

Computer Architecture

Lecture 17:

Latency Tolerance and Prefetching

Prof. Onur Mutlu

ETH Zürich

Fall 2017

22 November 2017

Summary of Last Week's Lectures

- Shared Cache Management
- Making Caching More Effective
- Heterogeneous Multi-Core Systems
- Bottleneck Acceleration

Today

- Quick Heterogeneous Systems Wrap-Up
- Memory Latency Tolerance
- Prefetching

Asymmetry via Frequency Boosting

Recall: How to Achieve Asymmetry

■ Static

- Type and power of cores fixed at design time
- Two approaches to design “faster cores”:
 - High frequency
 - Build a more complex, powerful core with entirely different uarch
- Is static asymmetry natural? (chip-wide variations in frequency)

■ Dynamic

- Type and power of cores change dynamically
- Two approaches to dynamically create “faster cores”:
 - Boost frequency dynamically (limited power budget)
 - Combine small cores to enable a more complex, powerful core
 - Is there a third, fourth, fifth approach?

Asymmetry via Boosting of Frequency

■ Static

- Due to process variations, cores might have different frequency
- Simply hardwire/design cores to have different frequencies

■ Dynamic

- Annavaram et al., “[Mitigating Amdahl's Law Through EPI Throttling](#),” ISCA 2005.
- Dynamic voltage and frequency scaling

EPI Throttling

- Goal: Minimize execution time of parallel programs while keeping power within a fixed budget
- For best scalar and throughput performance, vary energy expended per instruction (EPI) based on available parallelism
 - $P = \text{EPI} \bullet \text{IPS}$
 - P = fixed power budget
 - EPI = energy per instruction
 - IPS = aggregate instructions retired per second
- Idea: For a fixed power budget
 - Run sequential phases on high-EPI processor
 - Run parallel phases on multiple low-EPI processors

EPI Throttling via DVFS

- DVFS: Dynamic voltage frequency scaling
- In phases of low thread parallelism
 - Run a few cores at high supply voltage and high frequency
- In phases of high thread parallelism
 - Run many cores at low supply voltage and low frequency

Possible EPI Throttling Techniques

- Grochowski et al., “Best of both Latency and Throughput,” ICCD 2004.

Method	EPI Range	Time to Alter EPI	Throttle Action
Voltage/frequency scaling	1:2 to 1:4	100us (ramp Vcc)	Lower voltage and frequency
Asymmetric cores	1:4 to 1:6	10us (migrate 256KB L2 cache)	Migrate threads from large cores to small cores
Variable-size core	1:1 to 1:2	1us (fill 32KB L1 cache)	Reduce capacity of processor resources
Speculation control	1:1 to 1:1.4	10ns (pipeline latency)	Reduce amount of speculation

Boosting Frequency of a Small Core vs. Large Core

- Frequency boosting implemented on Intel Nehalem, IBM POWER7
- Advantages of Boosting Frequency
 - + Very simple to implement; no need to design a new core
 - + Parallel throughput does not degrade when TLP is high
 - + Preserves locality of boosted thread
- Disadvantages
 - Does not improve performance if thread is memory bound
 - Does not reduce Cycles per Instruction (remember the performance equation?)
 - Changing frequency/voltage can take longer than switching to a large core

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

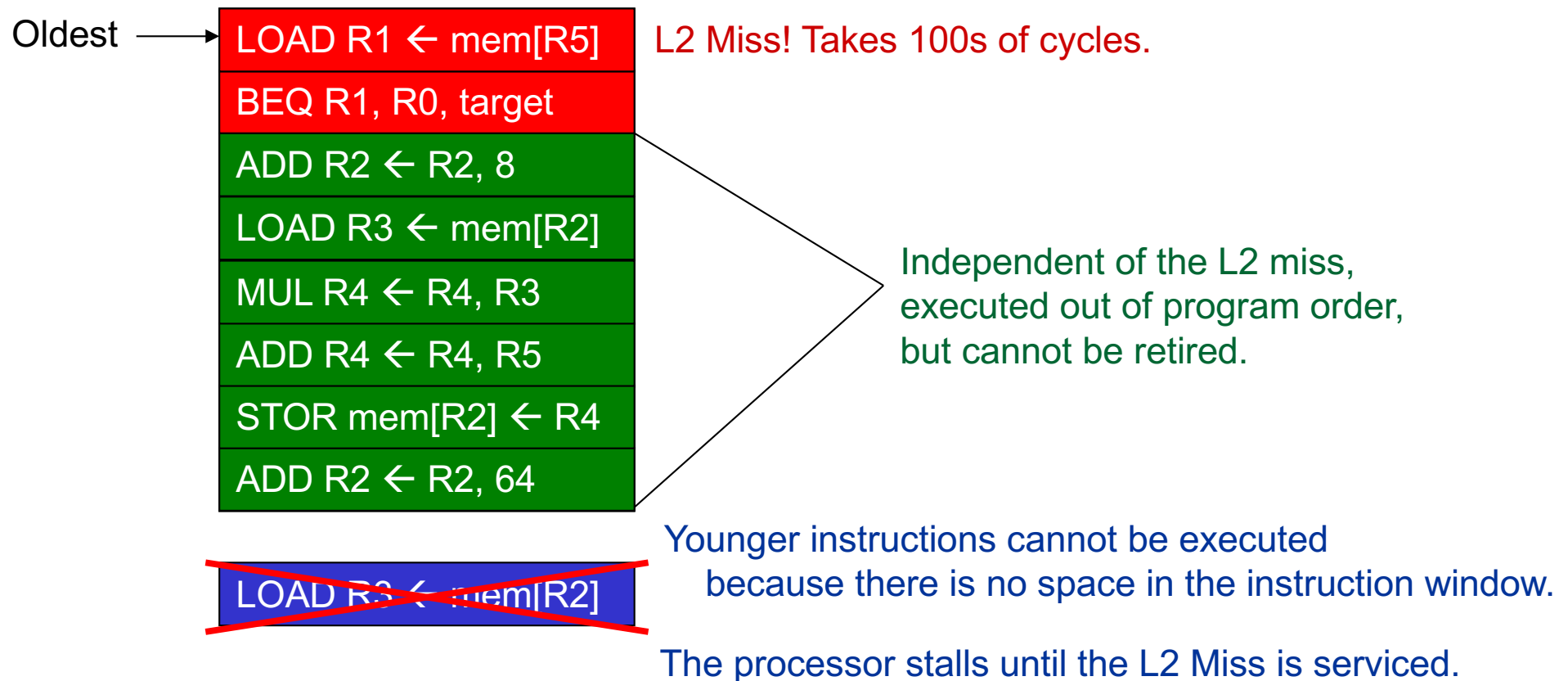
- An out-of-order execution processor tolerates latency of multi-cycle operations by executing independent instructions concurrently
 - It does so by buffering instructions in reservation stations and reorder buffer
 - Instruction window: Hardware resources needed to buffer all decoded but not yet retired/committed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

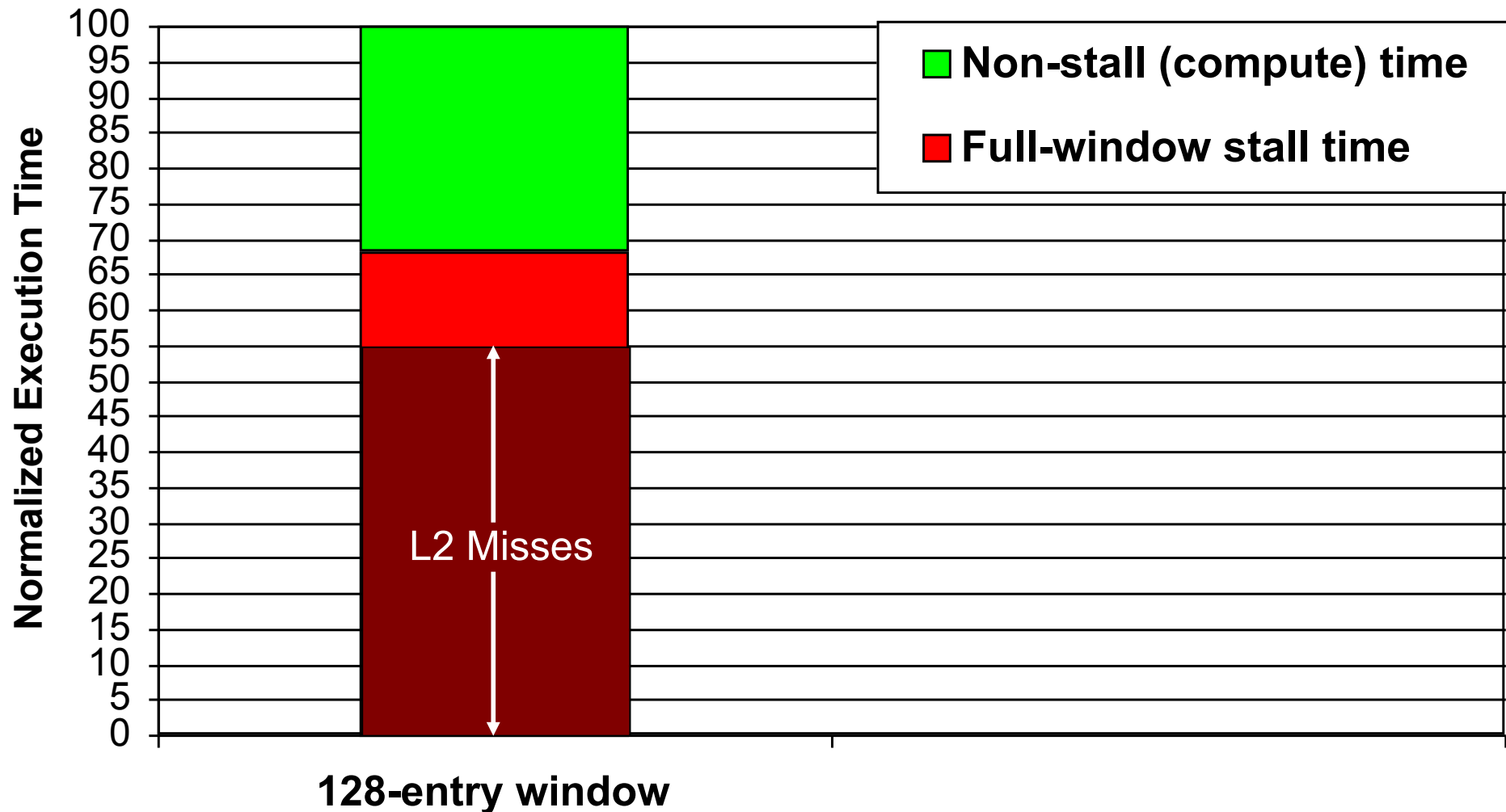
Full-window Stall Example

8-entry instruction window:



- 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 very easy to reduce it...
 - We looked at methods for reducing DRAM latency
 - Lee et al. "**Tiered-Latency DRAM**," HPCA 2013.
 - Lee et al., "**Adaptive-Latency DRAM**," HPCA 2015.
 - ...
- 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

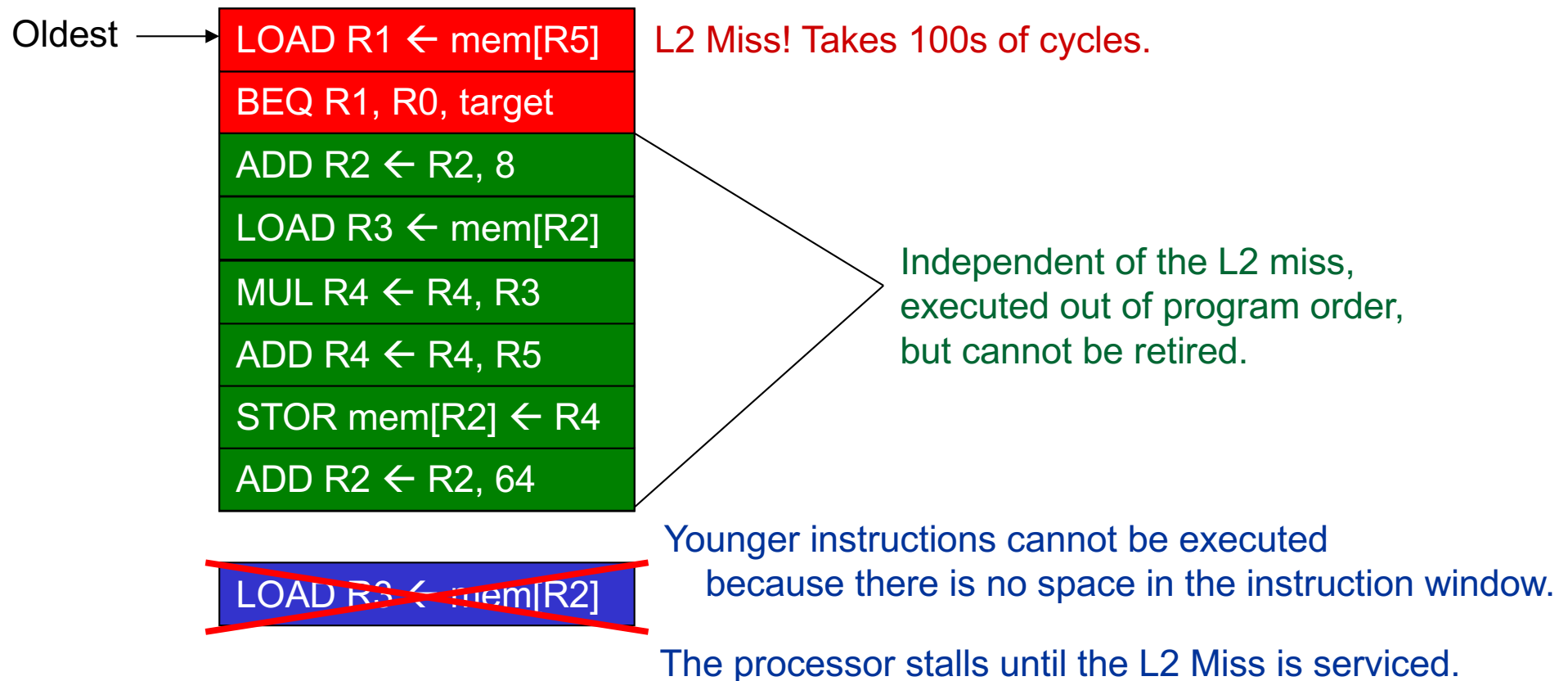
Memory Latency Tolerance Techniques

- **Caching** [initially by Bloom+, 1962 and later 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

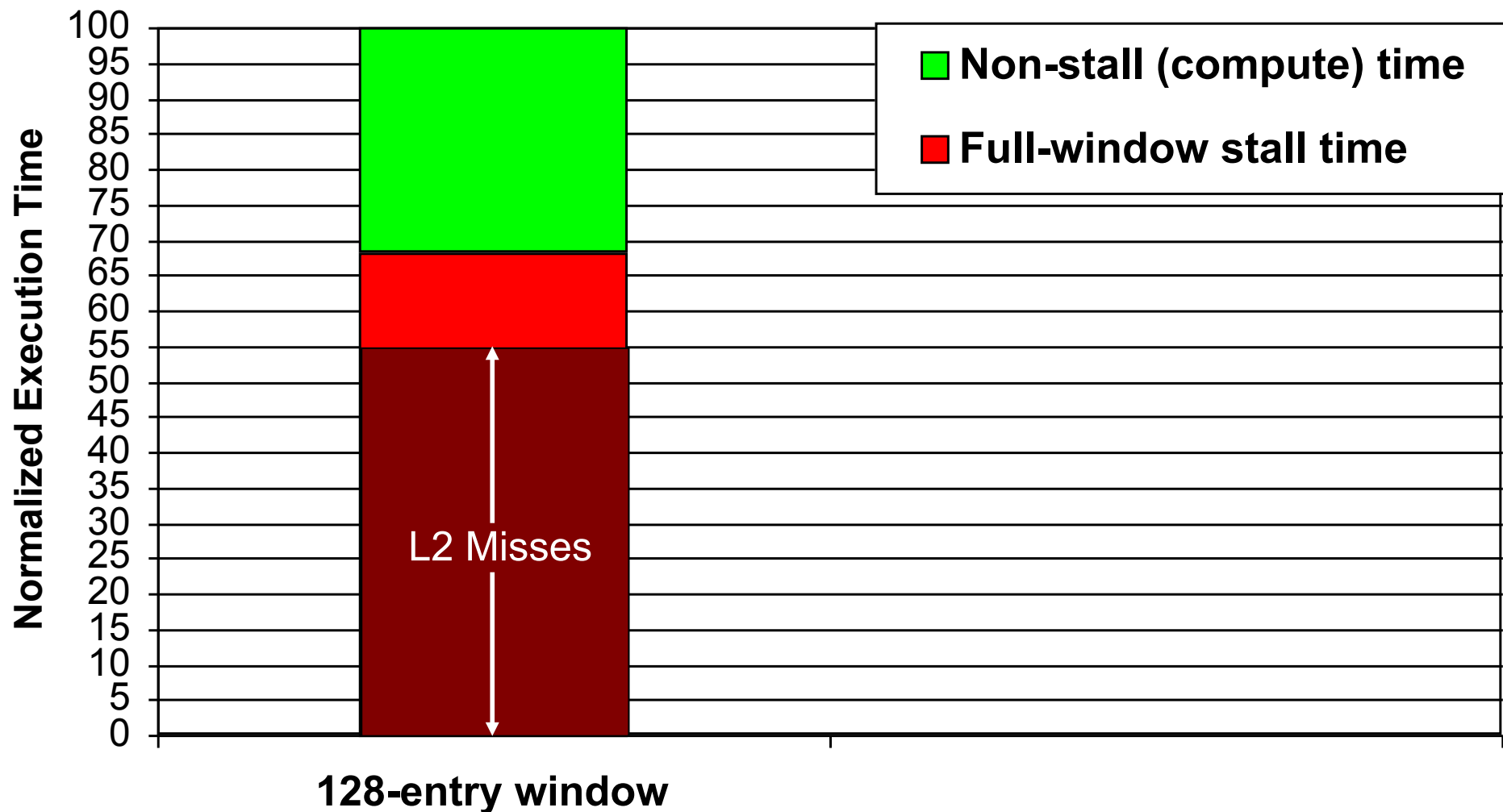
Small Windows: Full-window Stalls

8-entry instruction window:



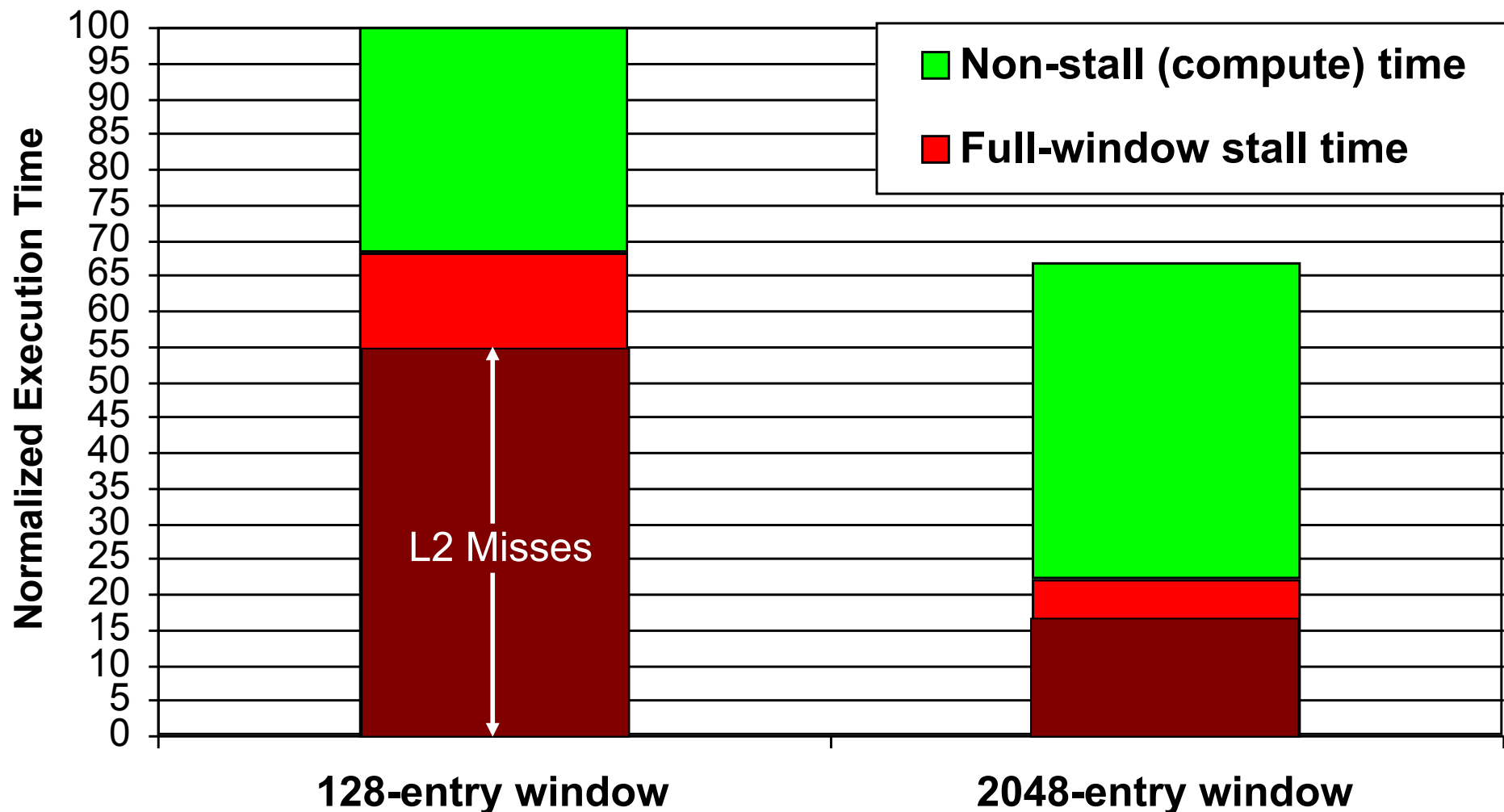
- Long-latency cache misses are responsible for most full-window stalls.

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

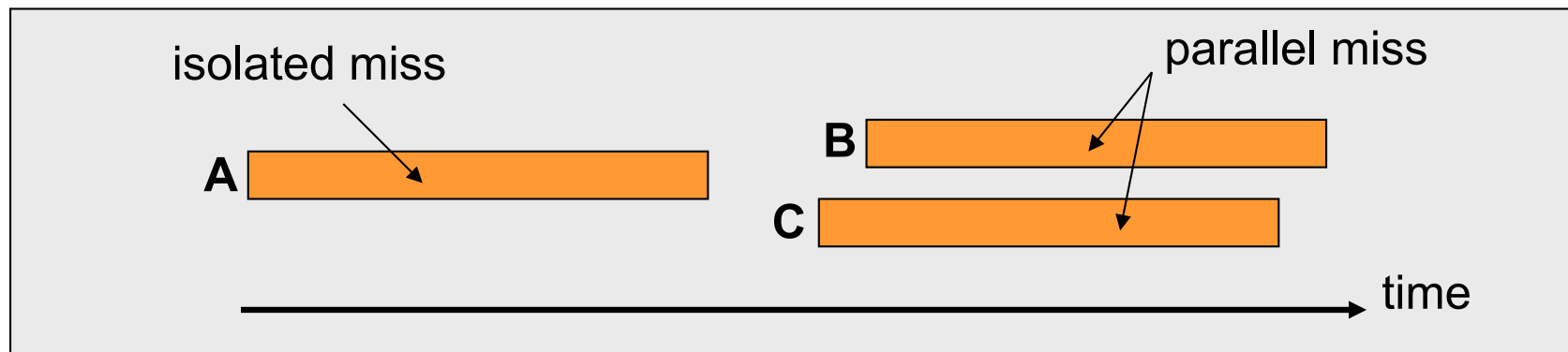
- Out-of-order execution requires large instruction windows to tolerate today's main memory latencies.
- As main memory latency increases, instruction window size should also increase to fully tolerate the memory latency.
- Building a large instruction window is a challenging task if we would like to 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



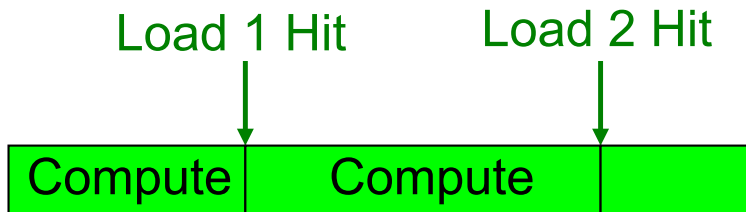
- Enables latency tolerance: **overlaps latency of different misses**
- How to generate multiple misses?
 - Out-of-order execution, multithreading, prefetching, **runahead**

Runahead Execution (I)

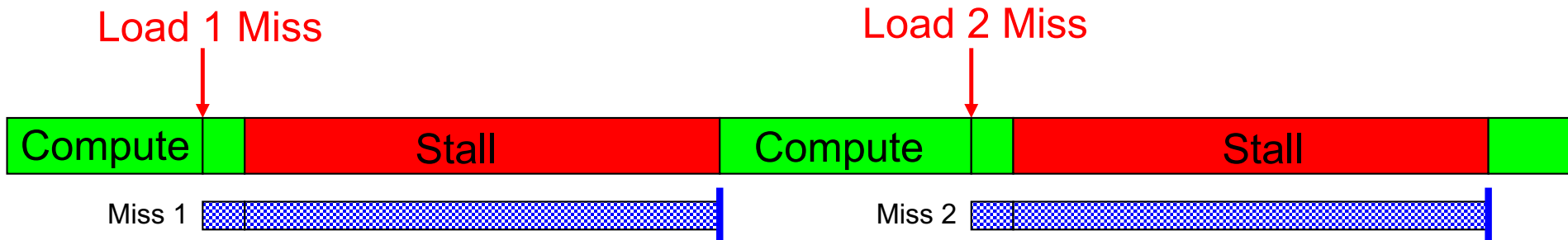
- A technique to obtain the memory-level parallelism benefits of a large instruction window
- When the oldest instruction is a long-latency cache miss:
 - Checkpoint architectural state and enter runahead mode
- In runahead mode:
 - Speculatively pre-execute instructions
 - The purpose of pre-execution 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

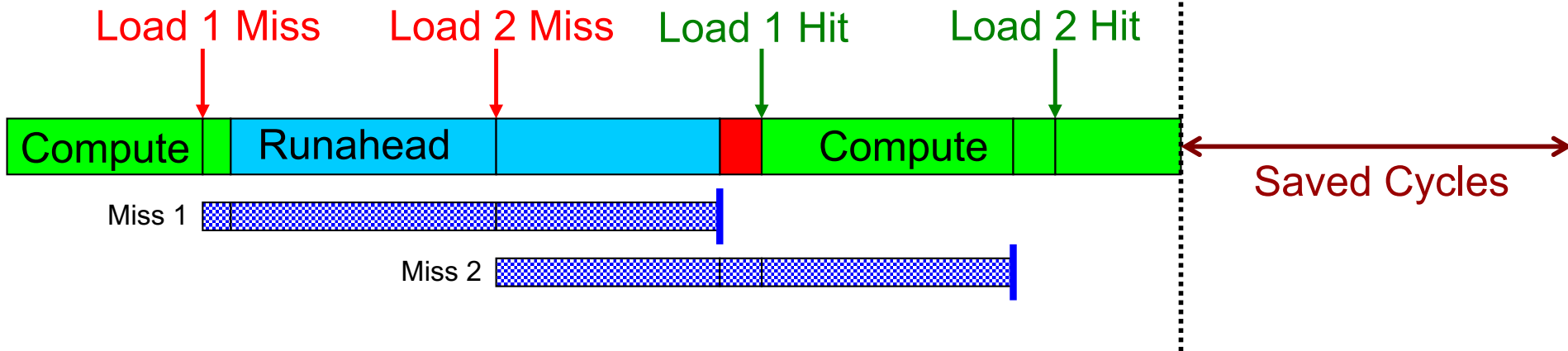
Perfect Caches:



Small Window:



Runahead:



Benefits of Runahead Execution

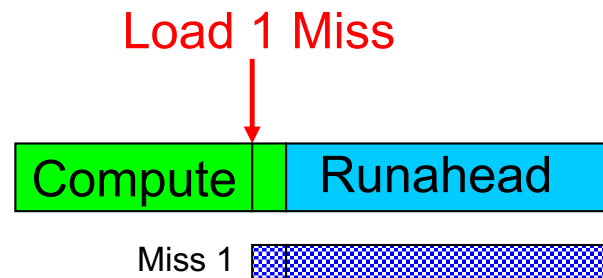
Instead of stalling during an L2 cache miss:

- 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
-

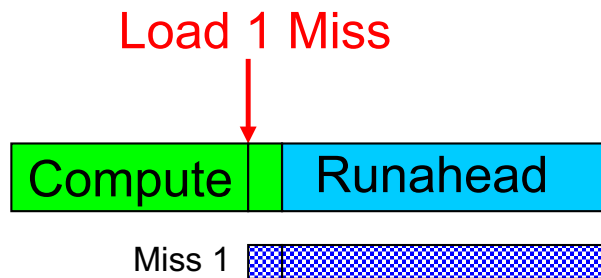
Instruction Processing in Runahead Mode



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

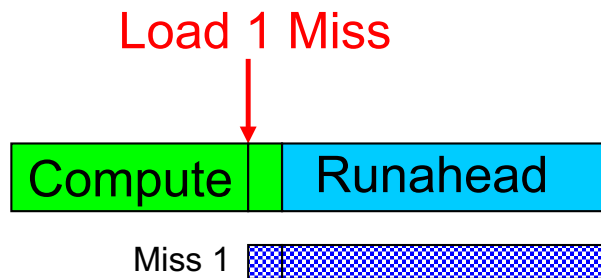
- It is purely speculative: **Architectural (software-visible) register/memory state is NOT updated in runahead mode.**
 - L2-miss dependent instructions are identified and treated specially.
 - They are quickly removed from the instruction window.
 - Their results are not trusted.
-

L2-Miss Dependent Instructions



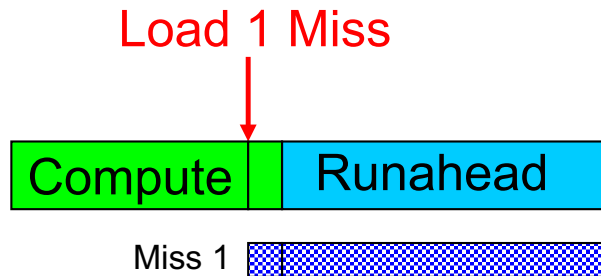
- Two types of results produced: INV and VALID
 - INV = Dependent on an L2 miss
 - INV results are marked using INV bits in the register file and store buffer.
 - INV values are not used for prefetching/branch resolution.
-

Removal of Instructions from Window



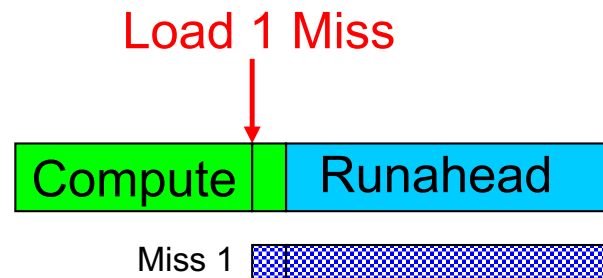
- Oldest instruction is examined for **pseudo-retirement**
 - An INV instruction is removed from window immediately.
 - A VALID instruction is removed when it completes execution.
 - **Pseudo-retired instructions free their allocated resources.**
 - This allows the processing of later instructions.
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

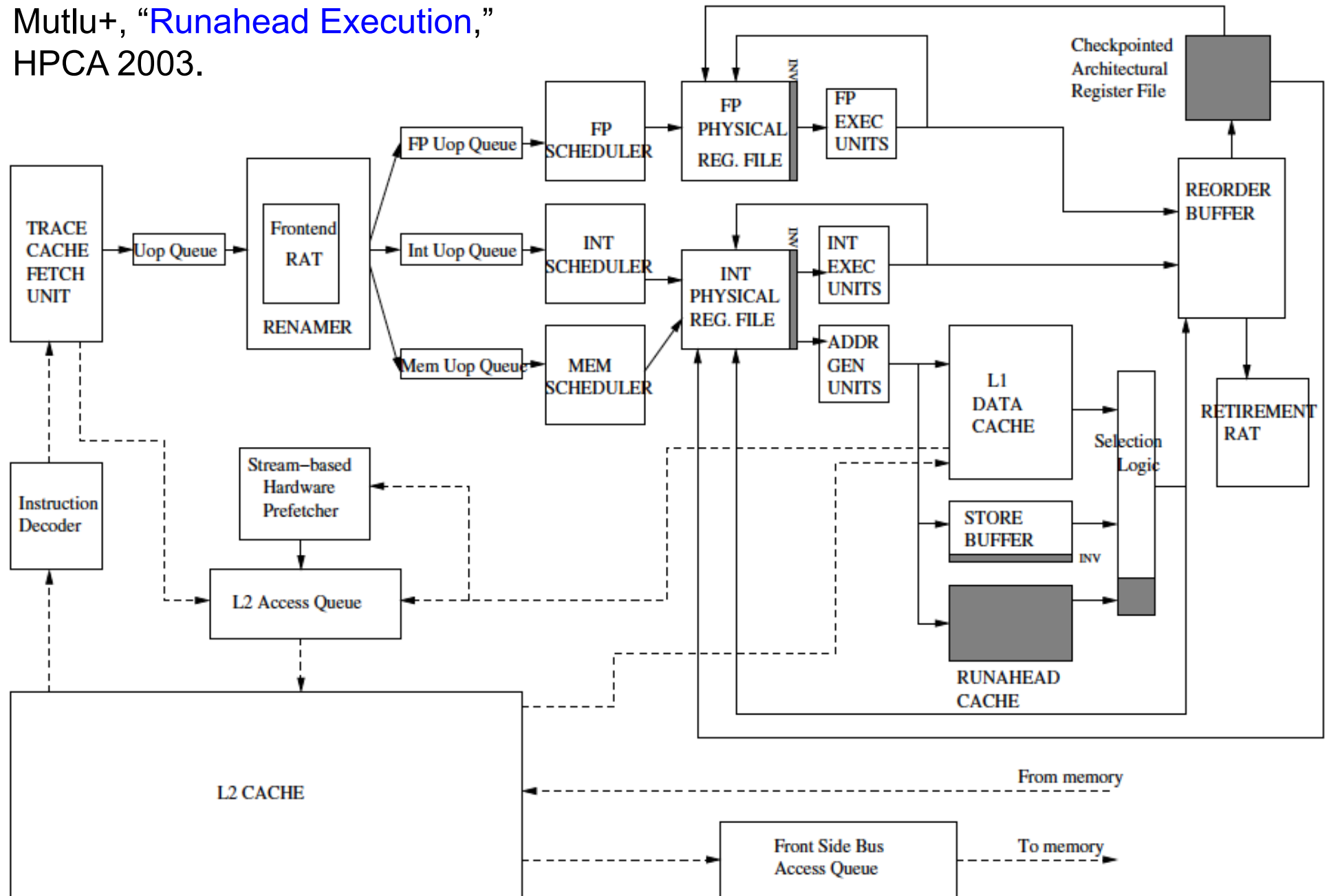
Branch Handling in Runahead Mode



- **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.**

A Runahead Processor Diagram

Mutlu+, “Runahead Execution,”
HPCA 2003.



Runahead Execution Pros and Cons

■ Advantages:

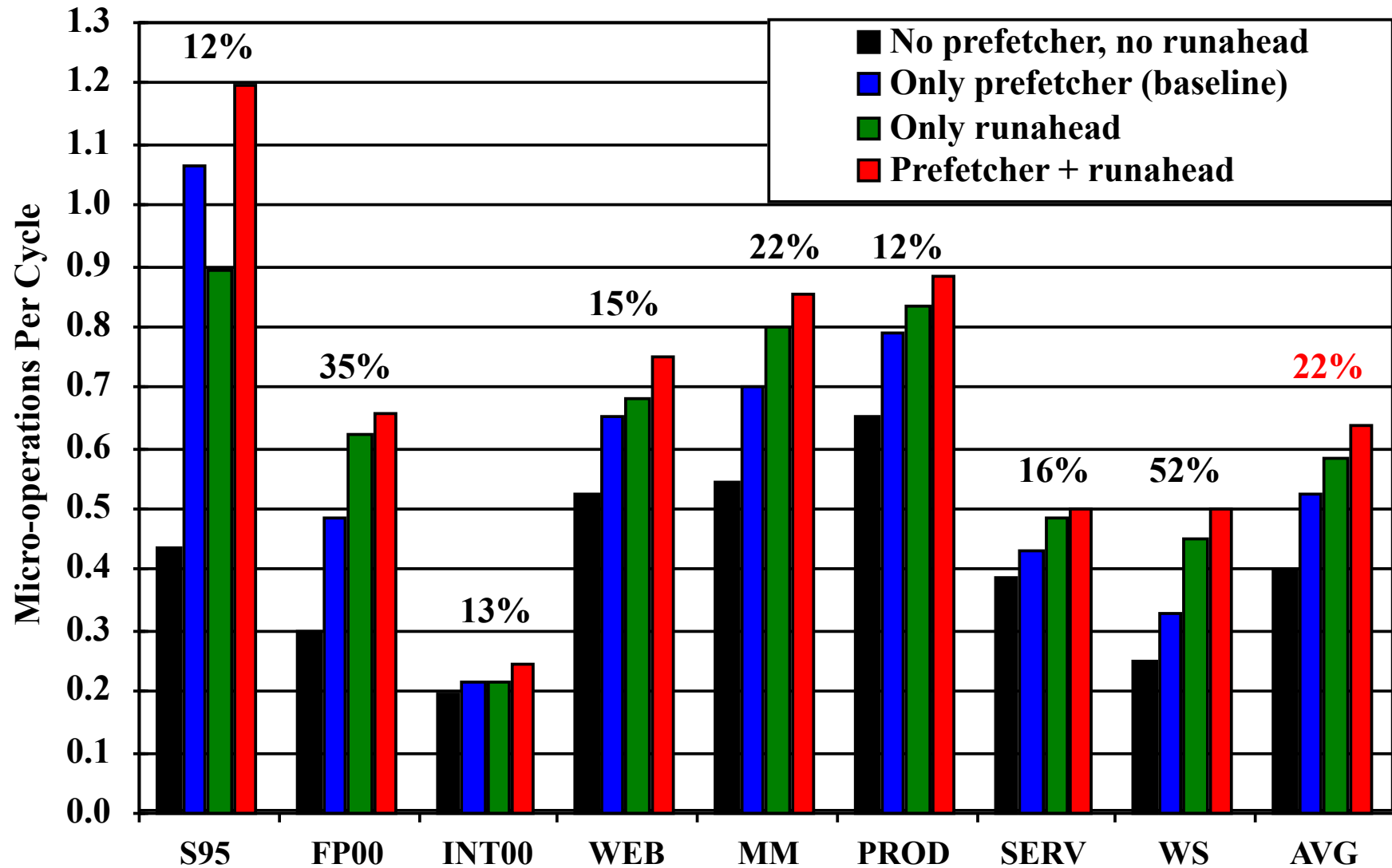
- + Very **accurate** prefetches for data/instructions (all cache levels)
 - + Follows the program path
- + **Simple to implement**, most of the hardware is already built in
- + Versus other pre-execution based prefetching mechanisms (as we will see):
 - + Uses the same thread context as main thread,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

■ Disadvantages/Limitations:

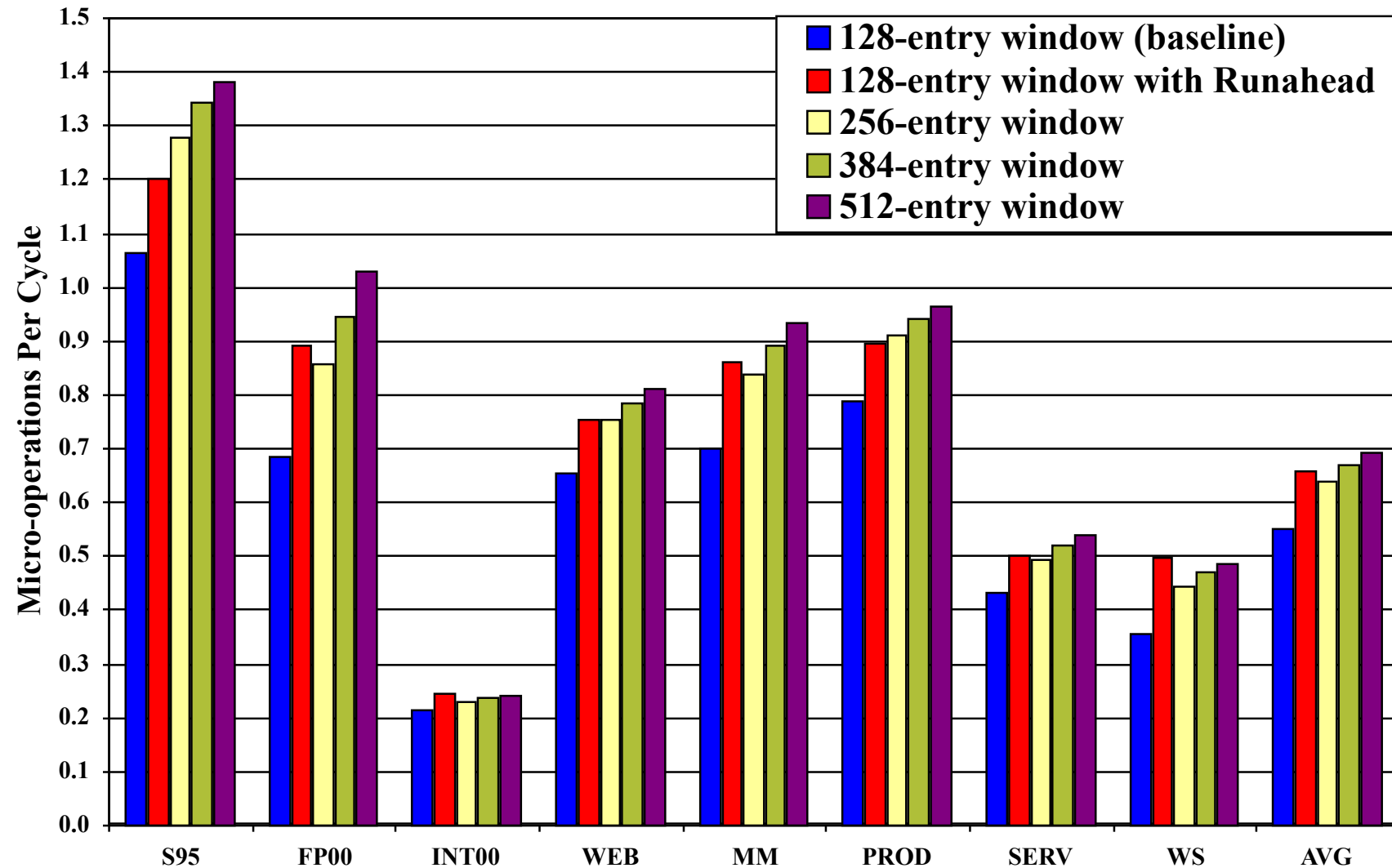
- **Extra executed instructions**
- Limited by branch prediction accuracy
- Cannot prefetch dependent cache misses
- **Effectiveness limited by available “memory-level parallelism” (MLP)**
- **Prefetch distance (how far ahead to prefetch) limited by memory latency**

■ Implemented in IBM POWER6, Sun “Rock”

Performance of Runahead Execution



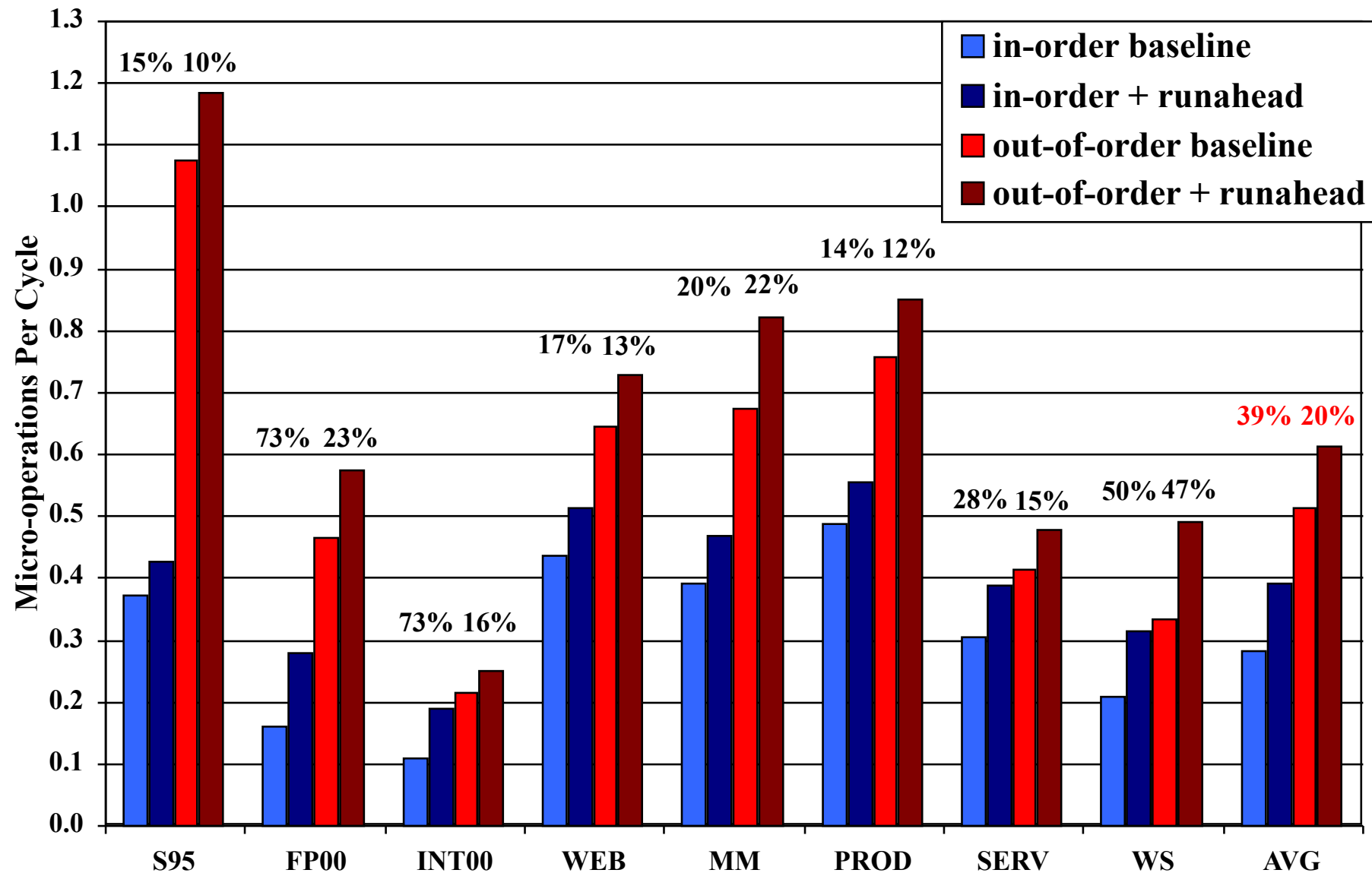
Runahead Execution vs. Large Windows



Runahead vs. A (Real) Large Window

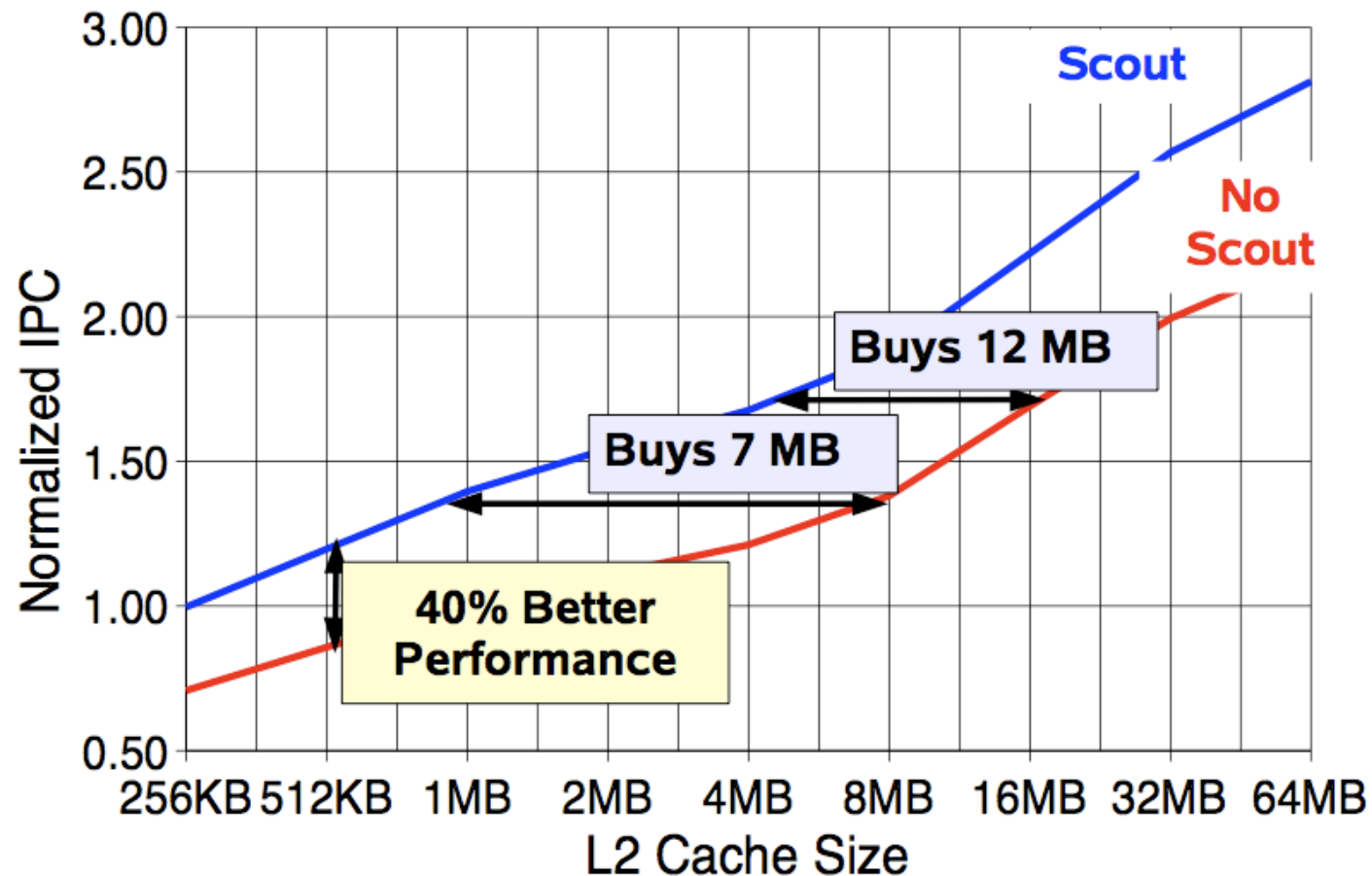
- When is one beneficial, when is the other?
- Pros and cons of each
- Which can tolerate floating-point operation latencies better?
- Which leads to less wasted execution?

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



Effect of Runahead in Sun ROCK

- Shailender Chaudhry talk, Aug 2008.



Generalizing the Idea

- Runahead on different long-latency operations?

More on Runahead Execution

- Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt,
"Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors"
Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA), pages 129-140, Anaheim, CA, February 2003. [Slides \(pdf\)](#)

Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors

Onur Mutlu § Jared Stark † Chris Wilkerson ‡ Yale N. Patt §

§ECE Department
The University of Texas at Austin
{onur,patt}@ece.utexas.edu

†Microprocessor Research
Intel Labs
jared.w.stark@intel.com

‡Desktop Platforms Group
Intel Corporation
chris.wilkerson@intel.com

More on Runahead Execution (Short)

- Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt,
"Runahead Execution: An Effective Alternative to Large Instruction Windows"
IEEE Micro, Special Issue: Micro's Top Picks from Microarchitecture Conferences (MICRO TOP PICKS), Vol. 23, No. 6, pages 20-25, November/December 2003.

RUNAHEAD EXECUTION: AN EFFECTIVE ALTERNATIVE TO LARGE INSTRUCTION WINDOWS

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]

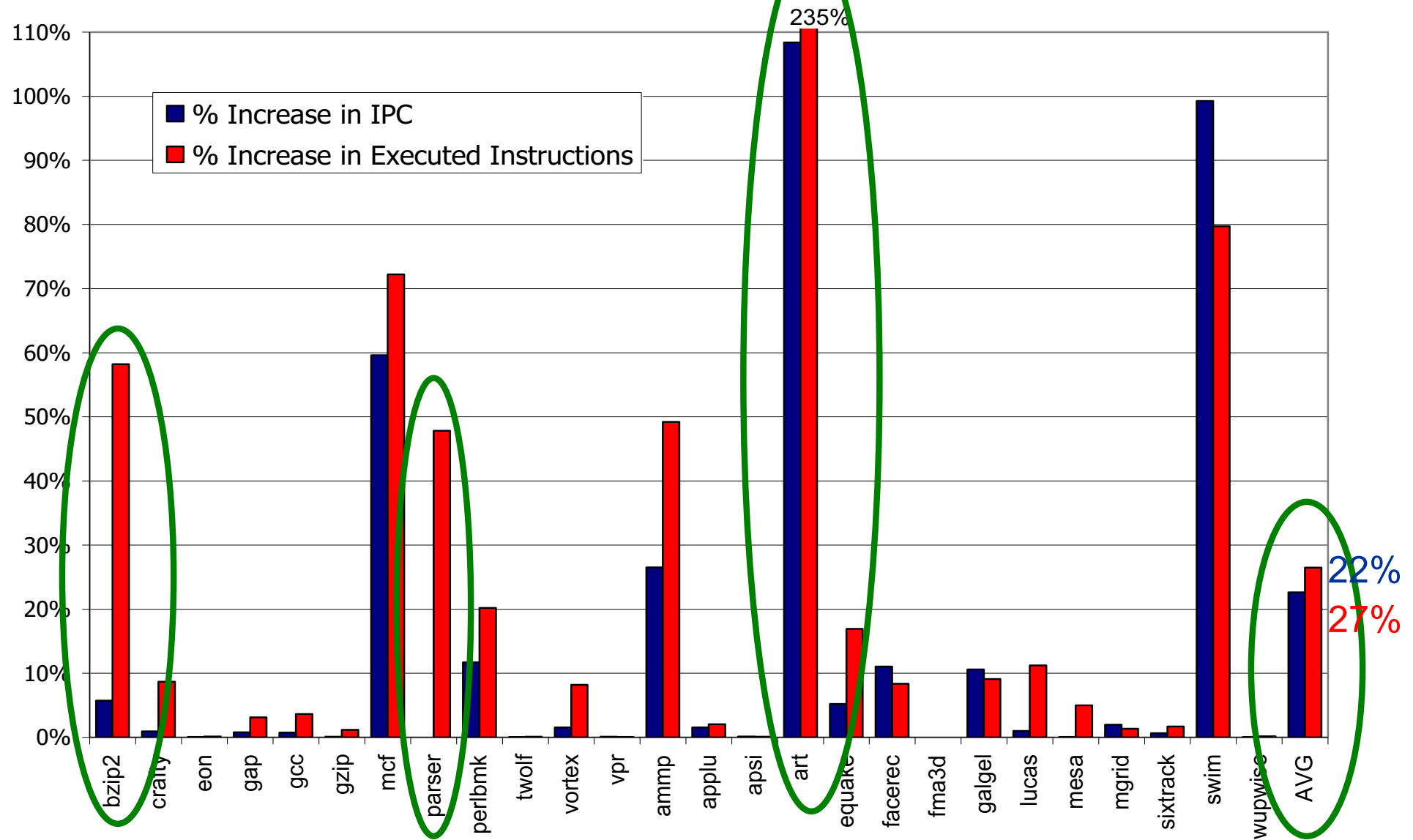
- **Ineffectiveness for pointer-intensive applications**

- 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

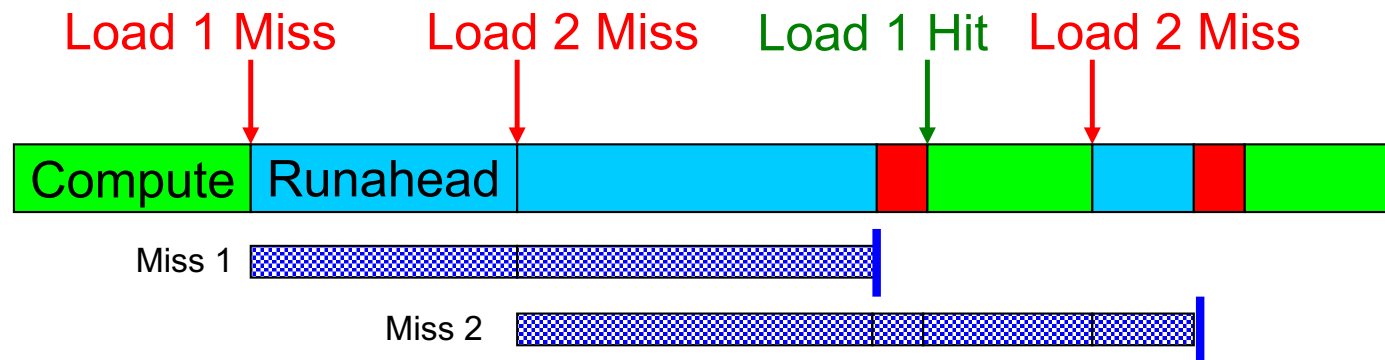


Causes of Inefficiency

- 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

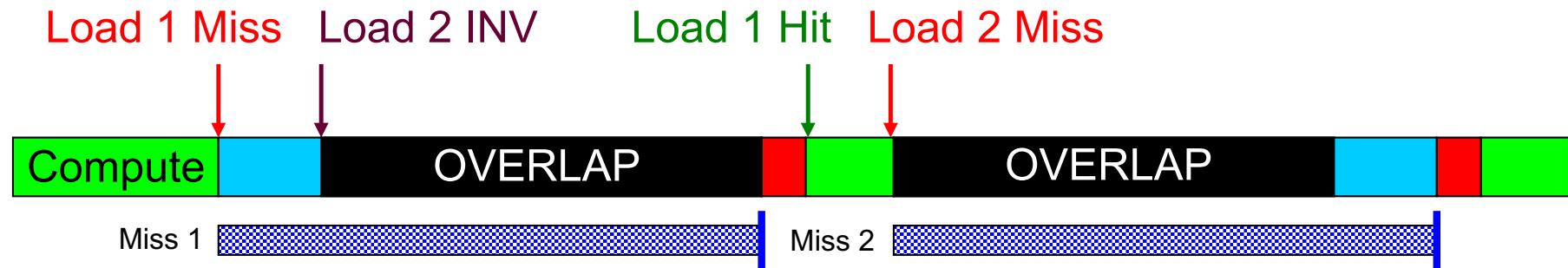
- Processor can initiate runahead mode due to an already in-flight L2 miss generated by
 - the prefetcher, wrong-path, or a previous runahead period



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

Overlapping Runahead Periods

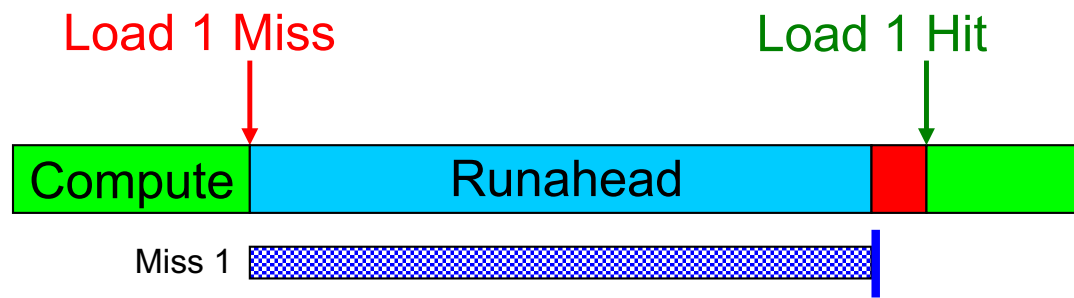
- Two runahead periods that execute the same instructions



- Second period is inefficient
-

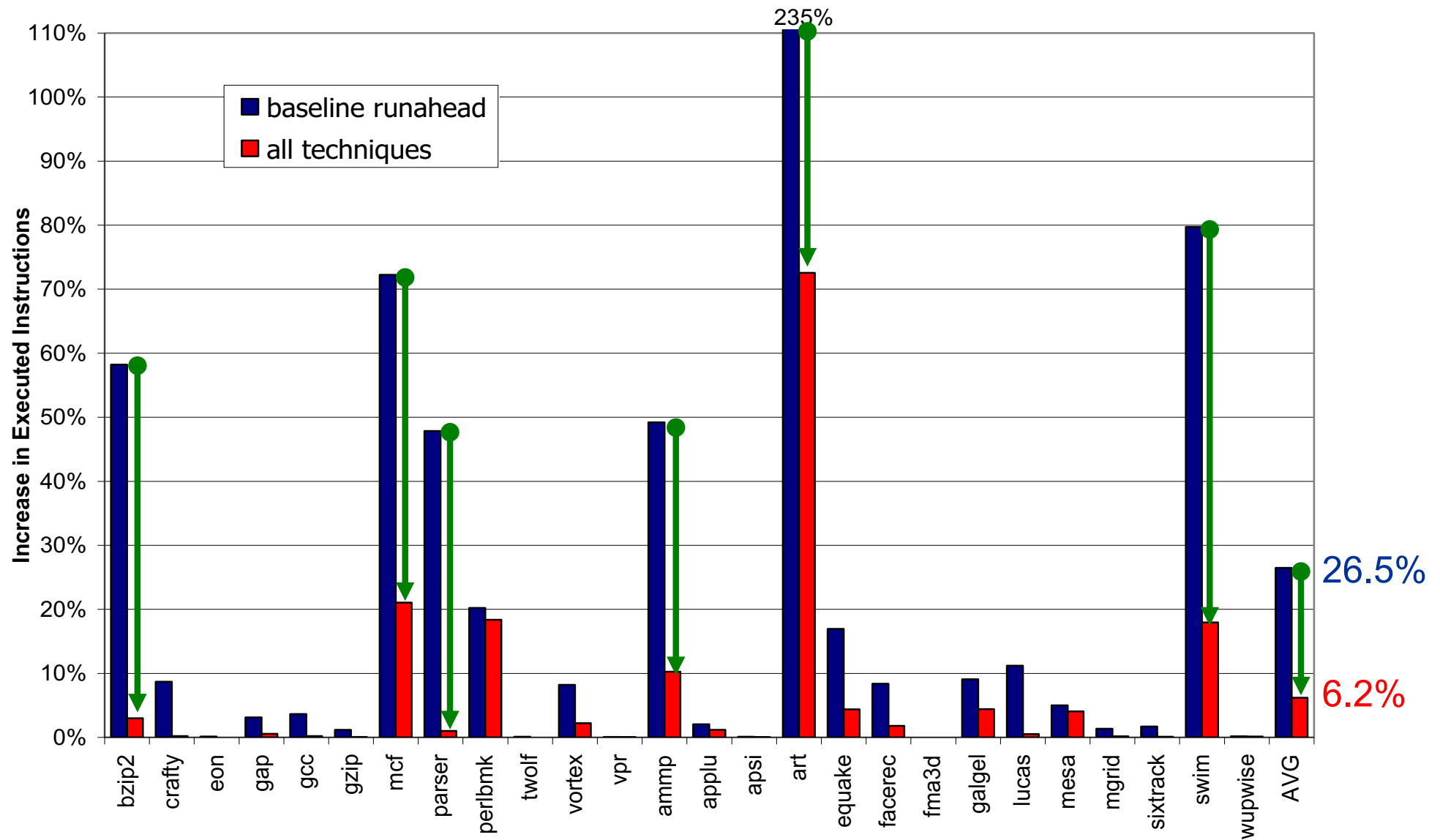
Useless Runahead Periods

- Periods that do not result in prefetches for normal mode

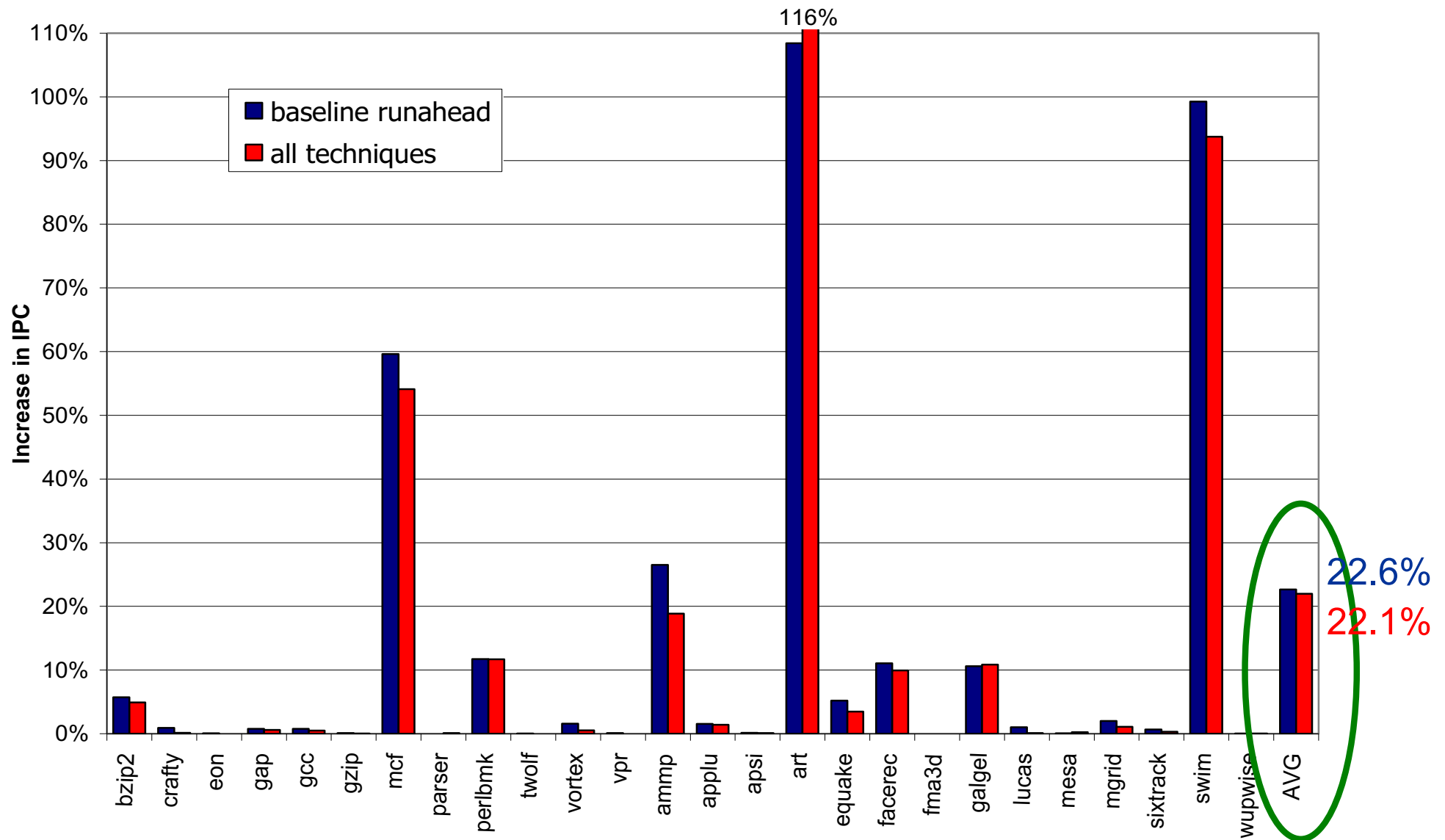


- They exist due to the lack of memory-level parallelism
 - Mechanism to eliminate useless periods:
 - Predict if a period will generate useful L2 misses
 - Estimate a period to be useful if it generated an L2 miss that cannot be captured by the instruction window
 - Useless period predictors are trained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

Overall Impact on Executed Instructions



Overall Impact on IPC



More on Efficient Runahead Execution

- Onur Mutlu, Hyesoon Kim, and Yale N. Patt,
"Techniques for Efficient Processing in Runahead Execution Engines"
Proceedings of the 32nd International Symposium on Computer Architecture (ISCA), pages 370-381, Madison, WI, June 2005. [Slides \(ppt\)](#) [Slides \(pdf\)](#)

Techniques for Efficient Processing in Runahead Execution Engines

Onur Mutlu Hyesoon Kim Yale N. Patt

Department of Electrical and Computer Engineering
University of Texas at Austin
{onur,hyesoon,patt}@ece.utexas.edu

More on Efficient Runahead Execution

- Onur Mutlu, Hyesoon Kim, and Yale N. Patt,
"Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance"
IEEE Micro, Special Issue: Micro's Top Picks from Microarchitecture Conferences (MICRO TOP PICKS), Vol. 26, No. 1, pages 10-20, January/February 2006.

EFFICIENT RUNAHEAD EXECUTION: POWER-EFFICIENT MEMORY LATENCY TOLERANCE

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**

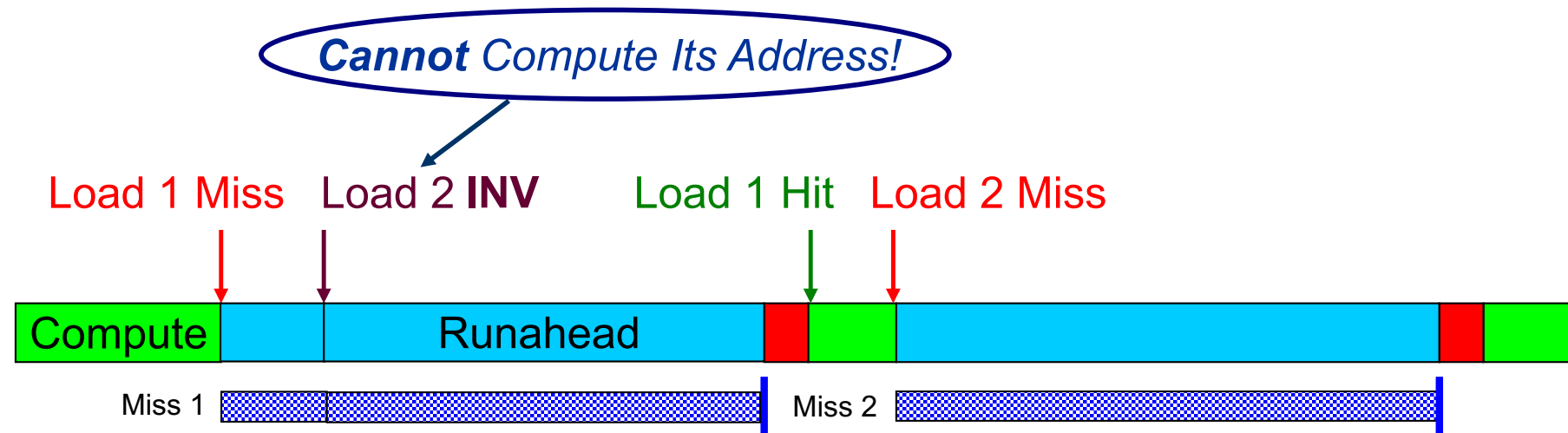
- Runahead cannot parallelize dependent L2 cache misses
- **Address-Value Delta (AVD) Prediction** [MICRO' 05]

- **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

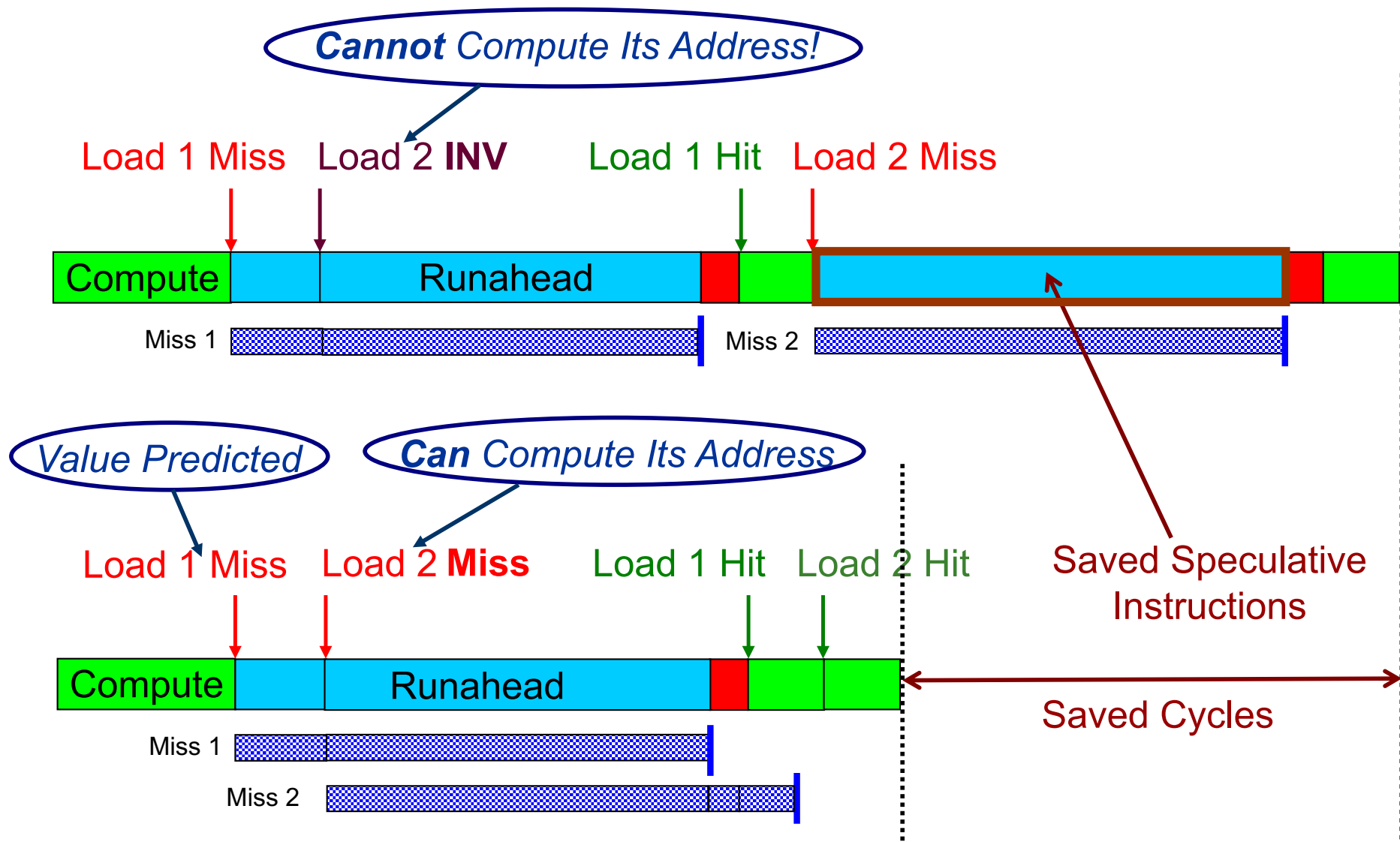


- Runahead execution cannot parallelize dependent misses
 - ❑ wasted opportunity to improve performance
 - ❑ wasted energy (useless pre-execution)
- Runahead performance would improve by 25% if this limitation were ideally overcome

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]

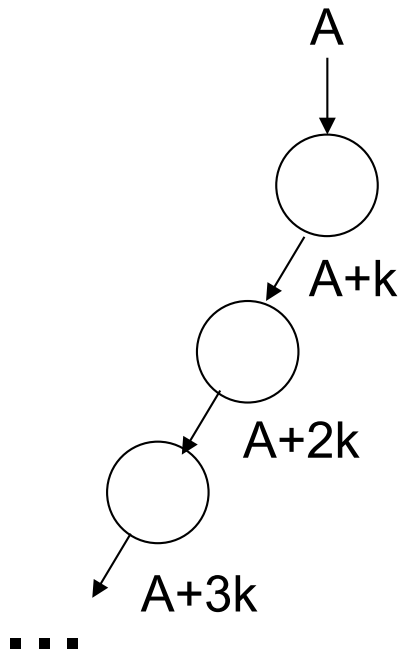
- Address-value delta (AVD) of a load instruction defined as:
$$\text{AVD} = \text{Effective Address of Load} - \text{Data Value of Load}$$
 - For some address loads, AVD is stable
 - An AVD predictor keeps track of the AVDs of address loads
 - When a load is an L2 miss in runahead mode, AVD predictor is consulted
 - If the predictor returns a stable (confident) AVD for that load, the value of the load is predicted
$$\text{Predicted Value} = \text{Effective Address} - \text{Predicted AVD}$$
-

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

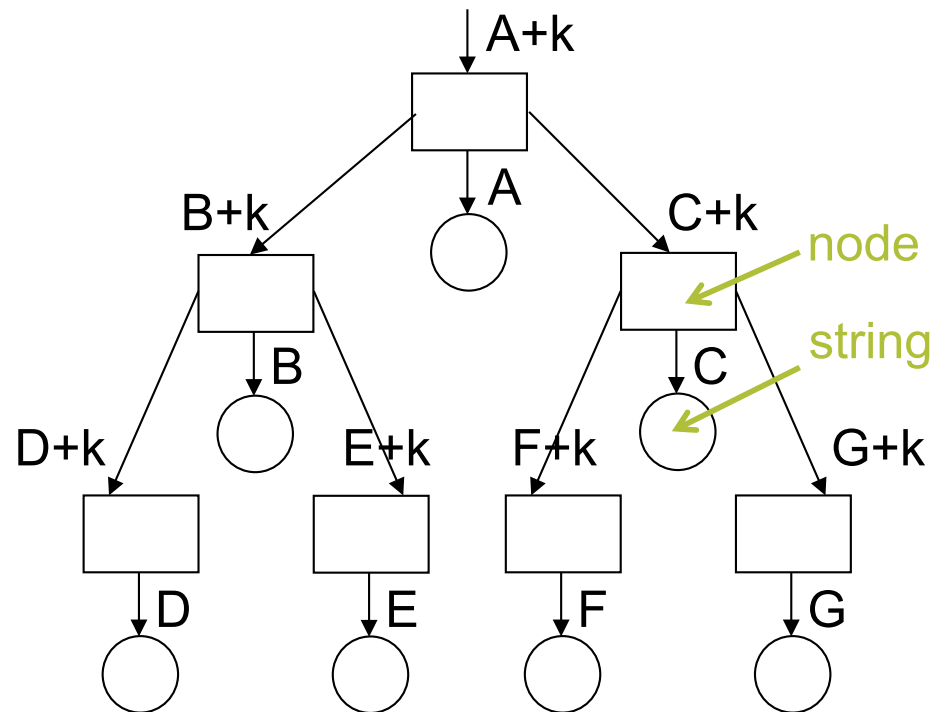
Stable AVD

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

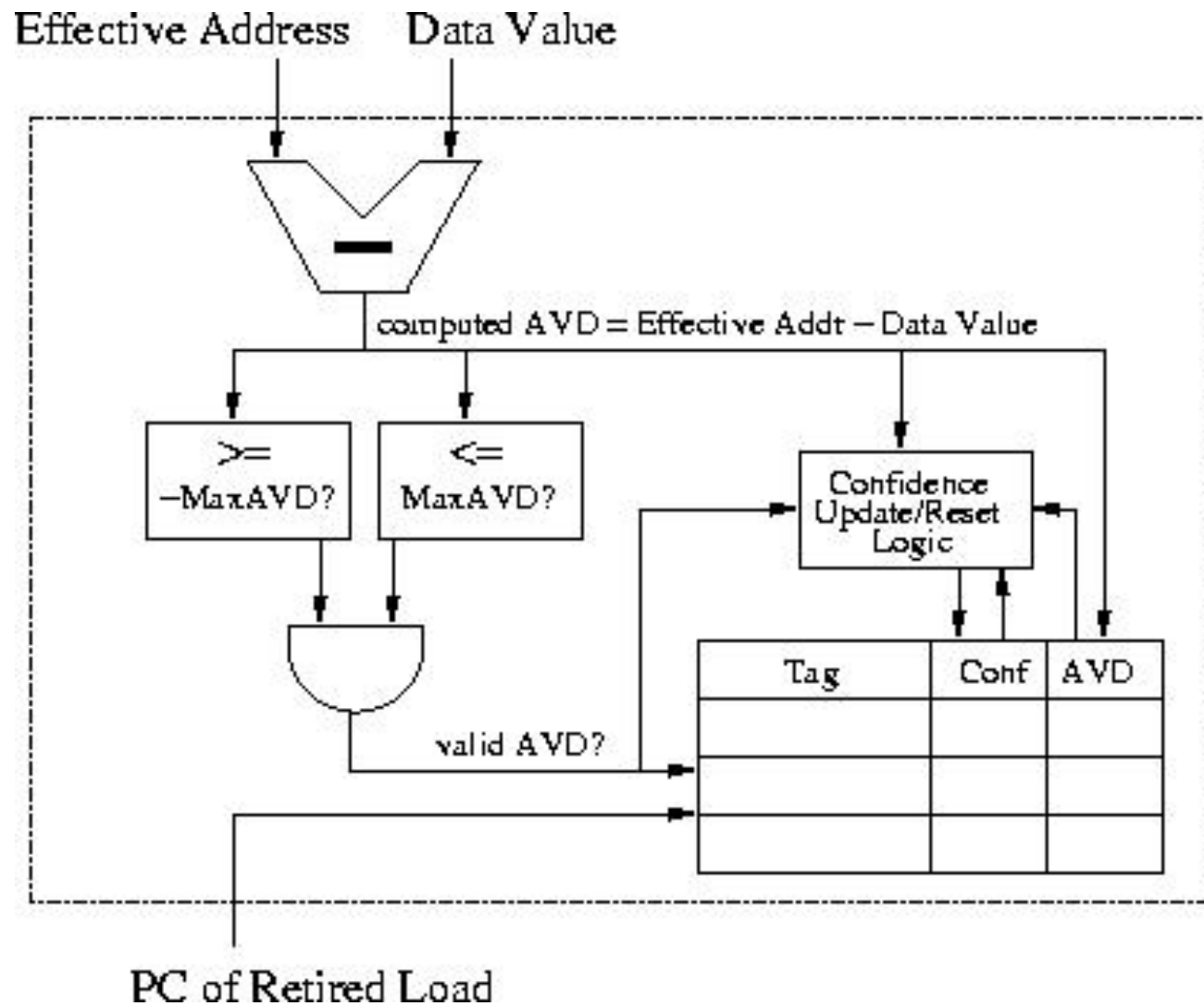
Identifying Address Loads in Hardware

- 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 for prediction
 - No need to value- predict the loads that will not generate addresses
 - Enables the predictor to be small

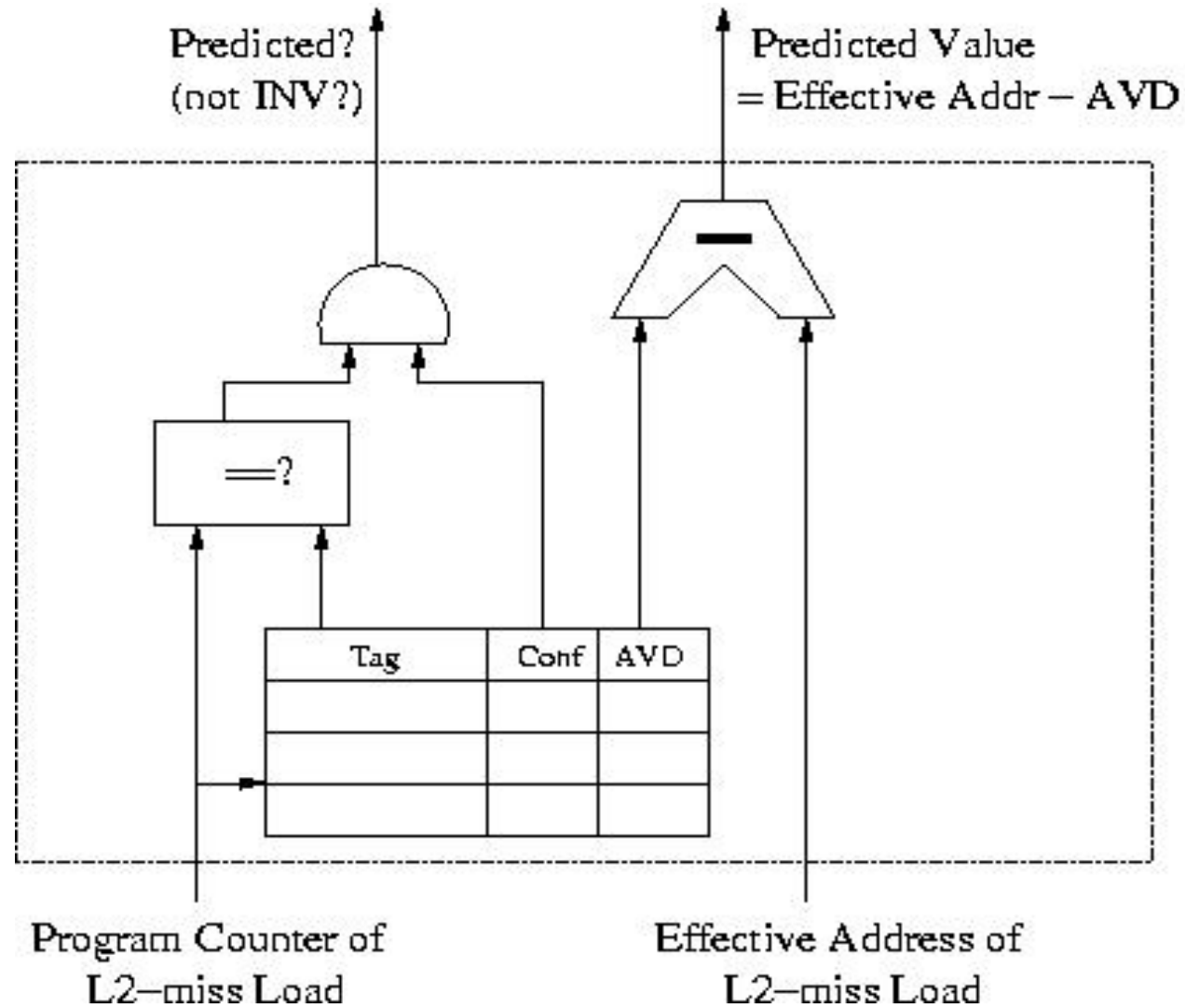
An Implementable AVD Predictor

- 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



Performance of AVD Prediction



More on AVD Prediction

- Onur Mutlu, Hyesoon Kim, and Yale N. Patt,
"Address-Value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns"
Proceedings of the 38th International Symposium on Microarchitecture (MICRO), pages 233-244, Barcelona, Spain, November 2005. [Slides \(ppt\)](#)[Slides \(pdf\)](#)

Address-Value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns

Onur Mutlu Hyesoon Kim Yale N. Patt

Department of Electrical and Computer Engineering
University of Texas at Austin
{onur,hyesoon,patt}@ece.utexas.edu

More on AVD Prediction (II)

- Onur Mutlu, Hyesoon Kim, and Yale N. Patt,
"Address-Value Delta (AVD) Prediction: A Hardware Technique for Efficiently Parallelizing Dependent Cache Misses"
IEEE Transactions on Computers (TC), Vol. 55, No. 12, pages 1491-1508, December 2006.

Address-Value Delta (AVD) Prediction: A Hardware Technique for Efficiently Parallelizing Dependent Cache Misses

Onur Mutlu, *Member, IEEE*, Hyesoon Kim, *Student Member, IEEE*, and
Yale N. Patt, *Fellow, IEEE*

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

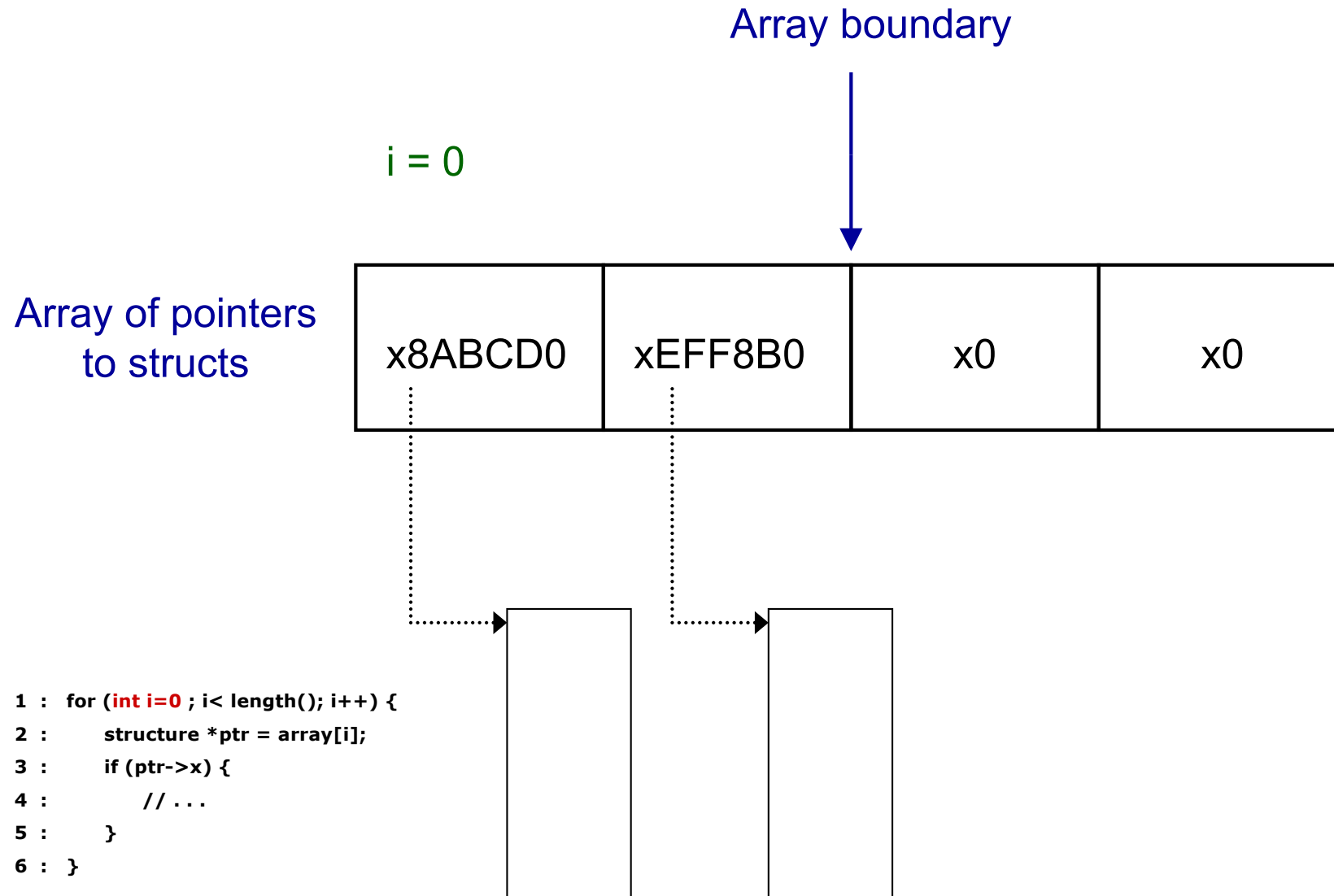
Why Does a WPE Occur?

- 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.

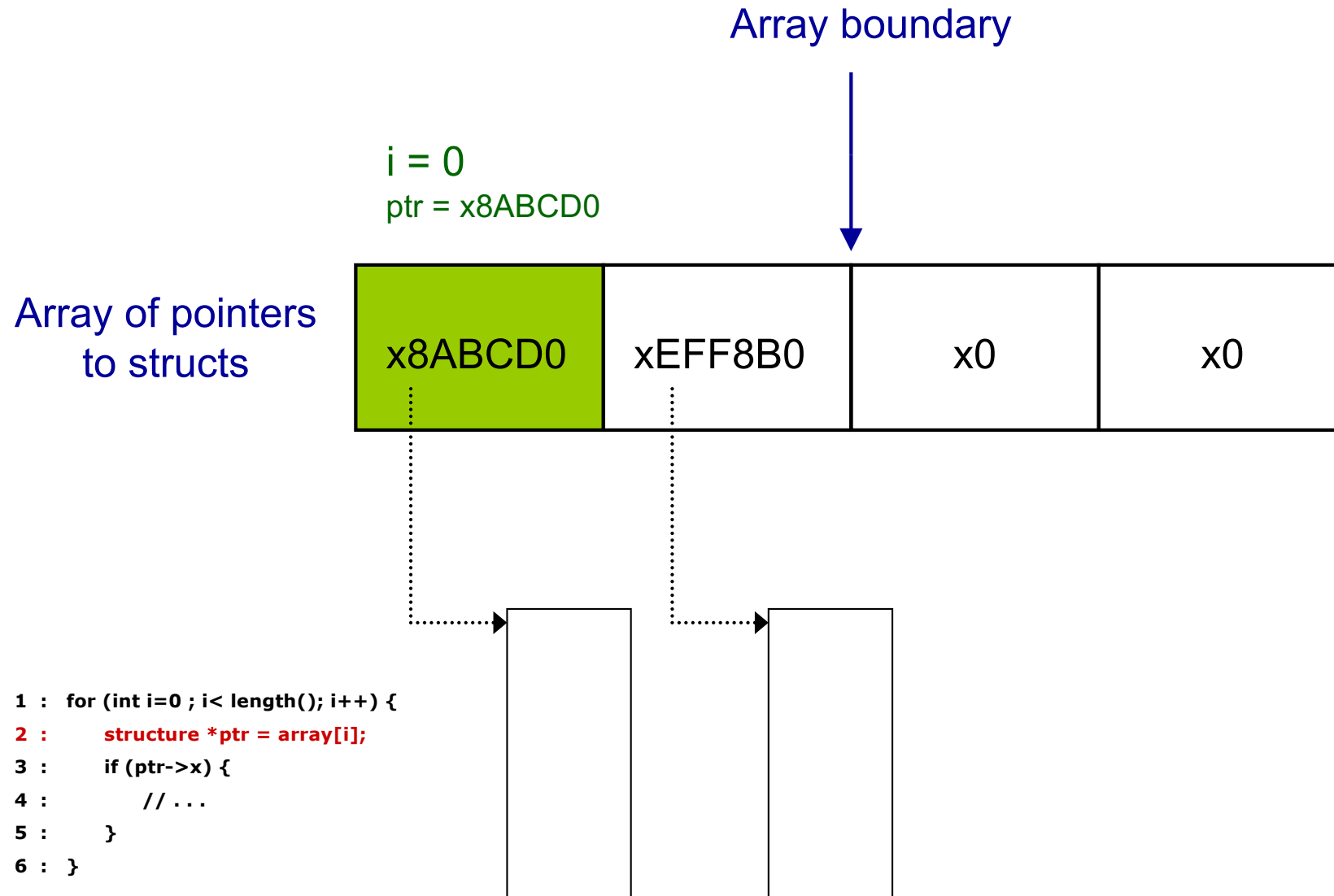
WPE Example from *eon*: NULL pointer dereference

```
1 : for (int i=0 ; i< length(); i++) {  
2 :     structure *ptr = array[i];  
3 :     if (ptr->x) {  
4 :         // ...  
5 :     }  
6 : }
```

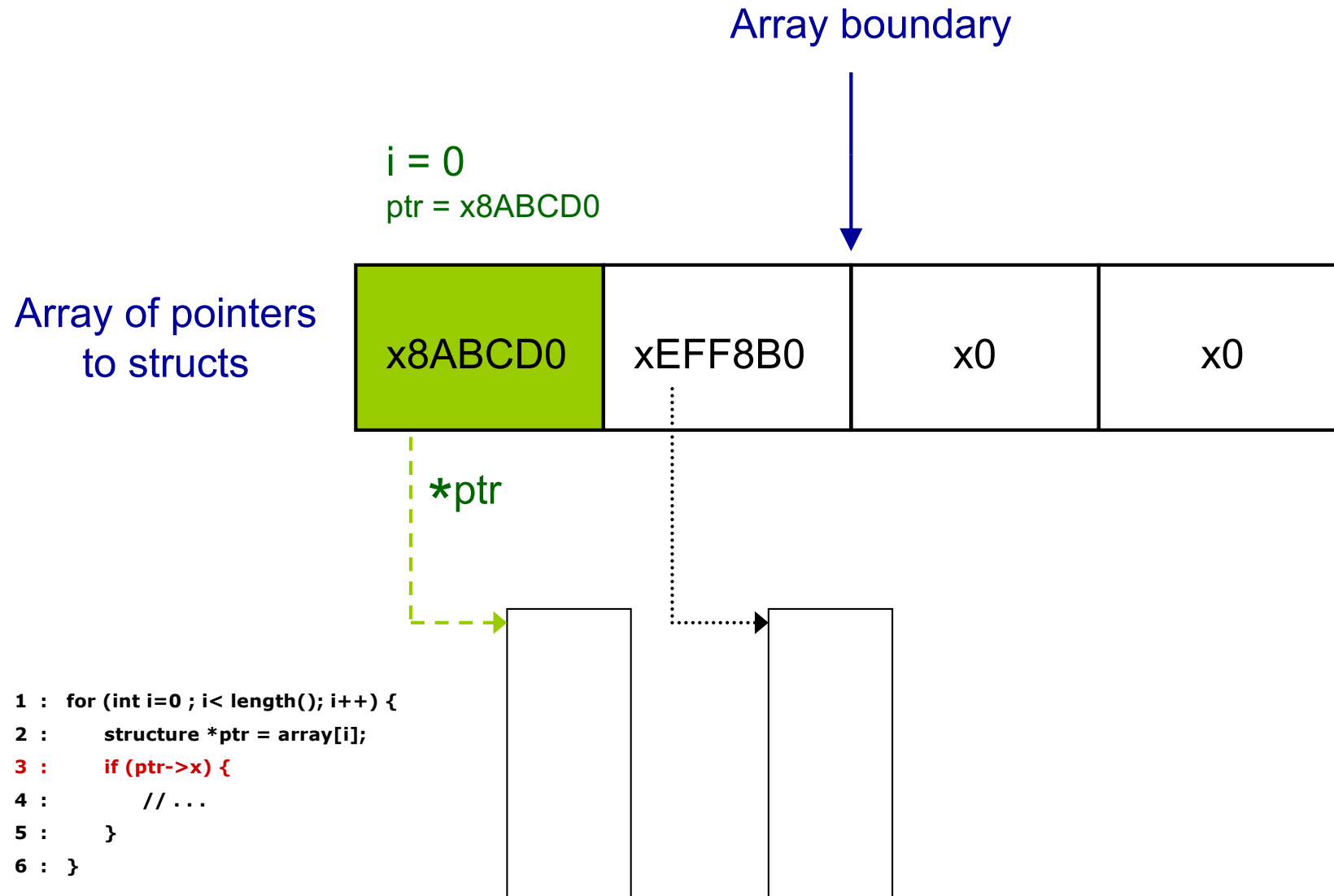
Beginning of the loop



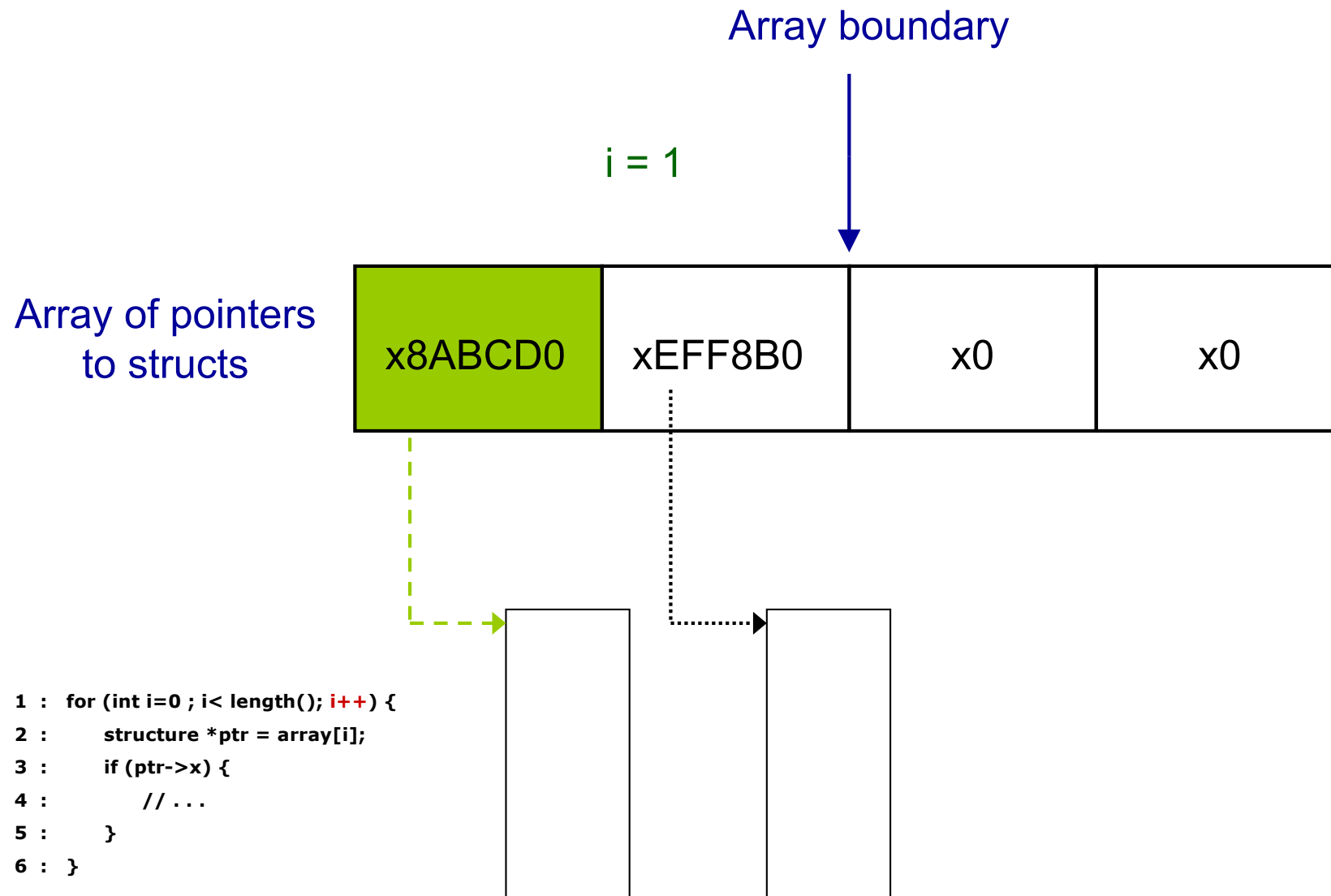
First iteration



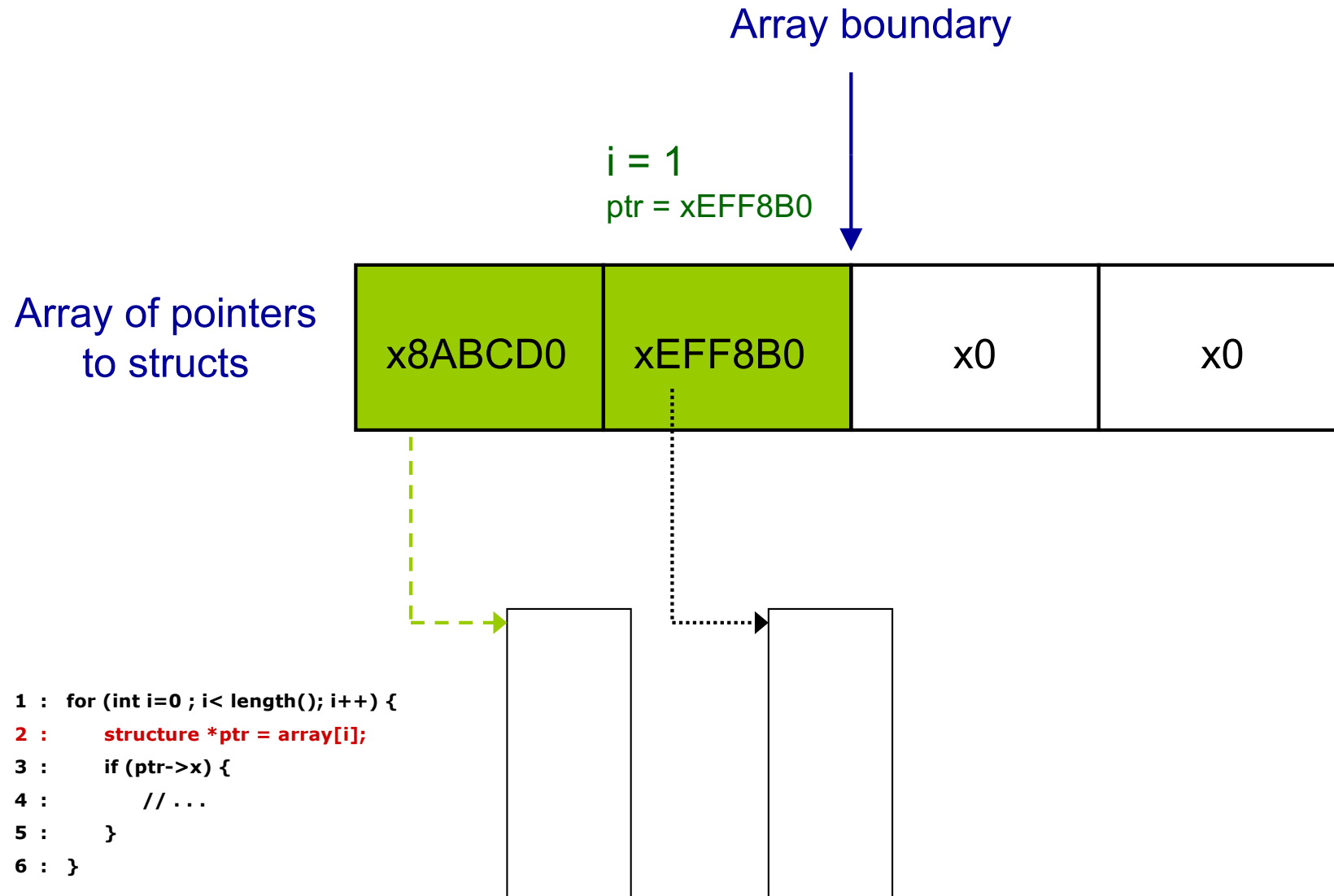
First iteration



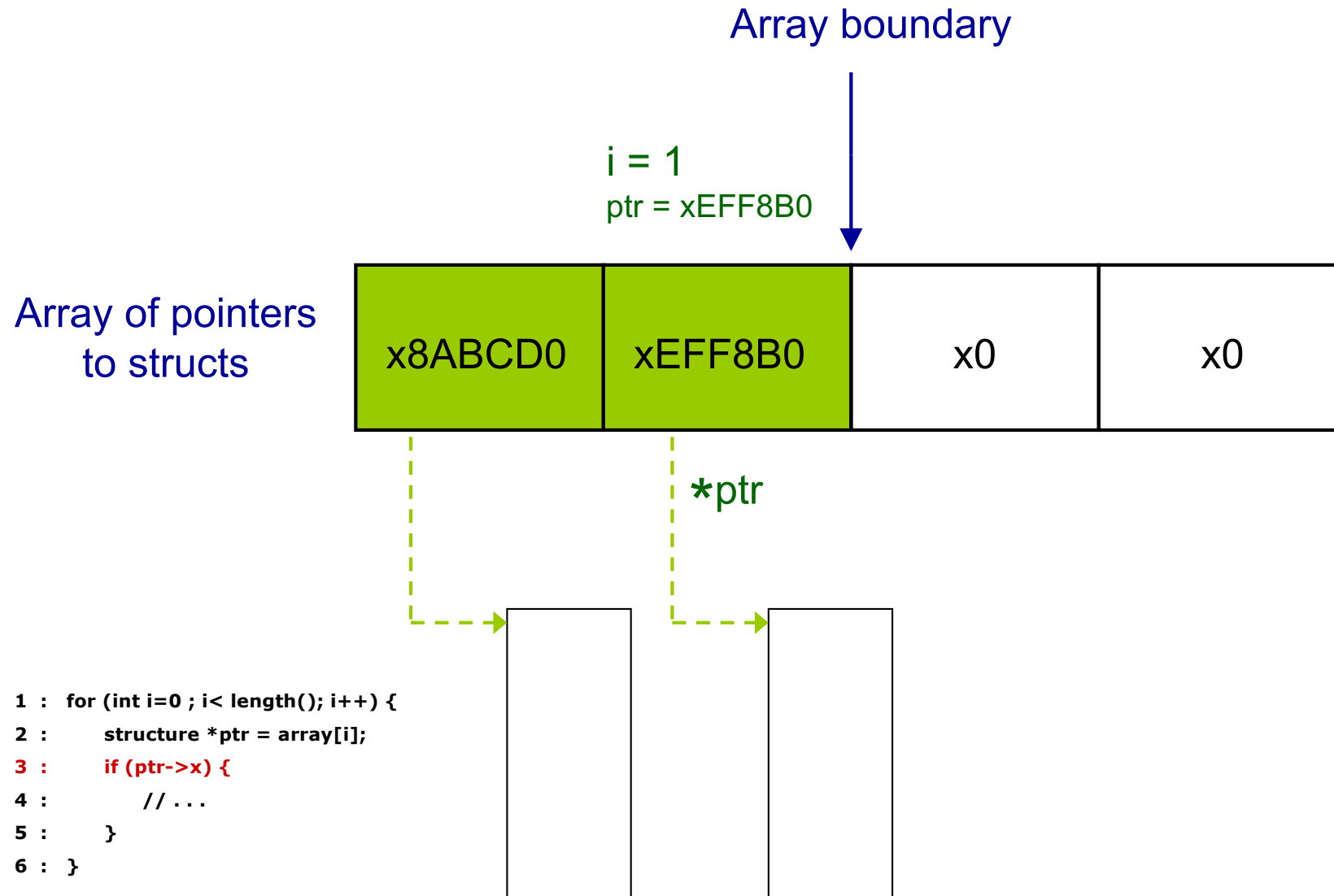
Loop branch correctly predicted



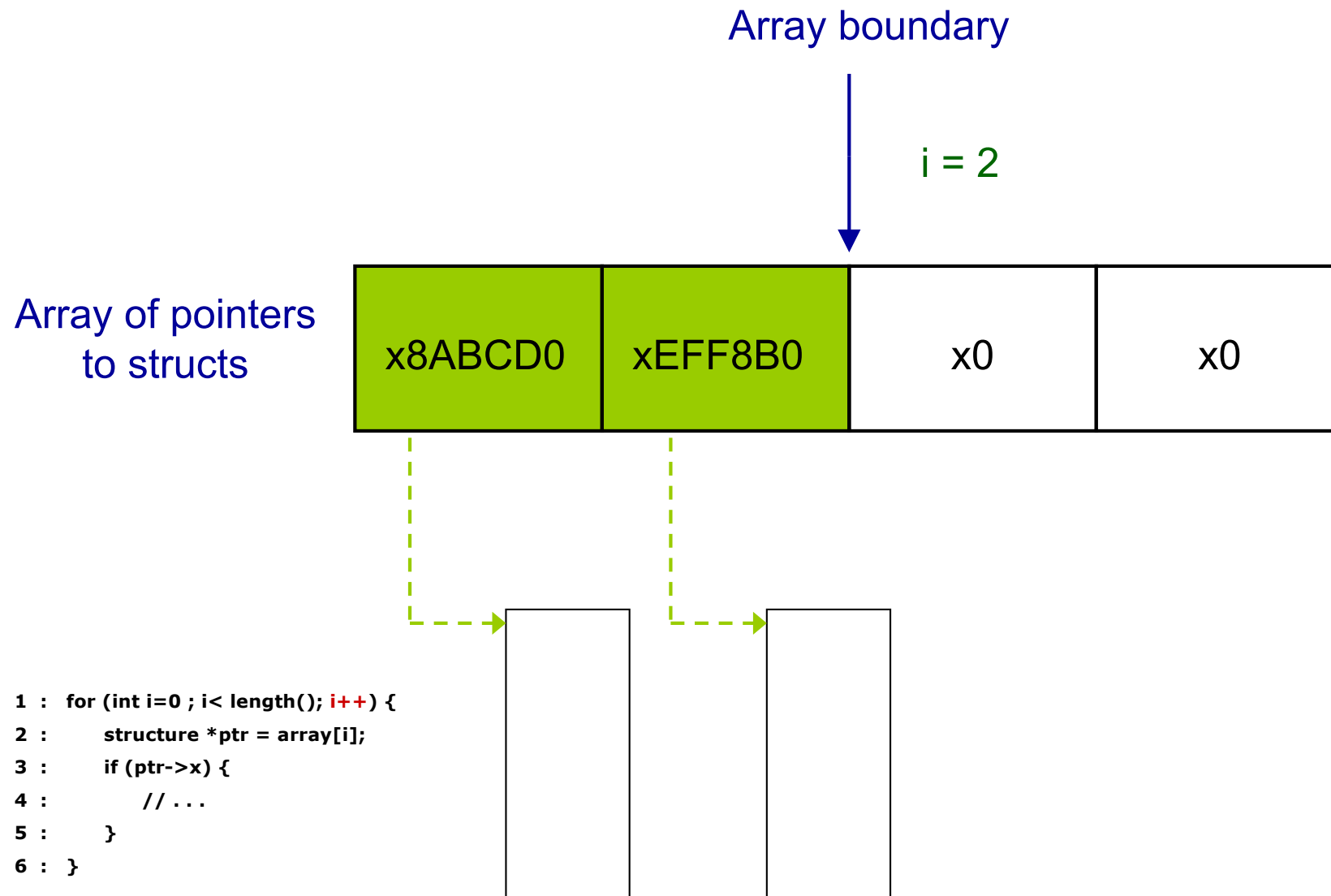
Second iteration



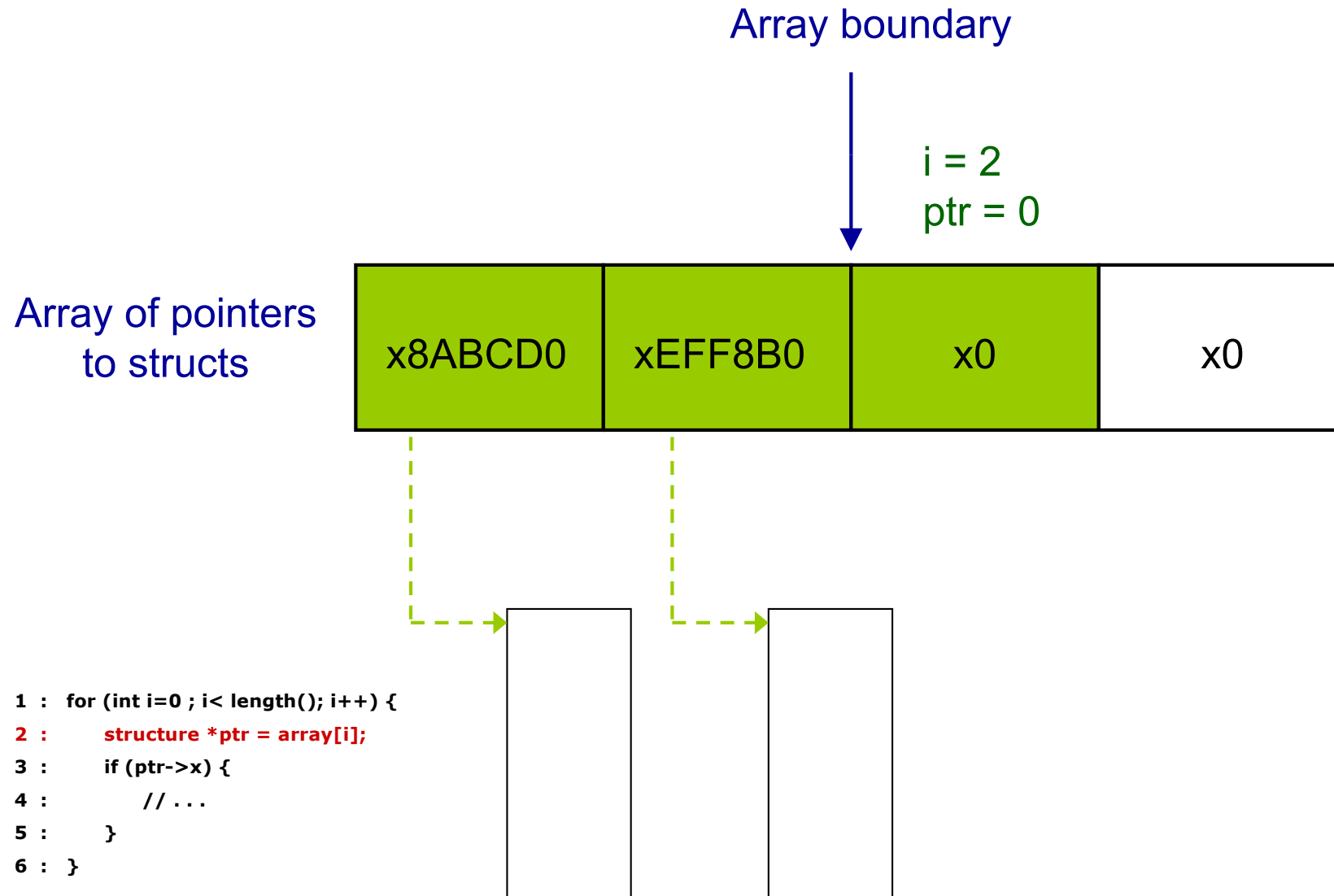
Second iteration



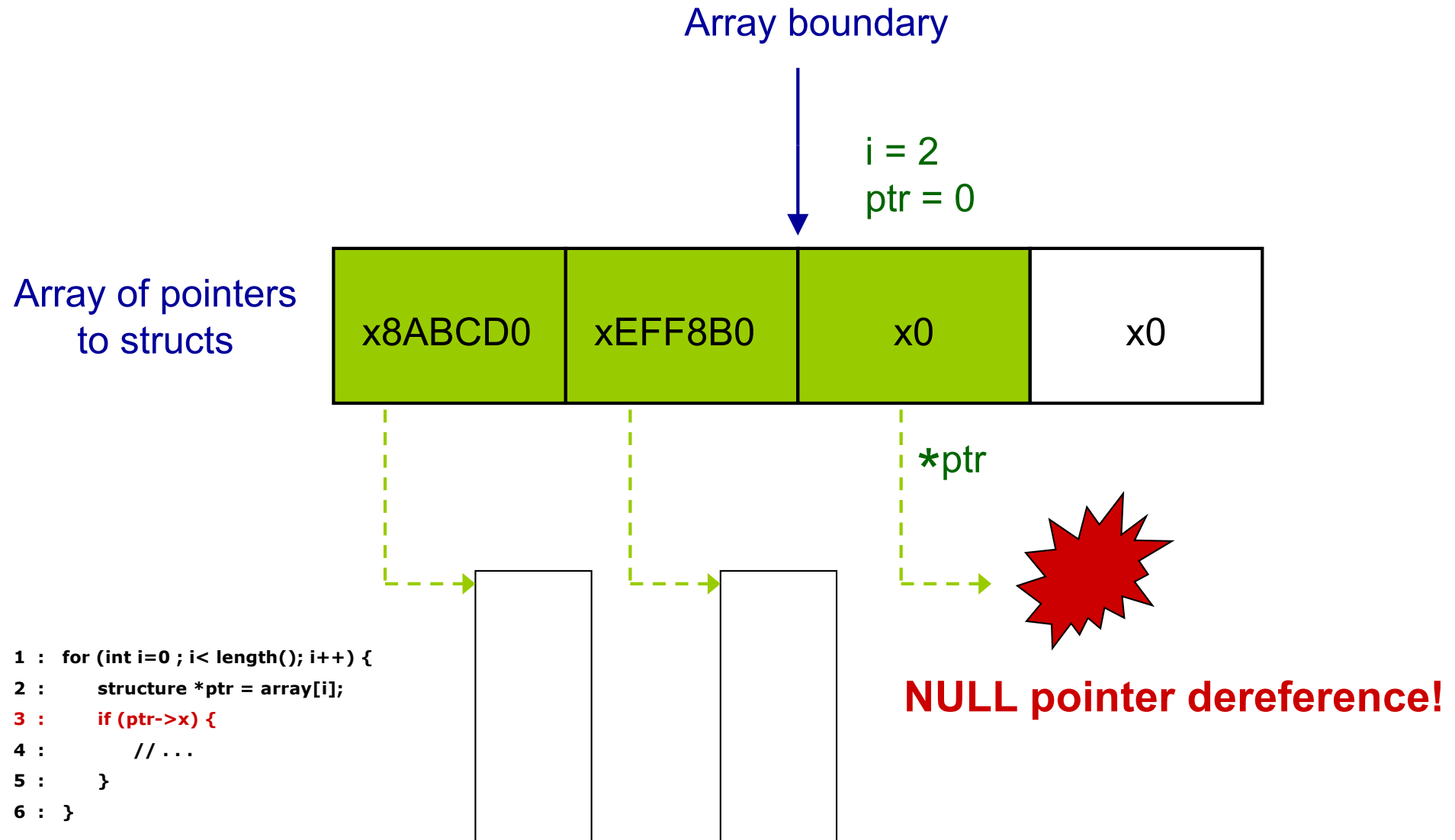
Loop exit branch mispredicted



Third iteration on wrong path



Wrong Path Event



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?

More on Wrong Path Events

- David N. Armstrong, Hyesoon Kim, Onur Mutlu, and Yale N. Patt, **"Wrong Path Events: Exploiting Unusual and Illegal Program Behavior for Early Misprediction Detection and Recovery"** *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, pages 119-128, Portland, OR, December 2004. [Slides \(pdf\)](#)[Slides \(ppt\)](#)

Wrong Path Events: Exploiting Unusual and Illegal Program Behavior for Early Misprediction Detection and Recovery

David N. Armstrong Hyesoon Kim Onur Mutlu Yale N. Patt

Department of Electrical and Computer Engineering
The University of Texas at Austin
{dna,hyesoon,onur,patt}@ece.utexas.edu

Why Is This Important?

- A modern processor spends significant amount of time fetching/executing instructions on the wrong path

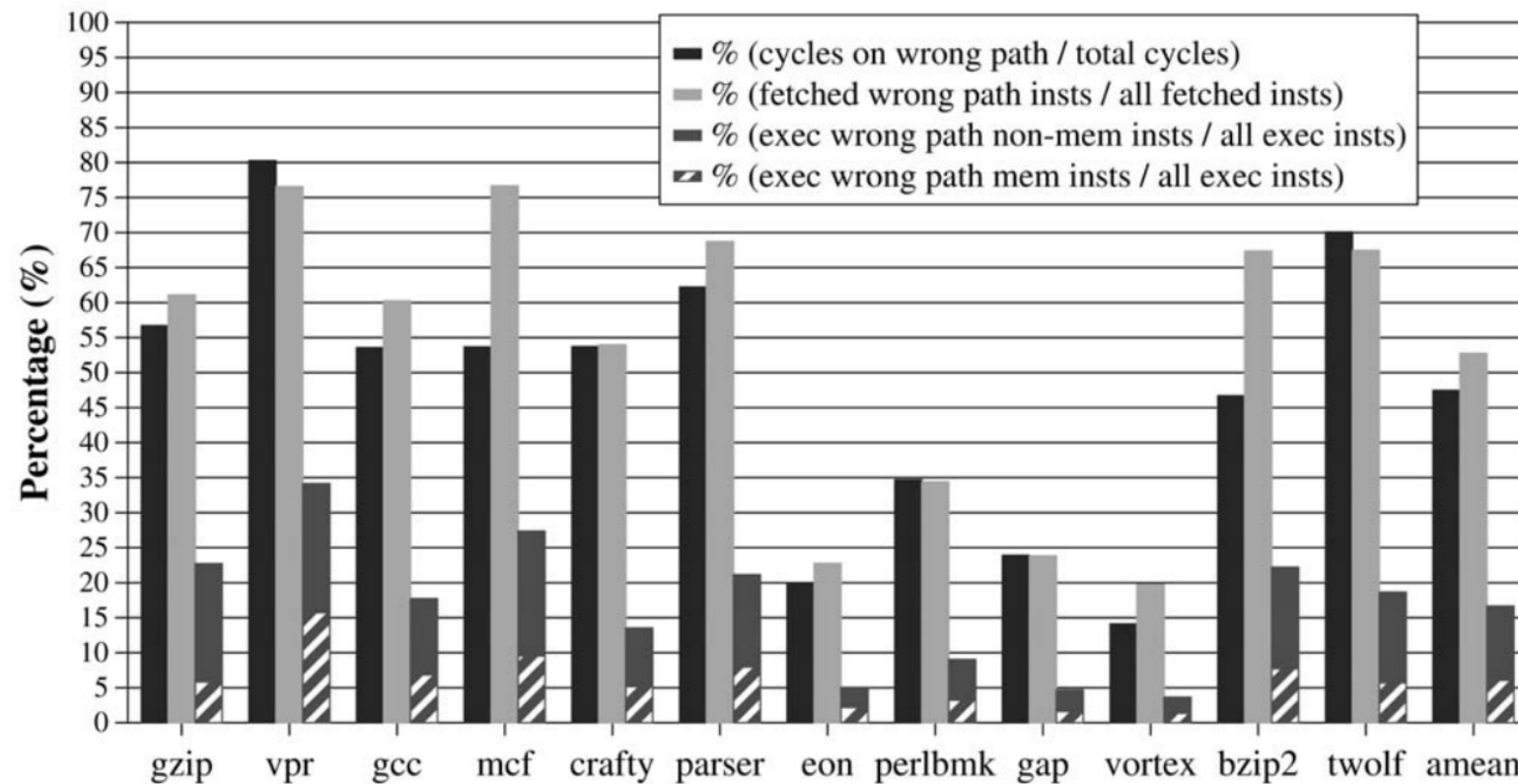


Fig. 1. Percentage of fetch cycles spent on the wrong path, percentage of instructions fetched on the wrong path, and percentage of instructions (memory and nonmemory) executed on the wrong path in the baseline processor for SPEC 2000 integer benchmarks.

A Lot of Time Spent on The Wrong Path

- A runahead processor, much more so...

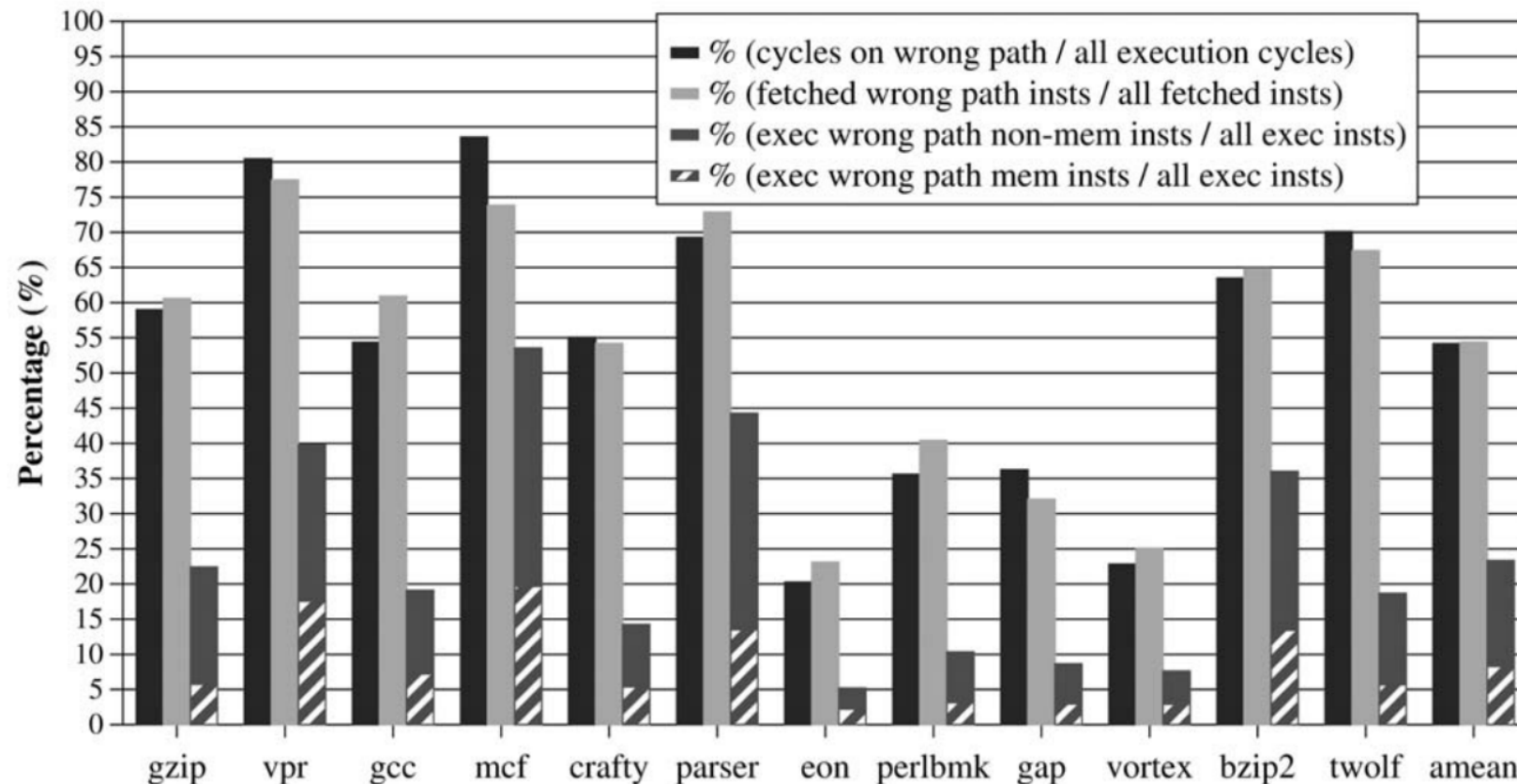


Fig. 20. Percentage of total cycles spent on the wrong path, percentage of instructions fetched on the wrong path, and percentage of instructions (memory and nonmemory) executed on the wrong path in the runahead processor.

Is Wrong-Path Execution Useless/Useful/Harmful?

4 WRONG PATH: TO MODEL OR NOT TO MODEL

In this section, we measure the error in IPC if wrong-path memory references are not simulated. We also evaluate the overall effect of wrong-path memory references on the IPC (retired Instructions Per Cycle) performance of a processor.

1. How important is it to correctly model wrong-path memory references? What is the error in the predicted performance if wrong-path references are not modeled?
2. Do wrong-path memory references affect performance positively or negatively? What is the relative significance on performance of prefetching, bandwidth consumption, and pollution caused by wrong-path references?
3. What kind of code structures lead to the positive effects of wrong-path memory references?
4. How do wrong-path memory references affect the performance of a runahead execution processor [7], [18] which implements an aggressive form of speculative execution?

Wrong Path Is Often Useful for Performance

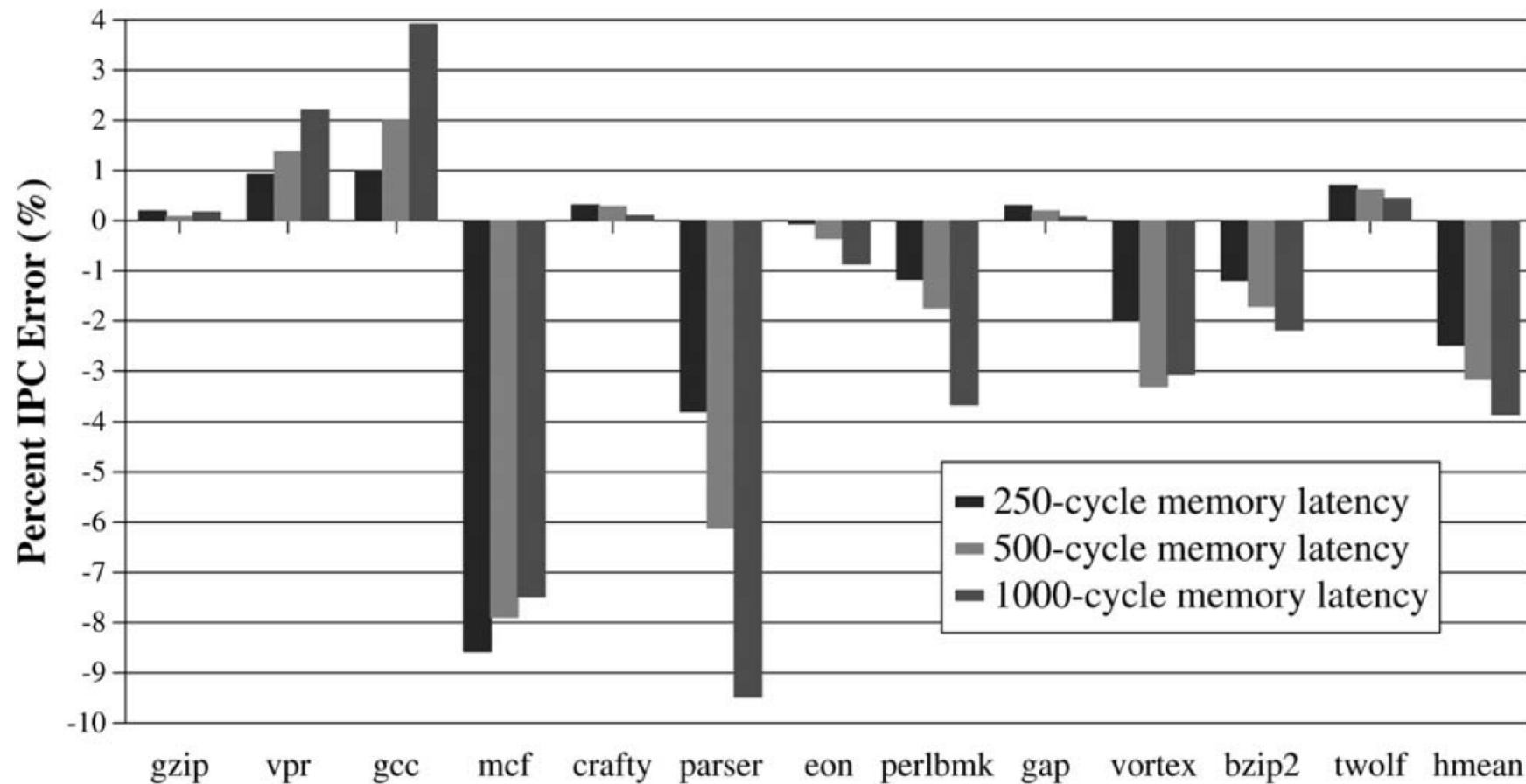


Fig. 7. Error in the IPC of the baseline processor with a stream prefetcher for three different memory latencies if wrong-path memory references are not simulated.

More So In Runahead Execution

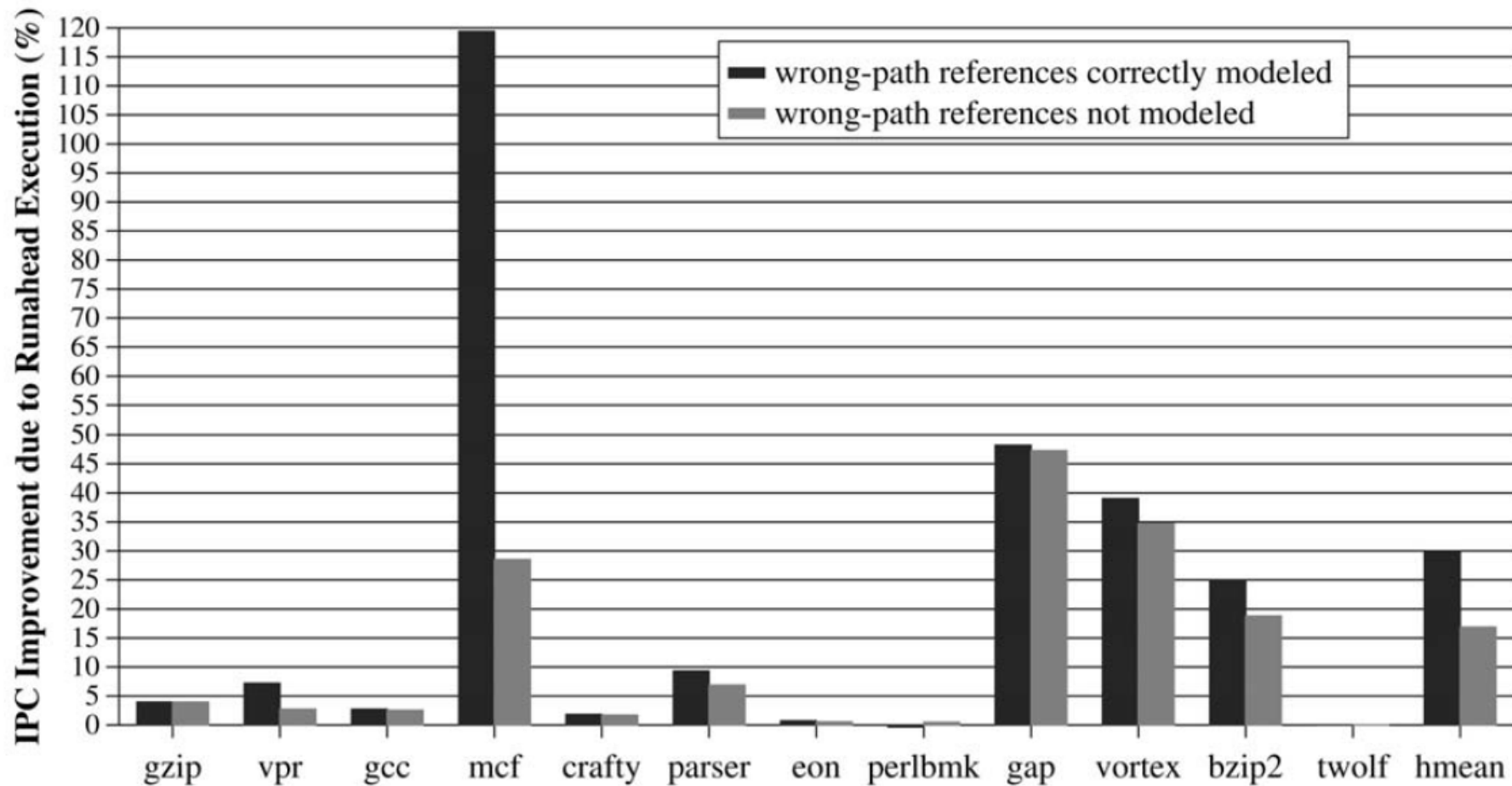


Fig. 19. IPC improvement of adding runahead execution to the baseline processor if wrong-path memory references are or are not modeled.

Why is Wrong Path Useful? (I)

- Control-independence: e.g., wrong-path execution of future loop iterations

```
1 :   arc_t *arc; // array of arc_t structures
2 :   // initialize arc (arc = ...)
3 :
4 :   for ( ; arc < stop_arcs; arc += size) {
5 :       if (arc->ident > 0) { // frequently mispredicted br.
6 :           // function calls and
7 :           // operations on the structure pointed to by arc
8 :           // ...
9 :       }
10: }
```

Fig. 16. An example from mcf showing wrong-path prefetching for later loop iterations.

Why is Wrong Path Useful? (II)

```
1 :  l = min; r = max;
2 :  cut = perm[ (long)( (l+r) / 2 ) ]->abs_cost;
3 :
4 :  do {
5 :    while( perm[l]->abs_cost > cut )
6 :      l++;
7 :    while( cut > perm[r]->abs_cost )
8 :      r--;
9 :
10:    if( l < r ) {
11:      xchange = perm[l];
12:      perm[l] = perm[r];
13:      perm[r] = xchange;
14:    }
15:    if( l <= r ) {
16:      l++; r--;
17:    }
18:  } while( l <= r );
```

Fig. 17. An example from mcf showing wrong-path prefetching between different loops.

Why is Wrong Path Useful? (III)

- Same data used in different control flow paths

```
1:  node_t *node;
2:  // initialize node
3:  // ...
4:
5:  while (node) {
6:
7:      if (node->orientation == UP) { // mispredicted branch
8:          node->potential = node->basic_arc->cost
9:                          + node->pred->potential;
10:     } else { /* == DOWN */
11:         node->potential = node->pred->potential
12:                         - node->basic_arc->cost;
13:         // ...
14:     }
15:     // control-flow independent point (re-convergent point)
16:     node = node->child;
17: }
```

Fig. 18. An example from mcf showing wrong-path prefetching in control-flow hammocks.

More on Wrong Path Execution (I)

- Onur Mutlu, Hyesoon Kim, David N. Armstrong, and Yale N. Patt,
"Understanding the Effects of Wrong-Path Memory References on Processor Performance"
Proceedings of the 3rd Workshop on Memory Performance Issues (WMPI), pages 56-64, Munchen, Germany, June 2004. [Slides](#)
[\(pdf\)](#)

Understanding The Effects of Wrong-Path Memory References on Processor Performance

Onur Mutlu Hyesoon Kim David N. Armstrong Yale N. Patt

Department of Electrical and Computer Engineering
The University of Texas at Austin
{onur,hyesoon,dna,patt}@ece.utexas.edu

More on Wrong Path Execution (II)

- Onur Mutlu, Hyesoon Kim, David N. Armstrong, and Yale N. Patt, **"An Analysis of the Performance Impact of Wrong-Path Memory References on Out-of-Order and Runahead Execution Processors"** *IEEE Transactions on Computers (TC)*, Vol. 54, No. 12, pages 1556-1571, December 2005.

An Analysis of the Performance Impact of Wrong-Path Memory References on Out-of-Order and Runahead Execution Processors

Onur Mutlu, *Student Member, IEEE*, Hyesoon Kim, *Student Member, IEEE*,
David N. Armstrong, and Yale N. Patt, *Fellow, IEEE*

What If ...

- The system learned from wrong-path execution and used that learning for better execution of the program/system?
- An open research problem...

Computer Architecture

Lecture 17:

Latency Tolerance and Prefetching

Prof. Onur Mutlu

ETH Zürich

Fall 2017

22 November 2017

We did not cover the following slides in lecture.
These are for your preparation for the next lecture.

Prefetching

Outline of Prefetching Lecture(s)

- Why prefetch? Why could/does it work?
- The four questions
 - What (to prefetch), when, where, how
- Software prefetching
- Hardware prefetching algorithms
- Execution-based prefetching
- Prefetching performance
 - Coverage, accuracy, timeliness
 - Bandwidth consumption, cache pollution
- Prefetcher throttling
- Issues in multi-core (if we get to it)

Readings in Prefetching

■ Required:

- Jouppi, “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers,” ISCA 1990.
- Joseph and Grunwald, “Prefetching using Markov Predictors,” ISCA 1997.

■ Recommended:

- Mowry et al., “Design and Evaluation of a Compiler Algorithm for Prefetching,” ASPLOS 1992.
- Srinath et al., “Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers“, HPCA 2007.
- Mutlu et al., “Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors,” HPCA 2003.

Prefetching

- Idea: Fetch the data before it is needed (i.e. pre-fetch) by the program
- Why?
 - ❑ Memory latency is high. If we can prefetch accurately and early enough we can reduce/eliminate that latency.
 - ❑ Can eliminate compulsory cache misses
 - ❑ Can it eliminate all cache misses? Capacity, conflict?
- Involves predicting which address will be needed in the future
 - ❑ Works if programs have predictable miss address patterns

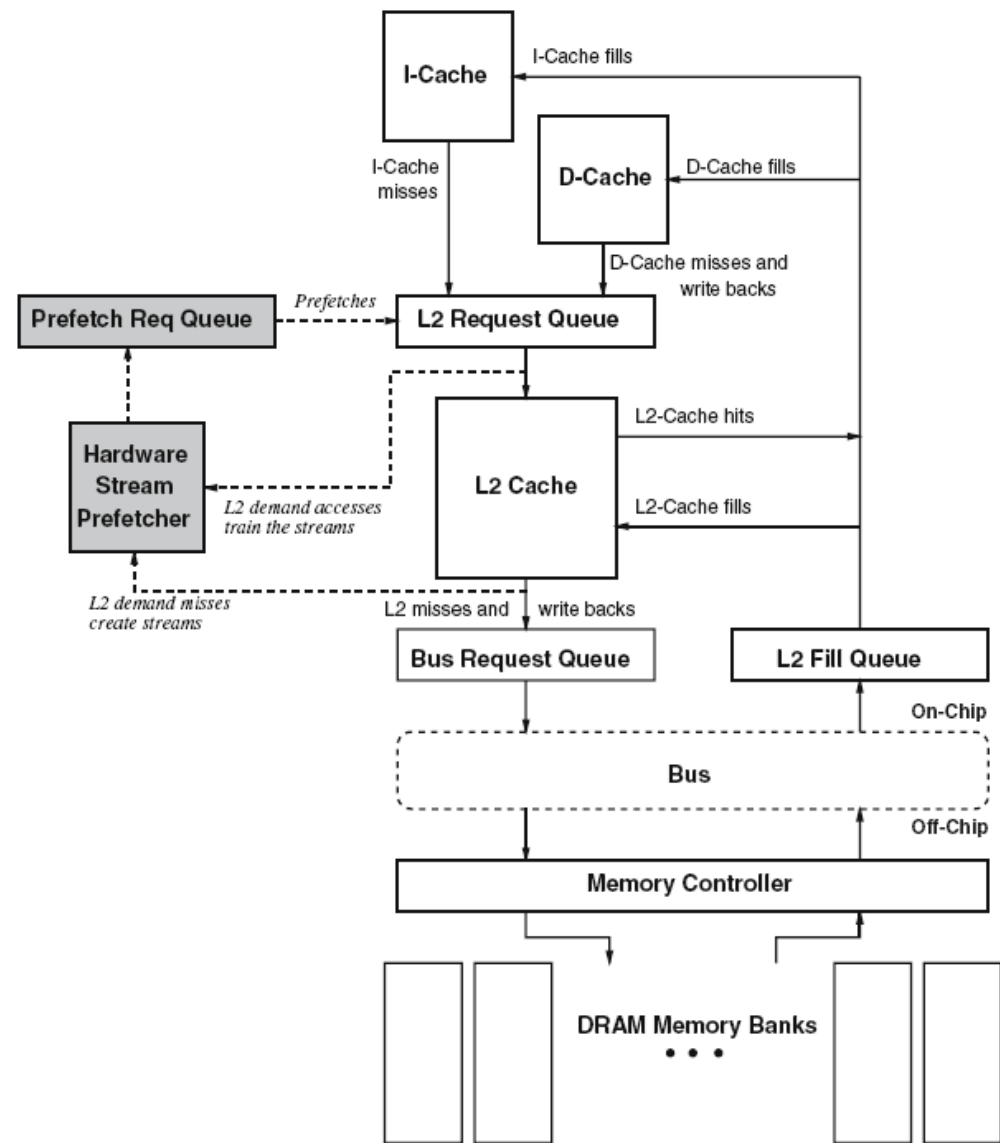
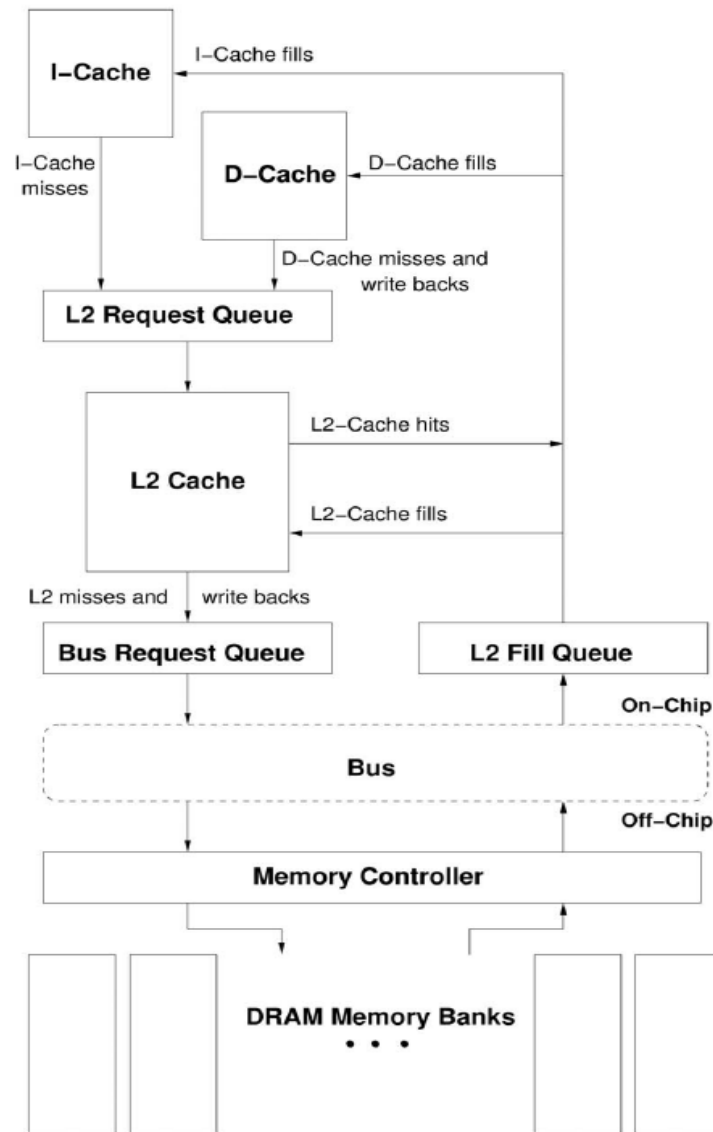
Prefetching and Correctness

- Does a misprediction in prefetching affect correctness?
- No, prefetched data at a “mispredicted” address is simply not used
- There is no need for state recovery
 - In contrast to branch misprediction or value misprediction

Basics

- In modern systems, prefetching is usually done in **cache block granularity**
- Prefetching is a technique that can reduce both
 - Miss rate
 - Miss latency
- Prefetching can be done by
 - hardware
 - compiler
 - programmer

How a HW Prefetcher Fits in the Memory System



Prefetching: The Four Questions

- What
 - **What** addresses to prefetch
- When
 - **When** to initiate a prefetch request
- Where
 - **Where** to place the prefetched data
- How
 - Software, hardware, execution-based, cooperative

Challenges in Prefetching: What

- **What** addresses to prefetch
 - Prefetching useless data wastes resources
 - Memory bandwidth
 - Cache or prefetch buffer space
 - Energy consumption
 - These could all be utilized by demand requests or more accurate prefetch requests
 - **Accurate** prediction of addresses to prefetch is important
 - Prefetch accuracy = used prefetches / sent prefetches
- **How do we know what to prefetch**
 - Predict based on past access patterns
 - Use the compiler's knowledge of data structures
- **Prefetching algorithm** determines what to prefetch

Challenges in Prefetching: When

- **When** to initiate a prefetch request
 - Prefetching too early
 - Prefetched data might not be used before it is evicted from storage
 - Prefetching too late
 - Might not hide the whole memory latency
- When a data item is prefetched affects the **timeliness** of the prefetcher
- Prefetcher can be made more timely by
 - Making it more **aggressive**: try to stay far ahead of the processor's access stream (hardware)
 - Moving the **prefetch instructions earlier in the code** (software)

Challenges in Prefetching: Where (I)

- **Where** to place the prefetched data
 - In cache
 - + Simple design, no need for separate buffers
 - Can evict useful demand data → cache pollution
 - In a separate **prefetch buffer**
 - + Demand data protected from prefetches → no cache pollution
 - More complex memory system design
 - Where to place the prefetch buffer
 - When to access the prefetch buffer (parallel vs. serial with cache)
 - When to move the data from the prefetch buffer to cache
 - How to size the prefetch buffer
 - Keeping the prefetch buffer coherent
- Many modern systems place prefetched data into the cache
 - Intel Pentium 4, Core2's, AMD systems, IBM POWER4,5,6, ...

Challenges in Prefetching: Where (II)

- Which level of cache to prefetch into?
 - Memory to L2, memory to L1. Advantages/disadvantages?
 - L2 to L1? (a separate prefetcher between levels)
- Where to place the prefetched data in the cache?
 - Do we treat prefetched blocks the same as demand-fetched blocks?
 - Prefetched blocks are not known to be needed
 - With LRU, a demand block is placed into the MRU position
- Do we skew the replacement policy such that it favors the demand-fetched blocks?
 - E.g., place all prefetches into the LRU position in a way?

Challenges in Prefetching: Where (III)

- **Where** to place the hardware prefetcher in the memory hierarchy?
 - ❑ In other words, what access patterns does the prefetcher see?
 - ❑ L1 hits and misses
 - ❑ L1 misses only
 - ❑ L2 misses only
- Seeing a more complete access pattern:
 - + Potentially better **accuracy** and **coverage** in prefetching
 - Prefetcher needs to examine more requests (bandwidth intensive, more ports into the prefetcher?)

Challenges in Prefetching: How

- **Software** prefetching
 - ❑ ISA provides prefetch instructions
 - ❑ Programmer or compiler inserts prefetch instructions (effort)
 - ❑ Usually works well only for “regular access patterns”
- **Hardware** prefetching
 - ❑ Hardware monitors processor accesses
 - ❑ Memorizes or finds patterns/strides
 - ❑ Generates prefetch addresses automatically
- **Execution-based** prefetchers
 - ❑ A “thread” is executed to prefetch data for the main program
 - ❑ Can be generated by either software/programmer or hardware

Software Prefetching (I)

- Idea: Compiler/programmer places prefetch instructions into appropriate places in code
- Mowry et al., “Design and Evaluation of a Compiler Algorithm for Prefetching,” ASPLOS 1992.
- Prefetch instructions prefetch data into caches
- Compiler or programmer can insert such instructions into the program

X86 PREFETCH Instruction

PREFETCHh—Prefetch Data Into Caches


Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
OF 18 /1	PREFETCHT0 <i>m8</i>	Valid	Valid	Move data from <i>m8</i> closer to the processor using T0 hint.
OF 18 /2	PREFETCHT1 <i>m8</i>	Valid	Valid	Move data from <i>m8</i> closer to the processor using T1 hint.
OF 18 /3	PREFETCHT2 <i>m8</i>	Valid	Valid	Move data from <i>m8</i> closer to the processor using T2 hint.
OF 18 /0	PREFETCHNTA <i>m8</i>	Valid	Valid	Move data from <i>m8</i> closer to the processor using NTA hint.

Description

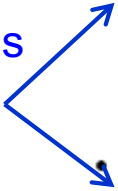
Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by a locality hint:

- T0 (temporal data)—prefetch data into all levels of the cache hierarchy.
 - Pentium III processor—1st- or 2nd-level cache.
 - Pentium 4 and Intel Xeon processors—2nd-level cache.
- T1 (temporal data with respect to first level cache)—prefetch data into level 2 cache and higher.
 - Pentium III processor—2nd-level cache.
 - Pentium 4 and Intel Xeon processors—2nd-level cache.
- T2 (temporal data with respect to second level cache)—prefetch data into level 2 cache and higher.
 - Pentium III processor—2nd-level cache.
 - Pentium 4 and Intel Xeon processors—2nd-level cache.
- NTA (non-temporal data with respect to all cache levels)—prefetch data into non-temporal cache structure and into a location close to the processor, minimizing cache pollution.
 - Pentium III processor—1st-level cache
 - Pentium 4 and Intel Xeon processors—2nd-level cache

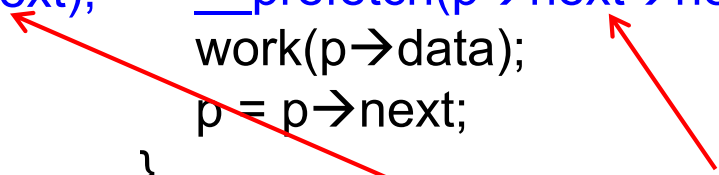
microarchitecture
dependent
specification



different instructions
for different cache
levels



Software Prefetching (II)

<pre>for (i=0; i<N; i++) { __prefetch(a[i+8]); __prefetch(b[i+8]); sum += a[i]*b[i]; }</pre>	<pre>while (p) { __prefetch(p→next); work(p→data); p = p→next; }</pre>	<pre>while (p) { __prefetch(p→next→next→next); work(p→data); p = p→next; }</pre>
		
		<p>Which one is better?</p>

- Can work for very regular array-based access patterns. Issues:
 - Prefetch instructions take up processing/execution bandwidth
 - **How early to prefetch?** Determining this is difficult
 - Prefetch distance depends on hardware implementation (memory latency, cache size, time between loop iterations) → portability?
 - Going too far back in code reduces accuracy (branches in between)
 - Need “special” prefetch instructions in ISA?
 - Alpha load into register 31 treated as prefetch (r31==0)
 - PowerPC *dcbt* (data cache block touch) instruction
 - Not easy to do for pointer-based data structures

Software Prefetching (III)

- Where should a compiler insert prefetches?
 - Prefetch for every load access?
 - Too bandwidth intensive (both memory and execution bandwidth)
 - Profile the code and determine loads that are likely to miss
 - What if profile input set is not representative?
 - How far ahead before the miss should the prefetch be inserted?
 - Profile and determine probability of use for various prefetch distances from the miss
 - What if profile input set is not representative?
 - Usually need to insert a prefetch far in advance to cover 100s of cycles of main memory latency → reduced accuracy

Hardware Prefetching (I)

- Idea: Specialized hardware observes load/store access patterns and prefetches data based on past access behavior
- Tradeoffs:
 - + Can be tuned to system implementation
 - + Does not waste instruction execution bandwidth
 - More hardware complexity to detect patterns
 - Software can be more efficient in some cases

Next-Line Prefetchers

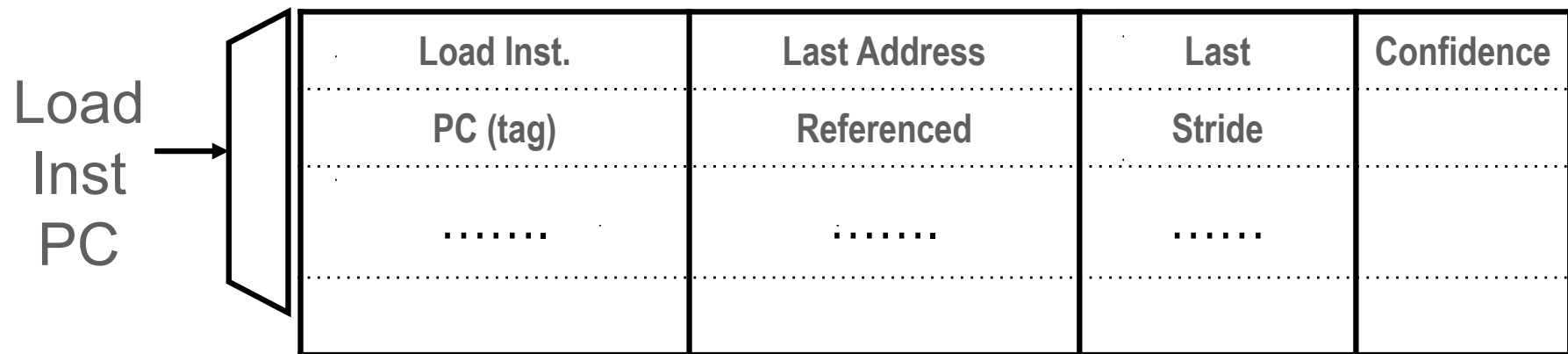
- Simplest form of hardware prefetching: always prefetch next N cache lines after a demand access (or a demand miss)
 - Next-line prefetcher (or next sequential prefetcher)
 - Tradeoffs:
 - + Simple to implement. No need for sophisticated pattern detection
 - + Works well for sequential/streaming access patterns (instructions?)
 - Can waste bandwidth with irregular patterns
 - And, even regular patterns:
 - What is the prefetch accuracy if access stride = 2 and $N = 1$?
 - What if the program is traversing memory from higher to lower addresses?
 - Also prefetch “previous” N cache lines?

Stride Prefetchers

- Two kinds
 - Instruction program counter (PC) based
 - Cache block address based

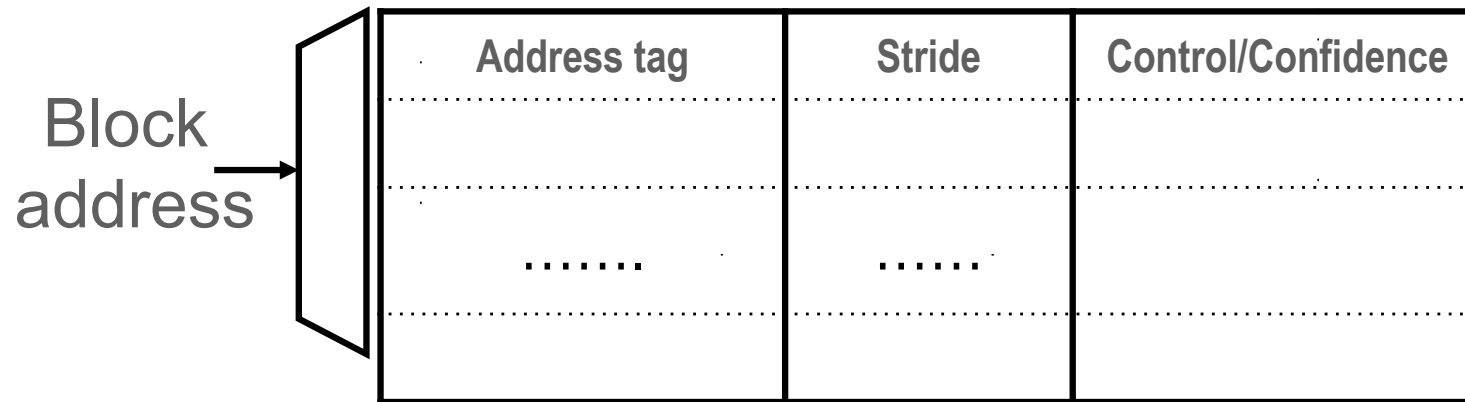
- Instruction based:
 - Baer and Chen, “An effective on-chip preloading scheme to reduce data access penalty,” SC 1991.
 - Idea:
 - Record the distance between the memory addresses referenced by a load instruction (i.e. stride of the load) as well as the last address referenced by the load
 - Next time the same load instruction is fetched, prefetch **last address + stride**

Instruction Based Stride Prefetching



- What is the problem with this?
 - How far can the prefetcher get ahead of the demand access stream?
 - Initiating the prefetch when the load is fetched the next time can be too late
 - Load will access the data cache soon after it is fetched!
 - Solutions:
 - Use **lookahead PC** to index the prefetcher table (**decouple frontend of the processor from backend**)
 - Prefetch ahead (**last address + N*stride**)
 - Generate **multiple prefetches**

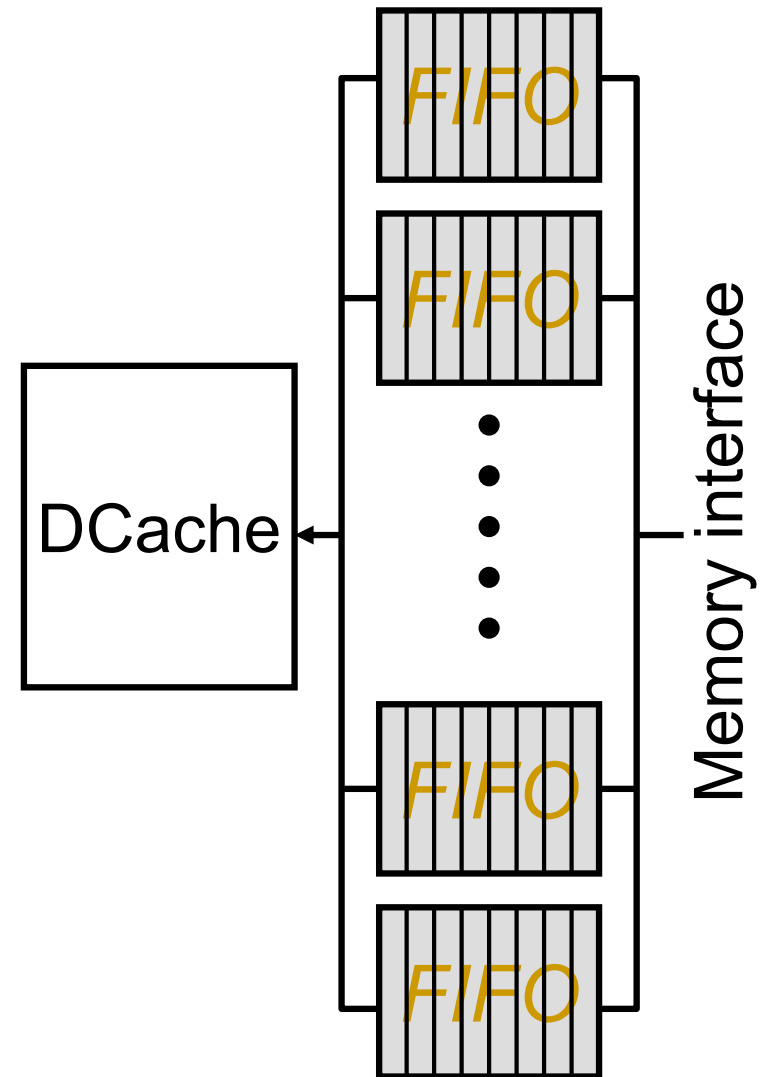
Cache-Block Address Based Stride Prefetching



- Can detect
 - $A, A+N, A+2N, A+3N, \dots$
 - **Stream buffers** are a special case of cache block address based stride prefetching where $N = 1$

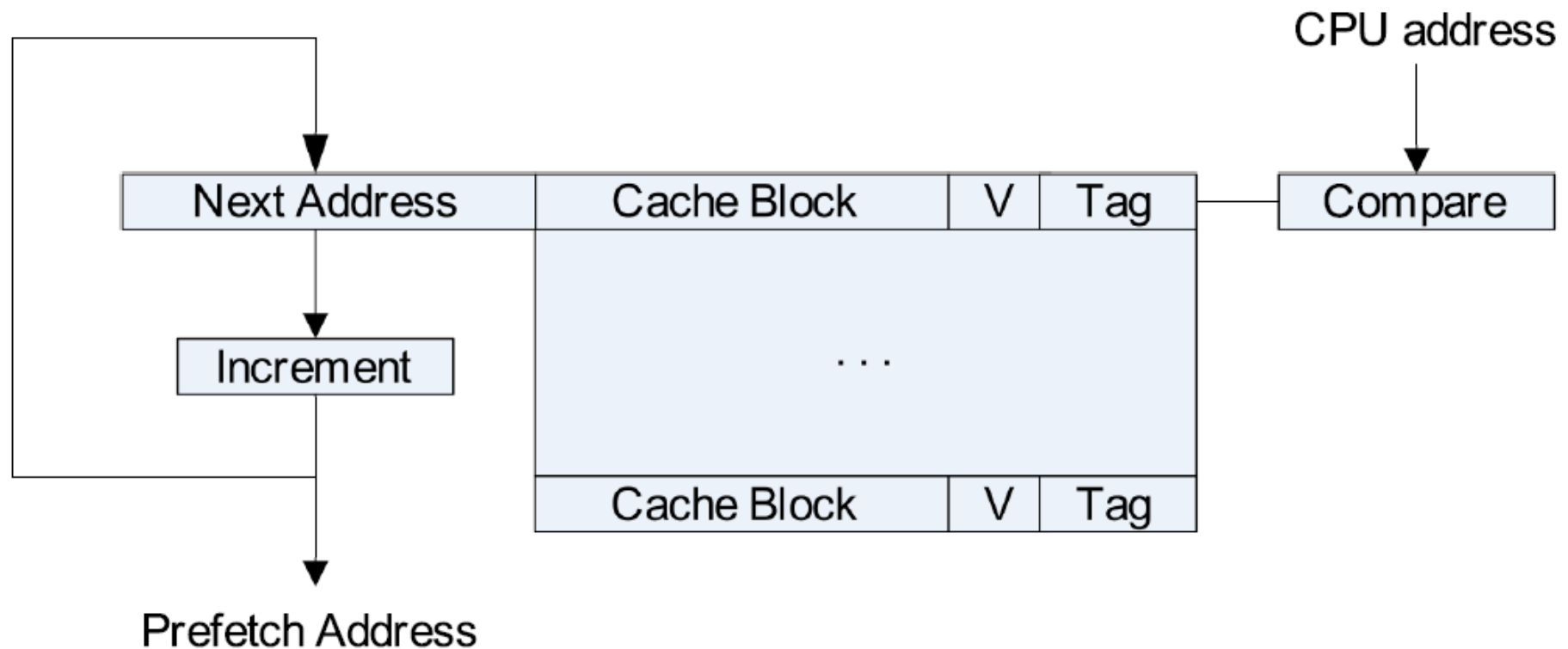
Stream Buffers (Jouppi, ISCA 1990)

- Each stream buffer holds one stream of sequentially prefetched cache lines
- On a load miss check the head of all stream buffers for an address match
 - if hit, pop the entry from FIFO, update the cache with data
 - if not, allocate a new stream buffer to the new miss address (may have to replace a stream buffer following LRU policy)
- Stream buffer FIFOs are continuously topped-off with subsequent cache lines whenever there is room and the bus is not busy

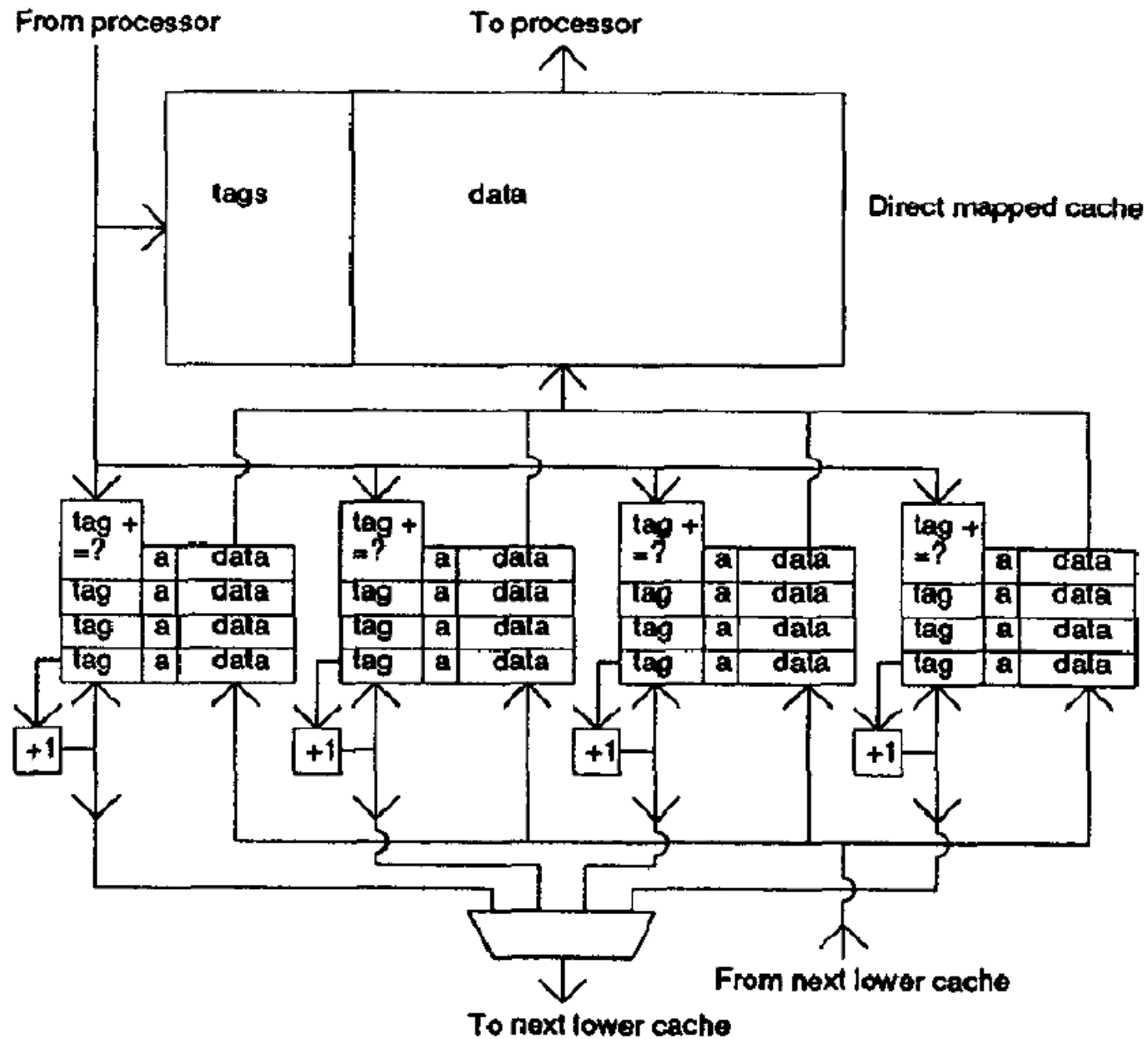


Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," ISCA 1990.

Stream Buffer Design



Stream Buffer Design



Tradeoffs in Stride Prefetching

- Instruction based stride prefetching vs. cache block address based stride prefetching
- The latter can exploit strides that occur due to the **interaction of multiple instructions**
- The latter can more easily get **further ahead** of the processor access stream
 - No need for lookahead PC
- The latter is more hardware intensive
 - Usually there are more data addresses to monitor than instructions

Locality Based Prefetchers

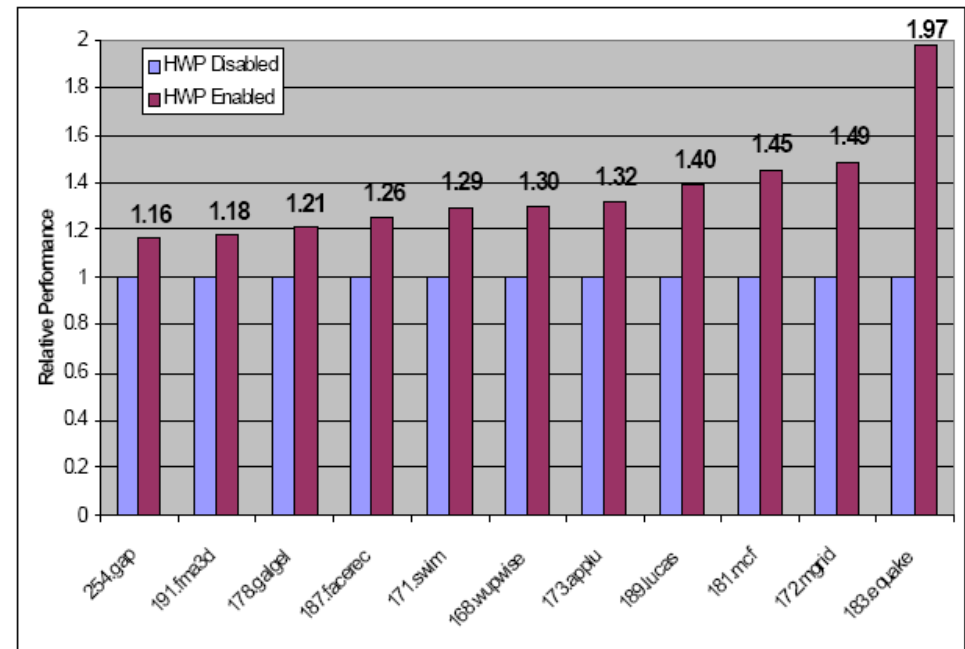
- In many applications access patterns are not perfectly strided
 - Some patterns look random to closeby addresses
 - How do you capture such accesses?
- Locality based prefetching
 - Srinath et al., “Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers“, HPCA 2007.

Pentium 4 (Like) Prefetcher (Srinath et al., HPCA 2007)

- Multiple tracking entries for a range of addresses
- **Invalid:** The tracking entry is not allocated a stream to keep track of. Initially, all tracking entries are in this state.
- **Allocated:** A demand (i.e. load/store) L2 miss allocates a tracking entry if the demand miss does not find any existing tracking entry for its cache-block address.
- **Training:** The prefetcher trains the direction (ascending or descending) of the stream based on the next two L2 misses that occur ± 16 cache blocks from the first miss. If the next two accesses in the stream are to ascending (descending) addresses, the direction of the tracking entry is set to 1 (0) and the entry transitions to *Monitor and Request state*.
- **Monitor and Request:** The tracking entry monitors the accesses to a memory region from a *start pointer (address A)* to an *end pointer (address P)*. The maximum distance between the start pointer and the end pointer is determined by *Prefetch Distance*, which indicates how far ahead of the demand access stream the prefetcher can send requests. If there is a demand L2 cache access to a cache block in the monitored memory region, the prefetcher requests cache blocks $[P+1, \dots, P+N]$ as prefetch requests (assuming the direction of the tracking entry is set to 1). N is called the *Prefetch Degree*. After sending the prefetch requests, the tracking entry starts monitoring the memory region between addresses $A+N$ to $P+N$ (i.e. effectively it moves the tracked memory region by N cache blocks).

Limitations of Locality-Based Prefetchers

- Bandwidth intensive
 - Why?
 - Can be fixed by
 - Stride detection
 - Feedback mechanisms



- Limited to prefetching closeby addresses
 - What about large jumps in addresses accessed?
- However, they work very well in real life
 - Single-core systems
 - Boggs et al., Intel Technology Journal, Feb 2004.

Prefetcher Performance (I)

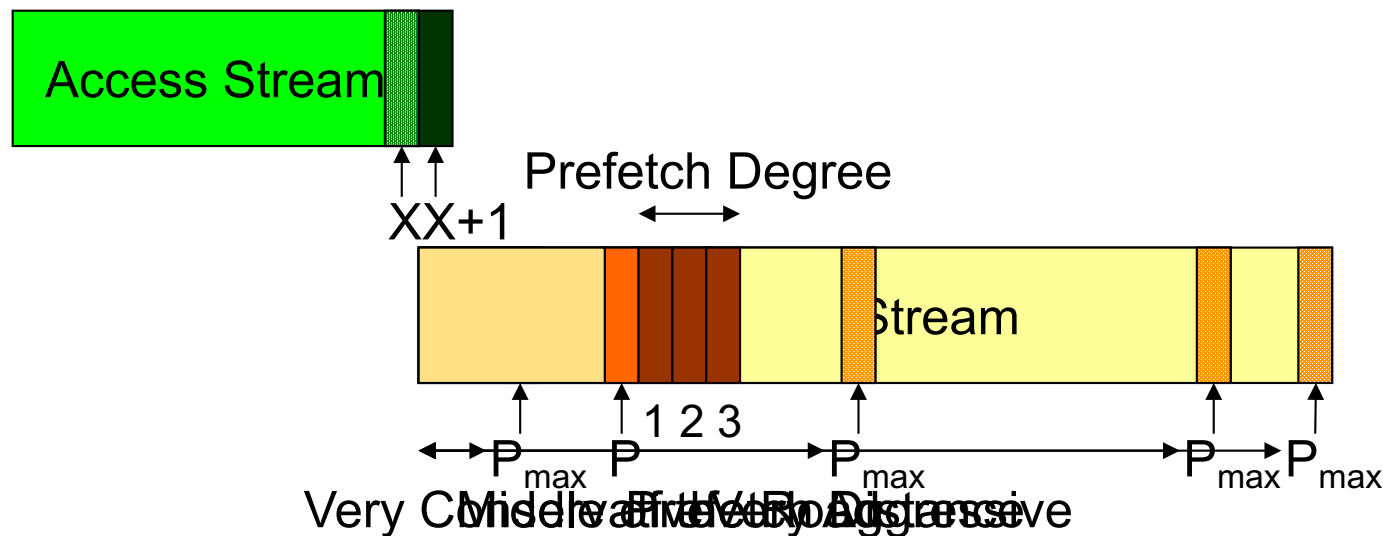
- **Accuracy** (used prefetches / sent prefetches)
- **Coverage** (prefetched misses / all misses)
- **Timeliness** (on-time prefetches / used prefetches)

- **Bandwidth consumption**
 - Memory bandwidth consumed with prefetcher / without prefetcher
 - Good news: **Can utilize idle bus bandwidth (if available)**

- **Cache pollution**
 - Extra demand misses due to prefetch placement in cache
 - More difficult to quantify but affects performance

Prefetcher Performance (II)

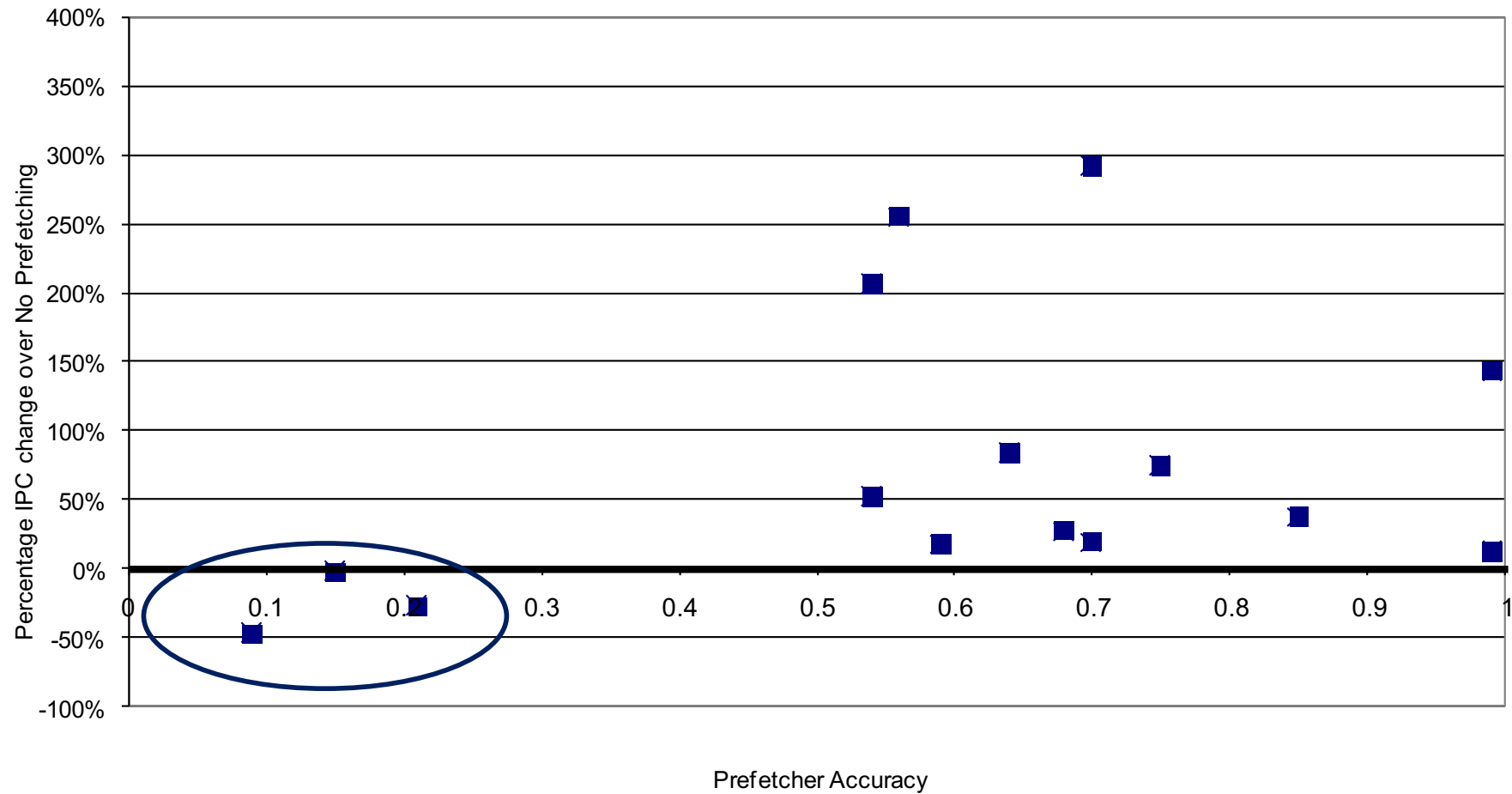
- Prefetcher aggressiveness affects all performance metrics
- Aggressiveness dependent on prefetcher type
- For most hardware prefetchers:
 - **Prefetch distance**: how far ahead of the demand stream
 - **Prefetch degree**: how many prefetches per demand access



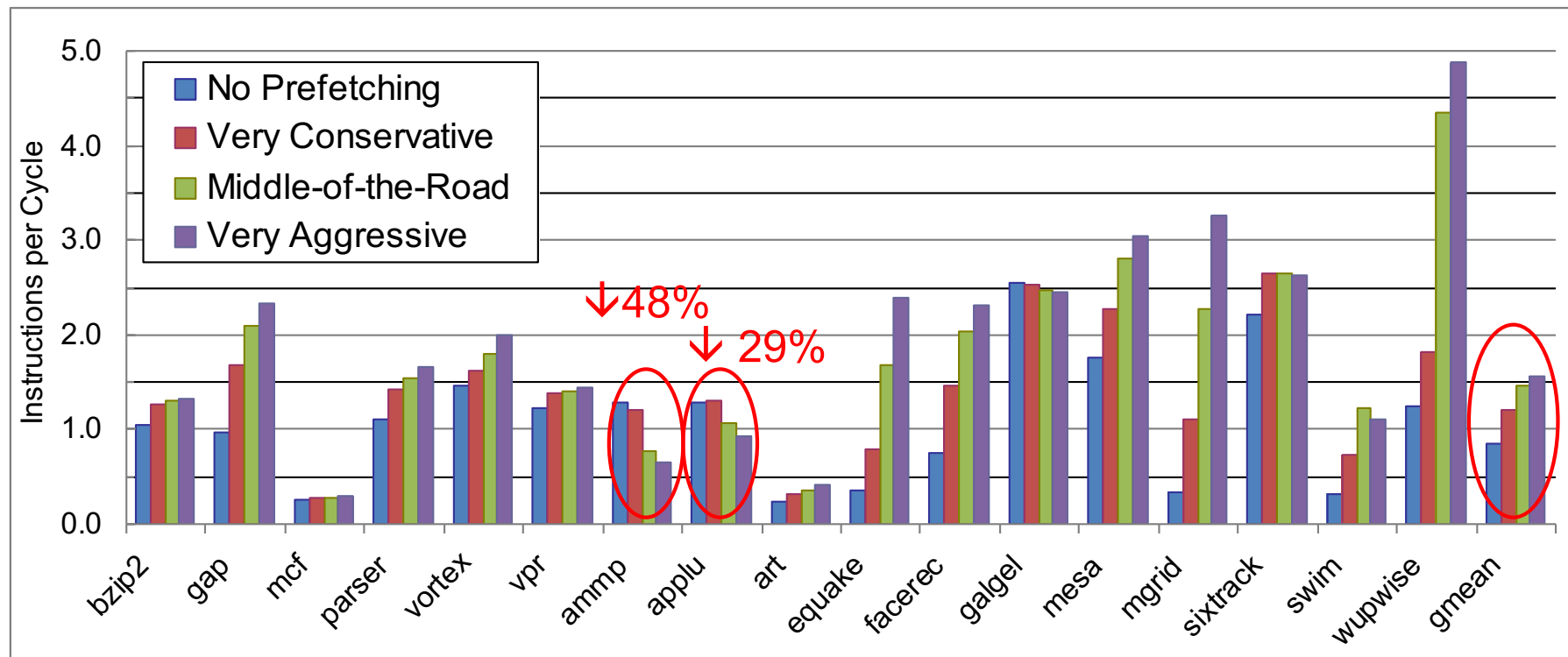
Prefetcher Performance (III)

- How do these metrics interact?
- **Very Aggressive Prefetcher** (large prefetch distance & degree)
 - Well ahead of the load access stream
 - Hides memory access latency better
 - More speculative
 - + Higher coverage, better timeliness
 - Likely lower accuracy, higher bandwidth and pollution
- **Very Conservative Prefetcher** (small prefetch distance & degree)
 - Closer to the load access stream
 - Might not hide memory access latency completely
 - Reduces potential for cache pollution and bandwidth contention
 - + Likely higher accuracy, lower bandwidth, less polluting
 - Likely lower coverage and less timely

Prefetcher Performance (IV)



Prefetcher Performance (V)

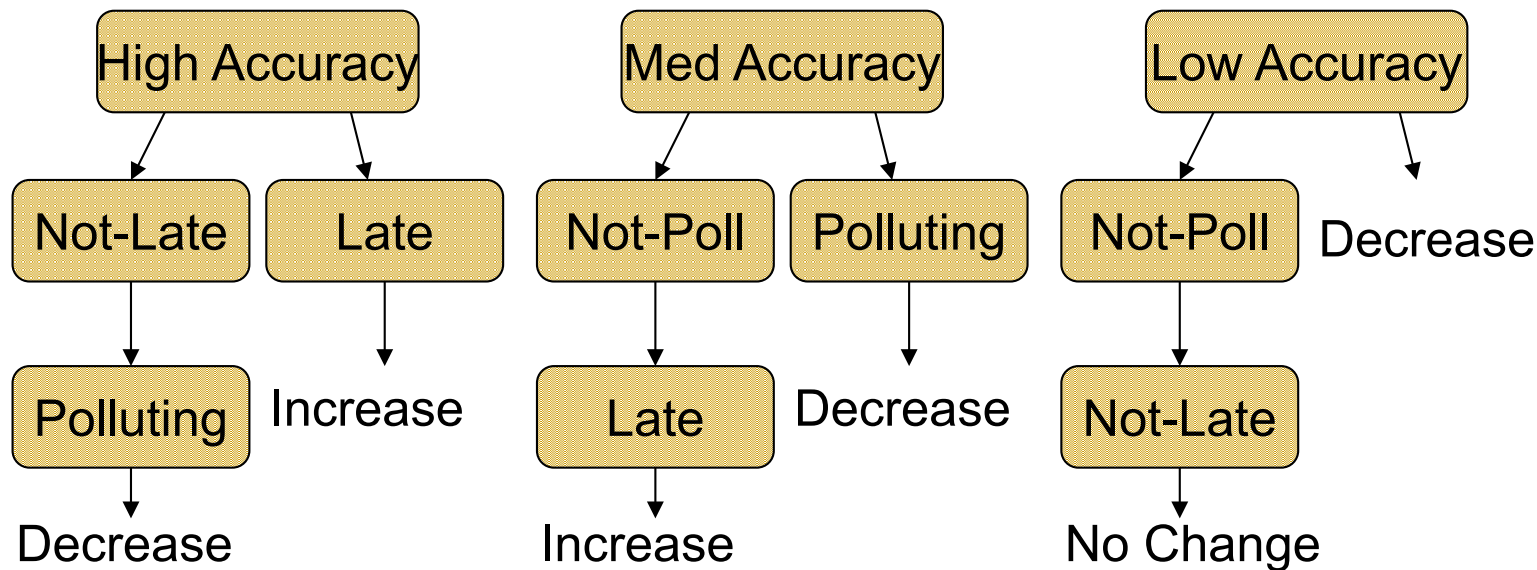


- Srinath et al., “Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers”, HPCA 2007.

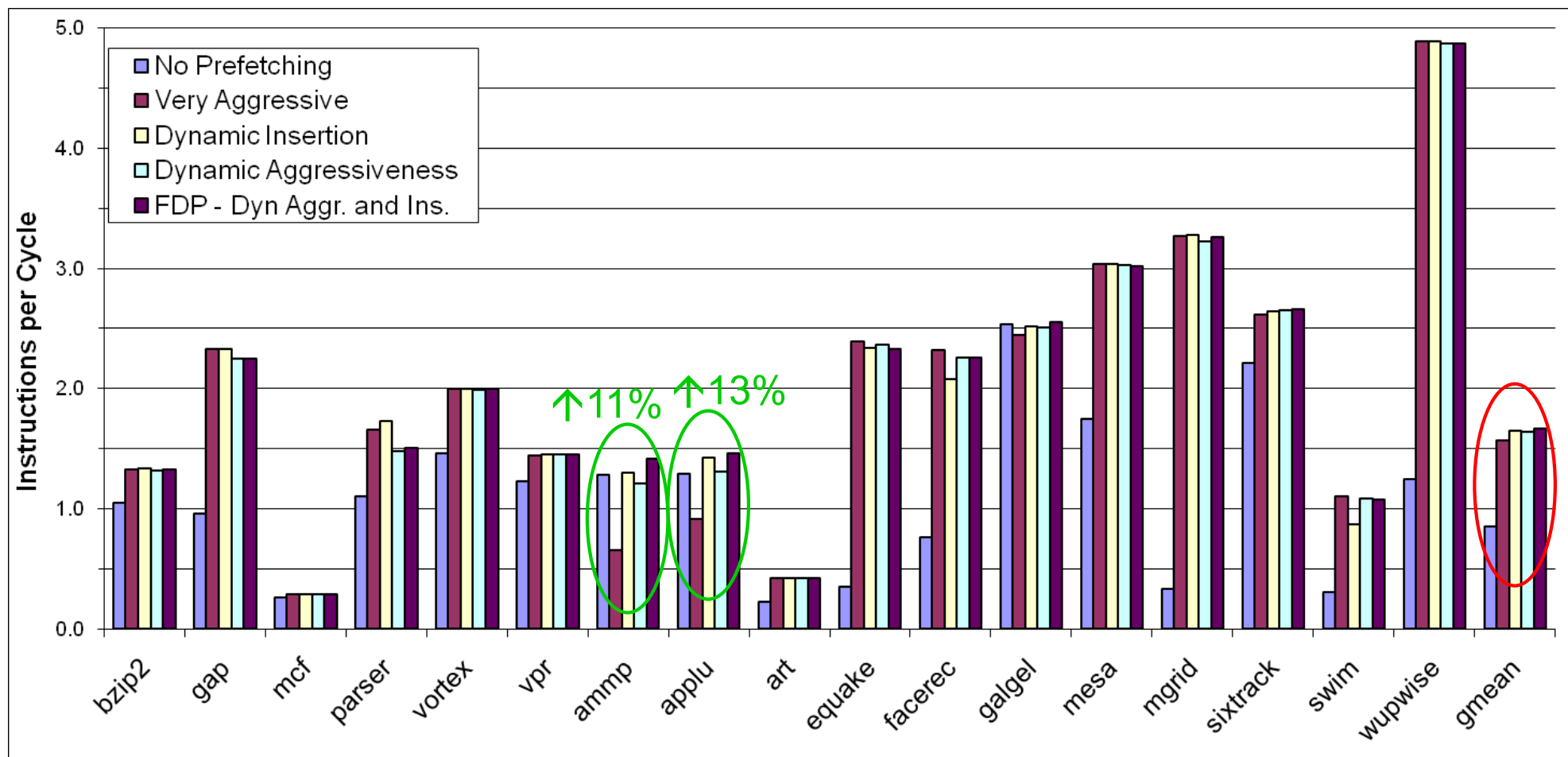
Feedback-Directed Prefetcher Throttling (I)

■ Idea:

- Dynamically monitor prefetcher performance metrics
- Throttle the prefetcher aggressiveness up/down based on past performance
- Change the location prefetches are inserted in cache based on past performance



Feedback-Directed Prefetcher Throttling (II)



- Srinath et al., “Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers”, HPCA 2007.

Feedback-Directed Prefetcher Throttling (III)

- BPKI - Memory Bus Accesses per 1000 retired Instructions
 - Includes effects of L2 demand misses as well as pollution induced misses and prefetches
- A measure of bus bandwidth usage

	No. Pref.	Very Cons	Mid	Very Aggr	FDP
IPC	0.85	1.21	1.47	1.57	1.67
BPKI	8.56	9.34	10.60	13.38	10.88

More on Feedback Directed Prefetching

- Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt,
"Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers"
Proceedings of the 13th International Symposium on High-Performance Computer Architecture (HPCA), pages 63-74, Phoenix, AZ, February 2007. [Slides \(ppt\)](#)

Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers

Santhosh Srinath^{†‡} Onur Mutlu[§] Hyesoon Kim[‡] Yale N. Patt[‡]

[†]Microsoft
ssri@microsoft.com

[§]Microsoft Research
onur@microsoft.com

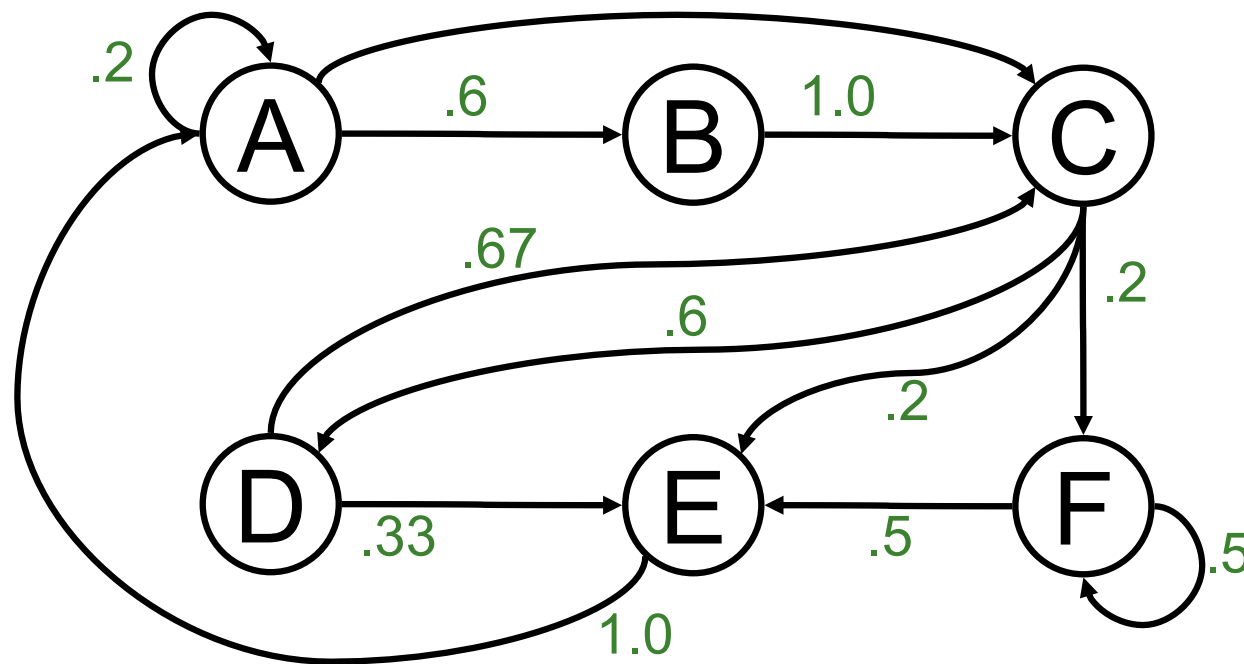
[‡]Department of Electrical and Computer Engineering
The University of Texas at Austin
{santhosh, hyesoon, patt}@ece.utexas.edu

How to Prefetch More Irregular Access Patterns?

- Regular patterns: Stride, stream prefetchers do well
- More irregular access patterns
 - Indirect array accesses
 - Linked data structures
 - Multiple regular strides (1,2,3,1,2,3,1,2,3,...)
 - Random patterns?
 - Generalized prefetcher for all patterns?
- Correlation based prefetchers
- Content-directed prefetchers
- Precomputation or execution-based prefetchers

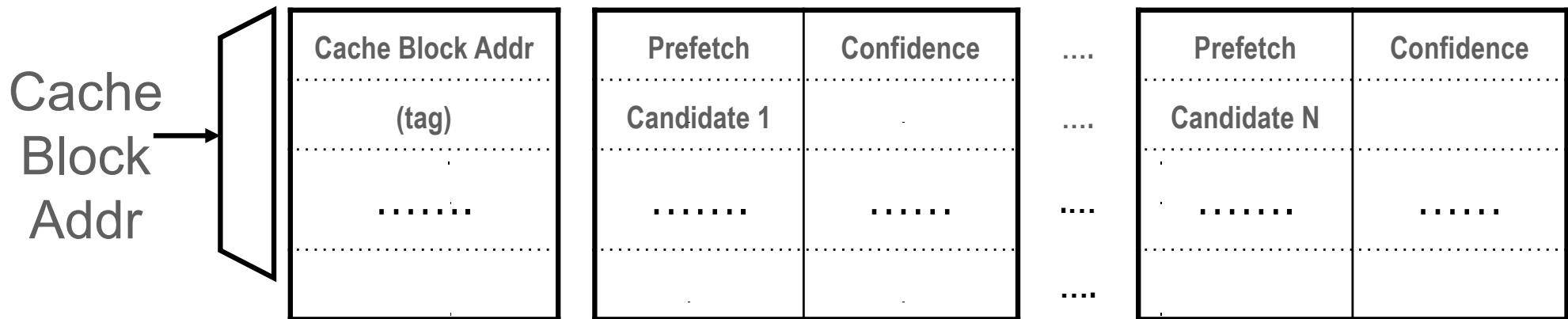
Address Correlation Based Prefetching (I)

- Consider the following history of cache block addresses
A, B, C, D, C, E, A, C, F, F, E, A, A, B, C, D, E, A, B, C, D, C
- After referencing a particular address (say A or E), are some addresses more likely to be referenced next



*Markov
Model*

Address Correlation Based Prefetching (II)



- Idea: Record the likely-next addresses (B, C, D) after seeing an address A
 - Next time A is accessed, prefetch B, C, D
 - A is said to be correlated with B, C, D
- Prefetch up to N next addresses to increase *coverage*
- Prefetch accuracy can be improved by using multiple addresses as key for the next address: (A, B) → (C)
(A,B) correlated with C
- Joseph and Grunwald, “Prefetching using Markov Predictors,” ISCA 1997.
 - Also called “Markov prefetchers”

Address Correlation Based Prefetching (III)

■ Advantages:

- Can cover **arbitrary access patterns**
 - Linked data structures
 - Streaming patterns (though not so efficiently!)

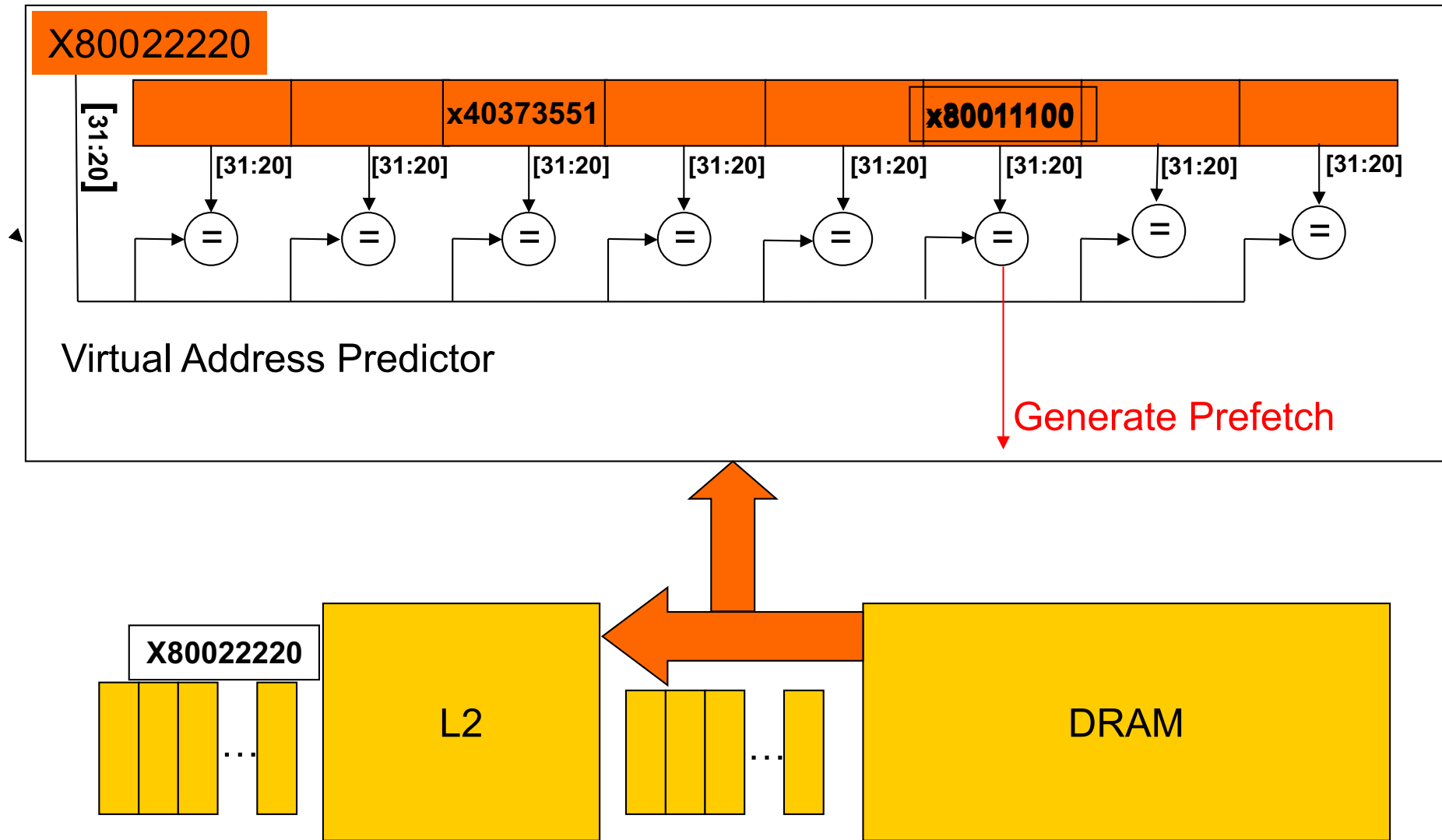
■ Disadvantages:

- **Correlation table** needs to be very large for high coverage
 - Recording every miss address and its subsequent miss addresses is infeasible
- **Can have low timeliness**: Lookahead is limited since a prefetch for the next access/miss is initiated right after previous
- Can consume a lot of **memory bandwidth**
 - Especially when Markov model probabilities (correlations) are low
- Cannot reduce **compulsory misses**

Content Directed Prefetching (I)

- A specialized prefetcher for pointer values
 - Idea: Identify pointers among all values in a fetched cache block and issue prefetch requests for them.
 - Cooksey et al., “A stateless, content-directed data prefetching mechanism,” ASPLOS 2002.
- + No need to memorize/record past addresses!
- + Can eliminate compulsory misses (never-seen pointers)
- Indiscriminately prefetches *all* pointers in a cache block
-
- How to identify pointer addresses:
 - Compare address sized values within cache block with cache block's address → if most-significant few bits match, pointer

Content Directed Prefetching (II)



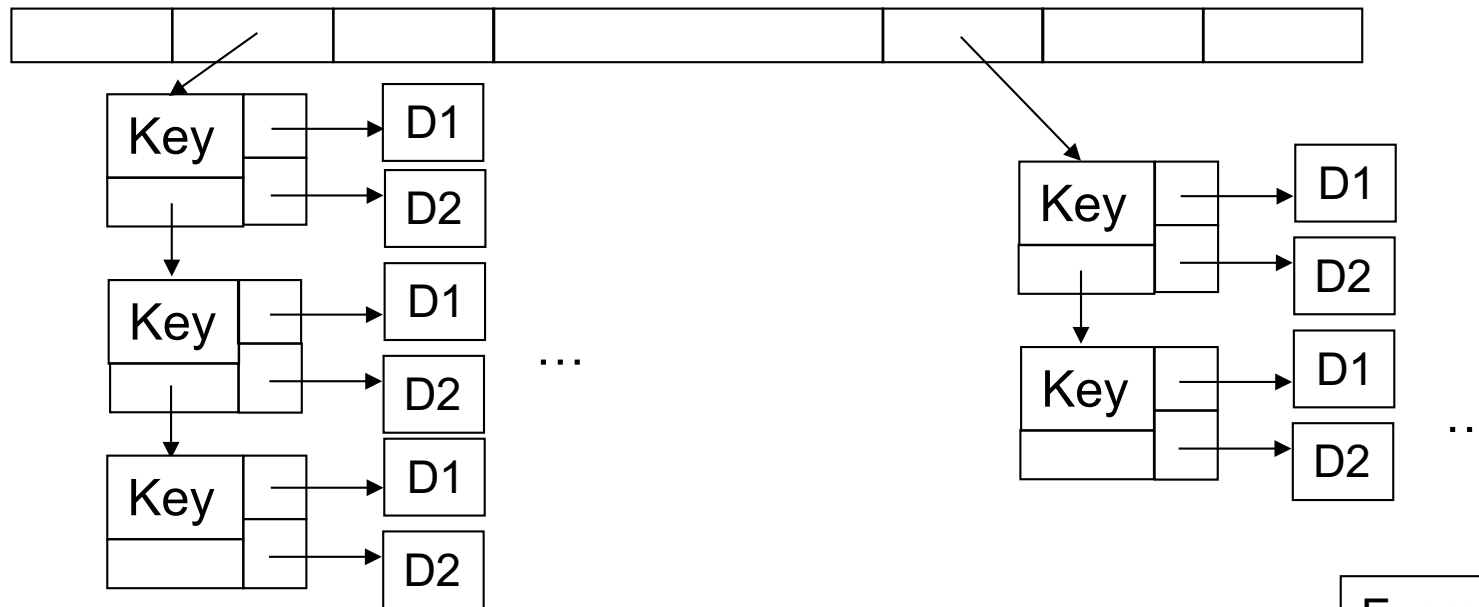
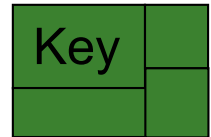
Making Content Directed Prefetching Efficient

- Hardware does not have enough information on pointers
- Software does (and can profile to get more information)
- Idea:
 - **Compiler** profiles and provides hints as to **which pointer addresses are likely-useful to prefetch.**
 - **Hardware** uses hints **to prefetch only likely-useful pointers.**
- Ebrahimi et al., “**Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems,**” HPCA 2009.

Shortcomings of CDP – An example

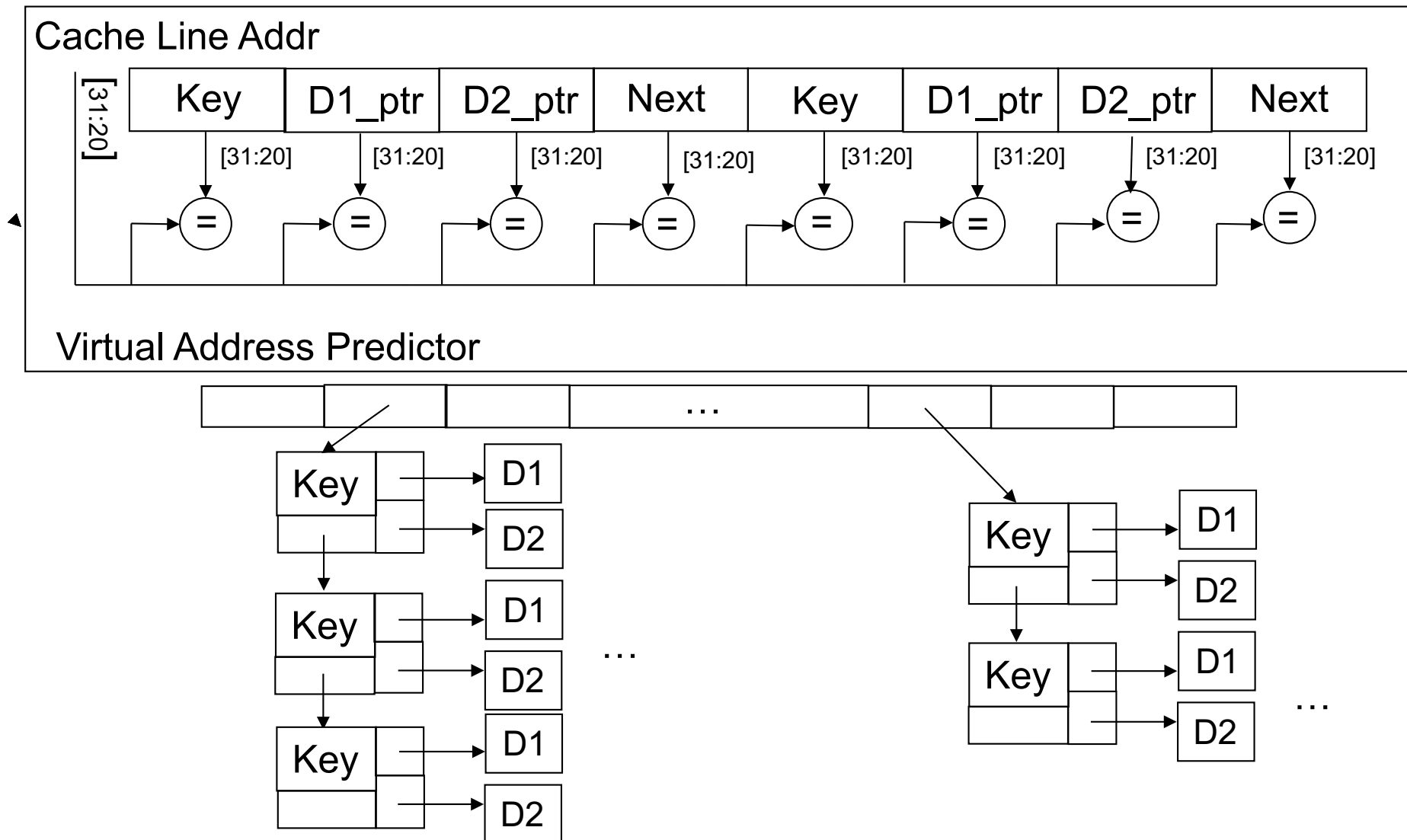
```
HashLookup(int Key) {  
    ...  
    for (node = head ; node -> Key != Key; node = node -> Next; ) ;  
    if (node) return node->D1;  
}
```

```
Struct node{  
    int Key;  
    int * D1_ptr;  
    int * D2_ptr;  
    node * Next;  
}
```



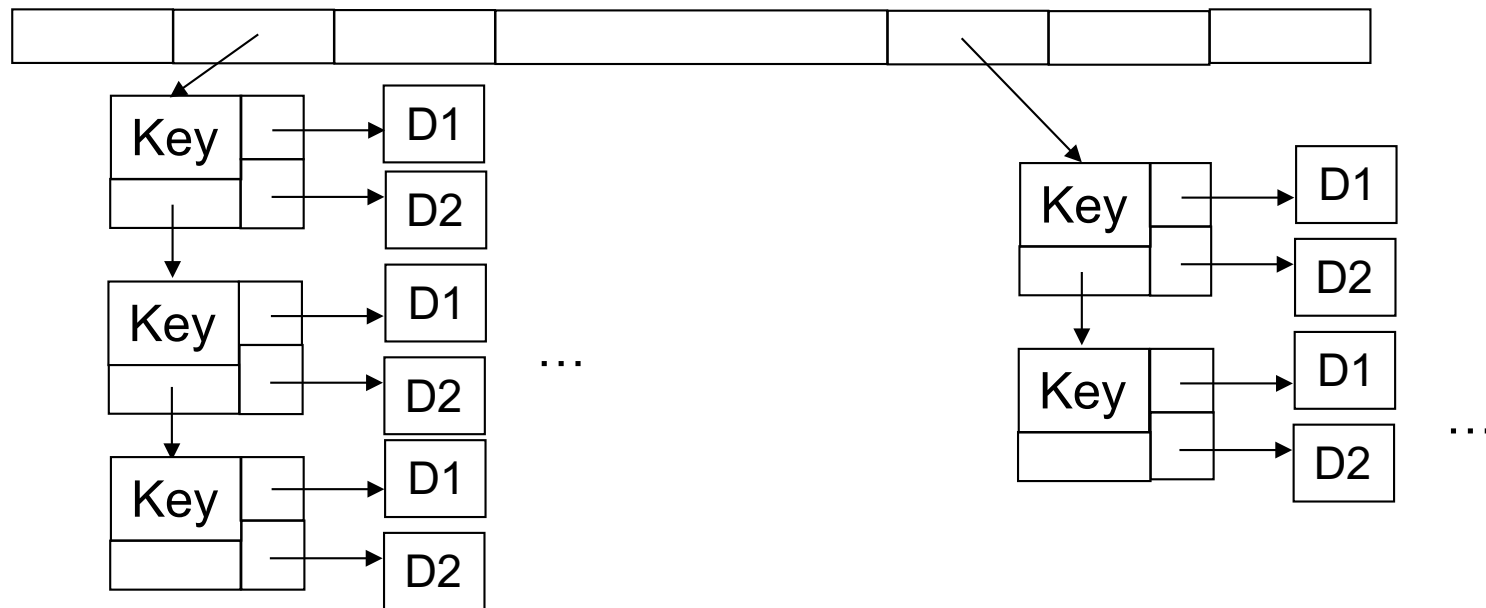
Example from mst

Shortcomings of CDP – An example

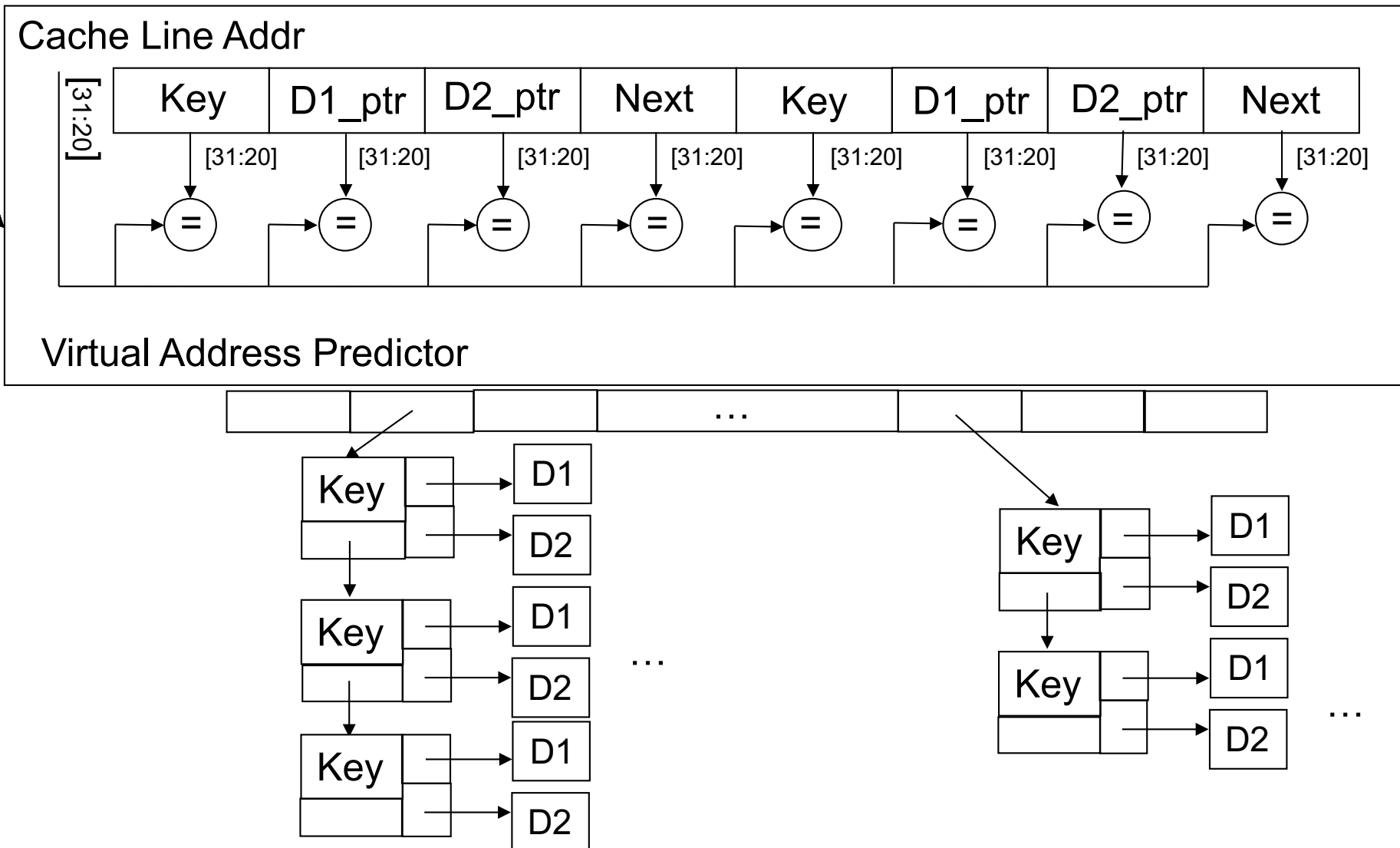


Shortcomings of CDP – An example

```
HashLookup(int Key) {
    ...
    for (node = head ; node -> Key != Key; node = node -> Next; ) ;
    if (node) return node -> D1;
}
```



Shortcomings of CDP – An example



More on Content Directed Prefetching

- Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt,
"Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems"
Proceedings of the 15th International Symposium on High-Performance Computer Architecture (HPCA), pages 7-17, Raleigh, NC, February 2009. [Slides \(ppt\)](#)

Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems

Eiman Ebrahimi[†] Onur Mutlu[§] Yale N. Patt[†]

[†]Department of Electrical and Computer Engineering
The University of Texas at Austin
{ebrahimi, patt}@ece.utexas.edu

[§]Computer Architecture Laboratory (CALCM)
Carnegie Mellon University
onur@cmu.edu

Hybrid Hardware Prefetchers

- Many different access patterns
 - Streaming, striding
 - Linked data structures
 - Localized random
 - Idea: Use multiple prefetchers to cover all patterns
-
- + Better prefetch coverage
 - More complexity
 - More bandwidth-intensive
 - Prefetchers start getting in each other's way (contention, pollution)
 - Need to manage accesses from each prefetcher

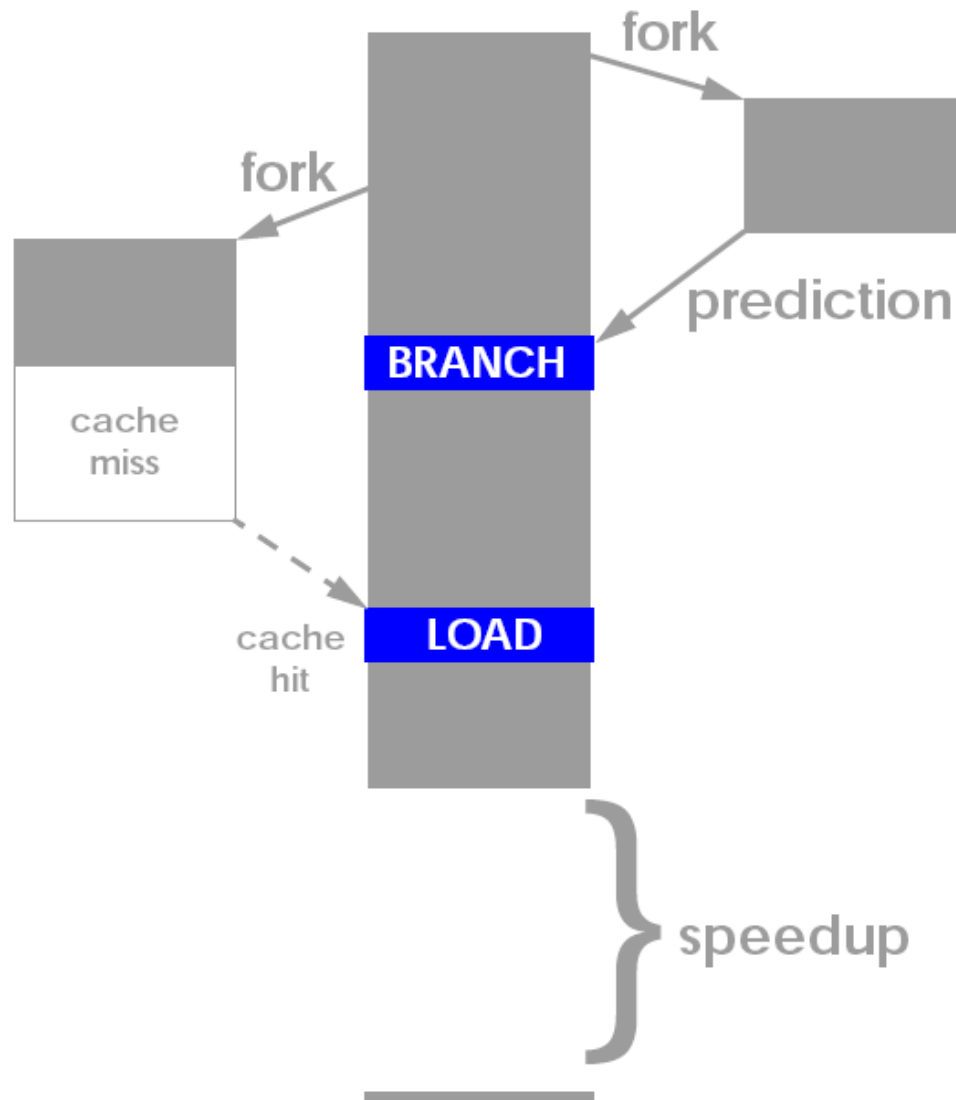
Execution-based Prefetchers (I)

- Idea: Pre-execute a piece of the (pruned) program solely for prefetching data
 - Only need to distill pieces that lead to cache misses
- **Speculative thread:** Pre-executed program piece can be considered a “thread”
- Speculative thread can be executed
 - On a separate processor/core
 - On a separate hardware thread context (think fine-grained multithreading)
 - On the same thread context in idle cycles (during cache misses)

Execution-based Prefetchers (II)

- How to construct the speculative thread:
 - Software based pruning and “spawn” instructions
 - Hardware based pruning and “spawn” instructions
 - Use the original program (no construction), but
 - Execute it faster without stalling and correctness constraints
- Speculative thread
 - Needs to discover misses before the main program
 - Avoid waiting/stalling and/or compute less
 - To get ahead, uses
 - Perform only address generation computation, branch prediction, value prediction (to predict “unknown” values)
 - Purely speculative so there is no need for recovery of main program if the speculative thread is incorrect

Thread-Based Pre-Execution



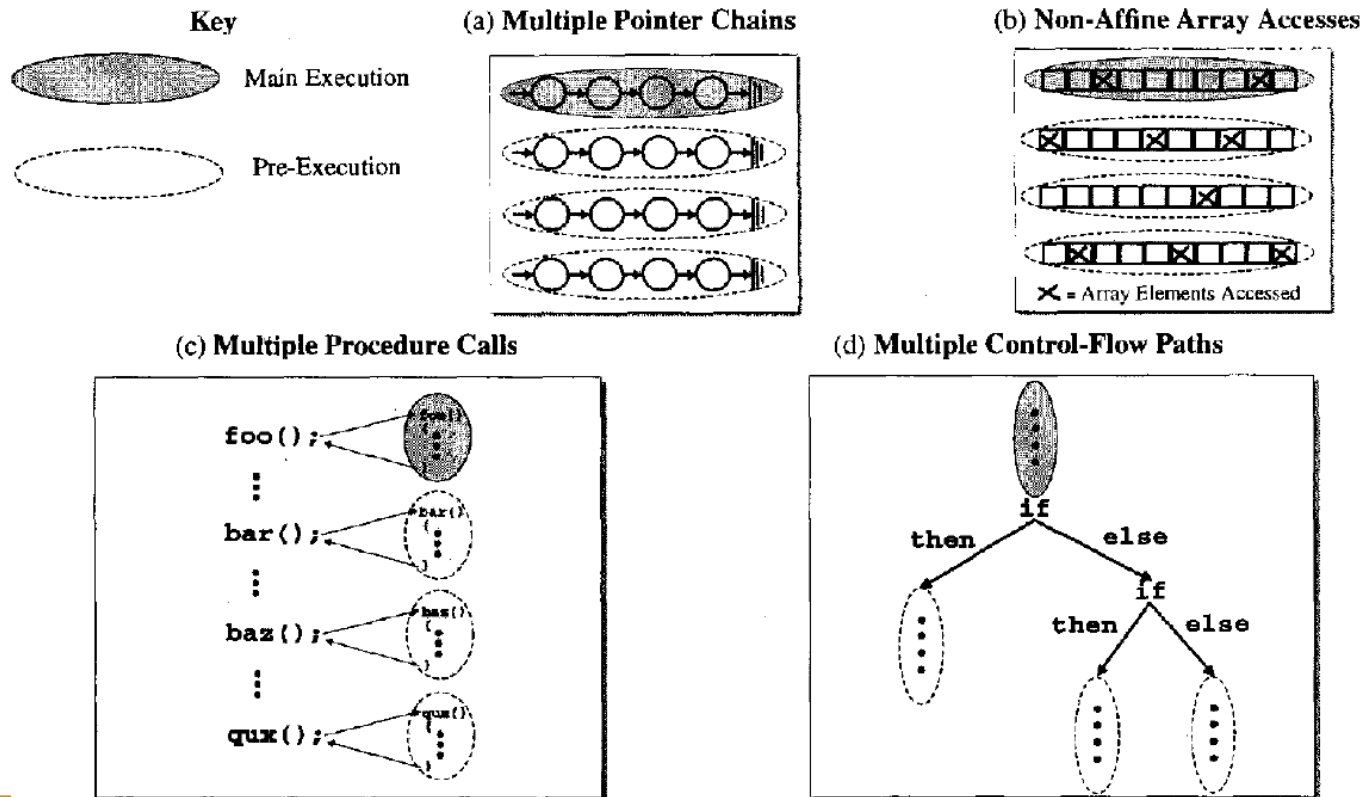
- Dubois and Song, “**Assisted Execution**,” USC Tech Report 1998.
- Chappell et al., “**Simultaneous Subordinate Microthreading (SSMT)**,” ISCA 1999.
- Zilles and Sohi, “**Execution-based Prediction Using Speculative Slices**”, ISCA 2001.

Thread-Based Pre-Execution Issues

- Where to execute the precomputation thread?
 1. Separate core (least contention with main thread)
 2. Separate thread context on the same core (more contention)
 3. Same core, same context
 - When the main thread is stalled
- When to spawn the precomputation thread?
 1. Insert spawn instructions well before the “problem” load
 - How far ahead?
 - Too early: prefetch might not be needed
 - Too late: prefetch might not be timely
 2. When the main thread is stalled
- When to terminate the precomputation thread?
 1. With pre-inserted CANCEL instructions
 2. Based on effectiveness/contention feedback (recall throttling)

Thread-Based Pre-Execution Issues

- What, when, where, how
 - Luk, “Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors,” ISCA 2001.
 - Many issues in software-based pre-execution discussed



An Example

(a) Original Code

```
register int i;
register arc_t *arcout;
for( i < trips; ){
    // loop over "trips" lists
    if (arcout[1].ident != FIXED) {
        ...
        first_of_sparse_list = arcout + 1;
    }
    ...
    arcin = (arc_t *)first_of_sparse_list
        → tail → mark;
    // traverse the list starting with
    // the first node just assigned
    while (arcin) {
        tail = arcin → tail;
        ...
        arcin = (arc_t *)tail → mark;
    }
    i++, arcout+=3;
}
```

(b) Code with Pre-Execution

```
register int i;
register arc_t *arcout;
for( i < trips; ){
    // loop over "trips" lists
    if (arcout[1].ident != FIXED) {
        ...
        first_of_sparse_list = arcout + 1;
    }
    ...
    // invoke a pre-execution starting
    // at END_FOR
    PreExecute_Start(END_FOR);
    arcin = (arc_t *)first_of_sparse_list
        → tail → mark;
    // traverse the list starting with
    // the first node just assigned
    while (arcin) {
        tail = arcin → tail;
        ...
        arcin = (arc_t *)tail → mark;
    }
    // terminate this pre-execution after
    // prefetching the entire list
    PreExecute_Stop();
END_FOR:
    // the target address of the pre-
    // execution
    i++, arcout+=3;
}
// terminate this pre-execution if we
// have passed the end of the for-loop
PreExecute_Stop();
```

The Spec2000 benchmark `mcf` spends roughly half of its execution time in a nested loop which traverses a set of linked lists. An abstract version of this loop is shown in Figure 2(a), in which the for-loop iterates over the lists and the while-loop visits the elements of each list. As we observe from the figure, the first node of each list is assigned by dereferencing the pointer `first_of_sparse_list`, whose value is in fact determined by `arcout`, an induction variable of the for-loop. Therefore, even when we are still working on the current list, the first and the remaining nodes on the next list can be loaded speculatively by pre-executing the next iteration of the for-loop.

Figure 2(b) shows a version of the program with pre-execution code inserted (shown in boldface). **END_FOR** is simply a label to denote the place where `arcout` gets updated. The new instruction **PreExecute_Start(END_FOR)** initiates a pre-execution thread, say T , starting at the PC represented by **END_FOR**. Right after the pre-execution begins, T 's registers that hold the values of `i` and `arcout` will be updated. Then `i`'s value is compared against `trips` to see if we have reached the end of the for-loop. If so, thread T will exit the for-loop and encounters a **PreExecute_Stop()**, which will terminate the pre-execution and free up T for future use. Otherwise, T will continue pre-executing the body of the for-loop, and hence compute the first node of the next list automatically. Finally, after traversing the entire list through the while-loop, the pre-execution will be terminated by another **PreExecute_Stop()**. Notice that any **PreExecute_Start()** instructions encountered during pre-execution are simply ignored as we do not allow nested pre-execution in order to keep our design simple. Similarly, **PreExecute_Stop()** instructions cannot terminate the main thread either.

Figure 2. Abstract versions of an important loop nest in the Spec2000 benchmark `mcf`. Loads that incur many cache misses are underlined.

Example ISA Extensions

Thread_ID = PreExecute_Start(Start_PC, Max_Insts):

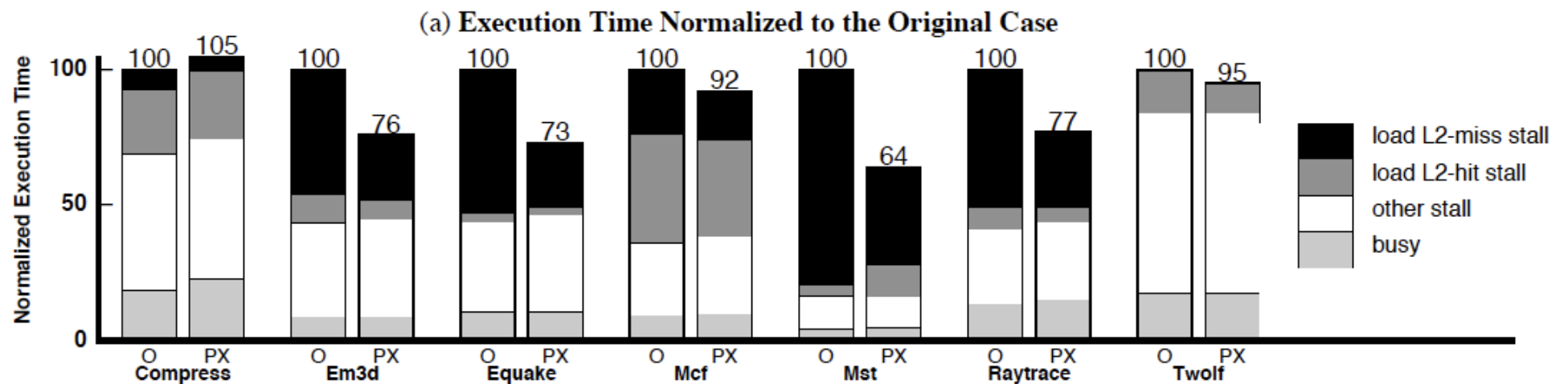
Request for an idle context to start pre-execution at *Start_PC* and stop when *Max_Insts* instructions have been executed; *Thread_ID* holds either the identity of the pre-execution thread or -1 if there is no idle context. This instruction has effect only if it is executed by the main thread.

PreExecute_Stop(): The thread that executes this instruction will be self terminated if it is a pre-execution thread; no effect otherwise.

PreExecute_Cancel(Thread_ID): Terminate the pre-execution thread with *Thread_ID*. This instruction has effect only if it is executed by the main thread.

Figure 4. Proposed instruction set extensions to support pre-execution. (C syntax is used to improve readability.)

Results on a Multithreaded Processor



Luk, “Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors,” ISCA 2001.

Problem Instructions

- Zilles and Sohi, “Execution-based Prediction Using Speculative Slices”, ISCA 2001.
- Zilles and Sohi, “Understanding the backward slices of performance degrading instructions,” ISCA 2000.

*Figure 2. Example problem instructions from heap insertion routine in **vpr**.*

```
struct s_heap **heap; // from [1..heap_size]
int heap_size; // # of slots in the heap
int heap_tail; // first unused slot in heap

void add_to_heap (struct s_heap *hptr) {
    ...
1.  heap[heap_tail] = hptr;
2.  int ifrom = heap_tail;
3.  int ito = ifrom/2;
4.  heap_tail++;
5.  while ((ito >= 1) &&
6.        (heap[ifrom]->cost < heap[ito]->cost))
7.      struct s_heap *temp_ptr = heap[ito];
8.      heap[ito] = heap[ifrom];
9.      heap[ifrom] = temp_ptr;
10.     ifrom = ito;
11.     ito = ifrom/2;
    }
}
```

branch misprediction (arrow pointing to line 6)

cache miss (arrow pointing to line 7)

Fork Point for Prefetching Thread

Figure 3. The **node_to_heap** function, which serves as the fork point for the slice that covers **add_to_heap**.

```
void node_to_heap (... , float cost, ...) {  
    struct s_heap *hptr; ← fork point  
    ...  
    hptr = alloc_heap_data();  
    hptr->cost = cost;  
    ...  
    add_to_heap (hptr);  
}
```


Pre-execution Thread Construction

Figure 4. Alpha assembly for the `add_to_heap` function. The instructions are annotated with the number of the line in Figure 2 to which they correspond. The problem instructions are in bold and the shaded instructions comprise the un-optimized slice.

```
node_to_heap:
... /* skips ~40 instructions */
2  lda    s1, 252(gp)    # &heap_tail
2  ldl    t2, 0(s1)      # ifrom = heap_tail
1  ldq    t5, -76(s1)    # &heap[0]
3  cmplt  t2, 0, t4      # see note
4  addl   t2, 0x1, t6    # heap_tail ++
1  s8addq t2, t5, t3      # &heap[heap_tail]
4  stl    t6, 0(s1)      # store heap_tail
1  stq    s0, 0(t3)      # heap[heap_tail]
3  addl   t2, t4, t4      # see note
3  sra    t4, 0x1, t4     # ito = ifrom/2
5  ble    t4, return    # (ito < 1)
loop:
6  s8addq t2, t5, a0      # &heap[ifrom]
6  s8addq t4, t5, t7      # &heap[ito]
11 cmplt  t4, 0, t9       # see note
10 move   t4, t2          # ifrom = ito
6  ldq    a2, 0(a0)       # heap[ifrom]
6  ldq    a4, 0(t7)       # heap[ito]
11 addl   t4, t9, t9      # see note
11 sra    t9, 0x1, t4      # ito = ifrom/2
6  lds    $f0, 4(a2)      # heap[ifrom]->cost
6  lds    $f1, 4(a4)      # heap[ito]->cost
6  cmpltl $f0,$f1,$f0      # (heap[ifrom]->cost
6  fbeq    $f0, return    # < heap[ito]->cost)
8  stq    a2, 0(t7)       # heap[ito]
9  stq    a4, 0(a0)       # heap[ifrom]
5  bgt    t4, loop        # (ito >= 1)
return:
... /* register restore code & return */
```

note: the divide by 2 operation is implemented by a 3 instruction sequence described in the strength reduction optimization.

Figure 5. Slice constructed for example problem instructions. Much smaller than the original code, the slice contains a loop that mimics the loop in the original code.

```
slice:
1  ldq    $6, 328(gp)    # &heap
2  ldl    $3, 252(gp)    # ito = heap_tail
slice_loop:
3,11 sra   $3, 0x1, $3    # ito /= 2
6  s8addq $3, $6, $16     # &heap[ito]
6  ldq    $18, 0($16)     # heap[ito]
6  lds    $f1, 4($18)     # heap[ito]->cost
6  cmptle $f1,$f17,$f31   # (heap[ito]->cost
                          # < cost) PRED
                          br    slice_loop

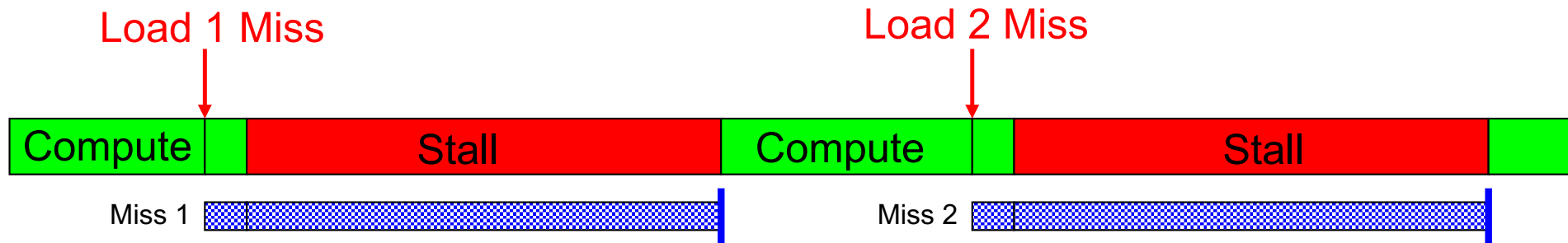
## Annotations
fork: on first instruction of node_to_heap
live-in: $f17<cost>, gp
max loop iterations: 4
```

Review: Runahead Execution

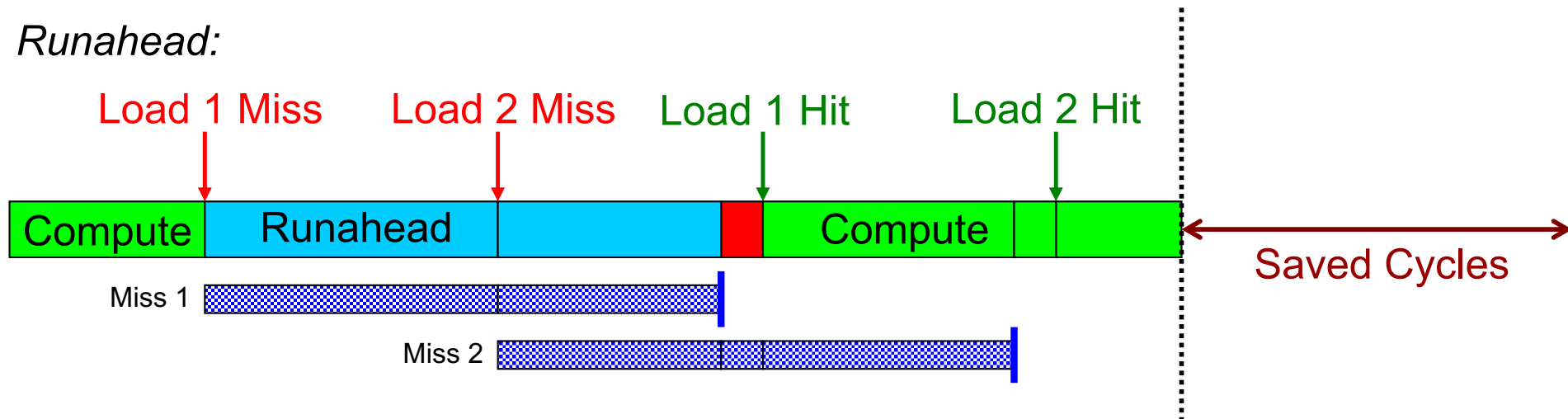
- A simple pre-execution method for prefetching purposes
- When the oldest instruction is a long-latency cache miss:
 - Checkpoint architectural state and enter runahead mode
- In runahead mode:
 - Speculatively pre-execute instructions
 - The purpose of pre-execution is to generate prefetches
 - L2-miss 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.

Review: Runahead Execution (Mutlu et al., HPCA 2003)

Small Window:



Runahead:



Runahead as an Execution-based Prefetcher

- Idea of an Execution-Based Prefetcher: Pre-execute a piece of the (pruned) program solely for prefetching data
- Idea of Runahead: Pre-execute the main program solely for prefetching data
- Advantages and disadvantages of runahead vs. other execution-based prefetchers?
- Can you make runahead even better by pruning the program portion executed in runahead mode?

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?

Execution-based Prefetchers: Pros and Cons

- + Can prefetch pretty much **any access pattern**
- + **Can be very low cost** (e.g., runahead execution)
 - + Especially if it uses the same hardware context
 - + Why? The processor is equipped to execute the program anyway
- + **Can be bandwidth-efficient** (e.g., runahead execution)
- Depend on **branch prediction and possibly value prediction accuracy**
 - Mispredicted branches dependent on missing data throw the thread off the correct execution path
- Can be **wasteful**
 - speculatively execute many instructions
 - can occupy a separate thread context
- Complexity in deciding when and what to pre-execute

Multi-Core Issues in Prefetching

Prefetching in Multi-Core (I)

- Prefetching shared data
 - Coherence misses
- Prefetch efficiency is a lot more important
 - Bus bandwidth more precious
 - Cache space more valuable
- One cores' prefetches interfere with other cores' requests
 - Cache conflicts
 - Bus contention
 - DRAM bank and row buffer contention

Prefetching in Multi-Core (II)

- Two key issues
 - How to prioritize prefetches vs. demands (of different cores)
 - How to control the aggressiveness of multiple prefetchers to achieve high overall performance
- Need to **coordinate the actions of independent prefetchers** for best system performance
 - Each prefetcher has different accuracy, coverage, timeliness

Some Ideas

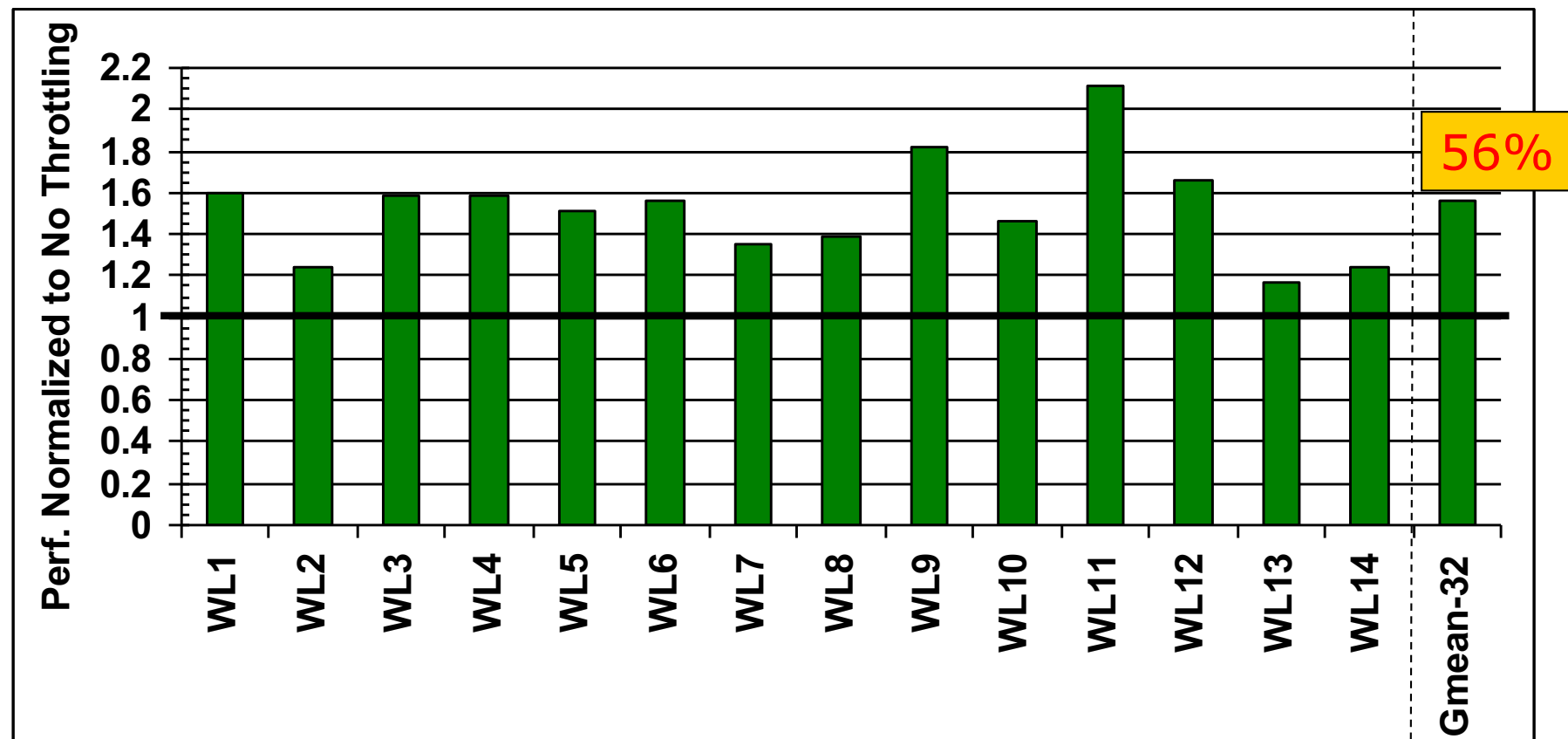
- Controlling prefetcher aggressiveness
 - Feedback directed prefetching [HPCA'07]
 - Coordinated control of multiple prefetchers [MICRO'09]
- How to prioritize prefetches vs. demands from cores
 - Prefetch-aware memory controllers and shared resource management [MICRO'08, ISCA'11]
- Bandwidth efficient prefetching of linked data structures
 - Through hardware/software cooperation (software hints) [HPCA'09]

Motivation

- Aggressive prefetching improves memory latency tolerance of many applications when they run alone
- Prefetching for concurrently-executing applications on a CMP can lead to
 - Significant **system performance** degradation and bandwidth waste
- **Problem:**
Prefetcher-caused inter-core interference
 - Prefetches of one application contend with prefetches and demands of other applications

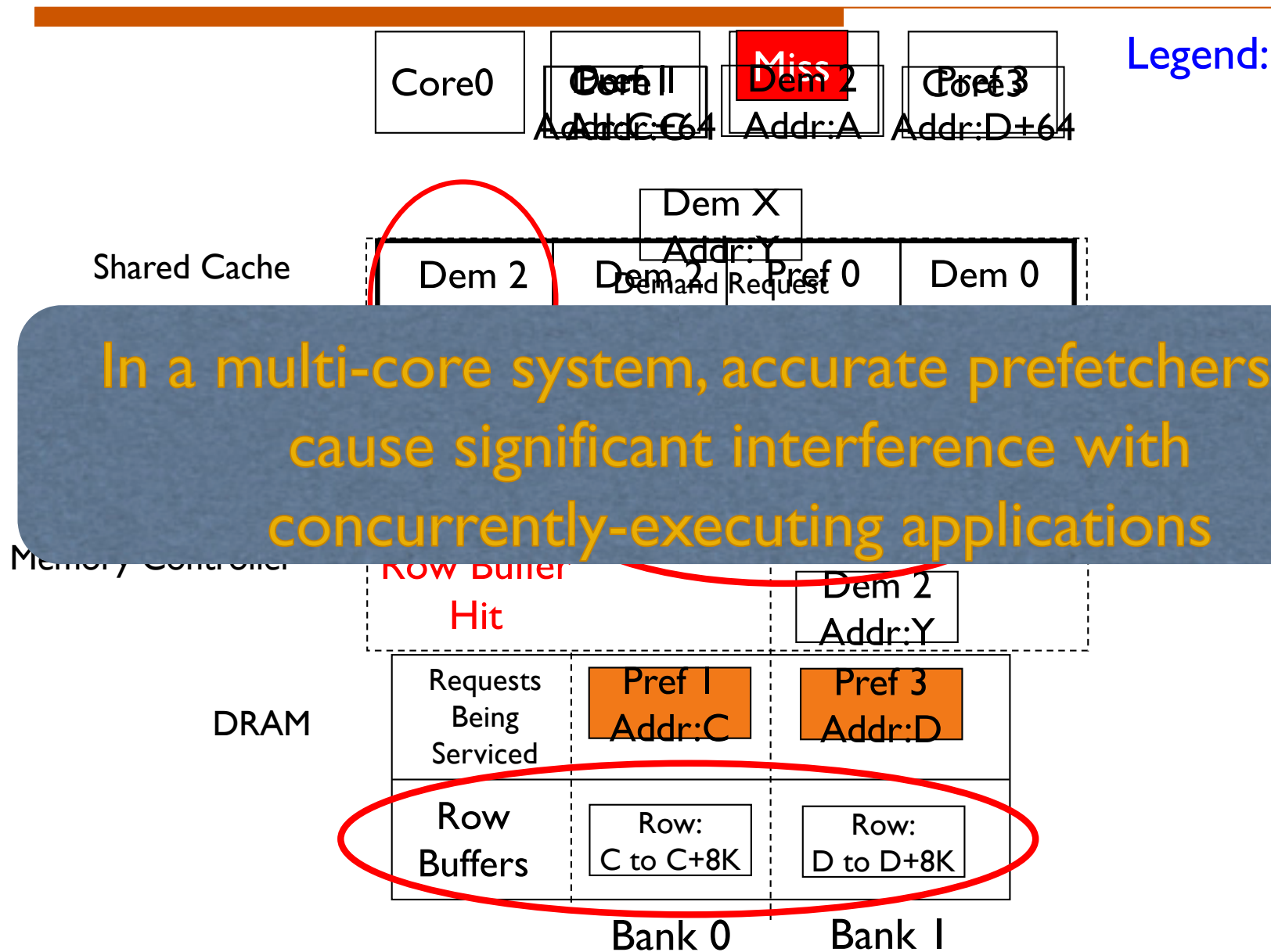
Potential Performance

System performance improvement of *ideally* removing all prefetcher-caused inter-core interference in shared resources

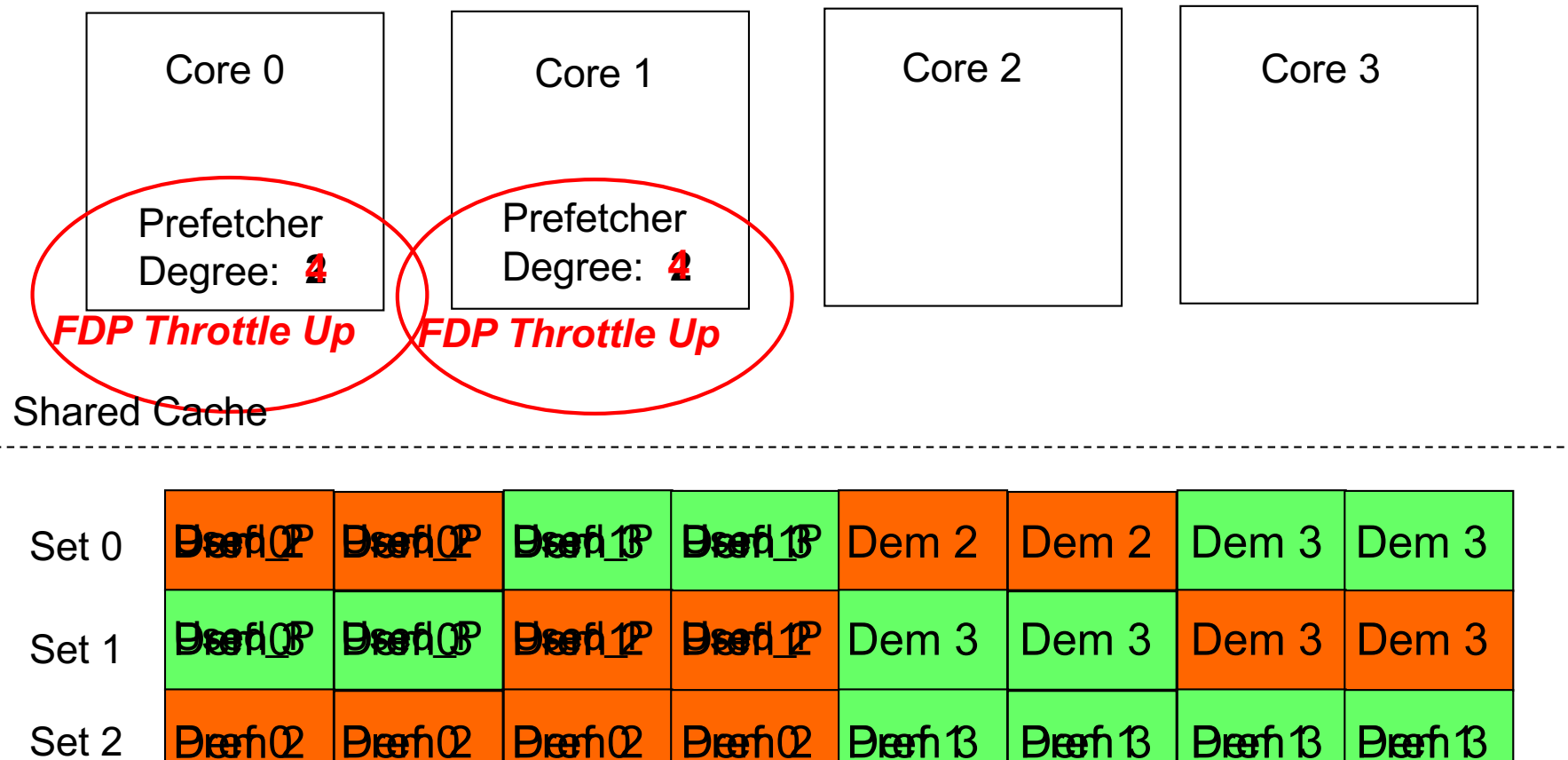


Exact workload combinations can be found in [Ebrahimi et al., MICRO 2009]

High Interference caused by Accurate Prefetchers



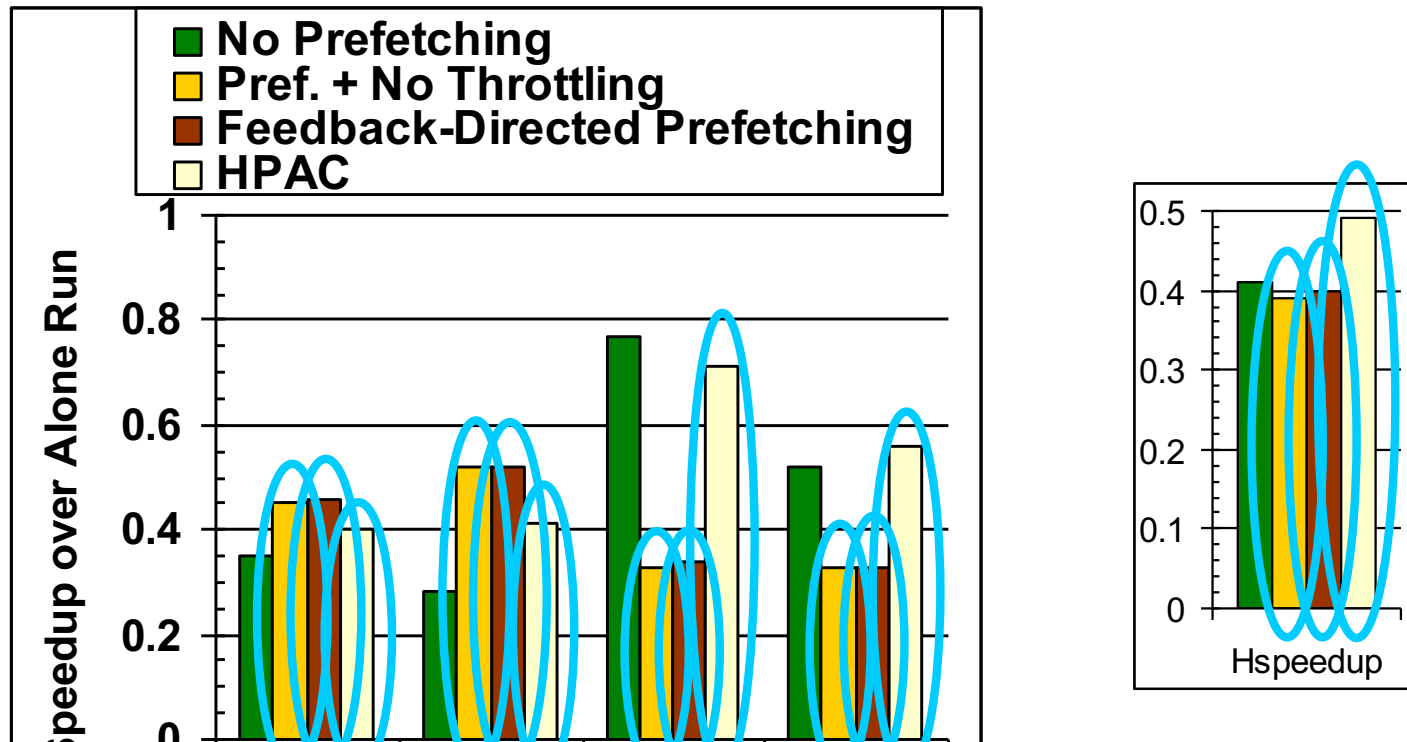
Shortcoming of Local Prefetcher Throttling



Local-only prefetcher control techniques
have no mechanism to detect inter-core interference

Shortcoming of Local-Only Prefetcher Control

4-core workload example: lbm_06 + swim_00 + crafty_00 + bzip2_00



Our Approach: Use both *global* and per-core feedback to determine each prefetcher's aggressiveness

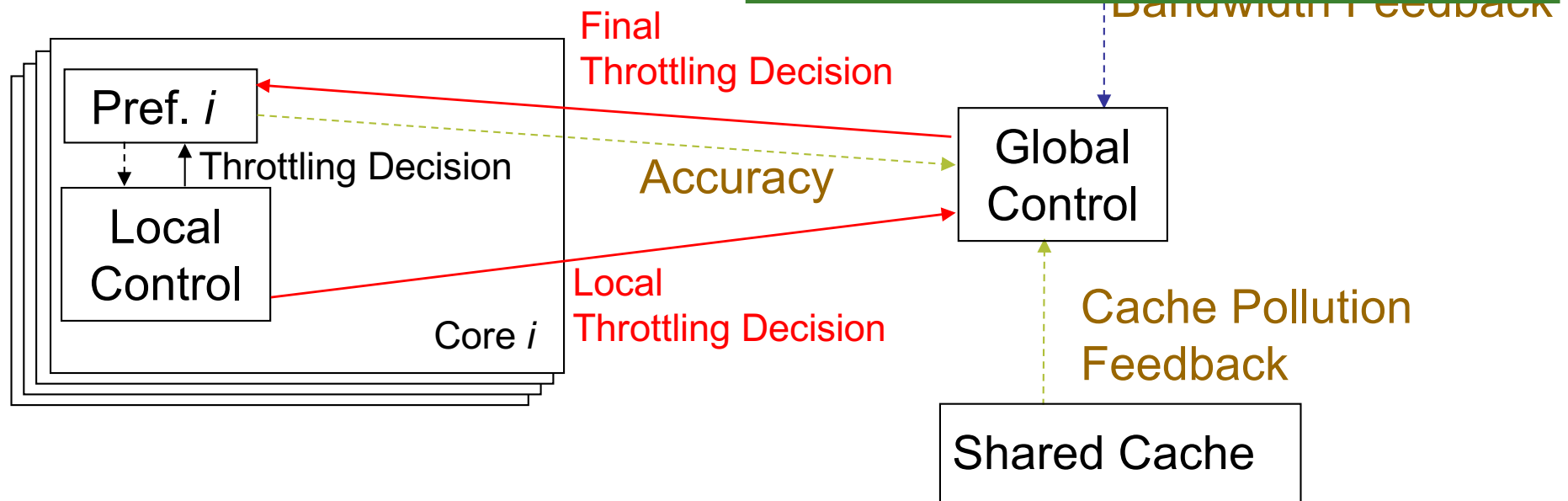
Prefetching in Multi-Core (II)

- Ideas for coordinating different prefetchers' actions
 - Utility-based prioritization
 - Prioritize prefetchers that provide the best marginal utility on system performance
 - Cost-benefit analysis
 - Compute cost-benefit of each prefetcher to drive prioritization
 - Heuristic based methods
 - Global controller overrides local controller's throttling decision based on interference and accuracy of prefetchers
 - Ebrahimi et al., "Coordinated Management of Multiple Prefetchers in Multi-Core Systems," MICRO 2009.

Hierarchical Prefetcher Throttling

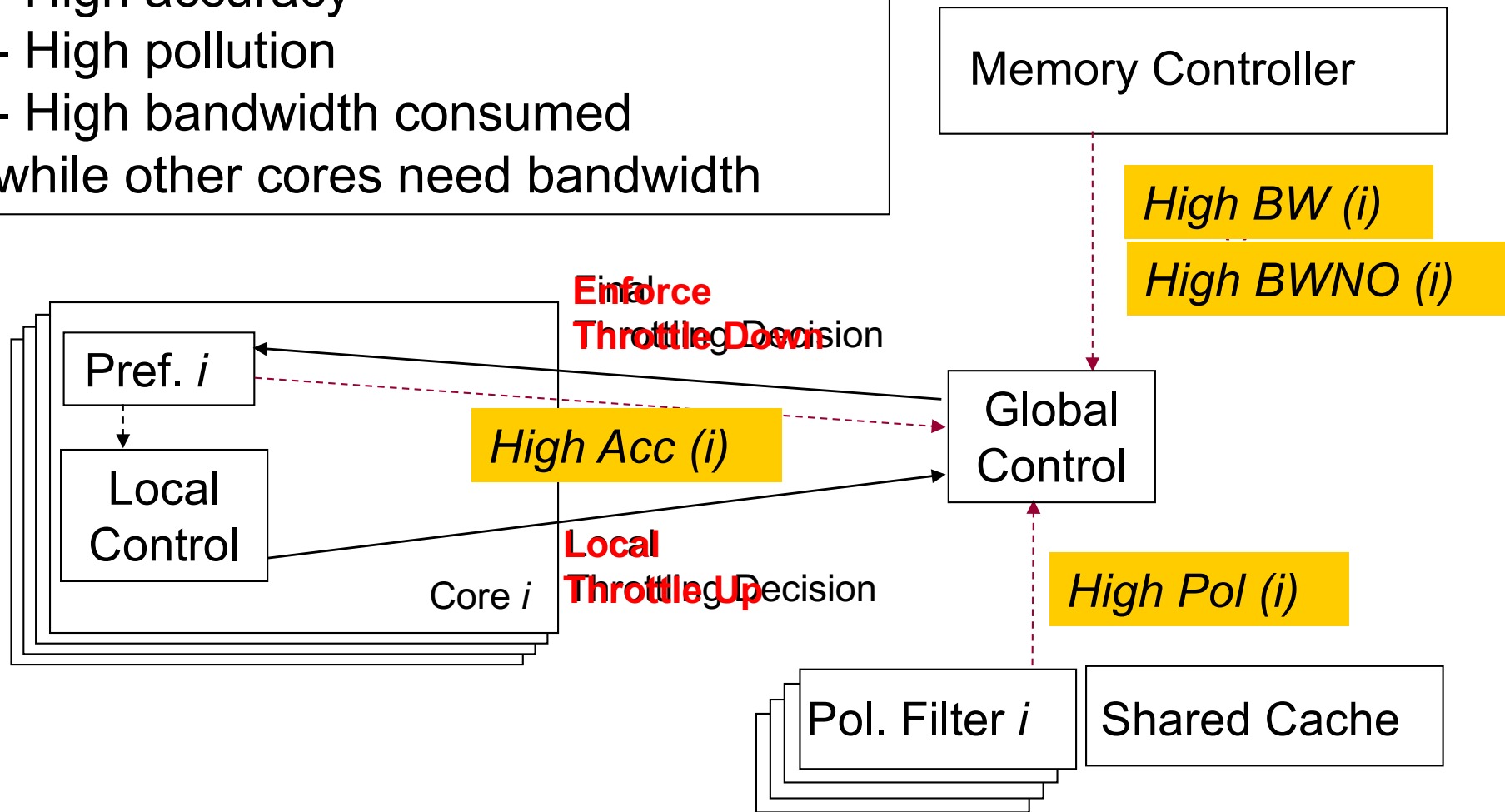
Local Control's goal: **maximize** or **overrides** the decisions made by prefetching performance of core i **independently** overall system performance

Global control's goal: Keep track of and control **prefetcher-caused** inter-core interference in shared memory system



Hierarchical Prefetcher Throttling Example

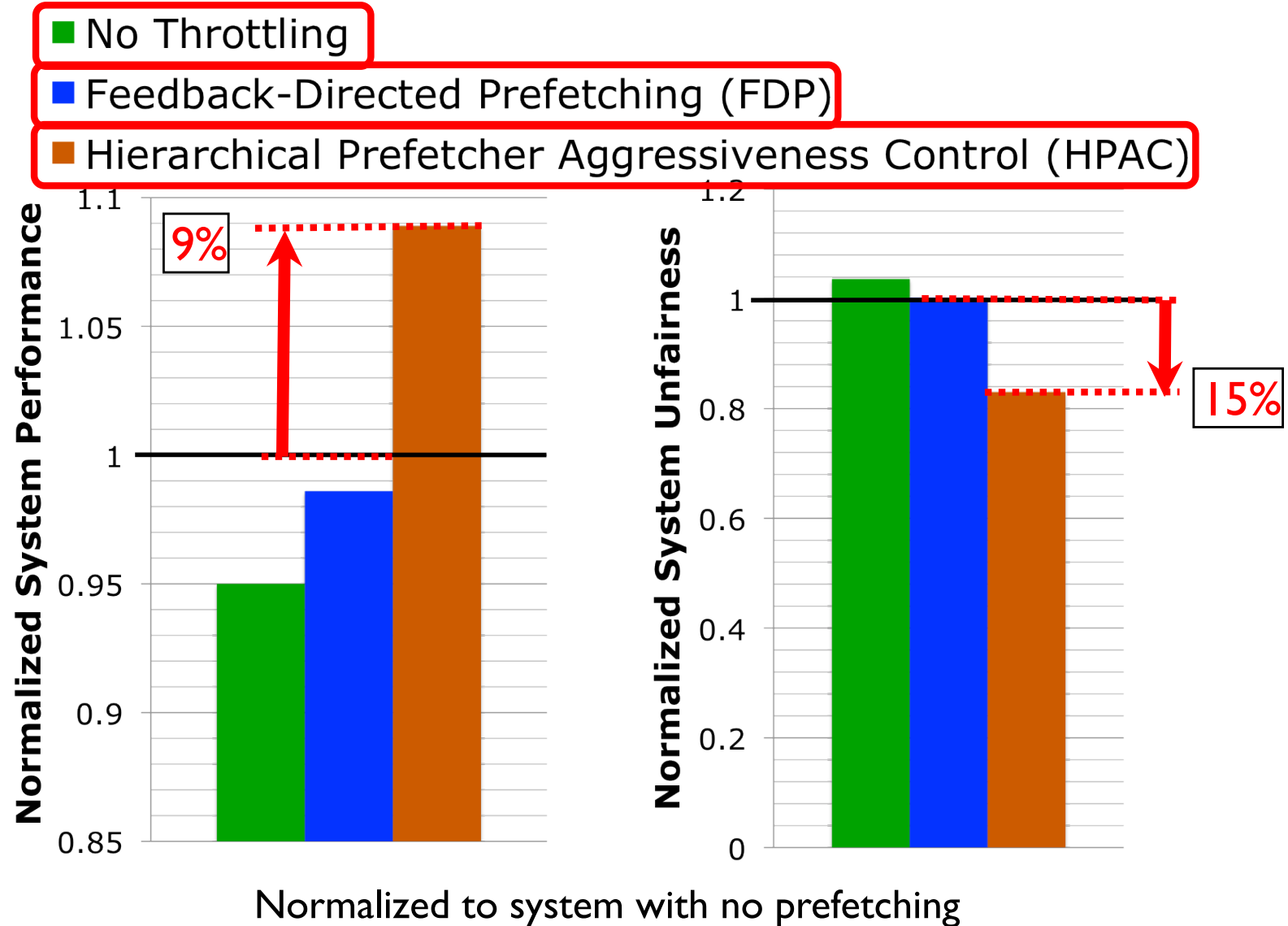
- High accuracy
 - High pollution
 - High bandwidth consumed
- while other cores need bandwidth



HPAC Control Policies

<i>Pol (i)</i>	<i>Acc (i)</i>	<i>BW (i)</i>	<i>BWNO (i)</i>	Interference Class	Action
Causing Low Pollution	Inaccurate	Low BW Consumption	Others' low BW need		
			Others' high BW need	Severe interference	throttle down
		High BW Consumption	Others' low BW need		
	Highly Accurate				
Causing High Pollution	Inaccurate	Low BW Consumption	Others' low BW need		
			Others' high BW need		
		High BW Consumption	Others' low BW need		
		High BW Consumption	Others' high BW need	Severe interference	throttle down

HPAC Evaluation



More on Coordinated Prefetcher Control

- Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt,
"Coordinated Control of Multiple Prefetchers in Multi-Core Systems"

*Proceedings of the 42nd International Symposium on
Microarchitecture (MICRO)*, pages 316-326, New York, NY, December
2009. [Slides \(ppt\)](#)

Coordinated Control of Multiple Prefetchers in Multi-Core Systems

Eiman Ebrahimi[†] Onur Mutlu[§] Chang Joo Lee[†] Yale N. Patt[†]

[†]Department of Electrical and Computer Engineering
The University of Texas at Austin
{ebrahimi, cjlee, patt}@ece.utexas.edu

[§]Computer Architecture Laboratory (CALCM)
Carnegie Mellon University
onur@cmu.edu

More on Prefetching in Multi-Core (I)

- Chang Joo Lee, Onur Mutlu, Veynu Narasiman, and Yale N. Patt,
"Prefetch-Aware DRAM Controllers"
*Proceedings of the 41st International Symposium on
Microarchitecture (MICRO)*, pages 200-209, Lake Como, Italy, November
2008. [Slides \(ppt\)](#)

Prefetch-Aware DRAM Controllers

Chang Joo Lee[†] Onur Mutlu[§] Veynu Narasiman[†] Yale N. Patt[†]

[†]Department of Electrical and Computer Engineering
The University of Texas at Austin
{cjlee, narasima, patt}@ece.utexas.edu

[§]Microsoft Research and Carnegie Mellon University
onur@{microsoft.com,cmu.edu}

More on Prefetching in Multi-Core (II)

- Chang Joo Lee, Veynu Narasiman, Onur Mutlu, and Yale N. Patt,
"Improving Memory Bank-Level Parallelism in the Presence of Prefetching"
Proceedings of the 42nd International Symposium on Microarchitecture (MICRO), pages 327-336, New York, NY, December 2009. [Slides \(ppt\)](#)

Improving Memory Bank-Level Parallelism in the Presence of Prefetching

Chang Joo Lee[†] Veynu Narasiman[†] Onur Mutlu[§] Yale N. Patt[†]

[†]Department of Electrical and Computer Engineering
The University of Texas at Austin
{cjlee, narasima, patt}@ece.utexas.edu

[§]Computer Architecture Laboratory (CALCM)
Carnegie Mellon University
onur@cmu.edu

More on Prefetching in Multi-Core (III)

- Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt,
"Prefetch-Aware Shared Resource Management for Multi-Core Systems"

Proceedings of the 38th International Symposium on Computer Architecture (ISCA), San Jose, CA, June 2011. Slides (pptx)

Prefetch-Aware Shared-Resource Management for Multi-Core Systems

Eiman Ebrahimi[†] Chang Joo Lee^{†‡} Onur Mutlu[§] Yale N. Patt[†]

[†]HPS Research Group
The University of Texas at Austin
{ebrahimi, patt}@hps.utexas.edu

[‡]Intel Corporation
chang.joo.lee@intel.com

[§]Carnegie Mellon University
onur@cmu.edu

More on Prefetching in Multi-Core (IV)

- Vivek Seshadri, Samihan Yedkar, Hongyi Xin, Onur Mutlu, Phillip P. Gibbons, Michael A. Kozuch, and Todd C. Mowry,
"Mitigating Prefetcher-Caused Pollution using Informed Caching Policies for Prefetched Blocks"
ACM Transactions on Architecture and Code Optimization (**TACO**), Vol. 11, No. 4, January 2015.
Presented at the 10th HiPEAC Conference, Amsterdam, Netherlands, January 2015.
[[Slides \(pptx\)](#)] [[pdf](#)]
[[Source Code](#)]

Mitigating Prefetcher-Caused Pollution Using Informed Caching Policies for Prefetched Blocks

VIVEK SESHADRI, SAMIHAN YEDKAR, HONGYI XIN, and ONUR MUTLU,
Carnegie Mellon University
PHILLIP B. GIBBONS and MICHAEL A. KOZUCH, Intel Pittsburgh
TODD C. MOWRY, Carnegie Mellon University

Prefetching in GPUs

- Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das,
"Orchestrated Scheduling and Prefetching for GPGPUs"
Proceedings of the 40th International Symposium on Computer Architecture (ISCA), Tel-Aviv, Israel, June 2013. [Slides \(pptx\)](#) [Slides \(pdf\)](#)

Orchestrated Scheduling and Prefetching for GPGPUs

Adwait Jog[†] Onur Kayiran[†] Asit K. Mishra[§] Mahmut T. Kandemir[†]

Onur Mutlu[‡] Ravishankar Iyer[§] Chita R. Das[†]

[†]The Pennsylvania State University [‡]Carnegie Mellon University

University Park, PA 16802

Pittsburgh, PA 15213

[§]Intel Labs

Hillsboro, OR 97124

{adwait, onur, kandemir, das}@cse.psu.edu onur@cmu.edu {asit.k.mishra, ravishankar.iyer}@intel.com

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

More on Runahead Enhancements

Eliminating Short Periods

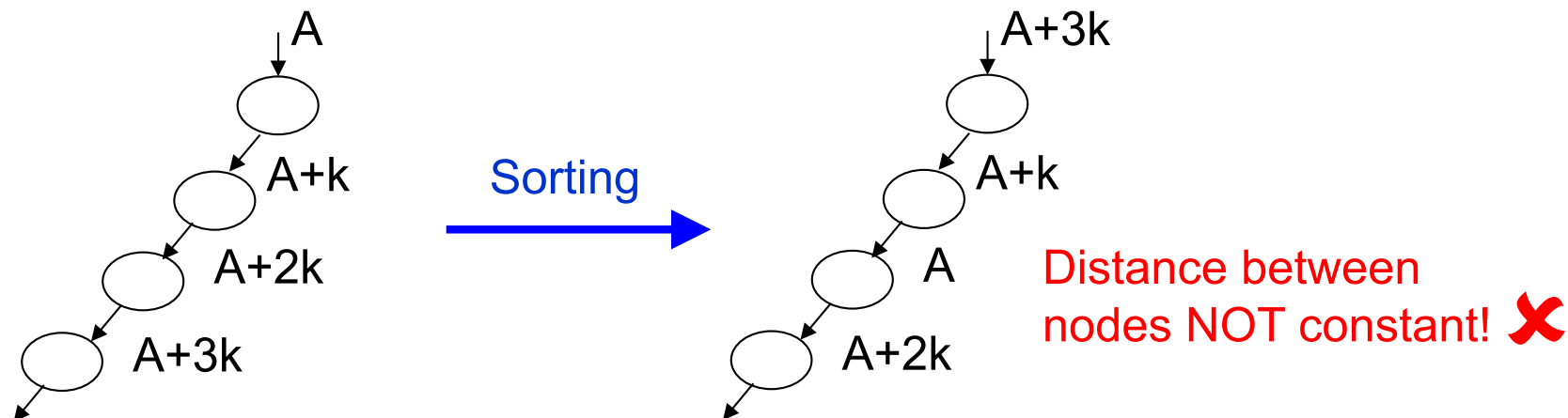
- Mechanism to eliminate short periods:
 - Record the number of cycles C an L2-miss has been in flight
 - If C is greater than a threshold T for an L2 miss, disable entry into runahead mode due to that miss
 - T can be determined statically (at design time) or dynamically
- $T=400$ for a minimum main 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

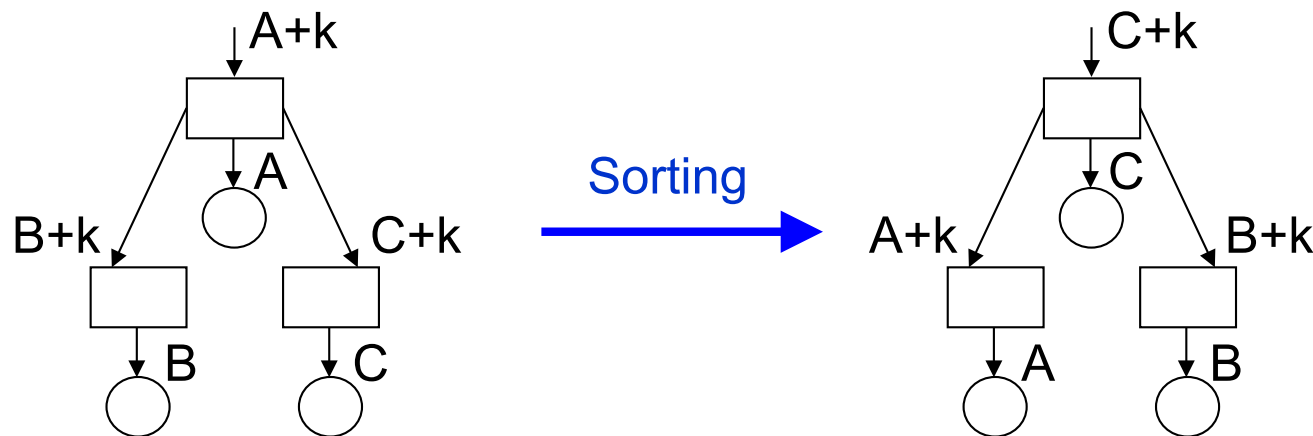
- Stable AVDs can be captured with a **stride value predictor**
- Stable AVDs disappear with the **re-organization of the data structure** (e.g., sorting)



- 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)



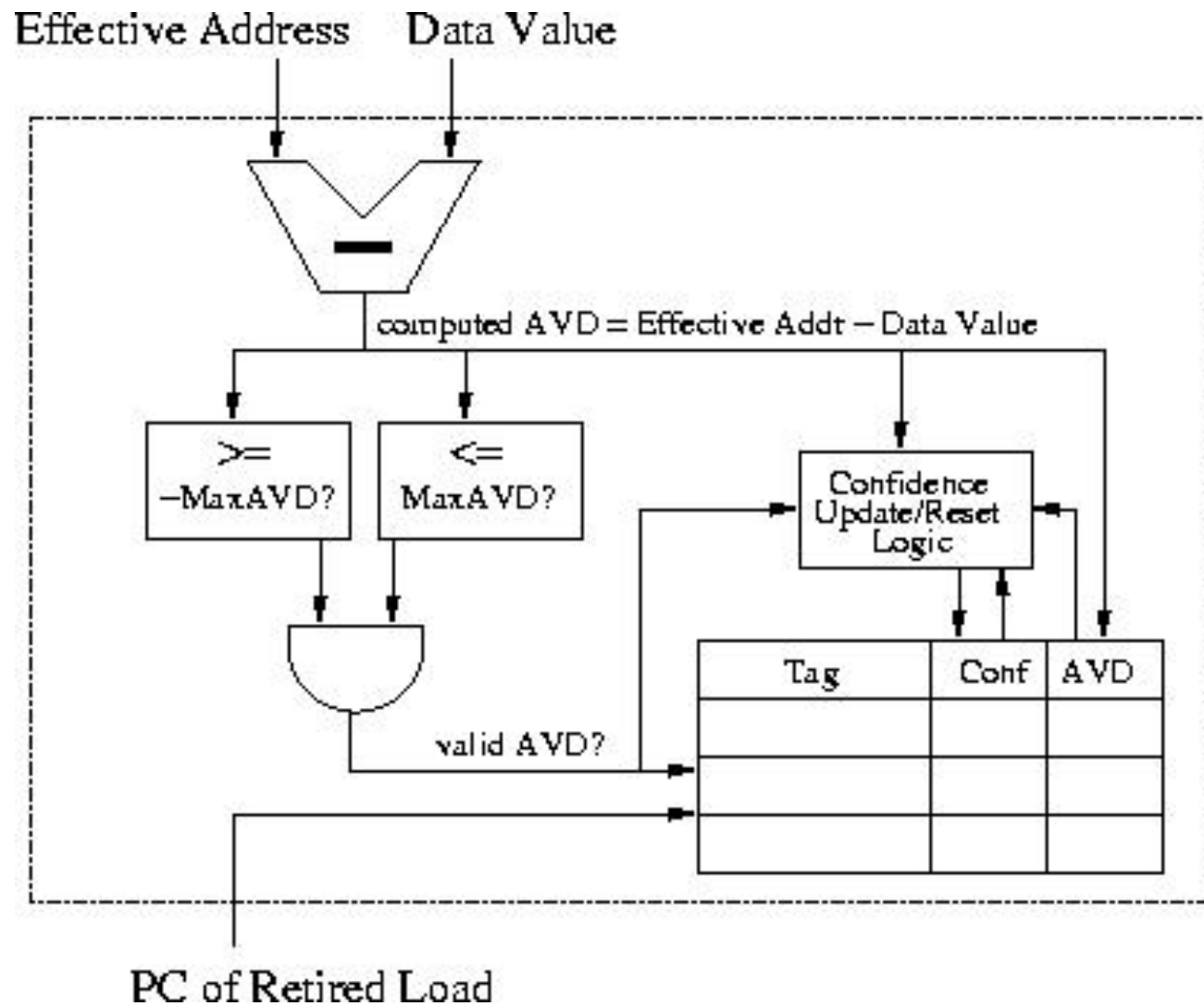
Distance between
node and **string**
still constant! ✓

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

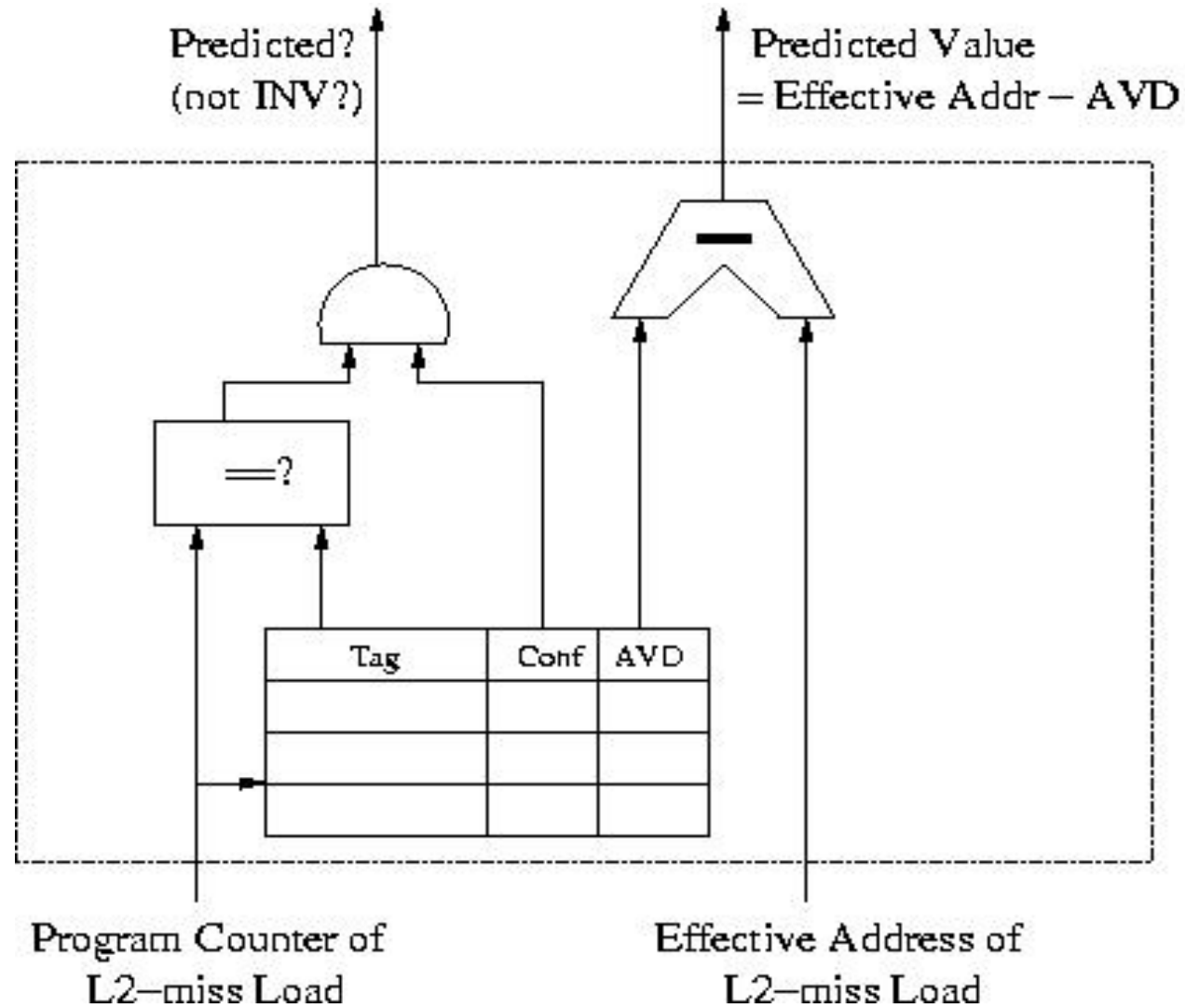
An Implementable AVD Predictor

- 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

