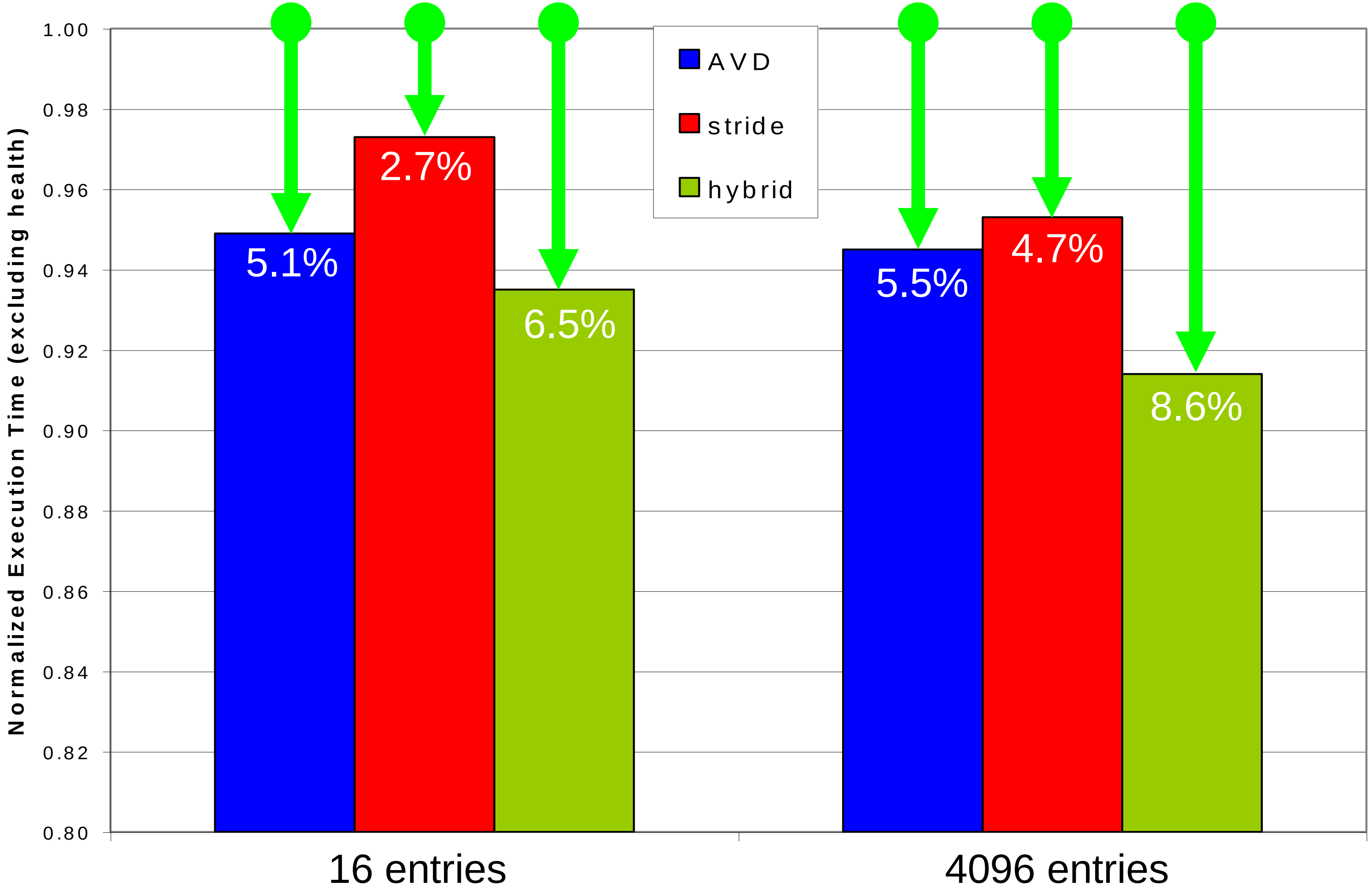


Baseline Processor

- Execution-driven Alpha simulator
- 8-wide superscalar processor
- 128-entry instruction window, 20-stage pipeline
- 64 KB, 4-way, 2-cycle L1 data and instruction caches
- 1 MB, 32-way, 10-cycle unified L2 cache
- 500-cycle minimum main memory latency
- 32 DRAM banks, 32-byte wide processor-memory bus (4:1 frequency ratio), 128 outstanding misses
    - Detailed memory model

- Pointer-intensive benchmarks from Olden and SPEC INT00

AVD vs. Stride VP Performance



Wrong Path Events

An Observation and A Question

- In an out-of-order processor, some instructions are executed on the mispredicted path (wrong-path instructions).

- Is the behavior of wrong-path instructions different from the behavior of correct-path instructions?
  - If so, we can use the difference in behavior for early misprediction detection and recovery.

What is a Wrong Path Event?

An instance of illegal or unusual behavior that is more likely to occur on the wrong path than on the correct path.

Wrong Path Event = WPE

Probability (wrong path | WPE) ~ 1

Why Does a WPE Occur?

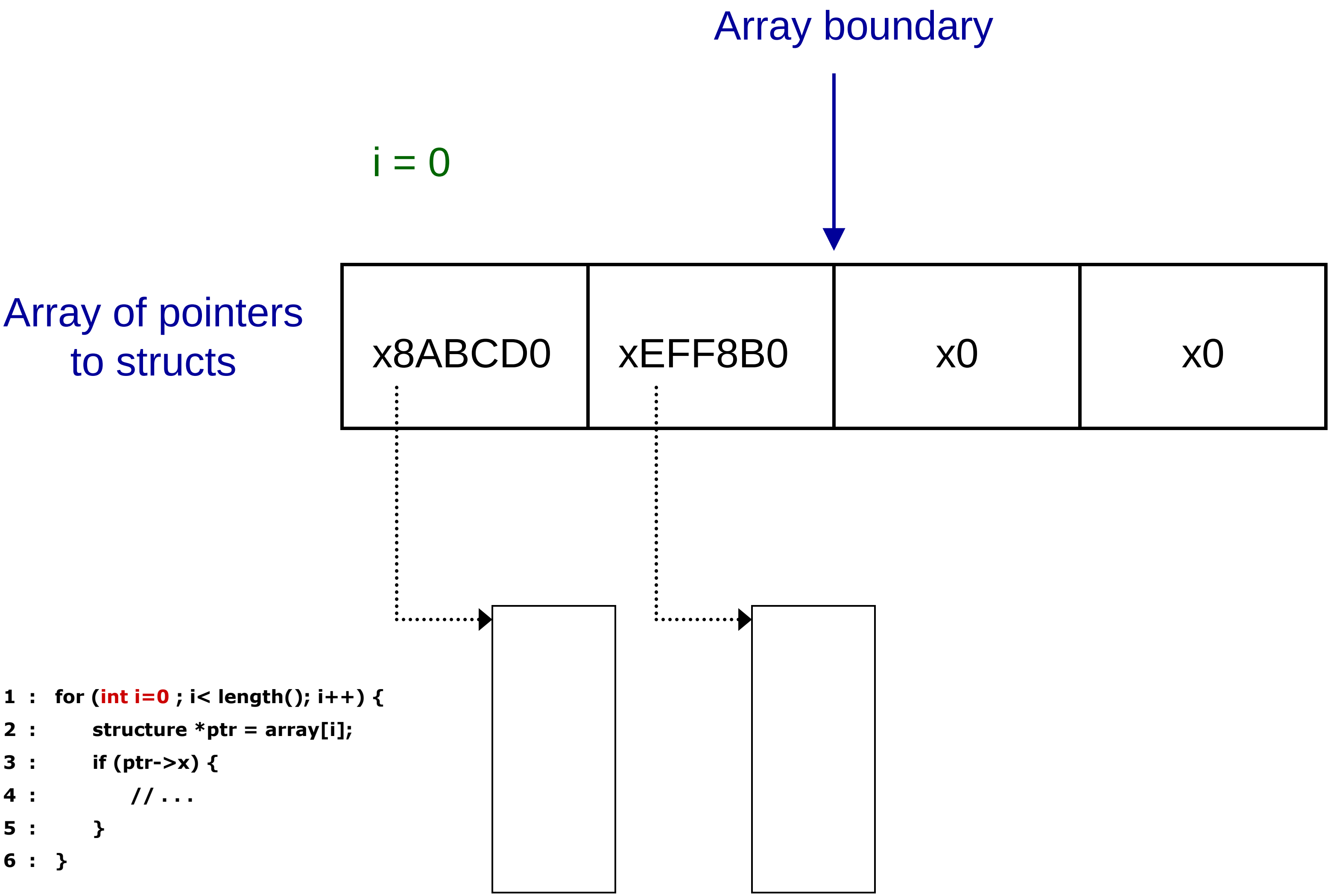
- A wrong-path instruction may be executed *before* the mispredicted branch is executed.
  - Because the mispredicted branch may be dependent on a long-latency instruction.

- The wrong-path instruction may consume a data value that is not properly initialized.

WPE Example from *eon*: NULL pointer dereference

```
for (int i=0 ; i< length(); i++) {
    structure *ptr = array[i];
    if (ptr->x) {
        // . . .
    }
}
```

Beginning of the loop



Array boundary

i = 0

Array of pointers
to structs

x8ABCD0	xEFF8B0	x0	x0

```
for (int i=0 ; i< length(); i++) {
    structure *ptr = array[i];
    if (ptr->x) {
        // . . .
    }
}
```

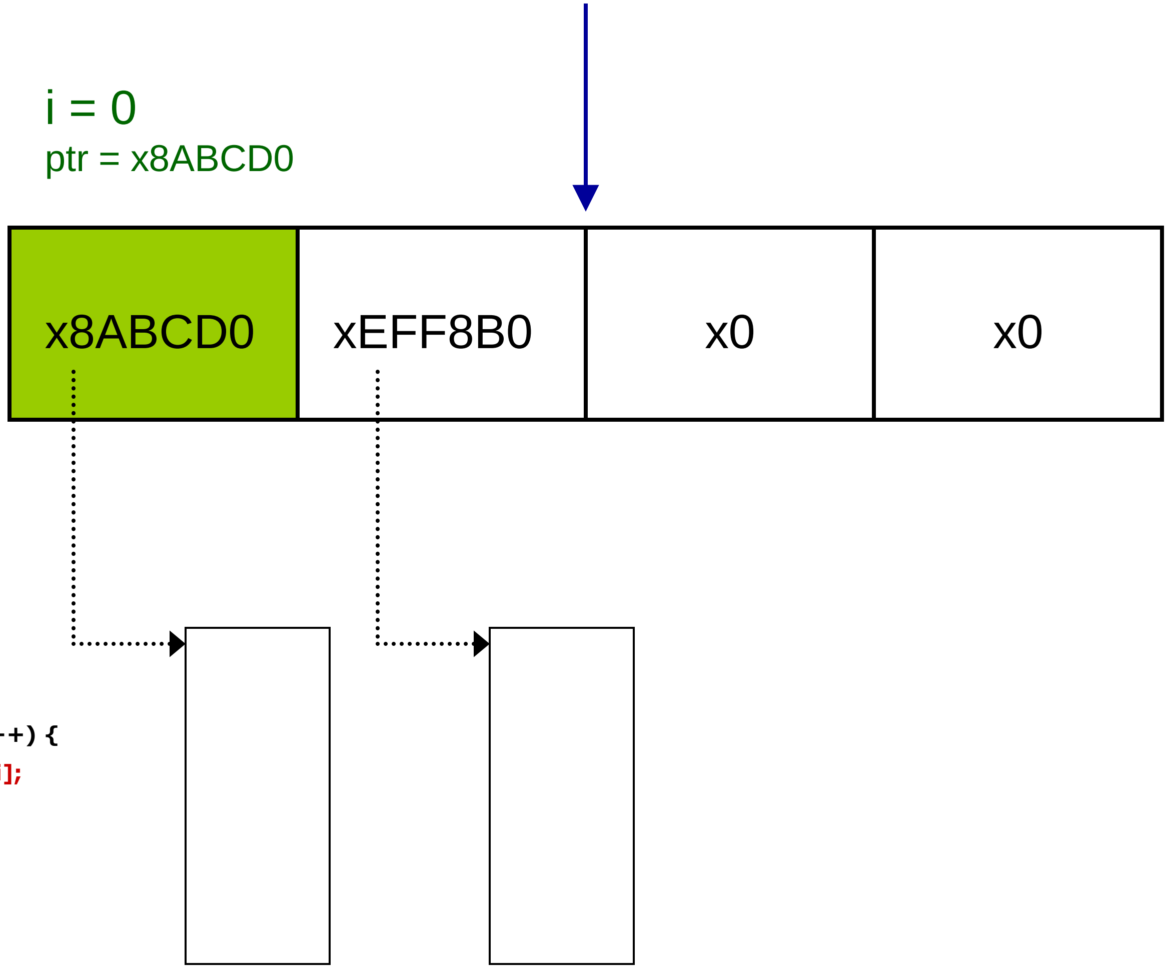
First iteration

Array boundary

i = 0

ptr = x8ABCD0

Array of pointers to structs



x8ABCD0

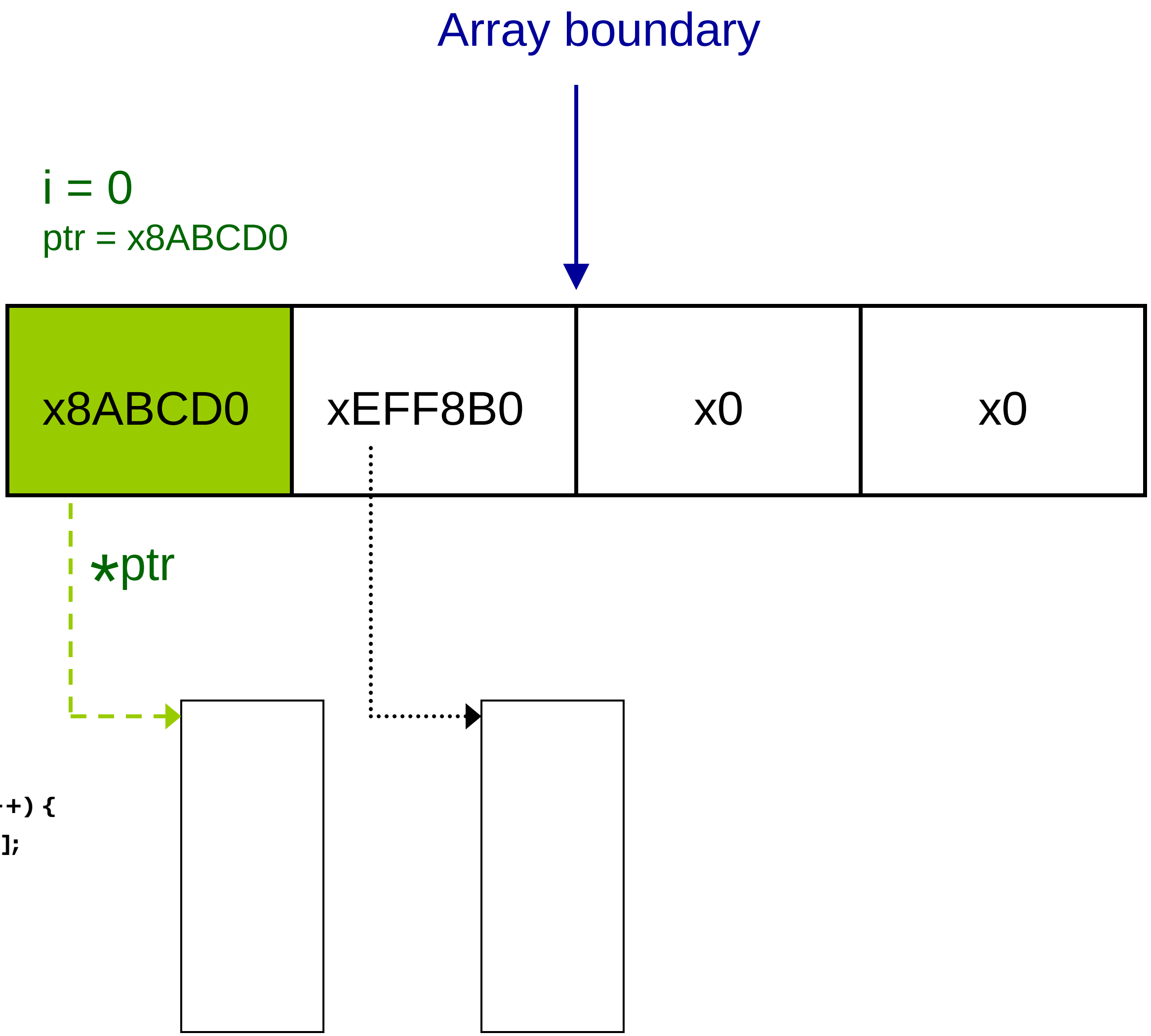
xEFF8B0

x0

x0

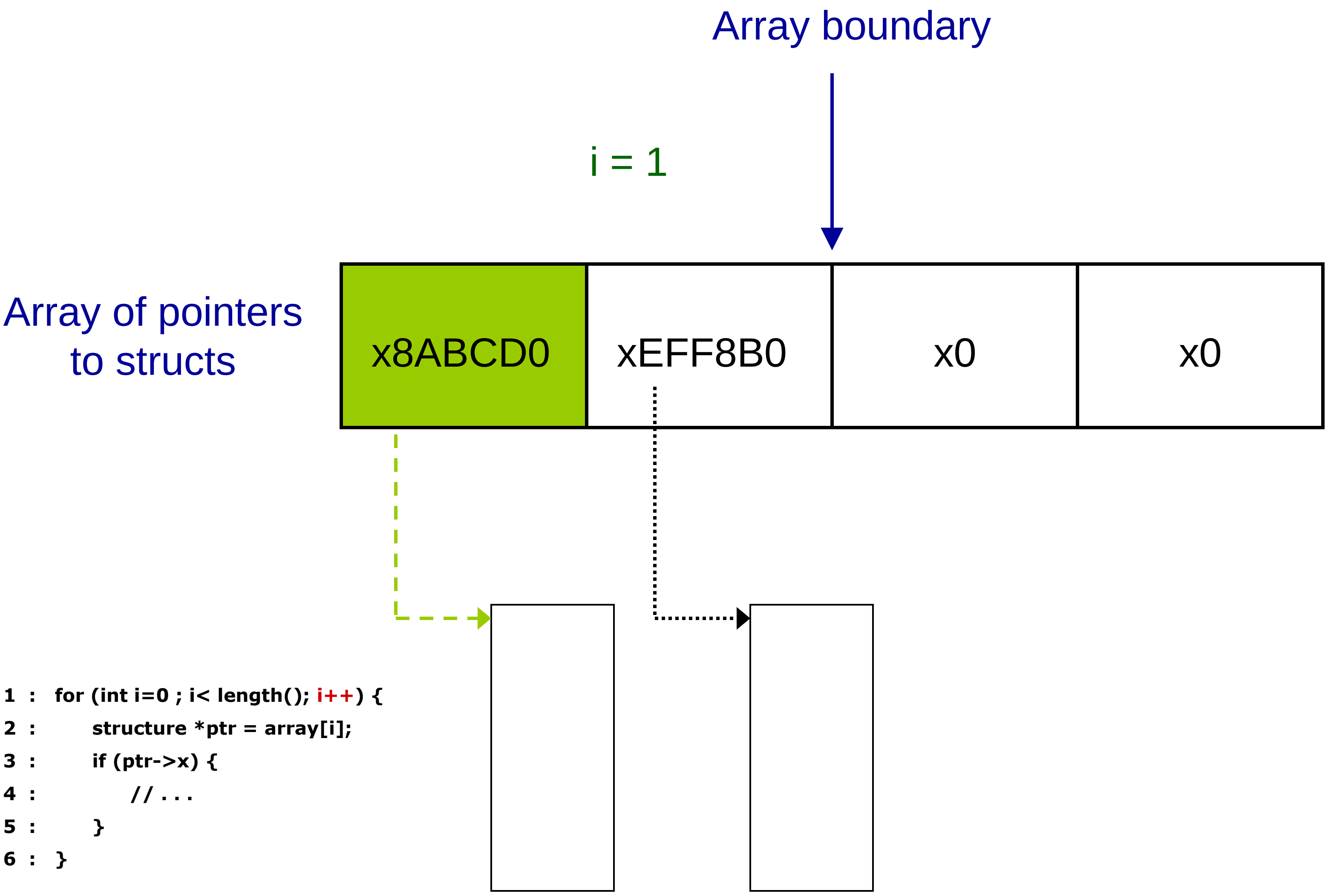
```
for (int i=0 ; i< length(); i++) {
    structure *ptr = array[i];
    if (ptr->x) {
        //...
    }
}
```

First iteration



```
for (int i=0 ; i< length(); i++) {
    structure *ptr = array[i];
    if (ptr->x) {
        //...
    }
}
```

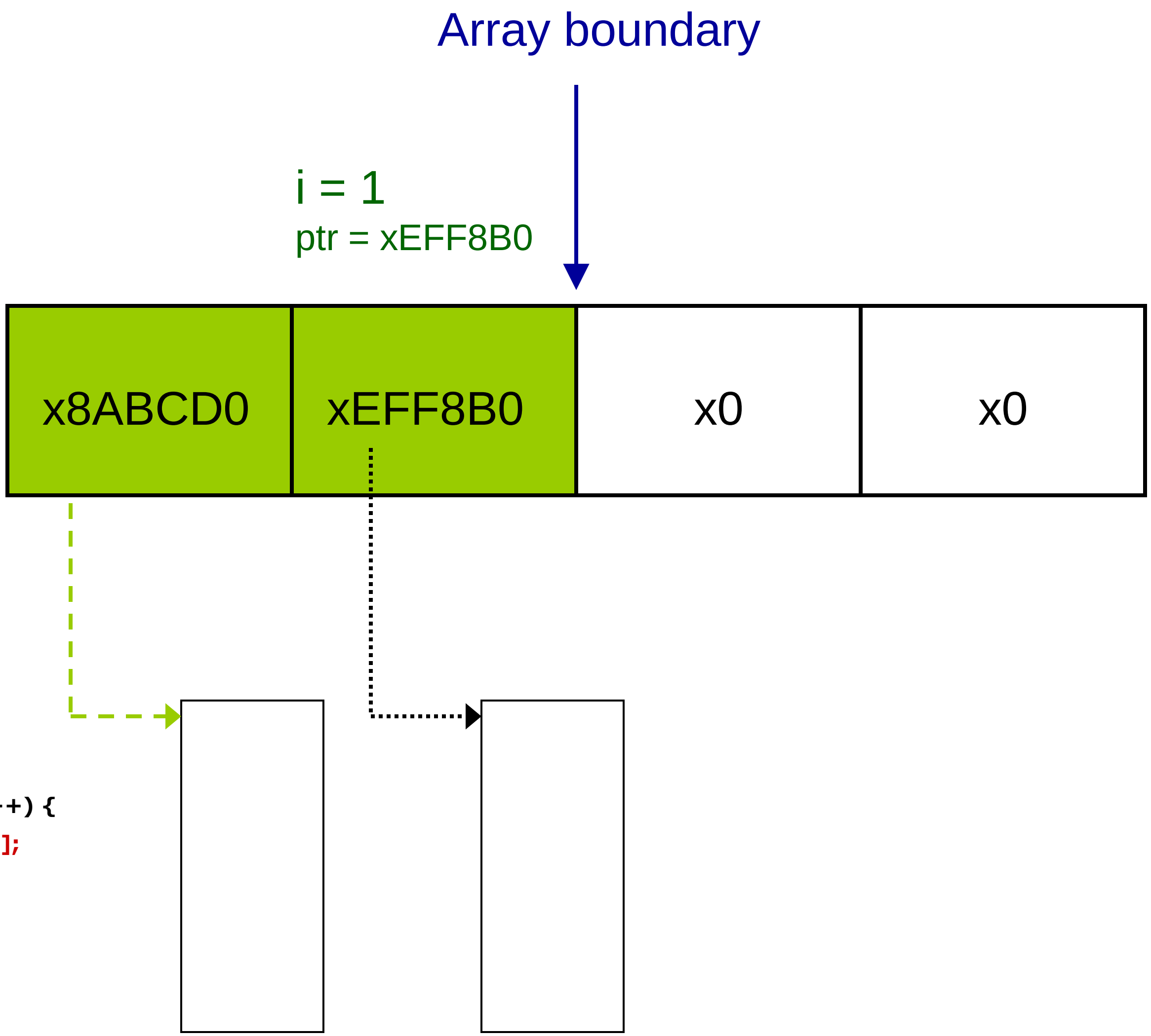
Loop branch correctly predicted



```
for (int i=0 ; i< length(); i++) {
      structure *ptr = array[i];
      if (ptr->x) {
            // . . .
      }
}
```

Second iteration

Array of pointers to structs



Array boundary

i = 1

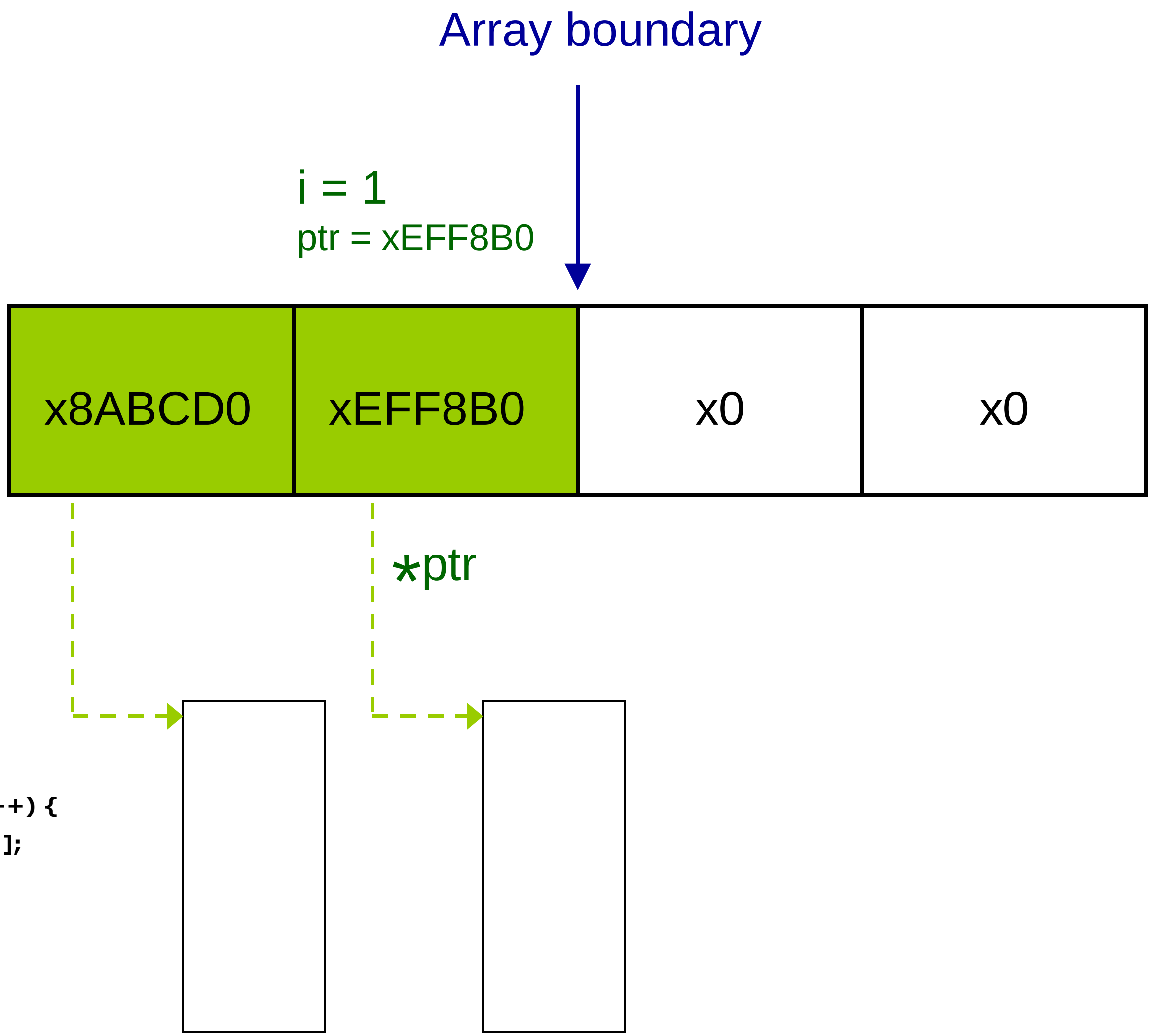
ptr = xEFF8B0

x8ABCD0	xEFF8B0	x0	x0

```
for (int i=0 ; i< length(); i++) {
    structure *ptr = array[i];
    if (ptr->x) {
        //...
    }
}
```

Second iteration

Array of pointers to structs



Array boundary

i = 1

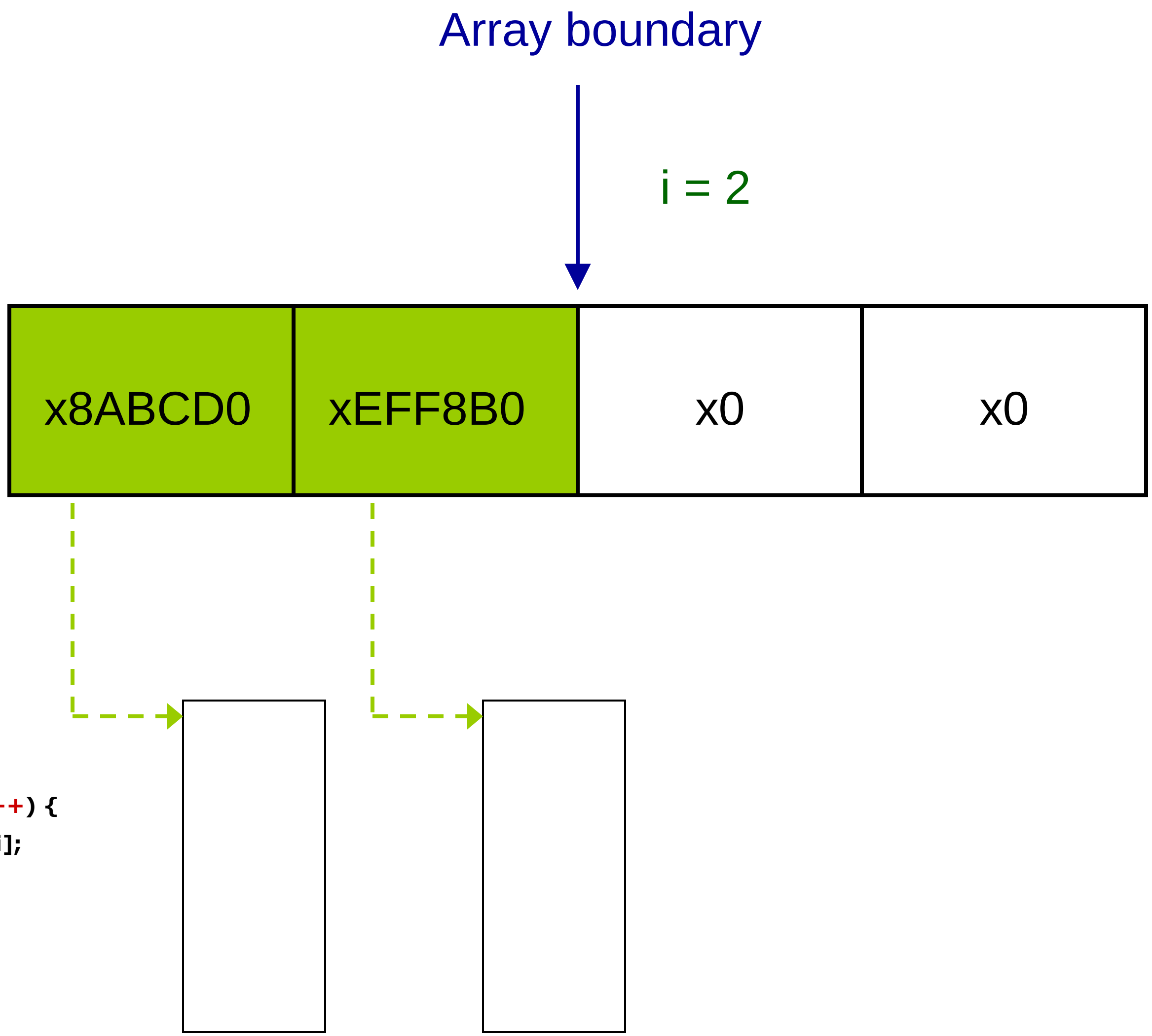
ptr = xEFF8B0

x8ABCD0 | xEFF8B0 | x0 | x0

*ptr

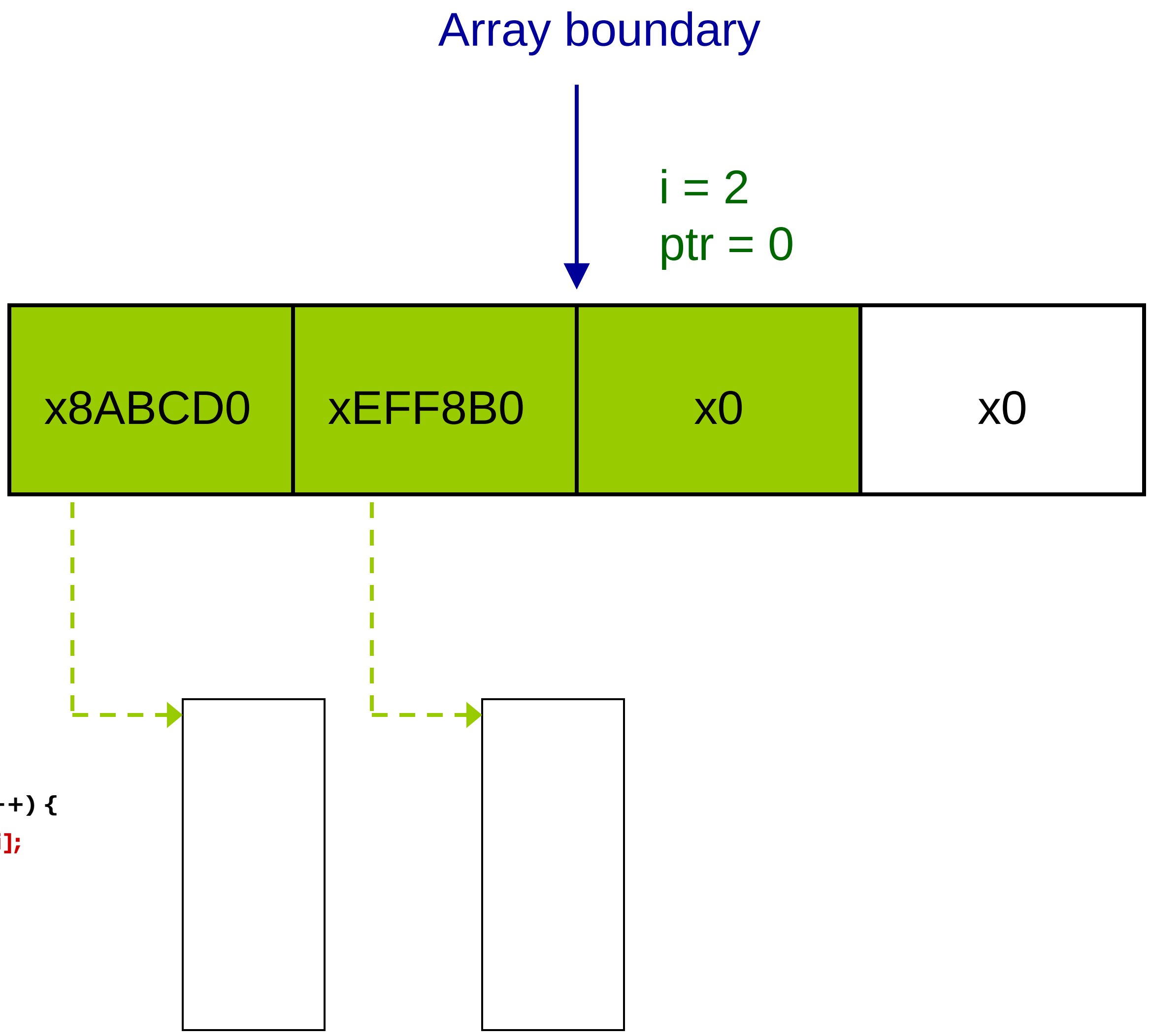
```
for (int i=0 ; i< length(); i++) {
    structure *ptr = array[i];
    if (ptr->x) {
        //...
    }
}
```

Loop exit branch mispredicted



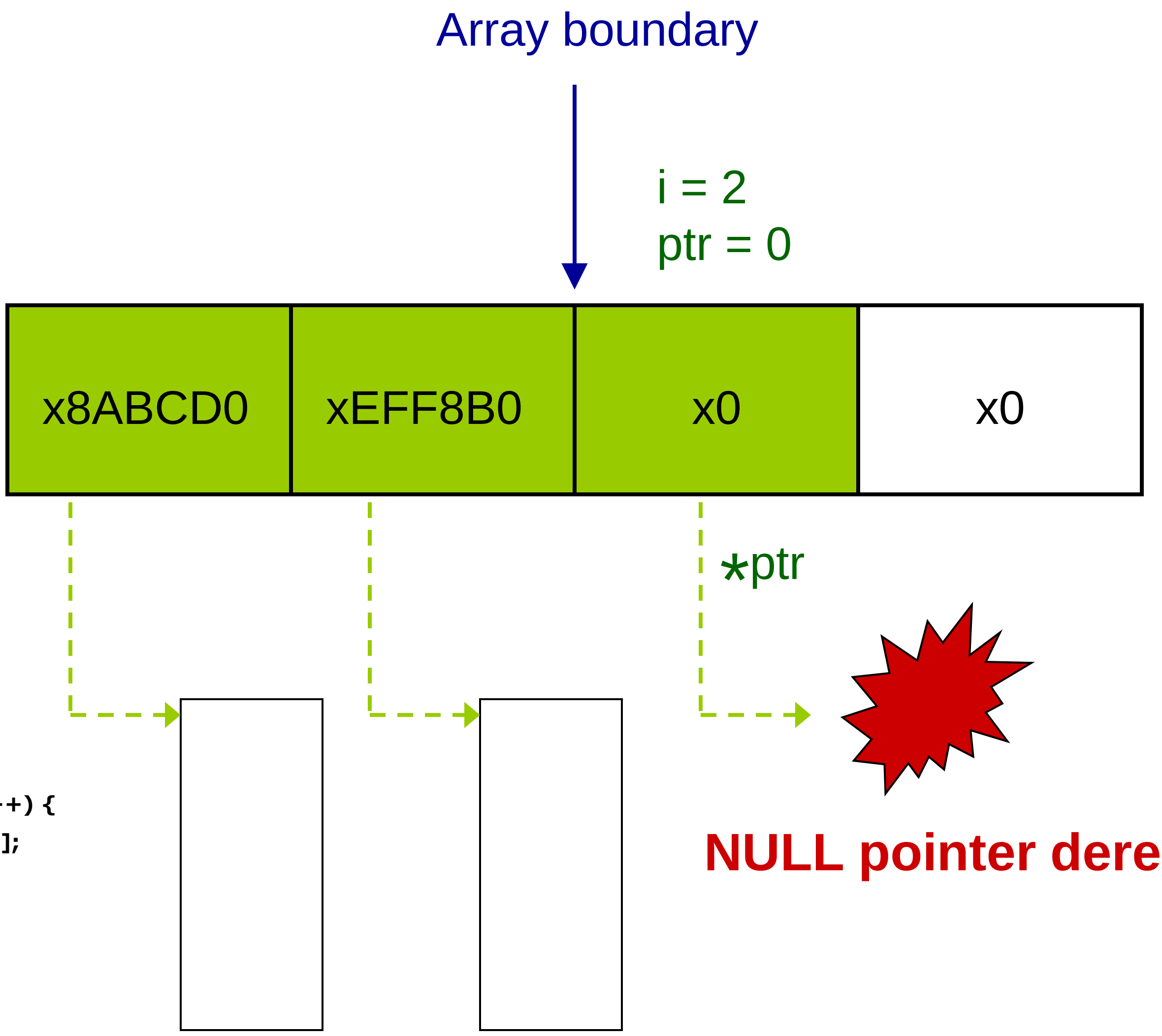
```
for (int i=0 ; i< length(); i++) {
    structure *ptr = array[i];
    if (ptr->x) {
        // . . .
    }
}
```

Third iteration on wrong path



```
for (int i=0 ; i< length(); i++) {
    structure *ptr = array[i];
    if (ptr->x) {
        // . . .
    }
}
```

Wrong Path Event



Array of pointers to structs

```
for (int i=0 ; i< length(); i++) {
    structure *ptr = array[i];
    if (ptr->x) {
        // . . .
    }
}
```

NULL pointer dereference!

Types of WPEs

- Due to memory instructions
  - NULL pointer dereference
  - Write to read-only page
  - Unaligned access (illegal in the Alpha ISA)
  - Access to an address out of segment range
  - Data access to code segment
  - Multiple concurrent TLB misses

Types of WPEs (continued)

- Due to control-flow instructions
  - Misprediction under misprediction
    - If three branches are executed and resolved as mispredicts while there are older unresolved branches in the processor, it is almost certain that one of the older unresolved branches is mispredicted.
  - Return address stack underflow
  - Unaligned instruction fetch address (illegal in Alpha)

- Due to arithmetic instructions
  - Some arithmetic exceptions
    - e.g. Divide by zero

Two Empirical Questions

1. How often do WPEs occur?

2. When do WPEs occur on the wrong path?