

18-447

Computer Architecture

Lecture 25: Memory Latency Tolerance II: Prefetching

Prof. Onur Mutlu

Carnegie Mellon University

Spring 2015, 4/1/2015

Announcements

- My office hours moved to 3:40-4:40pm today

Reminder on Assignments

- Lab 6 due this Friday (April 3)
 - C-level simulation of data caches and branch prediction
- Homework 6 will be due April 10
- Tentative Midterm II date: April 22
- The course will continue to move quickly... Keep your pace.
- Talk with the TAs and me if you need any help.
 - We cannot do or debug the assignments for you but we can give you suggestions
 - My goal is to enable you learn the material
 - You never know when you will use the principles you learn

Lab Late Days

- 3 more late days you can use on Lab 6 & 7
- Lab 8 will have special treatment

Where We Are in Lecture Schedule

- The memory hierarchy
 - Caches, caches, more caches
 - Virtualizing the memory hierarchy: Virtual Memory
 - Main memory: DRAM
 - Main memory control, scheduling
 - Memory latency tolerance techniques
 - Non-volatile memory
-
- Multiprocessors
 - Coherence and consistency
 - Interconnection networks
 - Multi-core issues (e.g., heterogeneous multi-core)

Upcoming Seminar on DRAM (April 3)

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

Required Reading

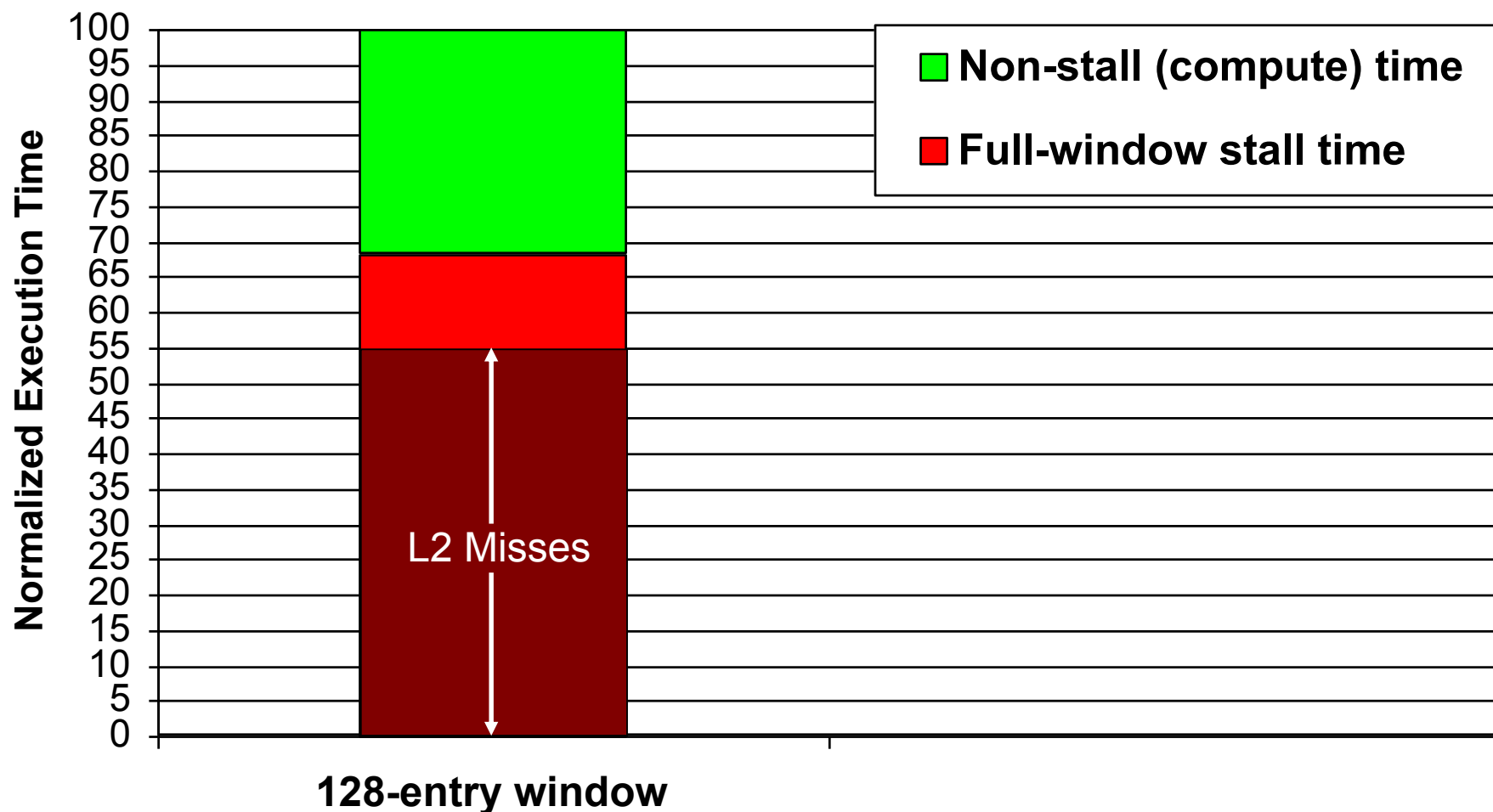
- Onur Mutlu, Justin Meza, and Lavanya Subramanian, **"The Main Memory System: Challenges and Opportunities"**

*Invited Article in Communications of the Korean Institute of Information Scientists and Engineers (**KIISE**), 2015.*

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

Tolerating Memory Latency

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

Memory Latency Tolerance Techniques

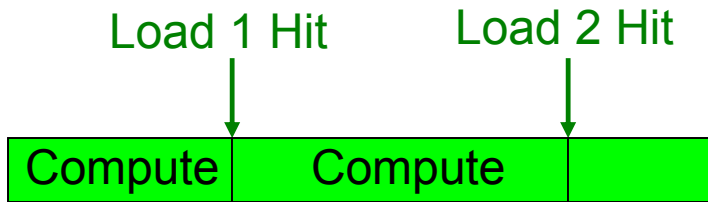
- **Caching** [initially by Wilkes, 1965]
 - ❑ Widely used, simple, effective, but inefficient, passive
 - ❑ Not all applications/phases exhibit temporal or spatial locality
- **Prefetching** [initially in IBM 360/91, 1967]
 - ❑ Works well for regular memory access patterns
 - ❑ Prefetching irregular access patterns is difficult, inaccurate, and hardware-intensive
- **Multithreading** [initially in CDC 6600, 1964]
 - ❑ Works well if there are multiple threads
 - ❑ Improving single thread performance using multithreading hardware is an ongoing research effort
- **Out-of-order execution** [initially by Tomasulo, 1967]
 - ❑ Tolerates irregular cache misses that cannot be prefetched
 - ❑ Requires extensive hardware resources for tolerating long latencies
 - ❑ **Runahead execution** alleviates this problem (as we will see today)

Runahead Execution (I)

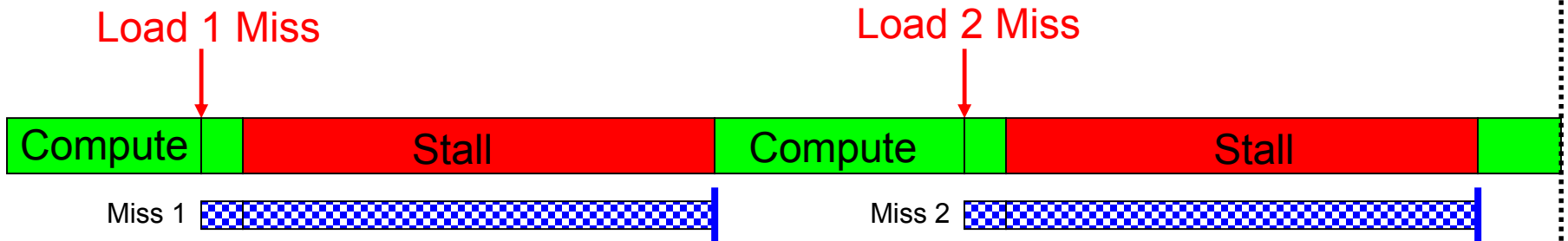
- A technique to obtain the memory-level parallelism benefits of a large instruction window
- When the oldest instruction is a long-latency cache miss:
 - Checkpoint architectural state and enter runahead mode
- In runahead mode:
 - Speculatively pre-execute instructions
 - The purpose of pre-execution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

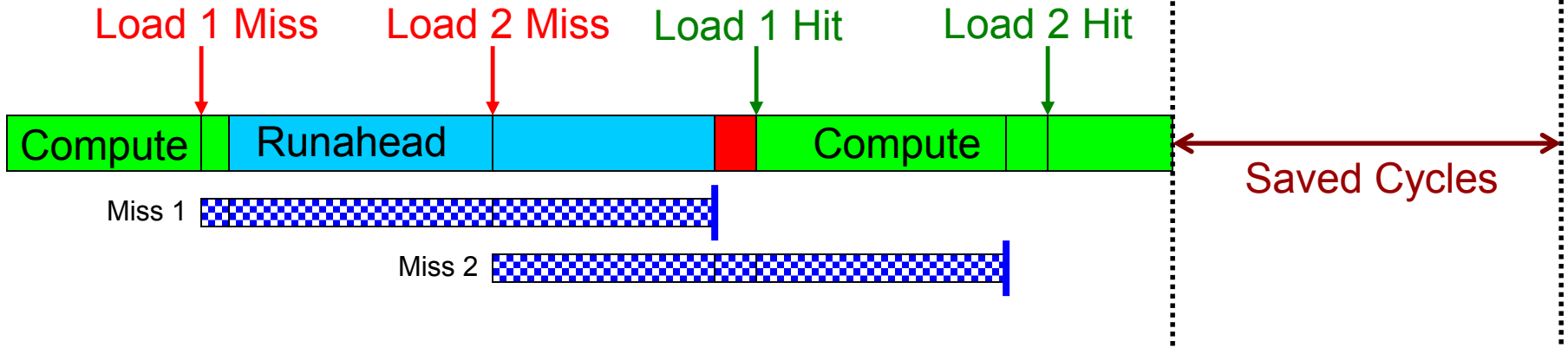
Perfect Caches:



Small Window:



Runahead:



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

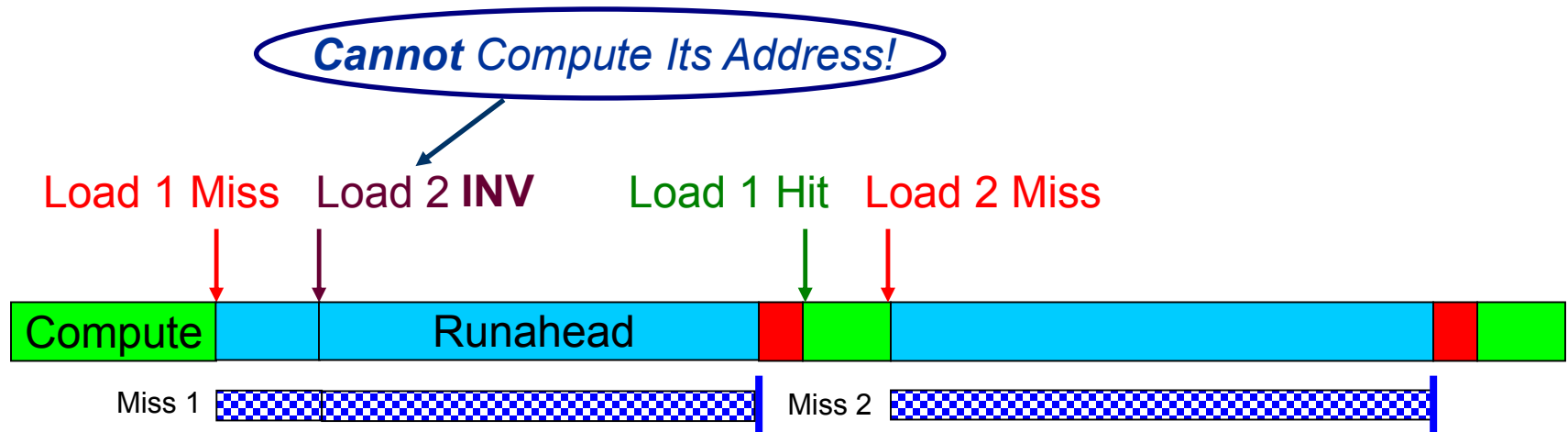
- ❑ Runahead cannot parallelize dependent L2 cache misses
- ❑ **Address-Value Delta (AVD) Prediction** [MICRO' 05]

■ Irresolvable branch mispredictions in runahead mode

- ❑ Cannot recover from a mispredicted L2-miss dependent branch
 - ❑ **Wrong Path Events** [MICRO' 04]
-

The Problem: Dependent Cache Misses

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

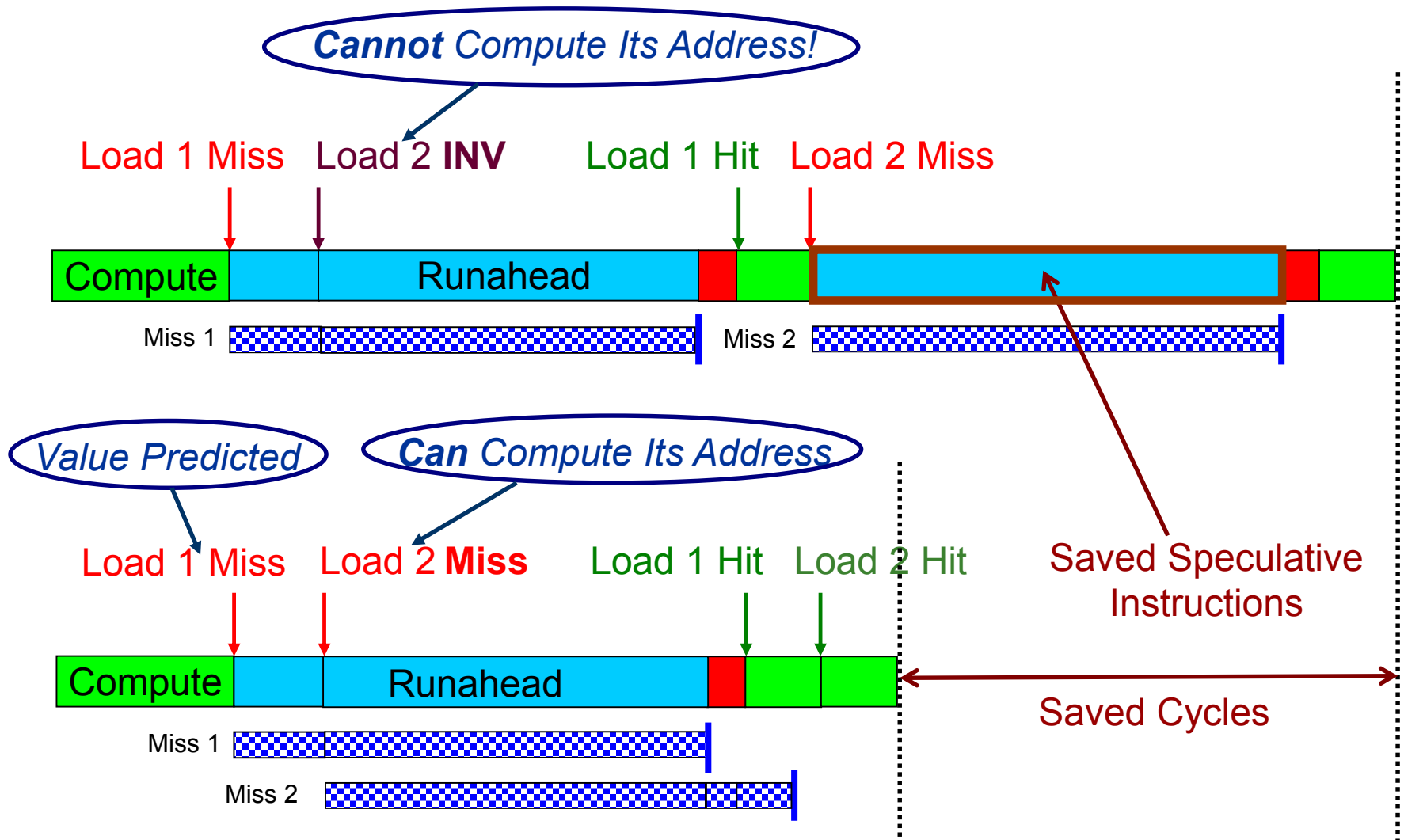


- Runahead execution cannot parallelize dependent misses
 - ❑ wasted opportunity to improve performance
 - ❑ wasted energy (useless pre-execution)
- Runahead performance would improve by 25% if this limitation were ideally overcome

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]

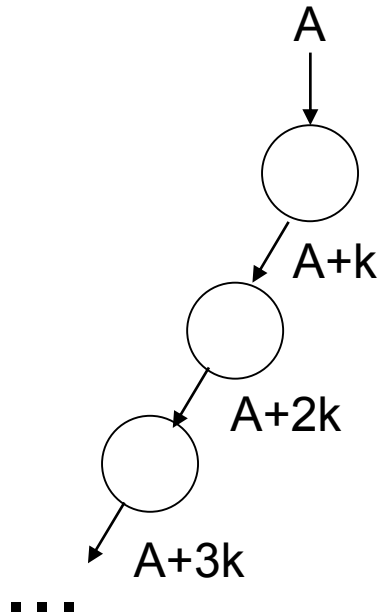
- Address-value delta (AVD) of a load instruction defined as:
$$\text{AVD} = \text{Effective Address of Load} - \text{Data Value of Load}$$
 - For some address loads, AVD is stable
 - An AVD predictor keeps track of the AVDs of address loads
 - When a load is an L2 miss in runahead mode, AVD predictor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

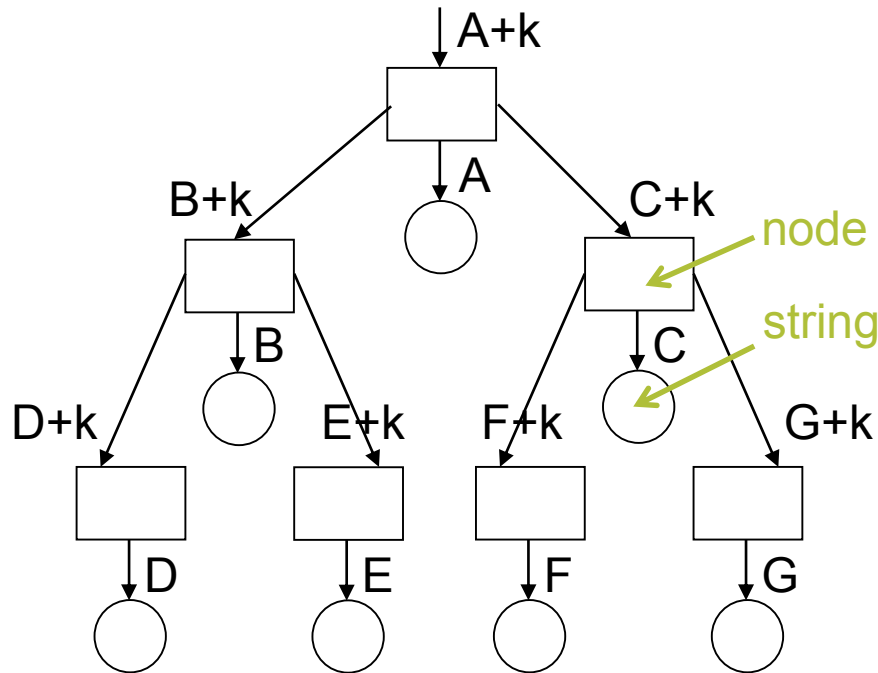
Stable AVD

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

Performance of AVD Prediction



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)

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

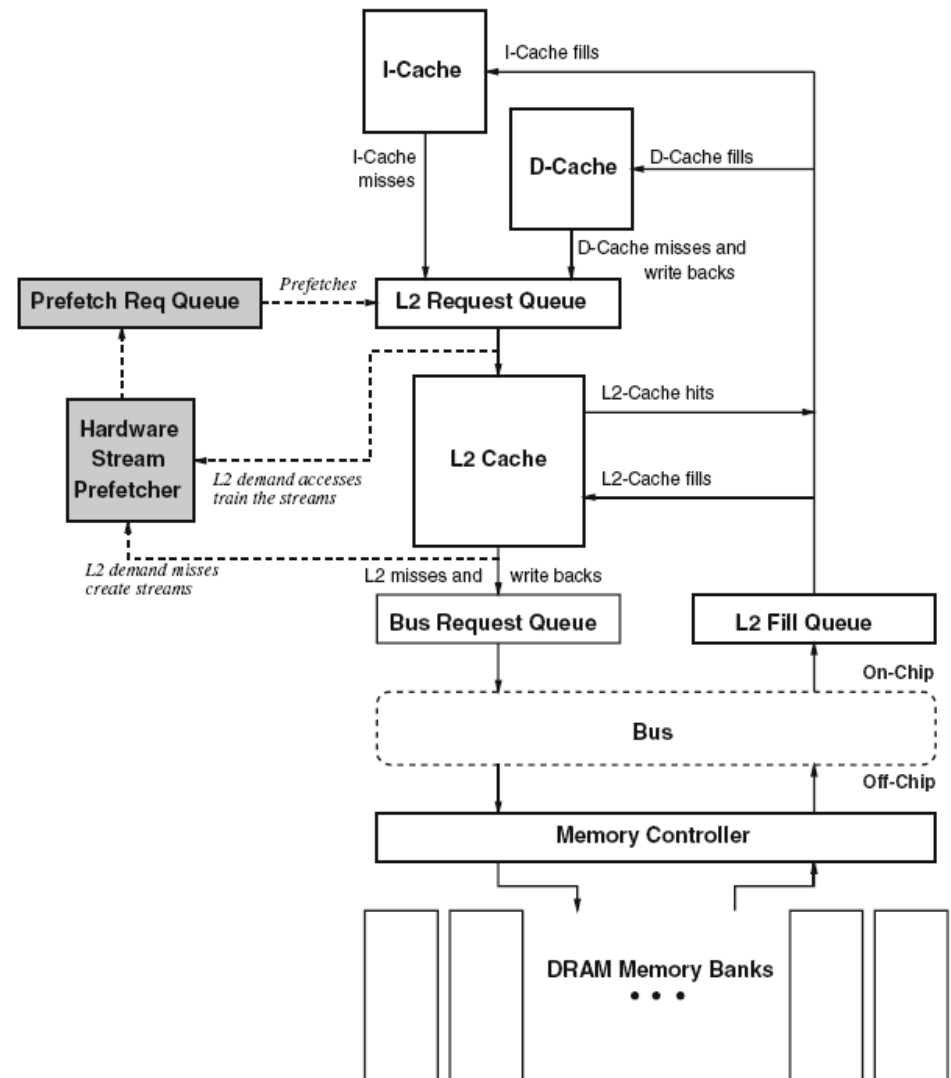
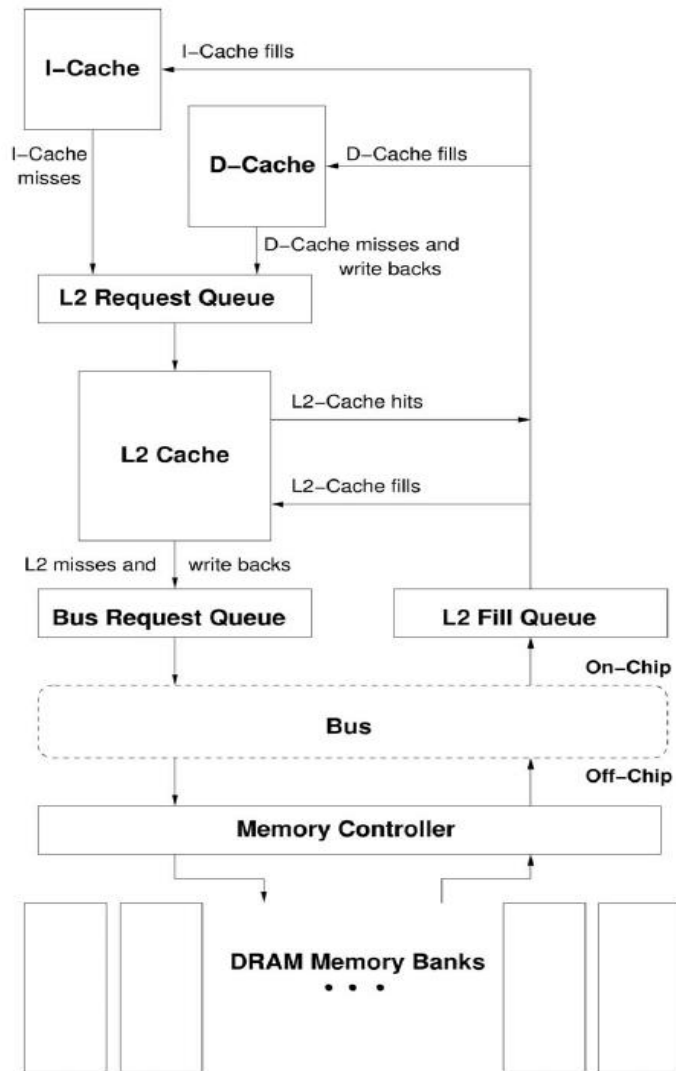
Prefetching and Correctness

- 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>
-----------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------

Which one is better?

- 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

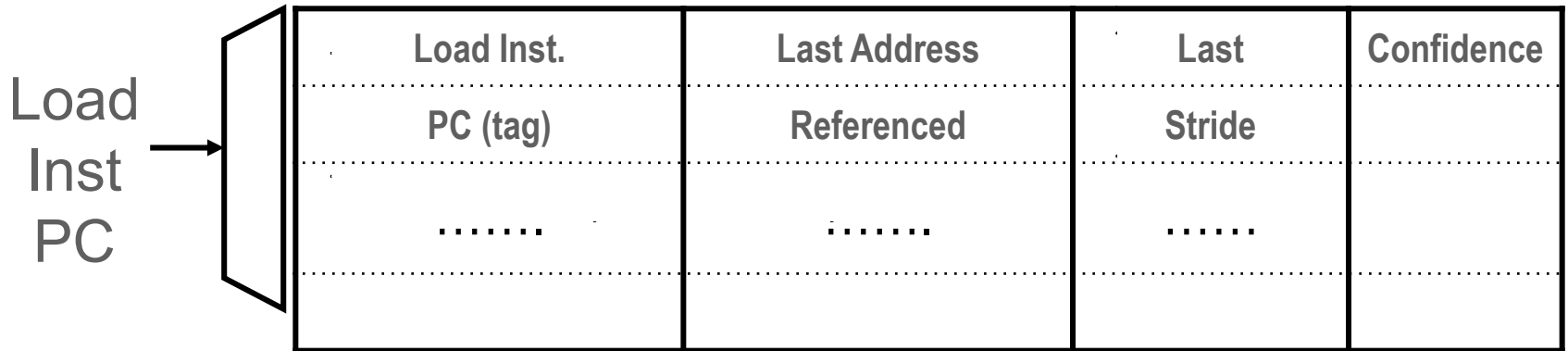
- Simplest form of hardware prefetching: always prefetch next N cache lines after a demand access (or a demand miss)
 - Next-line prefetcher (or next sequential prefetcher)
 - Tradeoffs:
 - + Simple to implement. No need for sophisticated pattern detection
 - + Works well for sequential/streaming access 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

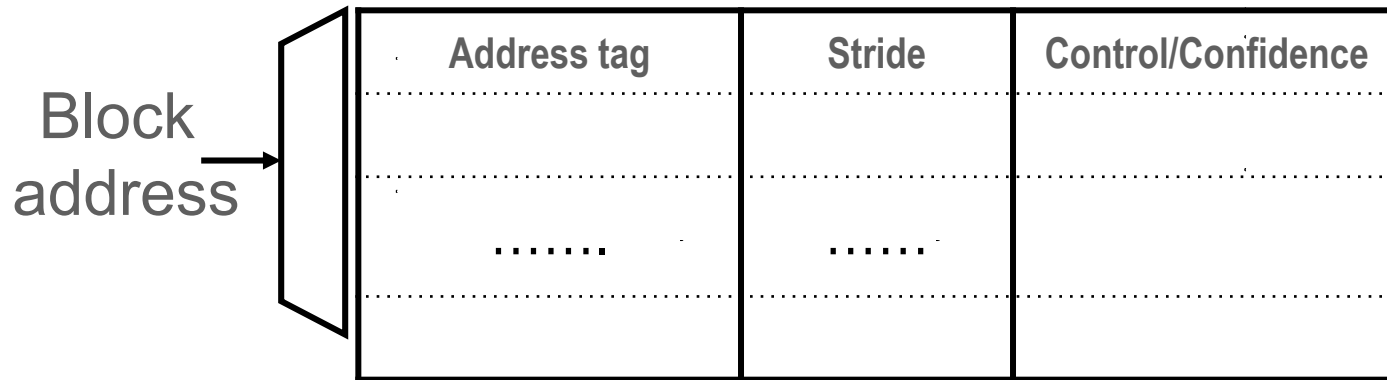
- Instruction based:
 - Baer and Chen, “An effective on-chip preloading scheme to reduce data access penalty,” SC 1991.
 - Idea:
 - Record the distance between the memory addresses referenced by a load instruction (i.e. stride of the load) as well as the last address referenced by the load
 - Next time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

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**

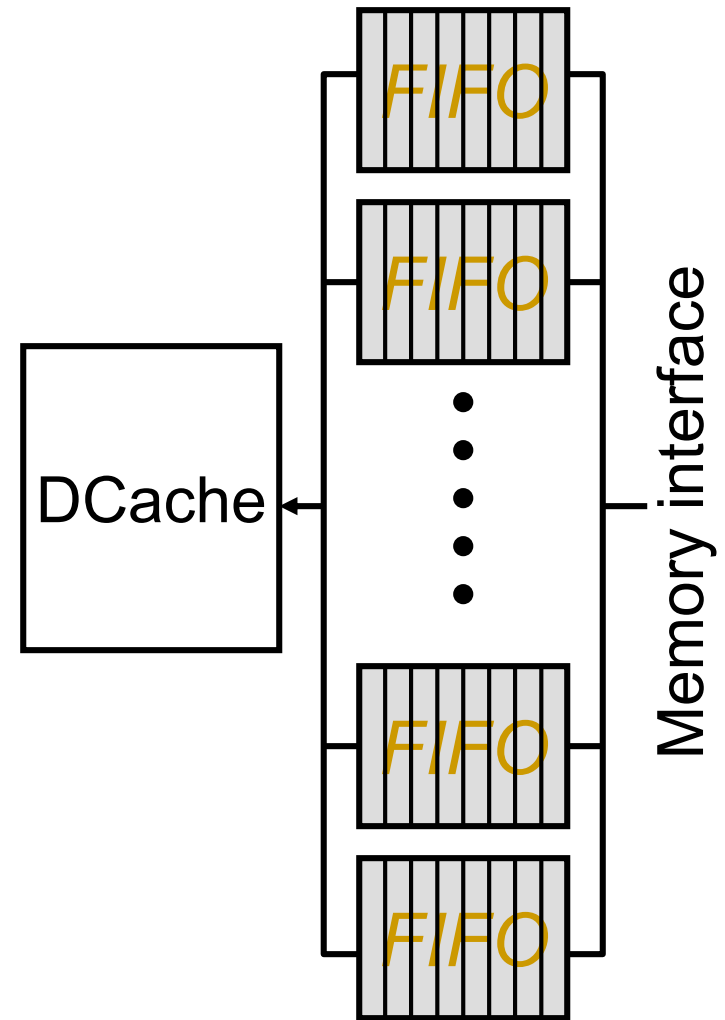
Cache-Block Address Based Stride Prefetching



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

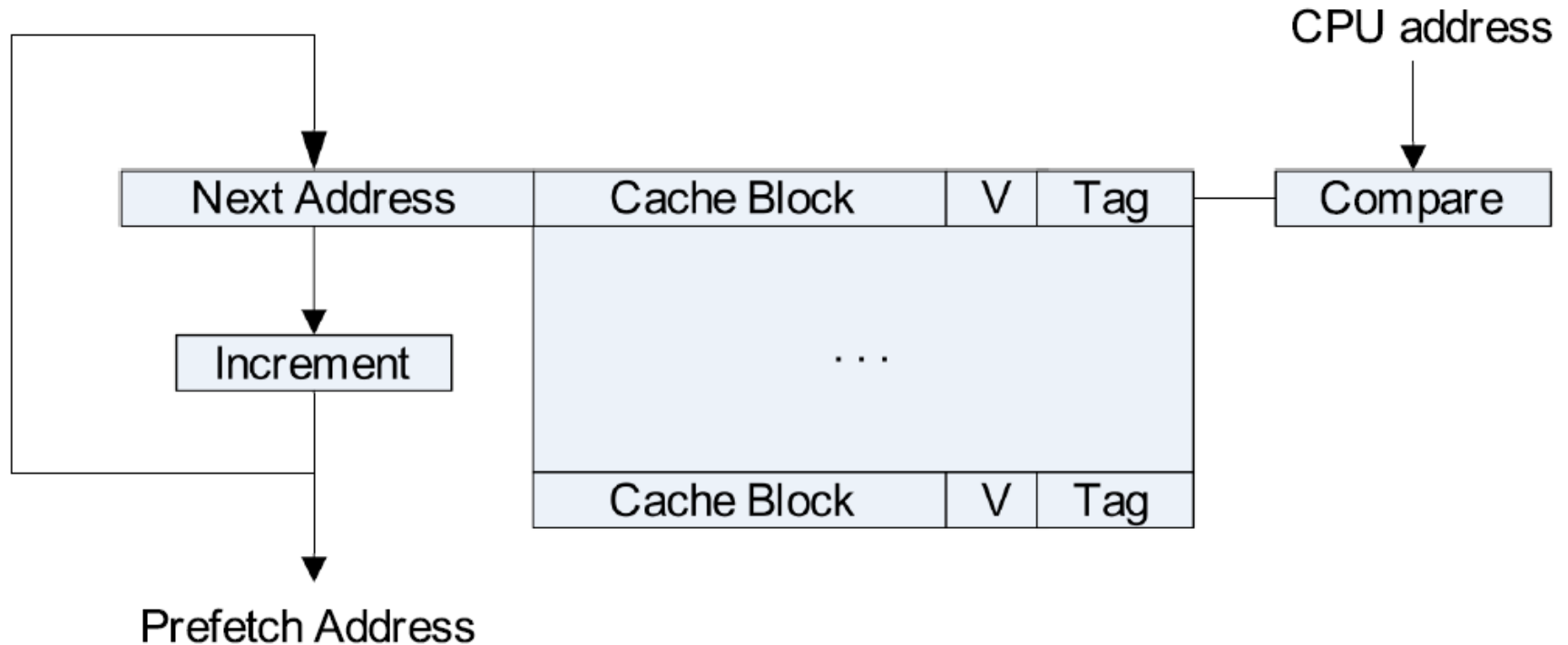
Stream Buffers (Jouppi, ISCA 1990)

- 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 recycle 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

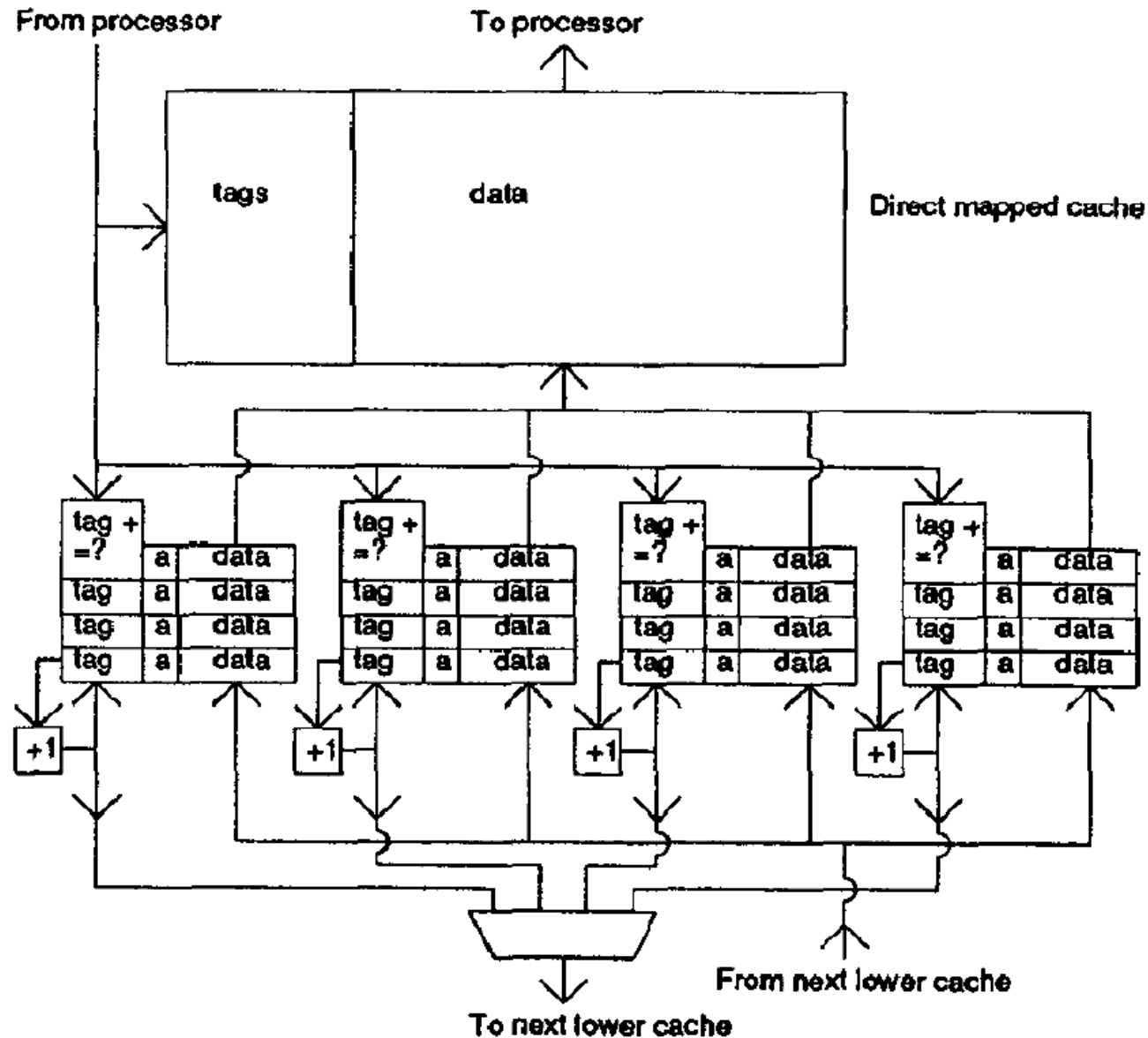


Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," ISCA 1990.

Stream Buffer Design



Stream Buffer Design



Prefetcher Performance (I)

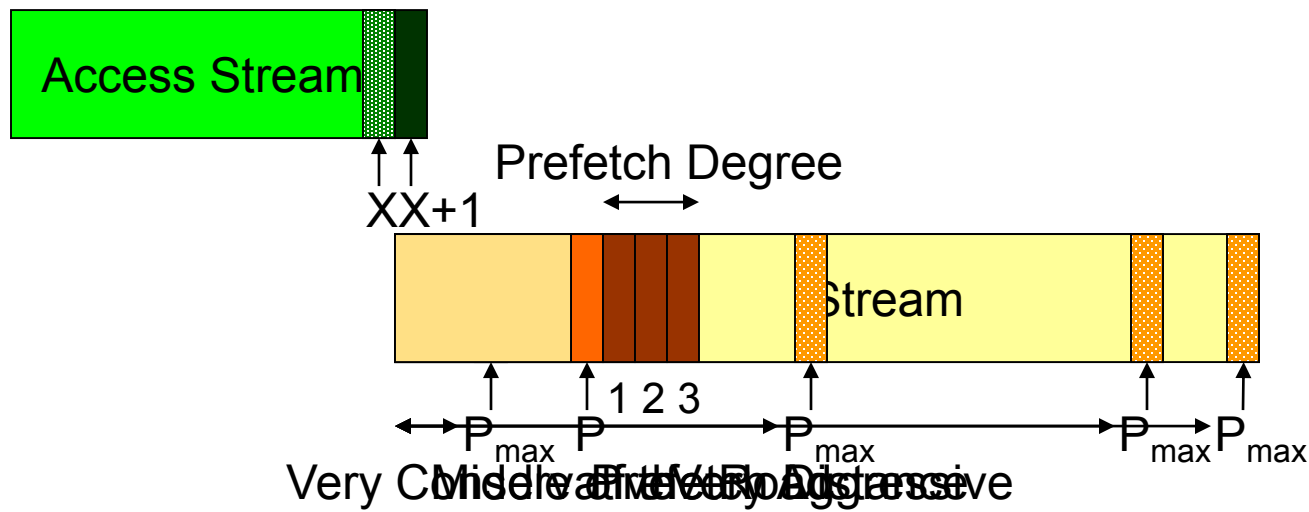
- **Accuracy** (used prefetches / sent prefetches)
- **Coverage** (prefetched misses / all misses)
- **Timeliness** (on-time prefetches / used prefetches)

- **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)

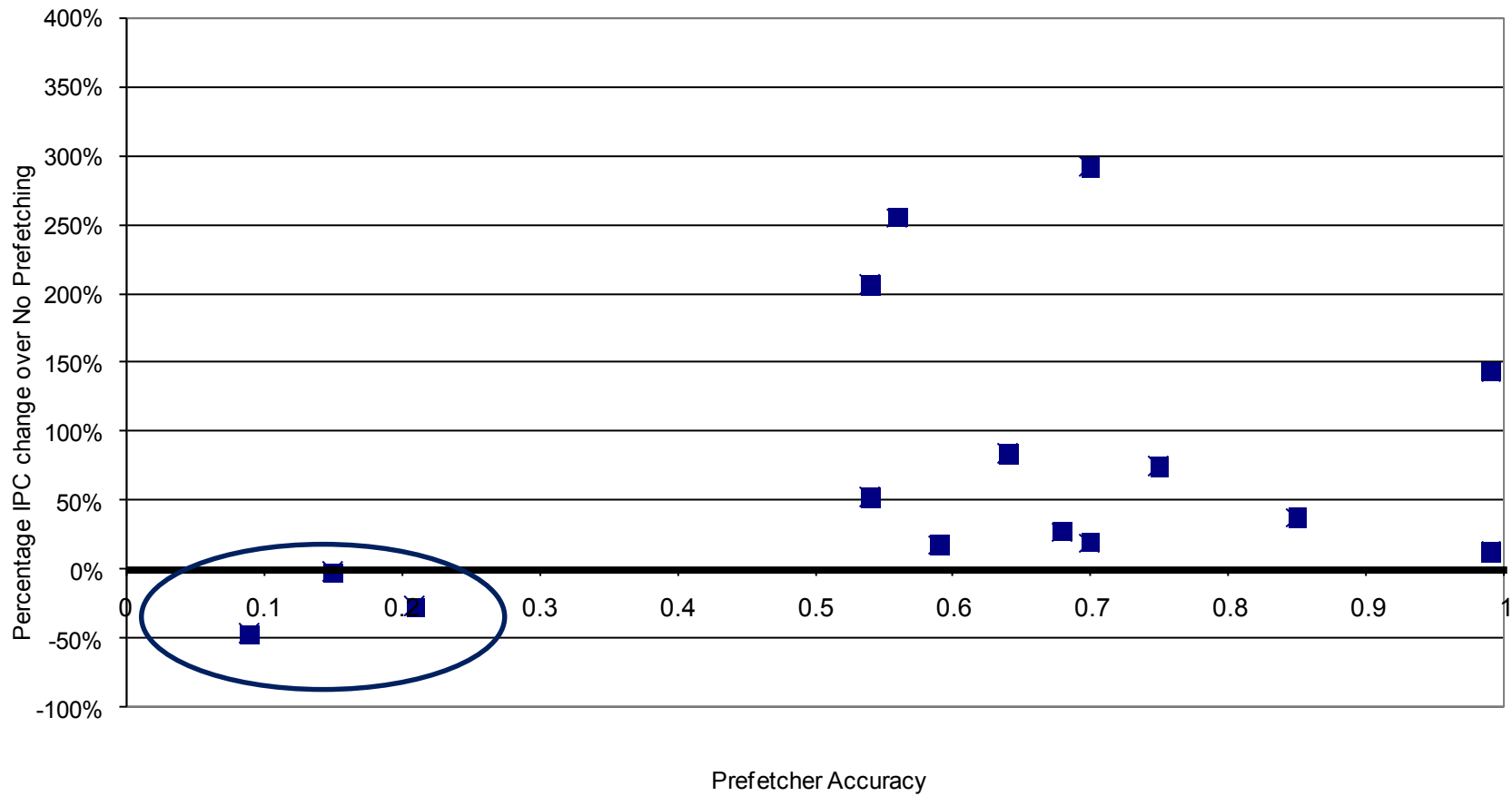
- Prefetcher aggressiveness affects all performance metrics
- Aggressiveness dependent on prefetcher type
- For most hardware prefetchers:
 - **Prefetch distance**: how far ahead of the demand stream
 - **Prefetch degree**: how many prefetches per demand access



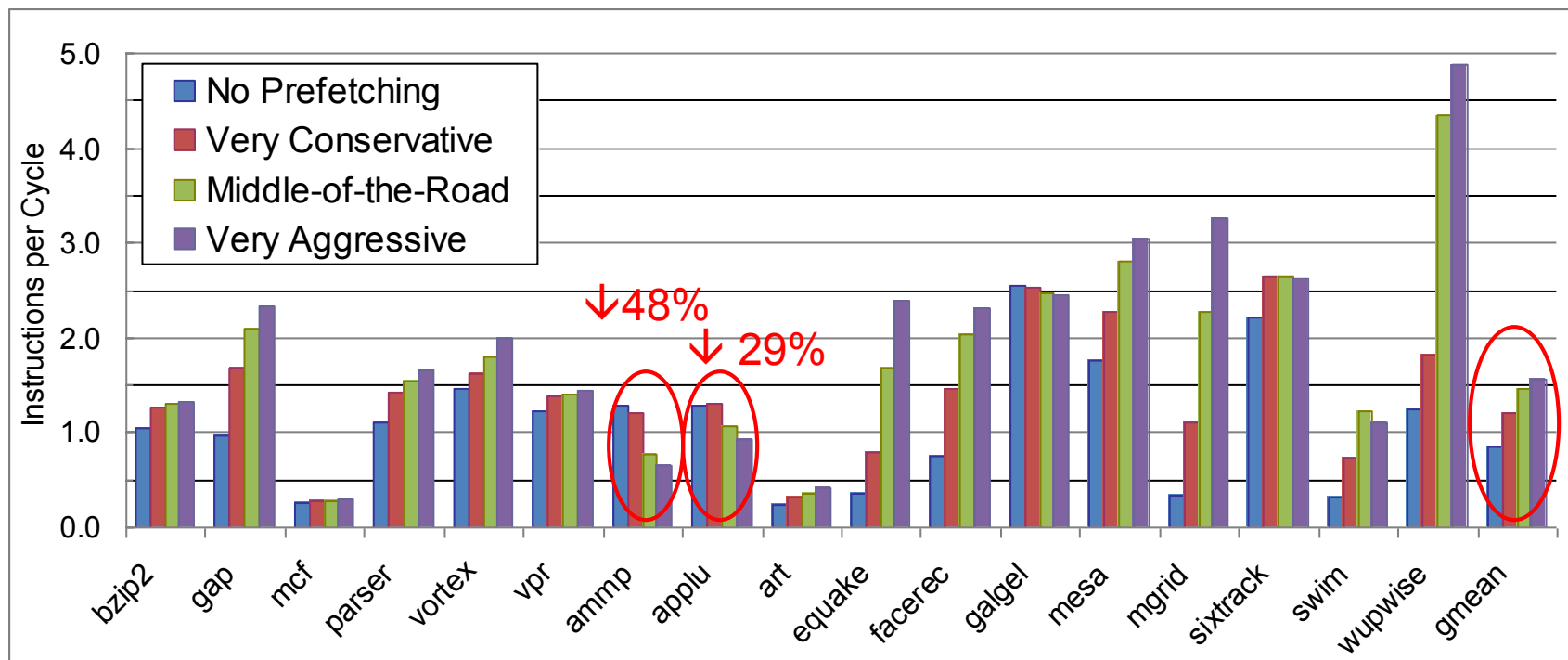
Prefetcher Performance (III)

- 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)

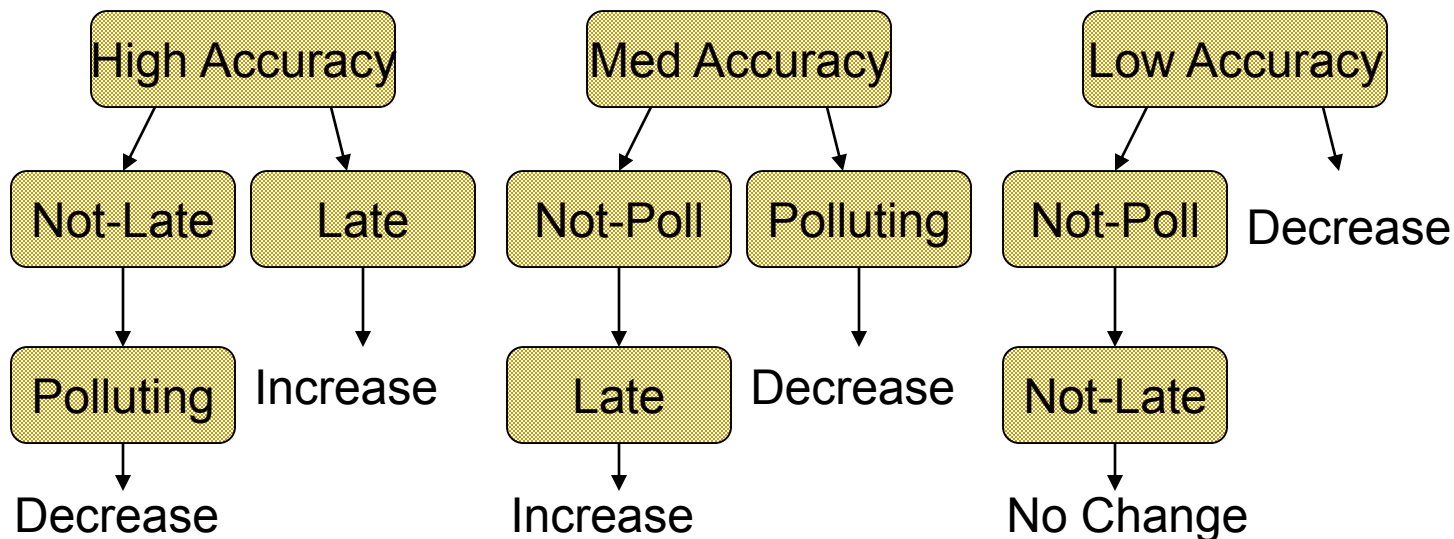


- Srinath et al., “Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers”, HPCA 2007.

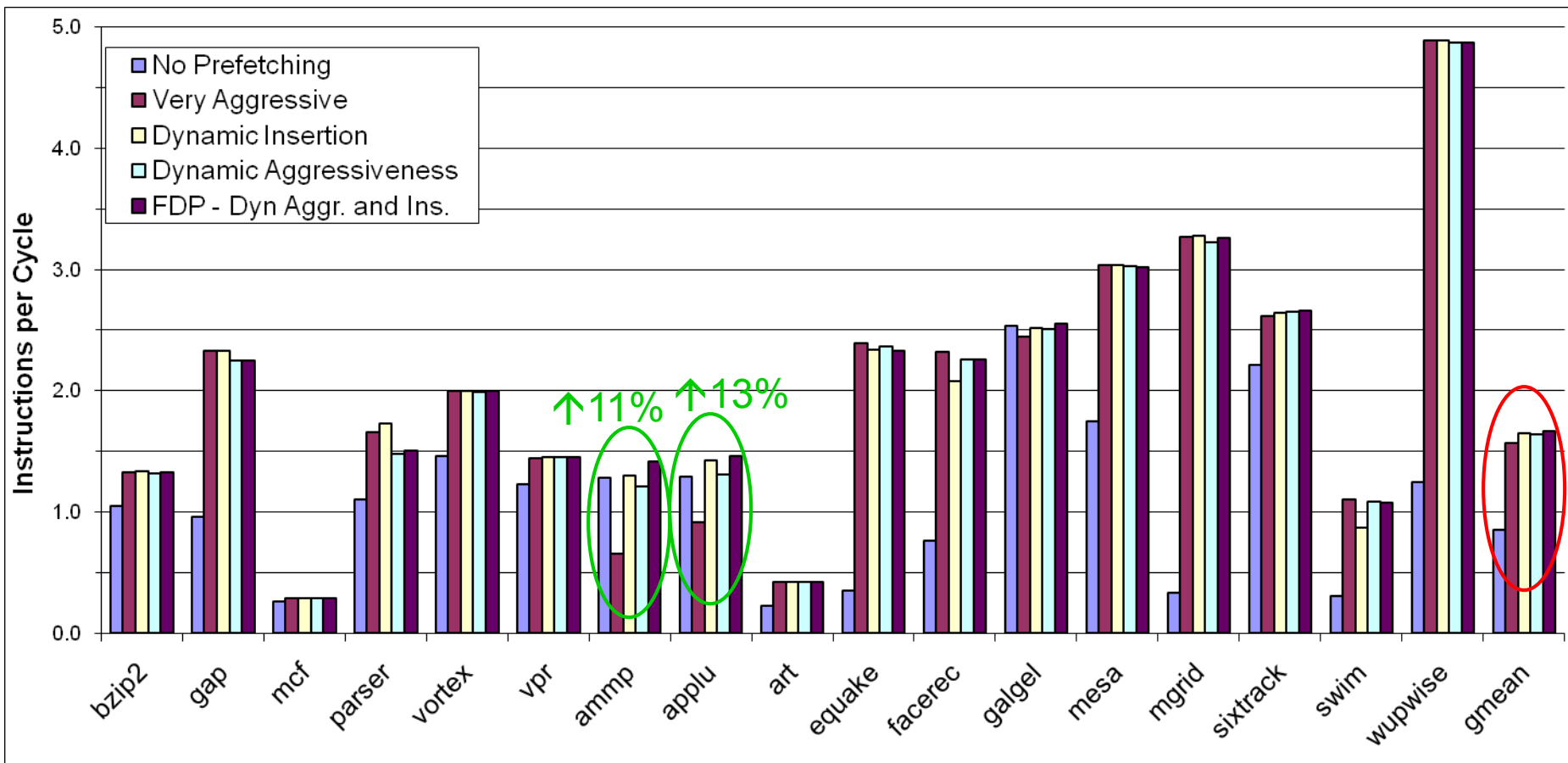
Feedback-Directed Prefetcher Throttling (I)

■ 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.

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