

# 18-447: Computer Architecture

## Lecture 22: Tolerating Memory Latency

Prof. Onur Mutlu

Carnegie Mellon University

Spring 2012, 4/16/2012

# Reminder: Homeworks

---

- Homework 6
  - Due today (April 16)
  - Topics: Main memory, caches, virtual memory

# Reminder: Lab Assignments

---

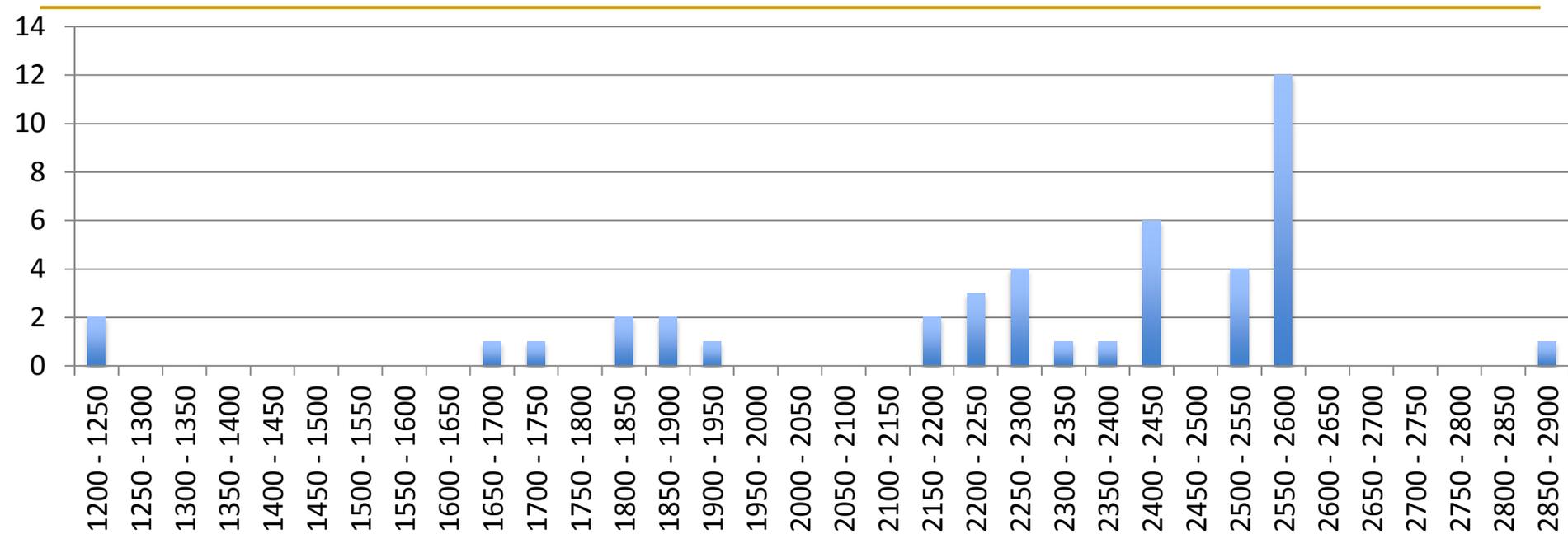
- Lab Assignment 6
  - Implementing a more realistic memory hierarchy
    - L2 cache model
    - DRAM, memory controller models
    - MSHRs, multiple outstanding misses
  - Due April 23
  - Extra credit: Prefetching

# Lab 4: Correct Designs and Extra Credit

---

Rank	Student	Crit. Path (ns)	Cycles	Execution Time (ns)	Relative Execution Time
1	<b>Eric Brunstad</b>	<b>10.425</b>	<b>34568</b>	<b>360371.4</b>	<b>1.00</b>
2	<b>Arthur Chang</b>	<b>10.686</b>	<b>34804</b>	<b>371915.5</b>	<b>1.03</b>
3	<b>Alex Crichton</b>	<b>10.85</b>	<b>34636</b>	<b>375800.6</b>	<b>1.04</b>
4	Jason Lin	11.312	34672	392209.7	1.09
5	Anish Phophaliya	10.593	37560	397873.0	1.10
6	James Wahawisan	9.16	44976	411980.2	1.14
7	Prerak Patel	11.315	37886	428680.1	1.19
8	Greg Nazario	12.23	35696	436562.1	1.21
9	Kee Young Lee	10.019	44976	450614.5	1.25
10	Jonathan Loh	13.731	33668	462295.3	1.28
11	Vikram Rajkumar	13.823	34932	482865.0	1.34
12	Justin Wagner	15.065	33728	508112.3	1.41
13	Daniel Jacobs	13.593	37782	513570.7	1.43
14	Mike Mu	14.055	36832	517673.8	1.44
15	Qiannan Zhang	13.484	38764	522693.8	1.45
16	Andrew Tan	16.754	34660	580693.6	1.61
17	Dennis Liang	16.722	37176	621657.1	1.73
18	Dev Gurjar	12.864	57332	737518.8	2.05
19	Winnie Woo	23.281	33976	790995.3	2.19

# Lab 5 Grades



<b>Average</b>	2285
<b>Median</b>	2405
<b>Max</b>	2855
<b>Min</b>	1206
<b>Max Possible (w/o Extra Credit)</b>	2595
<b>Total number of students</b>	43
<b>Fully correct</b>	13
<b>Attempted Extra Credit</b>	2

# Lab 5 Extra Credit

---

- Tyler Huberty

# Last Lecture

---

- Virtual memory

# Today

---

- Memory latency tolerance/reduction
- Prefetching

# Tolerating Memory Latency

# Latency Tolerance

---

- An out-of-order execution processor tolerates latency of multi-cycle operations by executing independent instructions concurrently
  - It does so by buffering instructions in reservation stations and reorder buffer
  - Instruction window: Hardware resources needed to buffer all decoded but not yet retired/committed instructions
- What if an instruction takes 500 cycles?
  - How large of an instruction window do we need to continue decoding?
  - How many cycles of latency can OoO tolerate?

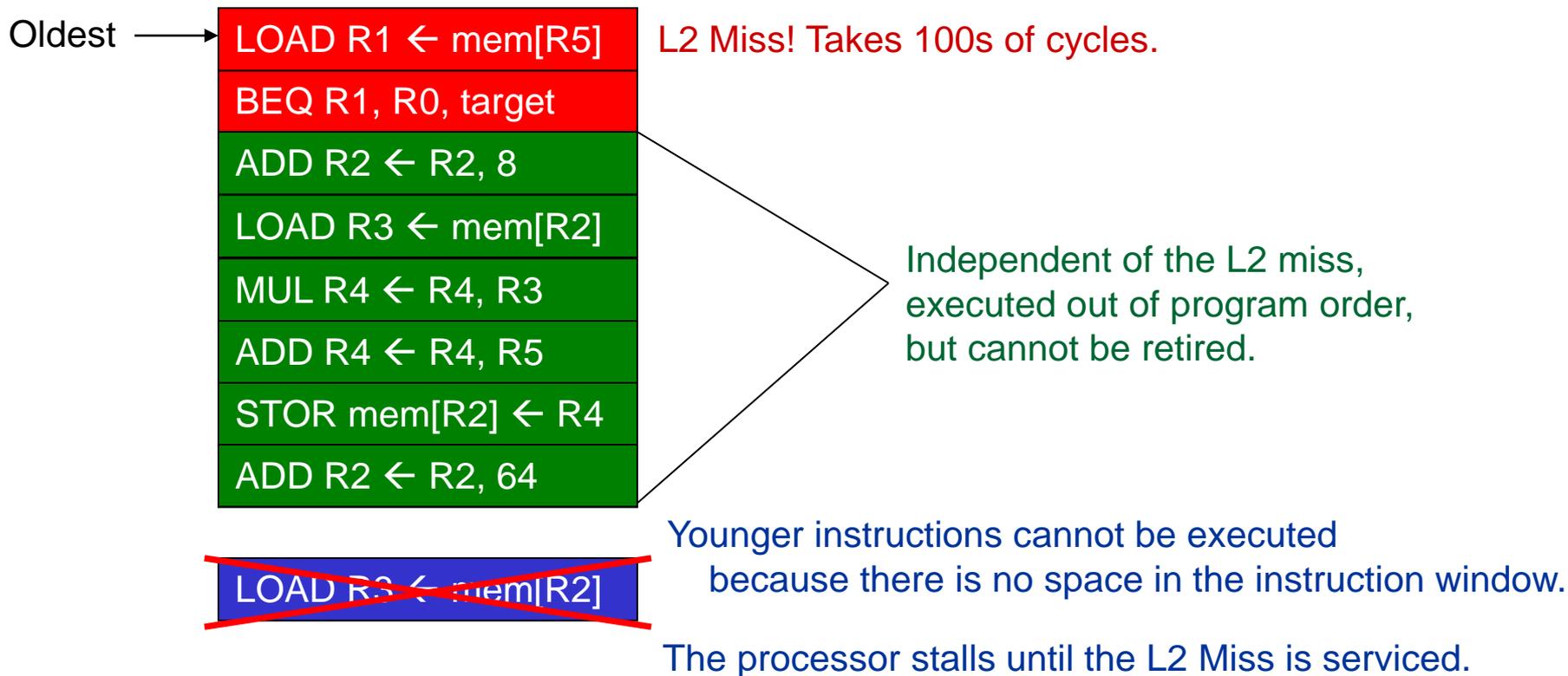
# Stalls due to Long-Latency Instructions

---

- When a **long-latency instruction** is not complete, it **blocks instruction retirement**.
  - Because we need to maintain precise exceptions
- Incoming instructions fill the instruction window (reorder buffer, reservation stations).
- Once the window is full, processor cannot place new instructions into the window.
  - This is called a **full-window stall**.
- A full-window stall prevents the processor from making progress in the execution of the program.

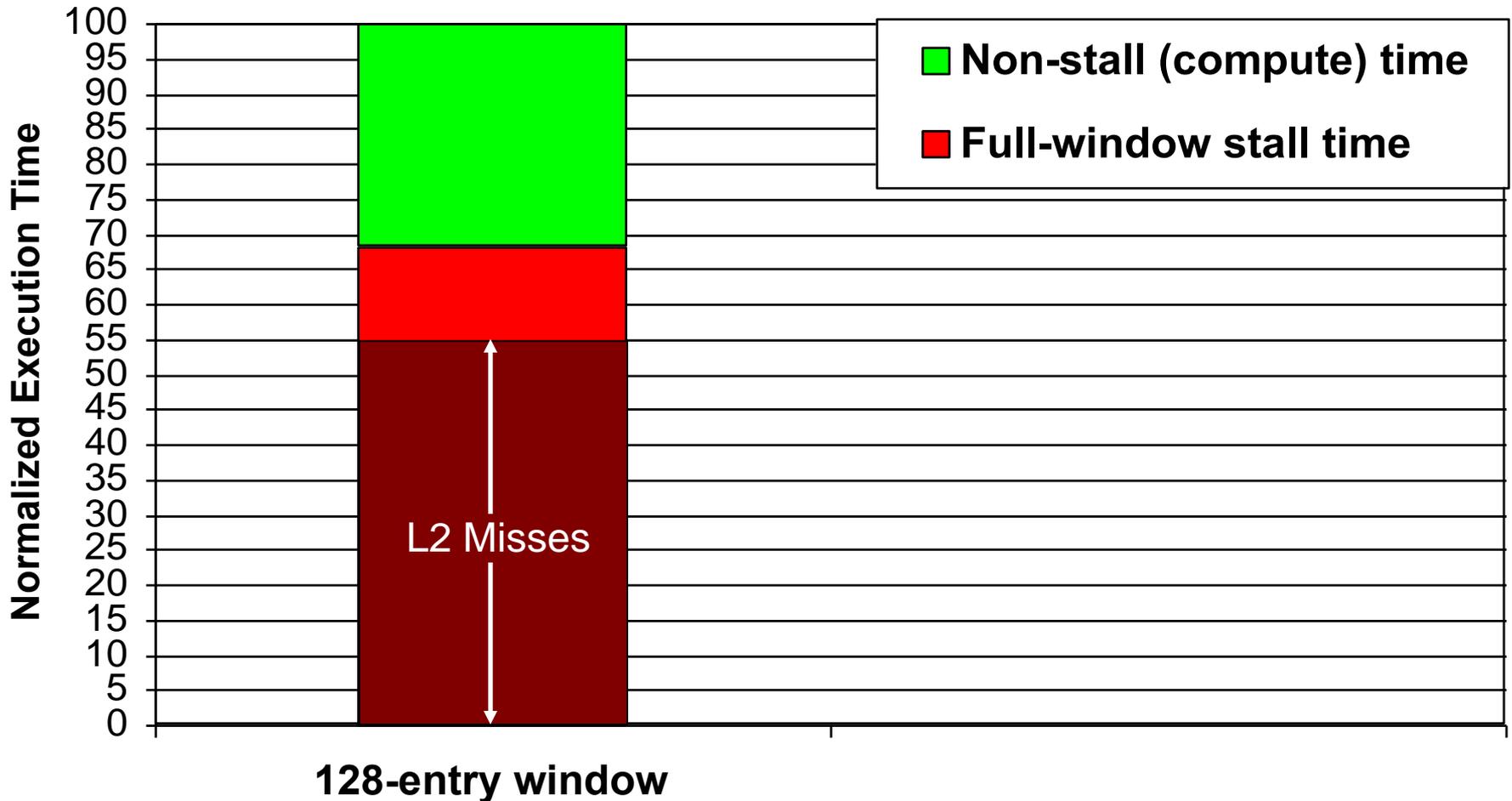
# Full-window Stall Example

8-entry instruction window:



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

# Cache Misses Responsible for Many Stalls



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

# How Do We Tolerate Stalls Due to Memory?

---

- Two major approaches
  - Reduce/eliminate stalls
  - Tolerate the effect of a stall when it happens
  
- Four fundamental techniques to achieve these
  - Caching
  - Prefetching
  - Multithreading
  - Out-of-order execution
  
- Many techniques have been developed to make these four fundamental techniques more effective in tolerating memory latency

# Memory Latency Tolerance Techniques

---

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

# Prefetching

# Today and Wednesday: Prefetching

---

- 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 (if we get to it)
- 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

---

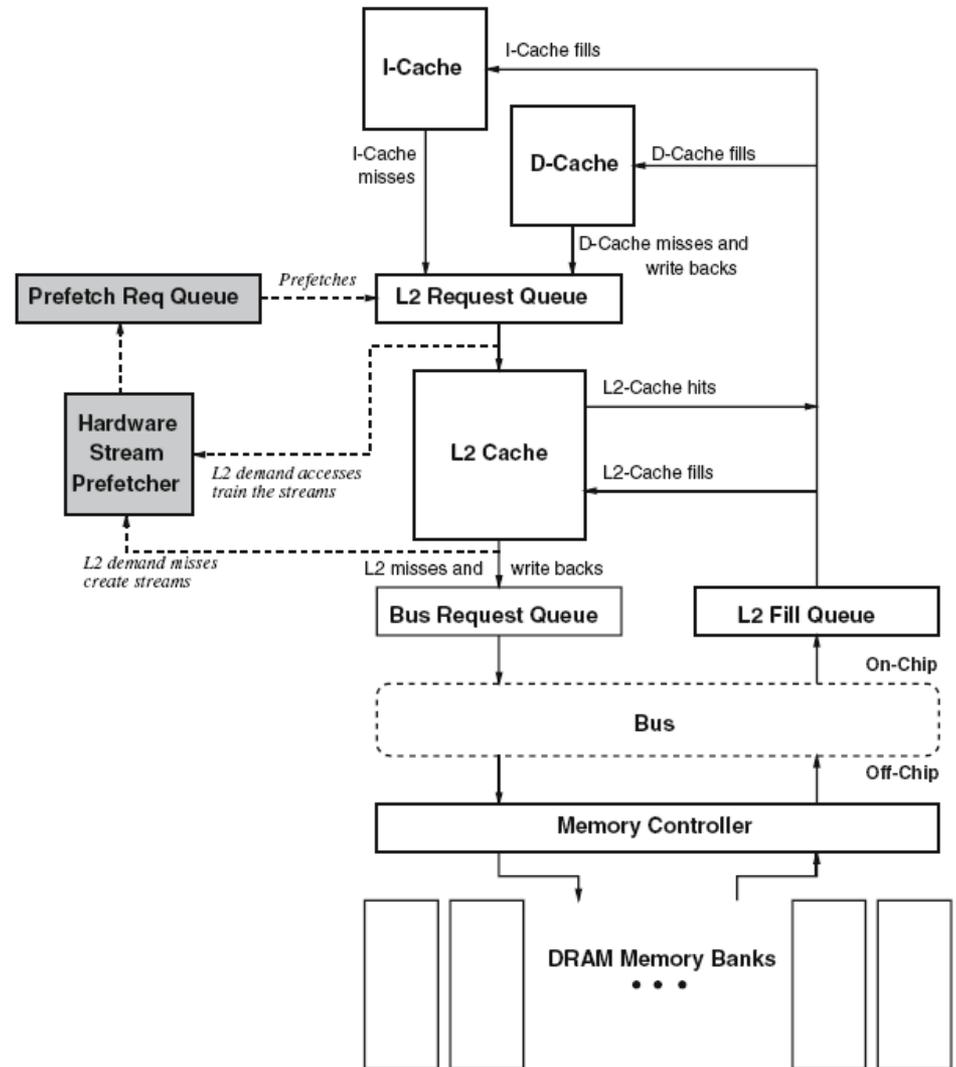
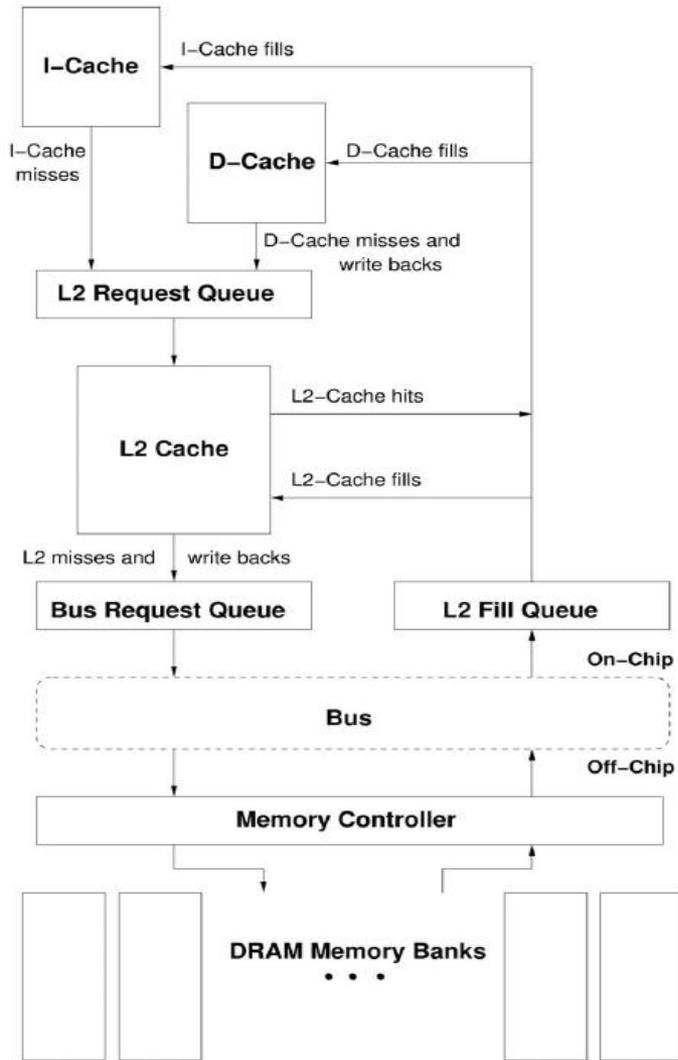
- Does a misprediction in prefetching affect correctness?
- No, prefetched data at a “mispredicted” address is simply not used
- There is no need for state recovery
- In contrast to branch misprediction or value misprediction

# 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)

---

```
for (i=0; i<N; i++) {
    __prefetch(a[i+8]);
    __prefetch(b[i+8]);
    sum += a[i]*b[i];
}

while (p) {
    __prefetch(p->next);
    work(p->data);
    p = p->next;
}

while (p) {
    __prefetch(p->next->next->next);
    work(p->data);
    p = p->next;
}
```

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?
    - Not really. 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

---

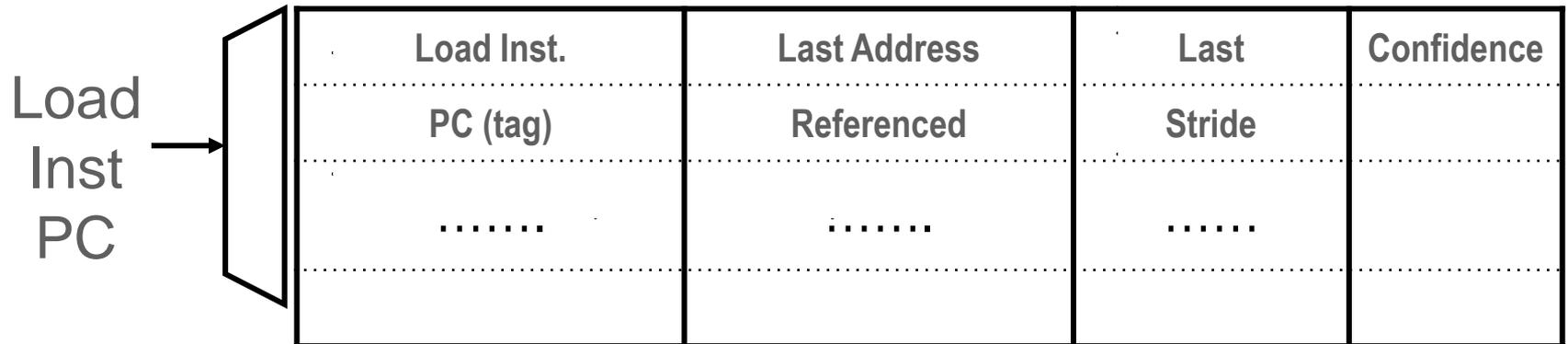
- Simplest form of hardware prefetching: always prefetch next N cache lines after a demand access (or a demand miss)
  - Next-line prefetcher (or next sequential prefetcher)
  - Tradeoffs:
    - + Simple to implement. No need for sophisticated pattern detection
    - + Works well for sequential/streaming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

---

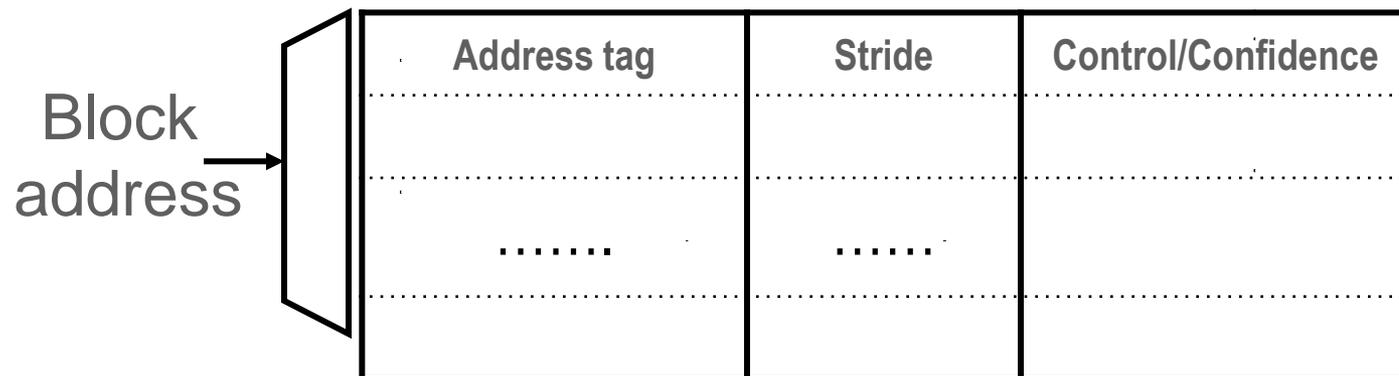
- Two kinds
  - Instruction program counter (PC) based
  - Cache block address based
- Instruction based:
  - Baer and Chen, “An effective on-chip preloading scheme to reduce data access penalty,” SC 1991.
  - Idea:
    - Record the distance between the memory addresses referenced by a load instruction (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?
  - Hint: how far can this get ahead? How much of the miss latency can the prefetch cover?
  - 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
    - Prefetch ahead ( $\text{last address} + N \cdot \text{stride}$ )
    - Generate multiple prefetches

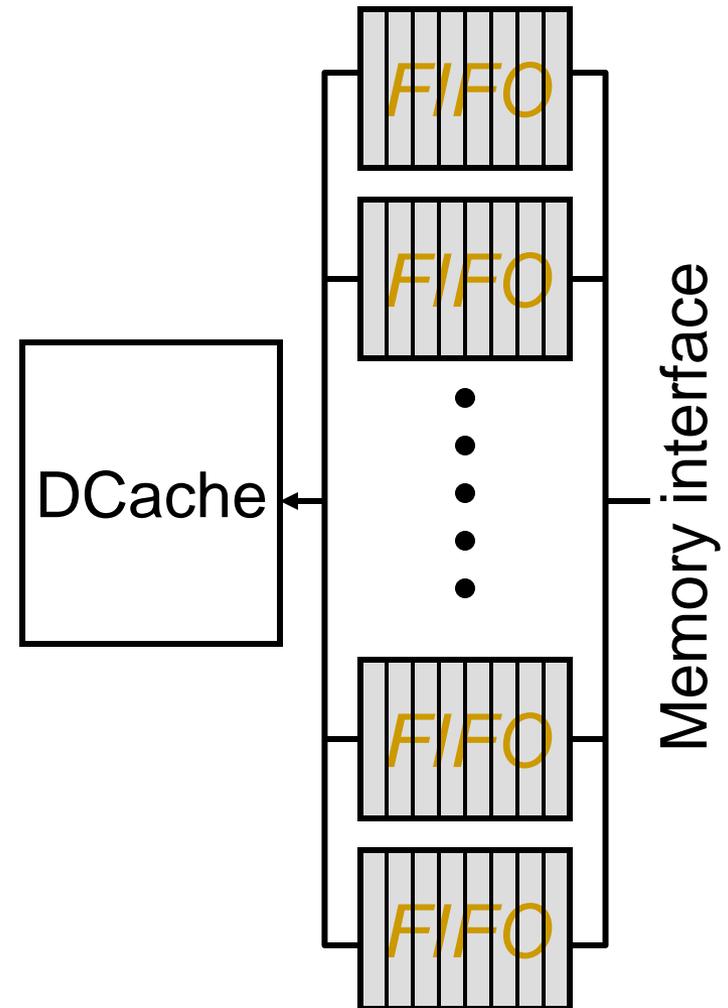
# Cache-Block Address Based Stride Prefetching



- Can detect
  - $A, A+N, A+2N, A+3N, \dots$
  - **Stream buffers** are a special case of cache block address based stride prefetching where  $N = 1$ 
    - Read the Jouppi paper
    - Stream buffer also has data storage in that paper (no prefetching into cache)

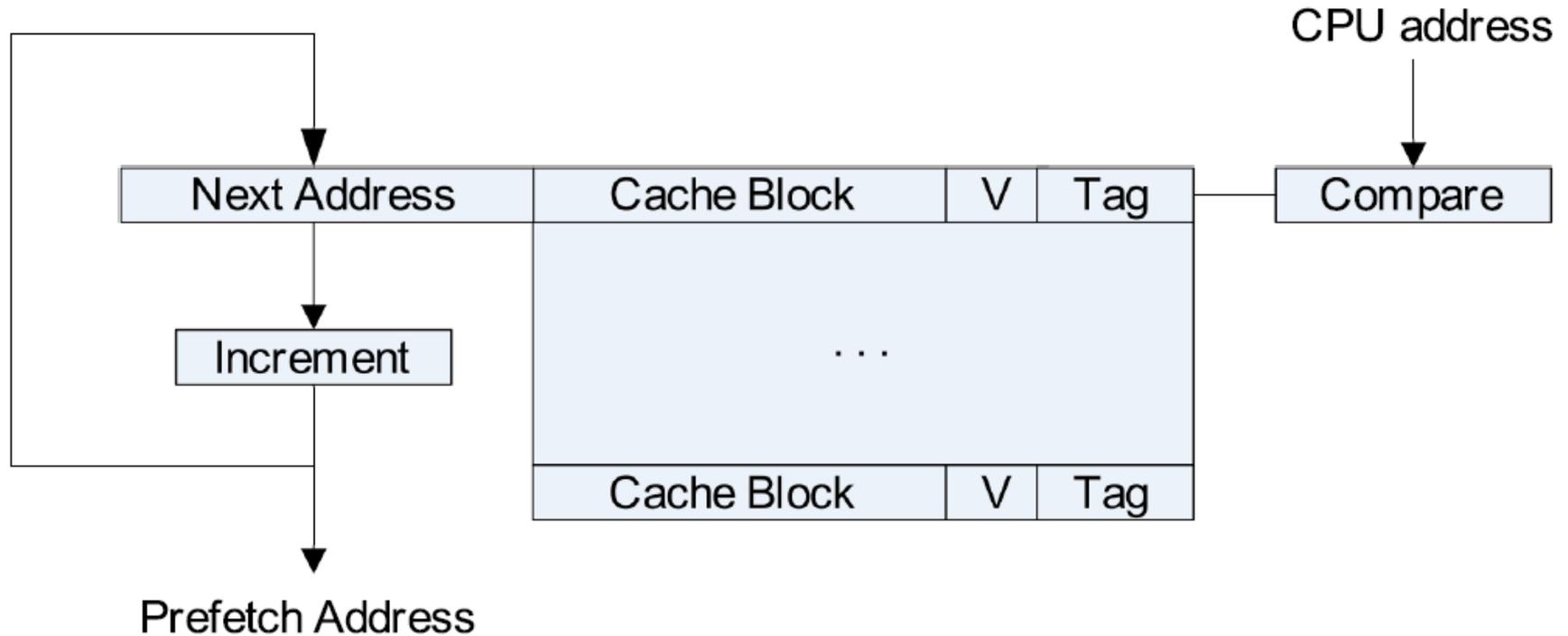
# Stream Buffers (Jouppi, ISCA 1990)

- 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

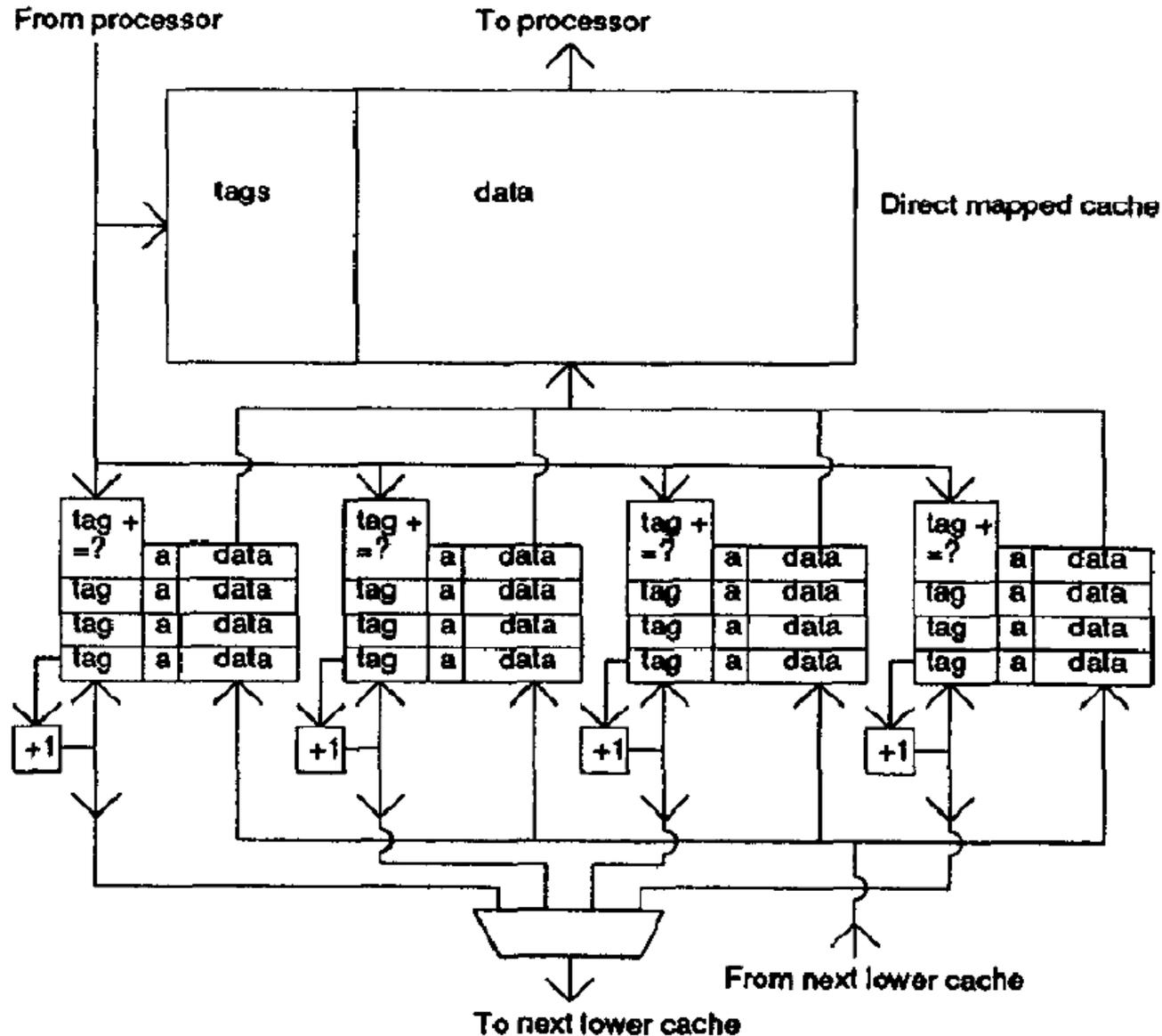


# Stream Buffer Design

---



# Stream Buffer Design



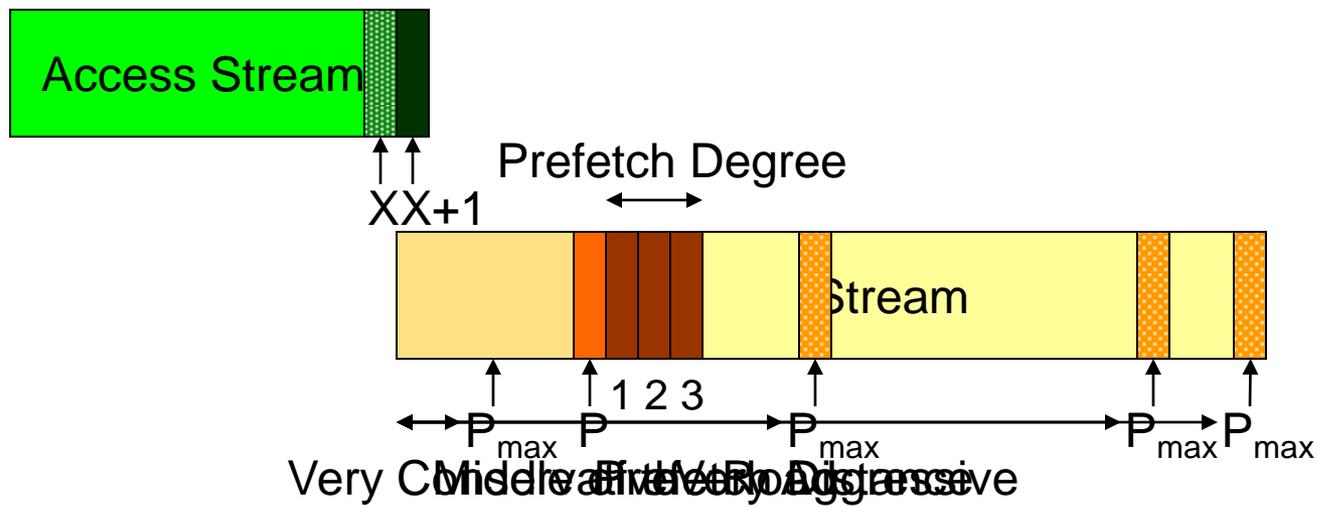
# Prefetcher Performance (I)

---

- **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

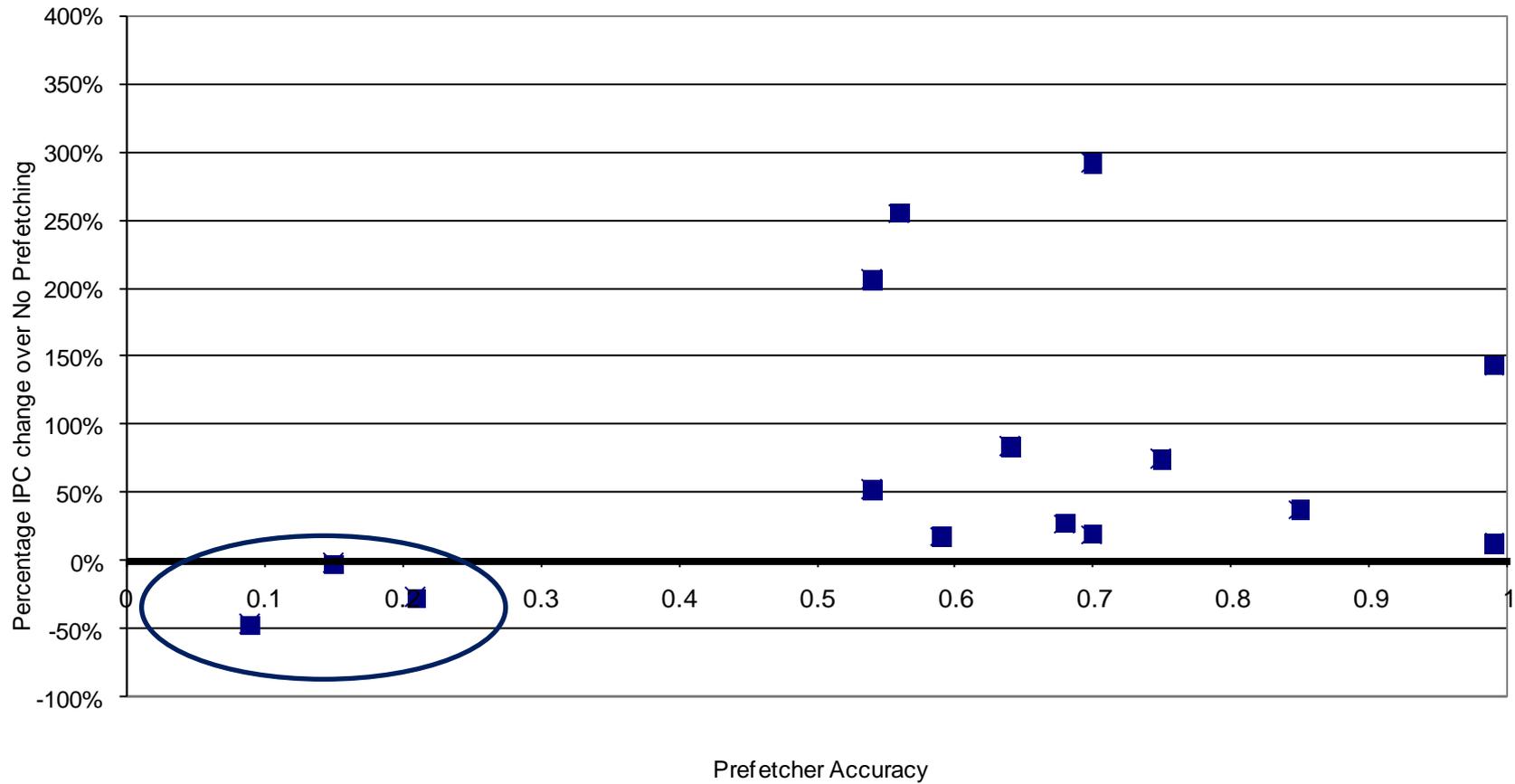


# Prefetcher Performance (III)

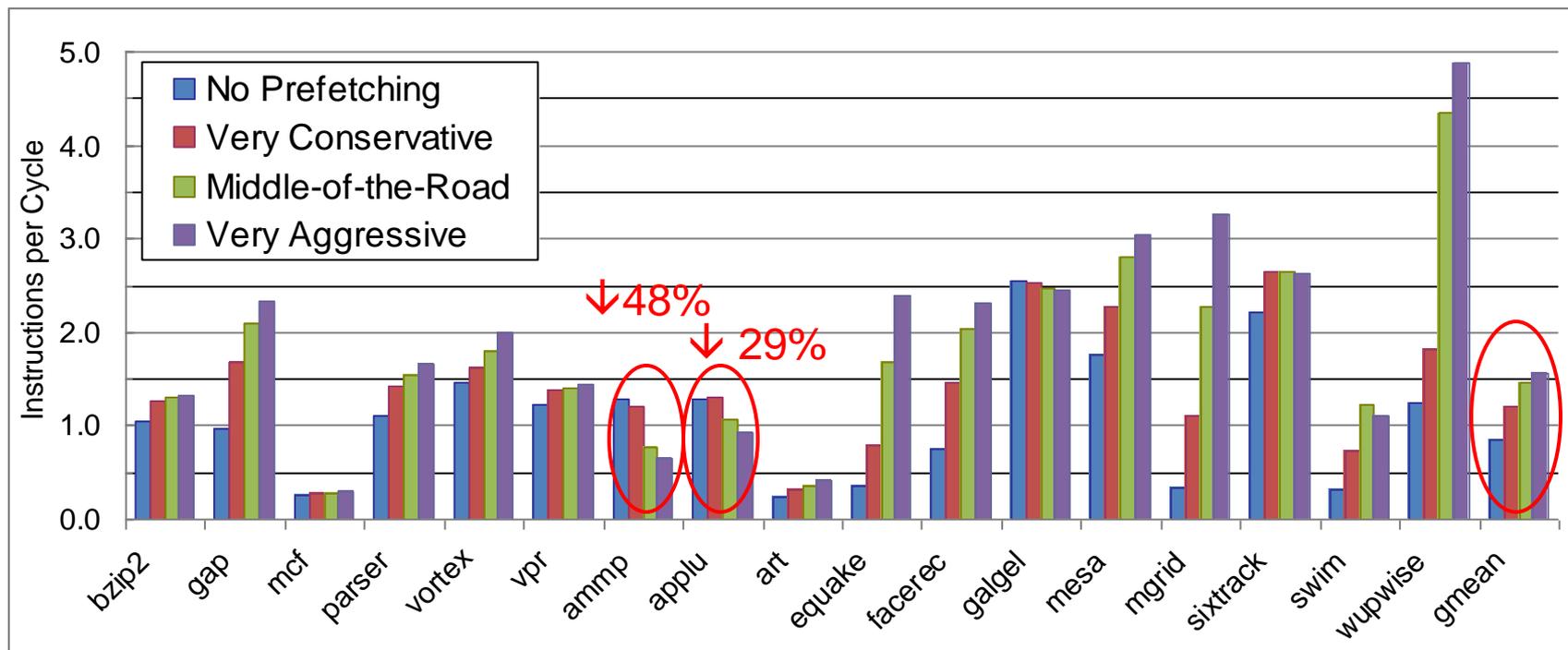
---

- How do these metrics interact?
- Very Aggressive
  - 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
  - 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)



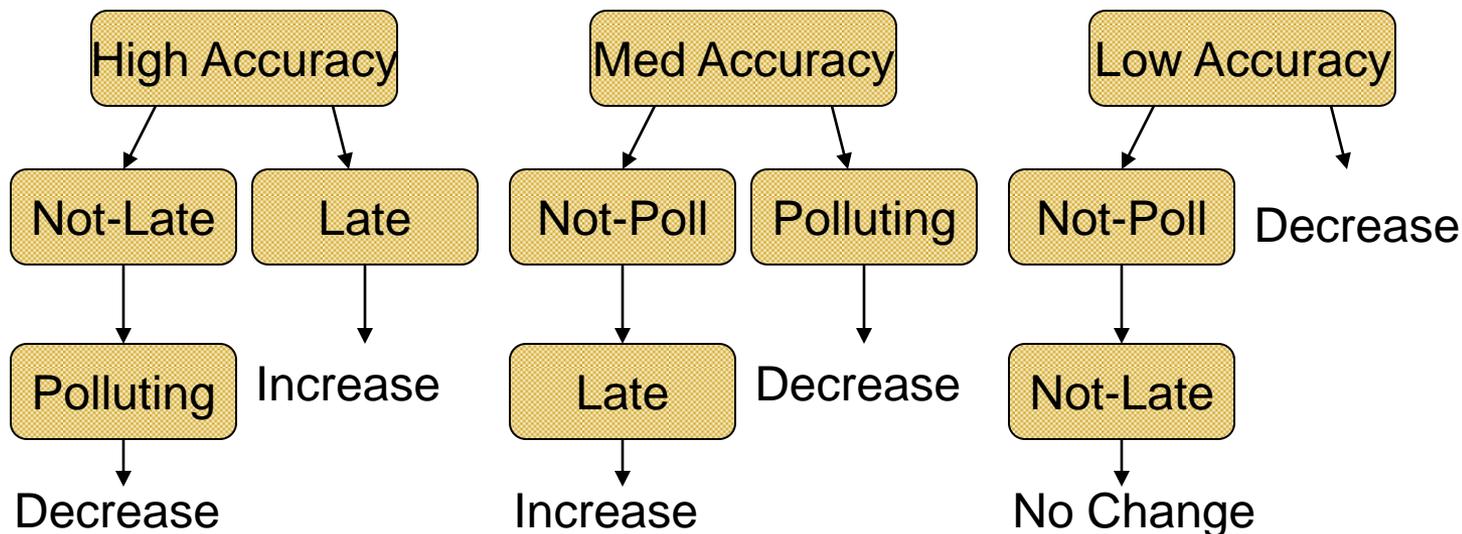
# Prefetcher Performance (V)



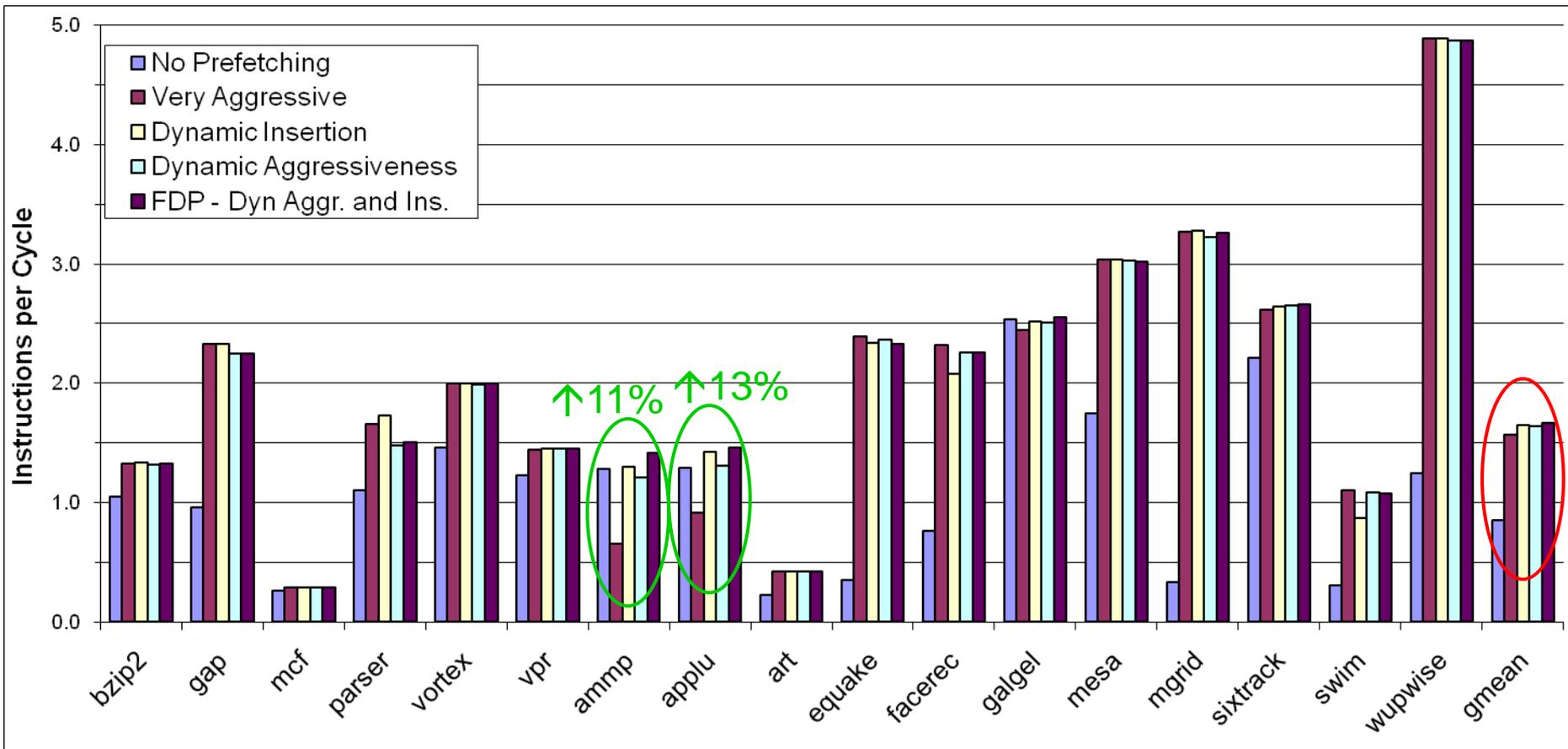
- Srinath et al., “Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers”, HPCA 2007.

# Feedback-Directed Prefetcher Throttling (I)

- Idea:
  - 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.