Computer Architecture
Lecture 18: Prefetching

Prof. Onur Mutlu
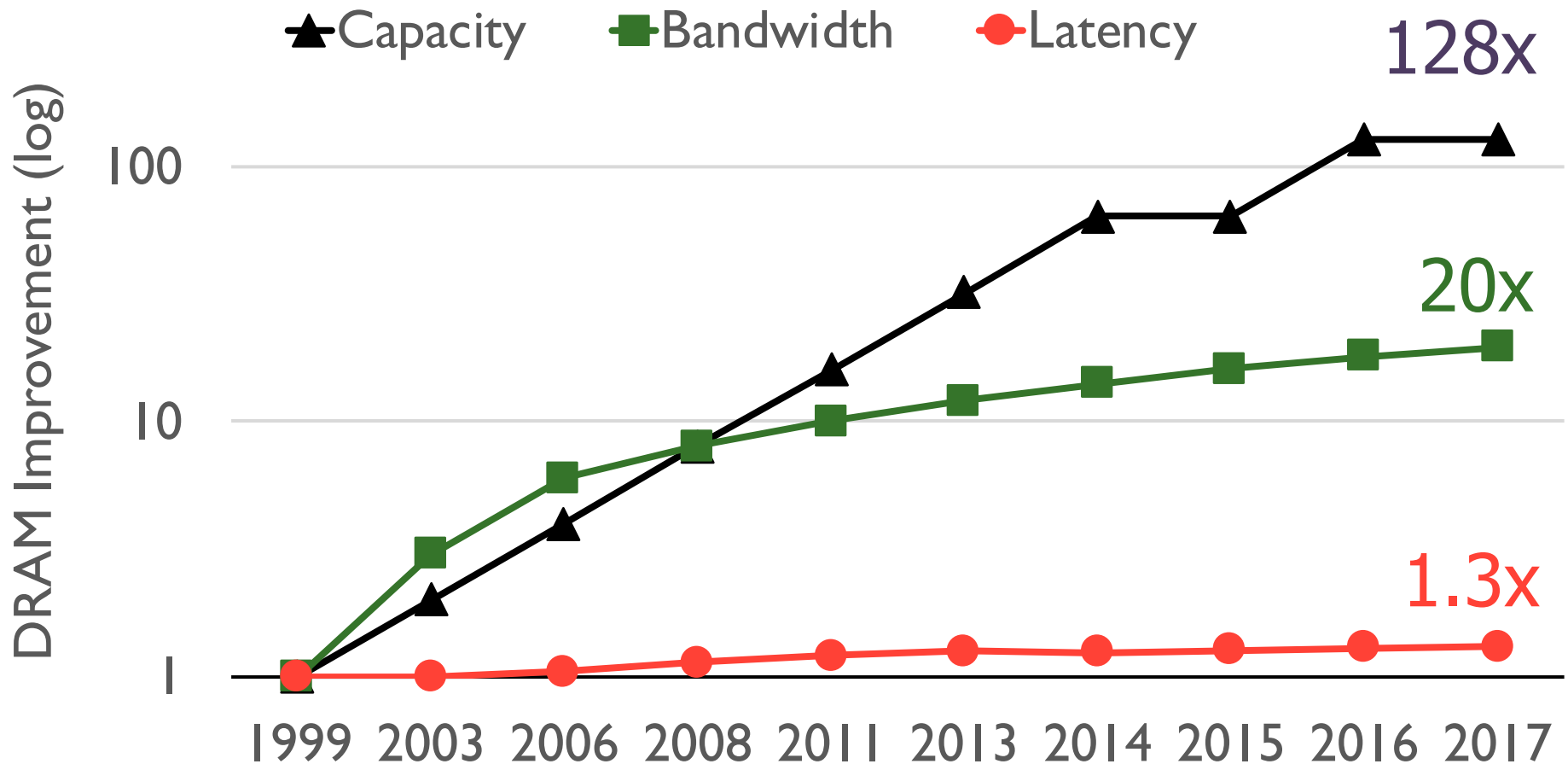
ETH Zürich

Fall 2020

26 November 2020

The (Memory) Latency Problem

Recall: Memory Latency Lags Behind

Memory latency remains almost constant
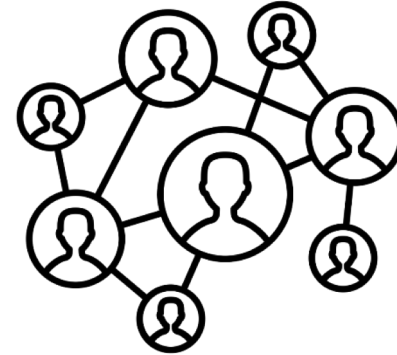
DRAM Latency Is Critical for Performance

In-memory Databases

[Mao+, EuroSys'12;
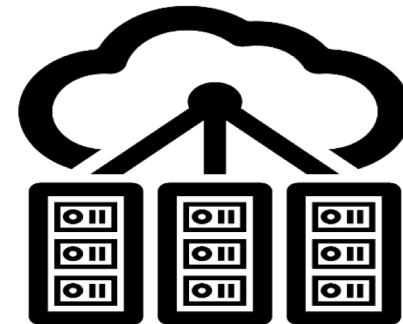 Clapp+ (**Intel**), IISWC'15]

Graph/Tree Processing

[Xu+, IISWC'12; Umuroglu+, FPL'15]

In-Memory Data Analytics

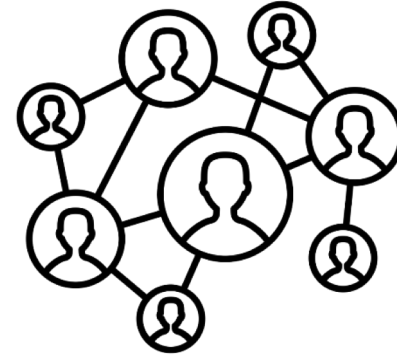[Clapp+ (**Intel**), IISWC'15;
 Awan+, BDCloud'15]

Datacenter Workloads

[Kanev+ (**Google**), ISCA'15]

DRAM Latency Is Critical for Performance
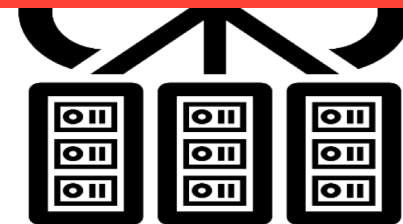
In-memory Databases

Graph/Tree Processing

Long memory latency → performance bottleneck

In-Memory Data Analytics
[Clapp+ (**Intel**), IISWC'15;
Awan+, BDCloud'15]

Datacenter Workloads
[Kanev+ (**Google**), ISCA'15]

SAFARI

New DRAM Types **Increase** Latency!

- Saugata Ghose, Tianshi Li, Nastaran Hajinazar, Damla Senol Cali, and Onur Mutlu,
  "Demystifying Workload–DRAM Interactions: An Experimental Study"
  Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (**SIGMETRICS**), Phoenix, AZ, USA, June 2019.
  [Preliminary arXiv Version]
  [Abstract]
  [Slides (pptx) (pdf)]
  [MemBen Benchmark Suite]
  [Source Code for GPGPUSim-Ramulator]

Demystifying Complex Workload–DRAM Interactions: An Experimental Study

Saugata Ghose[†]   Tianshi Li[†]   Nastaran Hajinazar[‡†]

Damla Senol Cali[†]   Onur Mutlu[§†]
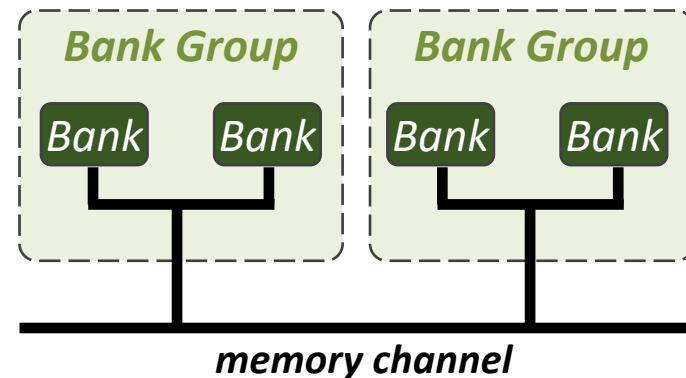
[†]Carnegie Mellon University   [‡]Simon Fraser University   [§]ETH Zürich

Modern DRAM Types: Comparison to DDR3

DRAM Type	Banks per Rank	Bank Groups	3D-Stacked	Low-Power
DDR3	8			
DDR4	16	✓		
GDDR5	16	✓		
HBM High-Bandwidth Memory	16		✓	
HMC Hybrid Memory Cube	256		✓	
Wide I/O	4		✓	✓
Wide I/O 2	8		✓	✓
LPDDR3	8			✓
LPDDR4	16			✓

increased latency

increased area/power

narrower rows, higher latency

■ **Bank groups**

memory channel

■ **3D-stacked DRAM**

high bandwidth with **Through-Silicon Vias (TSVs)**

Memory Layers

dedicated **Logic Layer**

▪ **New DRAM types often increase access latency in order to provide more banks, higher throughput**

▪ **Many applications can't make up for the increased latency**

- Especially true of common OS routines (e.g., file I/O, process forking)
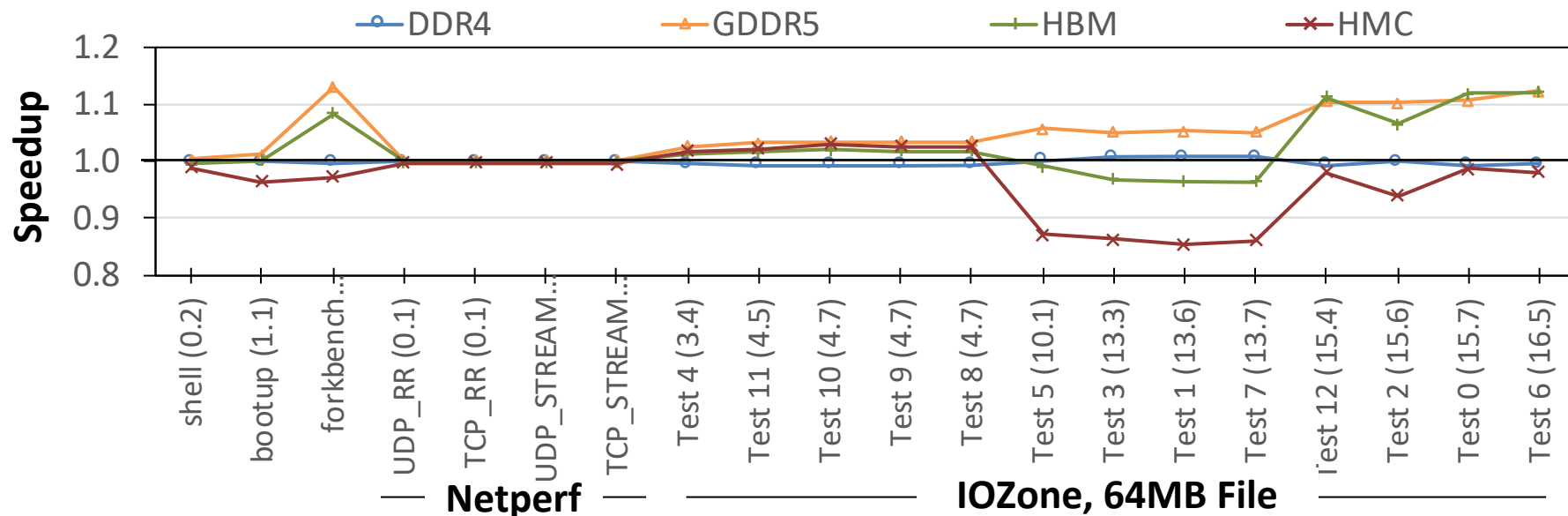
- A variety of desktop/scientific, server/cloud, GPGPU applications

Several applications don't benefit from more parallelism

1. **DRAM latency remains a critical bottleneck for many applications**

2. **Bank parallelism is not fully utilized by a wide variety of our applications**

3. **Spatial locality continues to provide significant performance benefits if it is exploited by the memory subsystem**

4. **For some classes of applications, low-power memory can provide energy savings without sacrificing significant performance**

New DRAM Types **Increase** Latency!

- Saugata Ghose, Tianshi Li, Nastaran Hajinazar, Damla Senol Cali, and Onur Mutlu,
  "Demystifying Workload–DRAM Interactions: An Experimental Study"
  Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (**SIGMETRICS**), Phoenix, AZ, USA, June 2019.
  [Preliminary arXiv Version]
  [Abstract]
  [Slides (pptx) (pdf)]
  [MemBen Benchmark Suite]
  [Source Code for GPGPUSim-Ramulator]

Demystifying Complex Workload–DRAM Interactions: An Experimental Study

Saugata Ghose[†]   Tianshi Li[†]   Nastaran Hajinazar[‡†]

Damla Senol Cali[†]   Onur Mutlu[§†]

[†]Carnegie Mellon University   [‡]Simon Fraser University   [§]ETH Zürich

Latency Reduction, Latency Tolerance, and Latency Hiding Techniques

Latency Reduction, Tolerance and Hiding

- **Fundamentally reduce latency as much as possible**
  - ❑ Data-centric approach
  - ❑ See Lecture 10: Low-Latency Memory
  - ❑ https://www.youtube.com/watch?v=vQd1YgOH1Mw

- **Hide latency seen by the processor**
  - ❑ Processor-centric approach
  - ❑ Caching, Prefetching

- **Tolerate (or, amortize) latency seen by the processor**
  - ❑ Processor-centric approach
  - ❑ Multithreading, Out-of-order Execution, Runahead Execution

Conventional Latency Tolerance Techniques

- Caching [initially by Wilkes, 1965]
  - Widely used, simple, effective, but inefficient, passive
  - Not all applications/phases exhibit temporal or spatial locality

- Prefetching [initially in IBM 360/91, 1967]
  - Works well for regular memory access patterns
  - Prefetching irregular access patterns is difficult, inaccurate, and hardware-intensive

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

- Out-of-order execution [initially by Tomasulo, 1967]
  - Tolerates cache misses that cannot be prefetched
  - Requires extensive hardware resources for tolerating long latencies

Lectures on Latency Tolerance

- **Caching**
  - http://www.youtube.com/watch?v=mZ7CPJKzwfM
  - http://www.youtube.com/watch?v=TsxQPLMXT60
  - http://www.youtube.com/watch?v=OUk96_Bz708
  - And more here: https://safari.ethz.ch/architecture/fall2018/doku.php?id=schedule

- **Prefetching**
  - Today
  - Also: http://www.youtube.com/watch?v=CLi04cG9aQ8

- **Multithreading**
  - http://www.youtube.com/watch?v=bu5dxKTvQVs
  - https://www.youtube.com/watch?v=iqi9wFqFiNU
  - https://www.youtube.com/watch?v=e8lfl6MbILg
  - https://www.youtube.com/watch?v=7vkDpZ1-hHM

- **Out-of-order Execution, Runahead Execution**
  - http://www.youtube.com/watch?v=EdYAKfx9JEA
  - http://www.youtube.com/watch?v=WExCvQAuTxo
  - http://www.youtube.com/watch?v=Kj3relihGF4

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
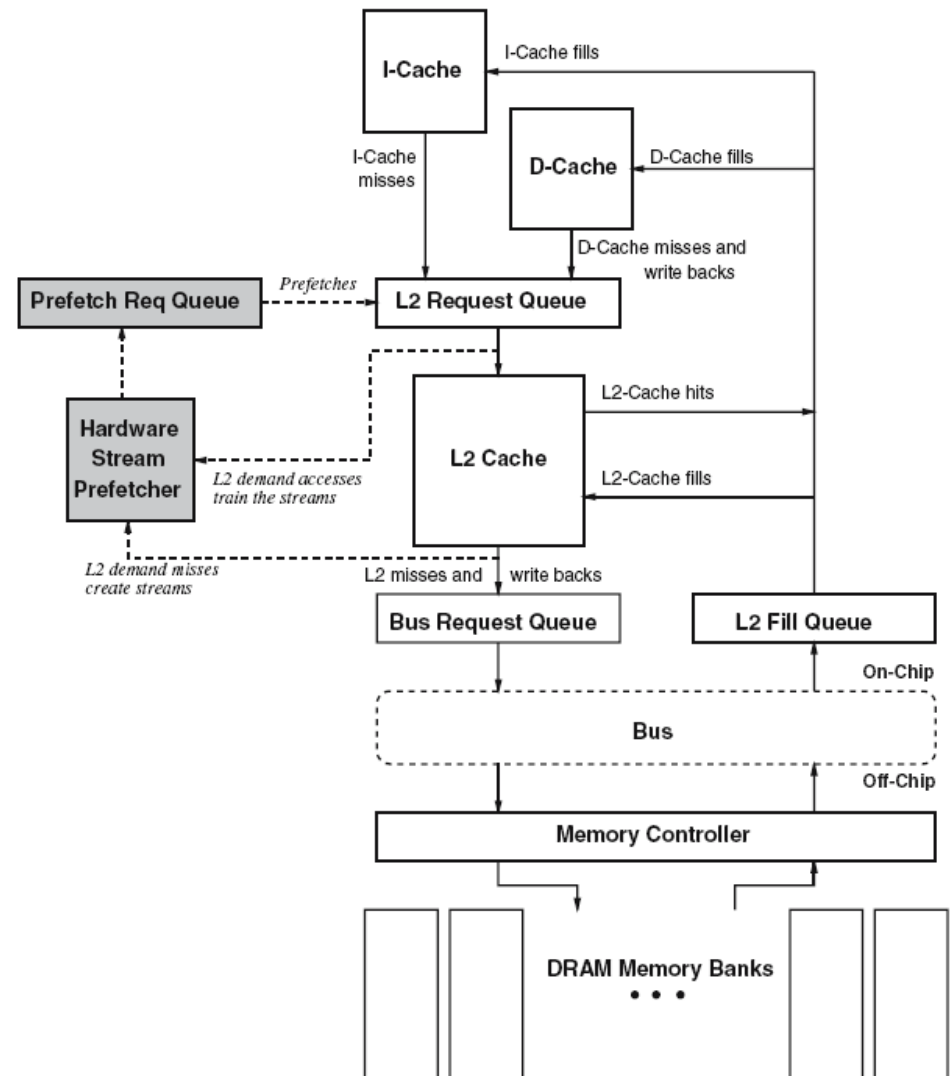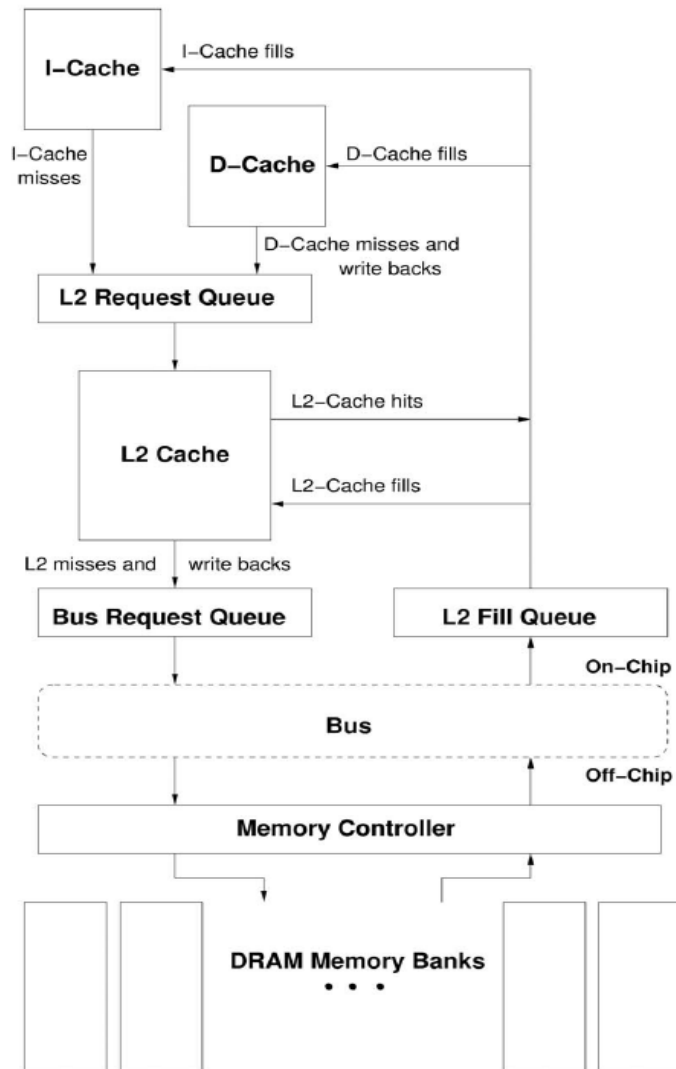
Prefetching: The Four Questions

- **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

PREFETCH*h*—Prefetch Data Into Caches

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

30

Software Prefetching (II)

```
for (i=0; i<N; i++) {         while (p) {                  while (p) {
    __prefetch(a[i+8]);           __prefetch(p→next);          __prefetch(p→next→next→next);
    __prefetch(b[i+8]);           work(p→data);                work(p→data);
    sum += a[i]*b[i];             p = p→next;                  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?

    ■ 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

Load Inst. PC (tag)	Last Address Referenced	Last Stride	Confidence
…….	…….	……	

Load Inst PC
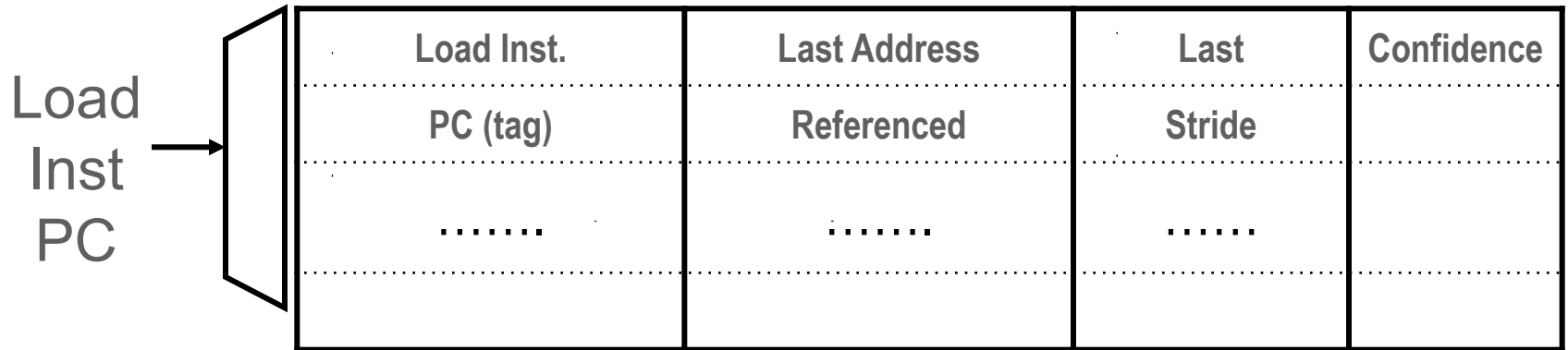
- **What is the problem with this?**
  - How far can the prefetcher get ahead of the demand access stream?
  - Initiating the prefetch when the load is fetched the next time can be too late
    - Load will access the data cache soon after it is fetched!
  - Solutions:
    - Use lookahead PC to index 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

Address tag	Stride	Control/Confidence
....	......	
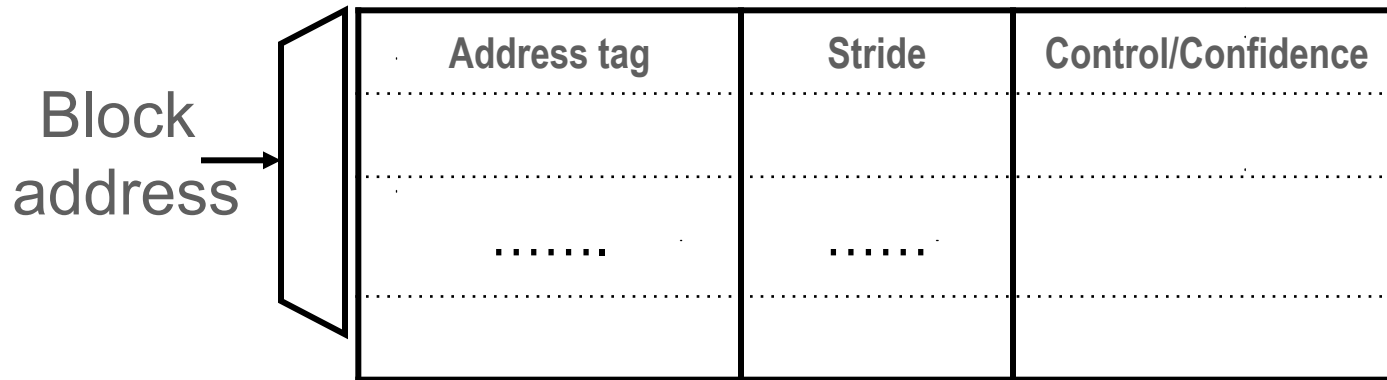
Block address →

- Can detect
  - A, A+N, A+2N, A+3N, …
  - 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
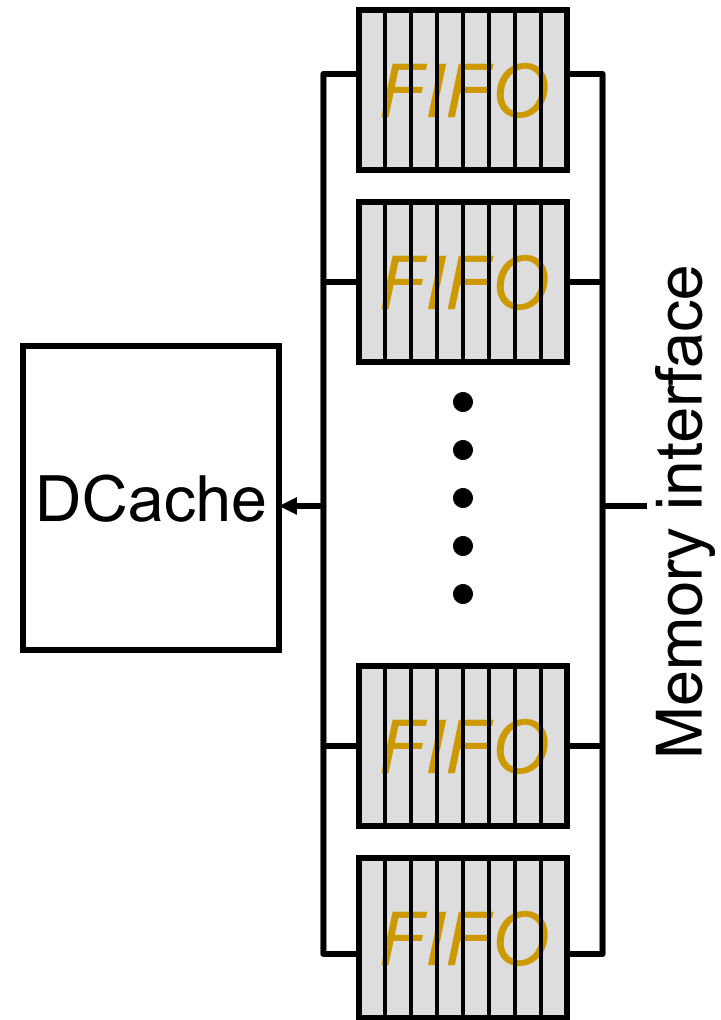
Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," ISCA 1990.

Stream Buffer Design

Stream Buffer Design
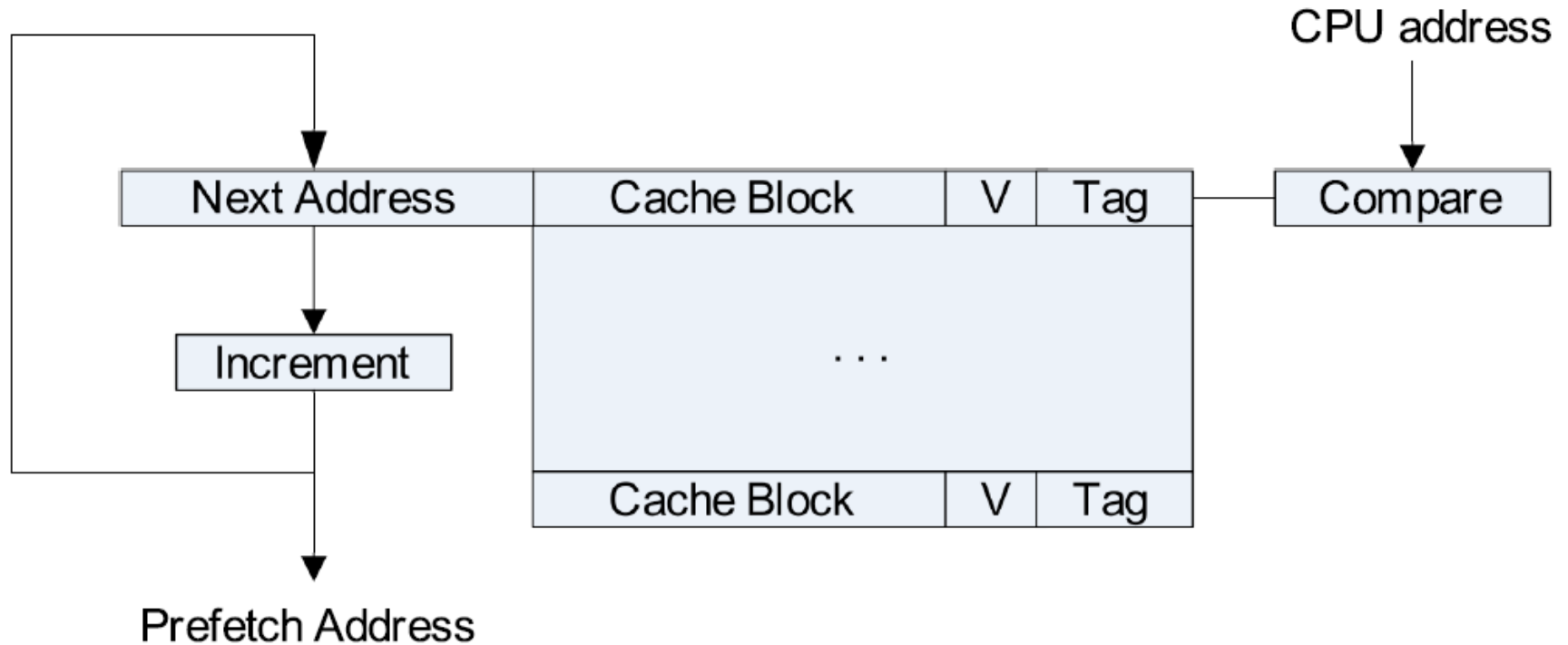
Tradeoffs in Stride Prefetching

- 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, ..., 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
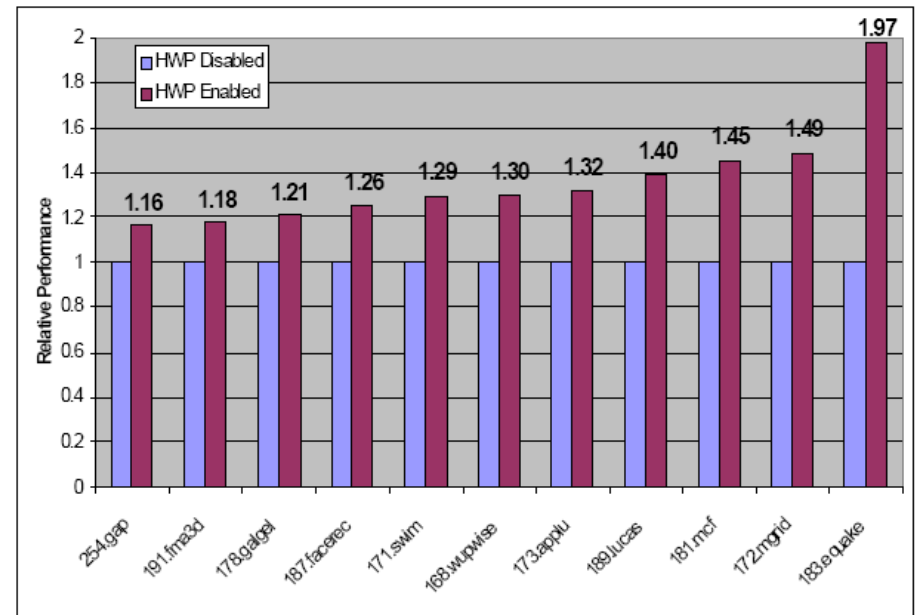
- **Limited to prefetching closeby addresses**
  - What about large jumps in addresses accessed?

- **However, they work very well in real life**
  - Single-core systems
  - Boggs et al., "The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology", 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
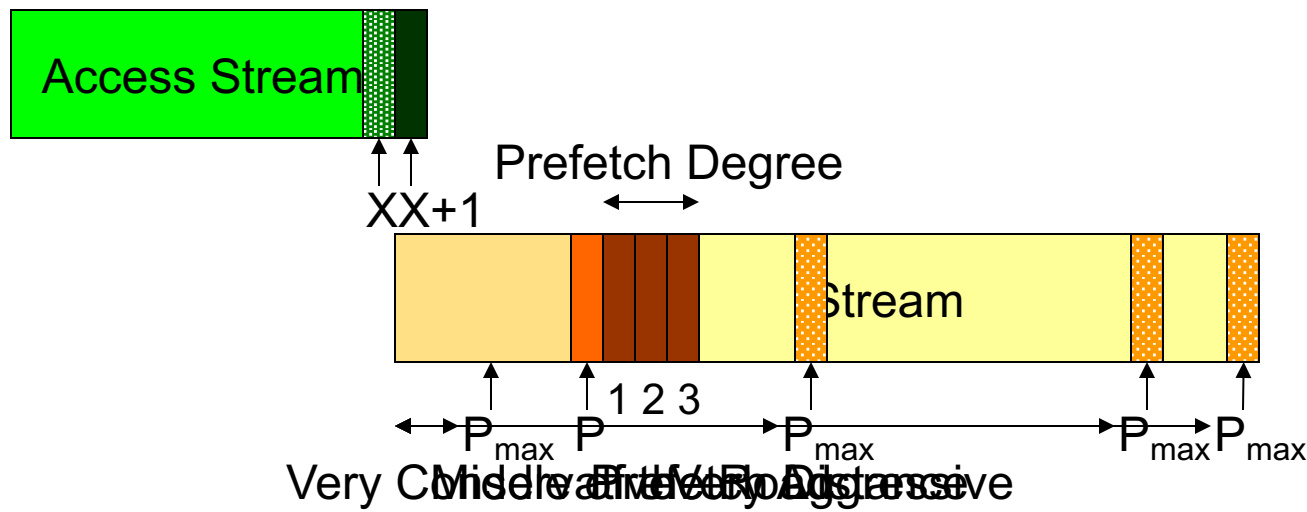
Prefetcher Performance (III)

- 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)
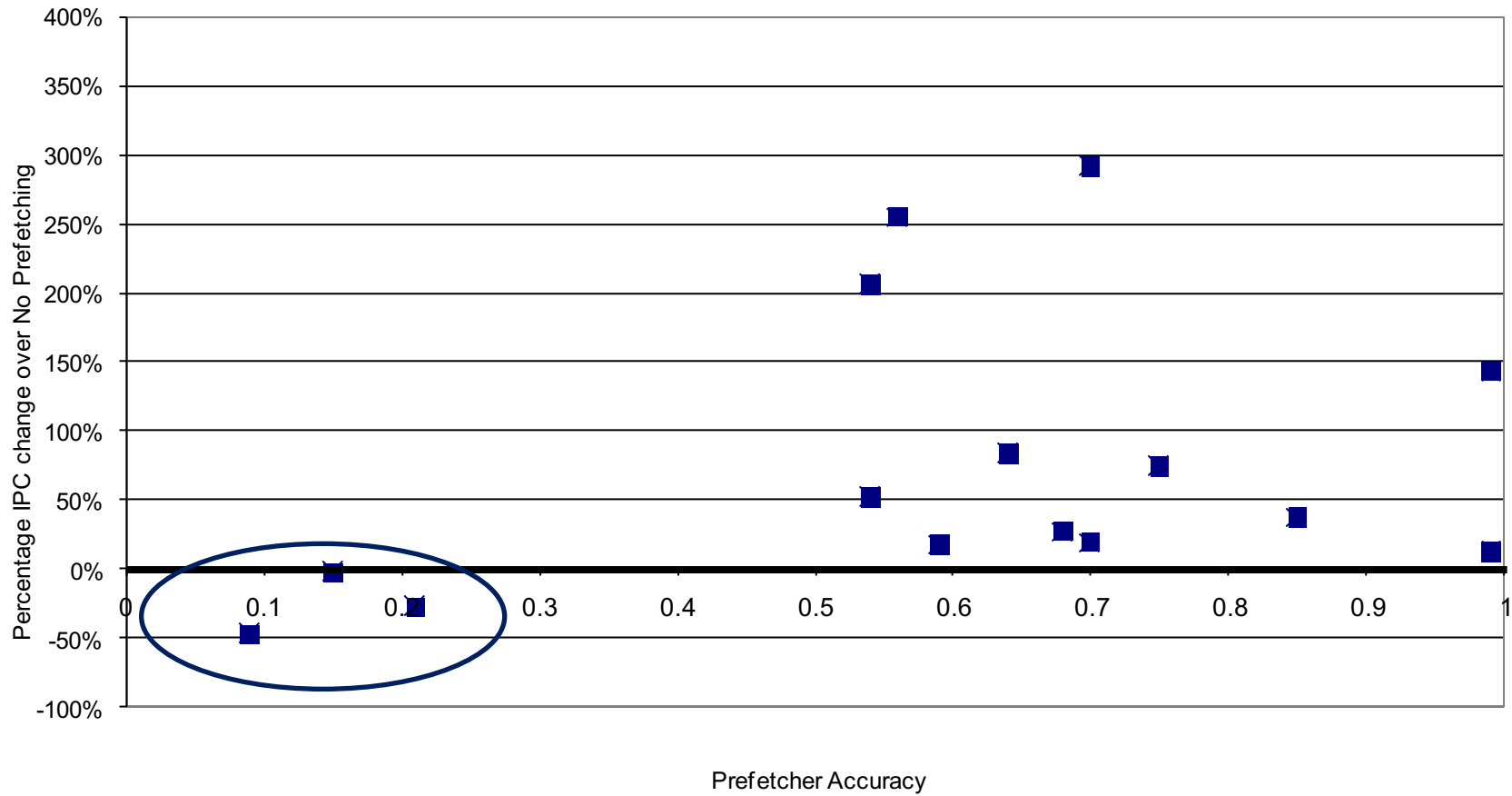
Prefetcher Performance (V)
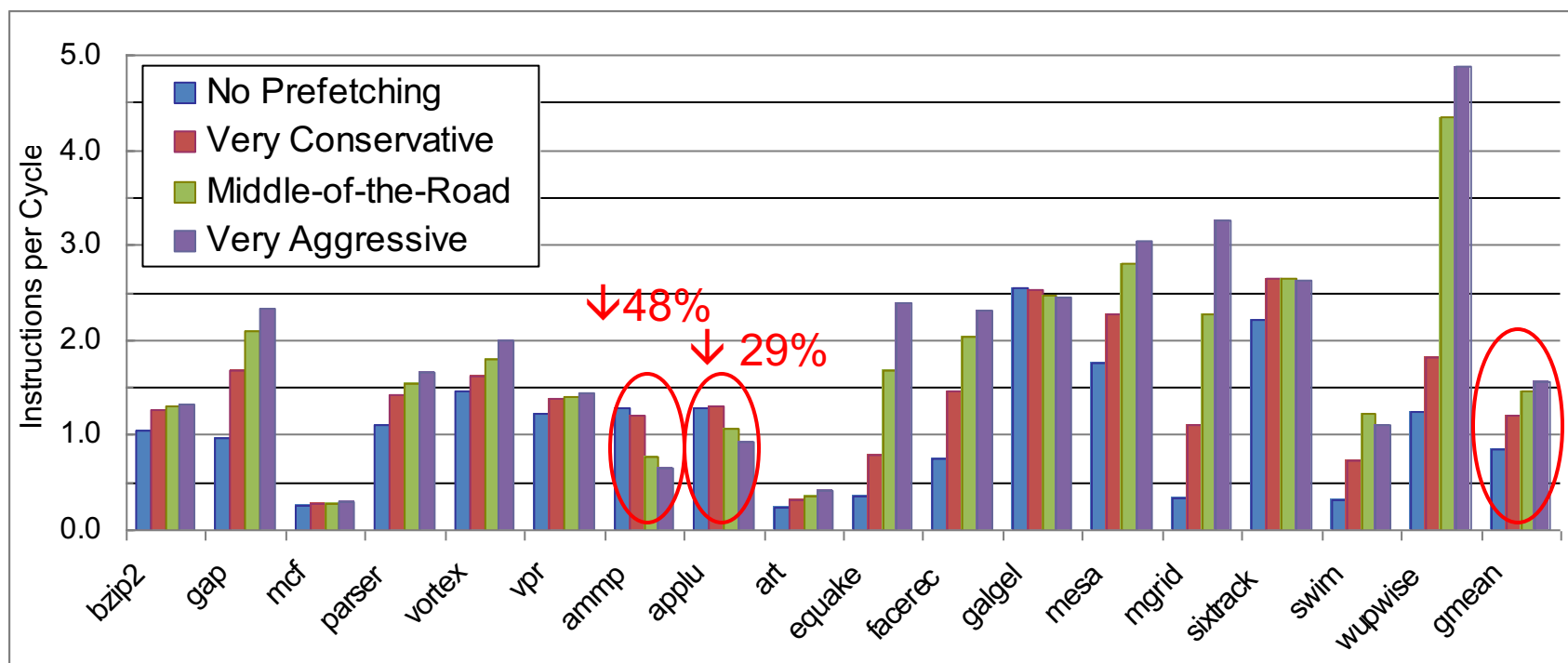
- Srinath et al., "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers", HPCA 2007.

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
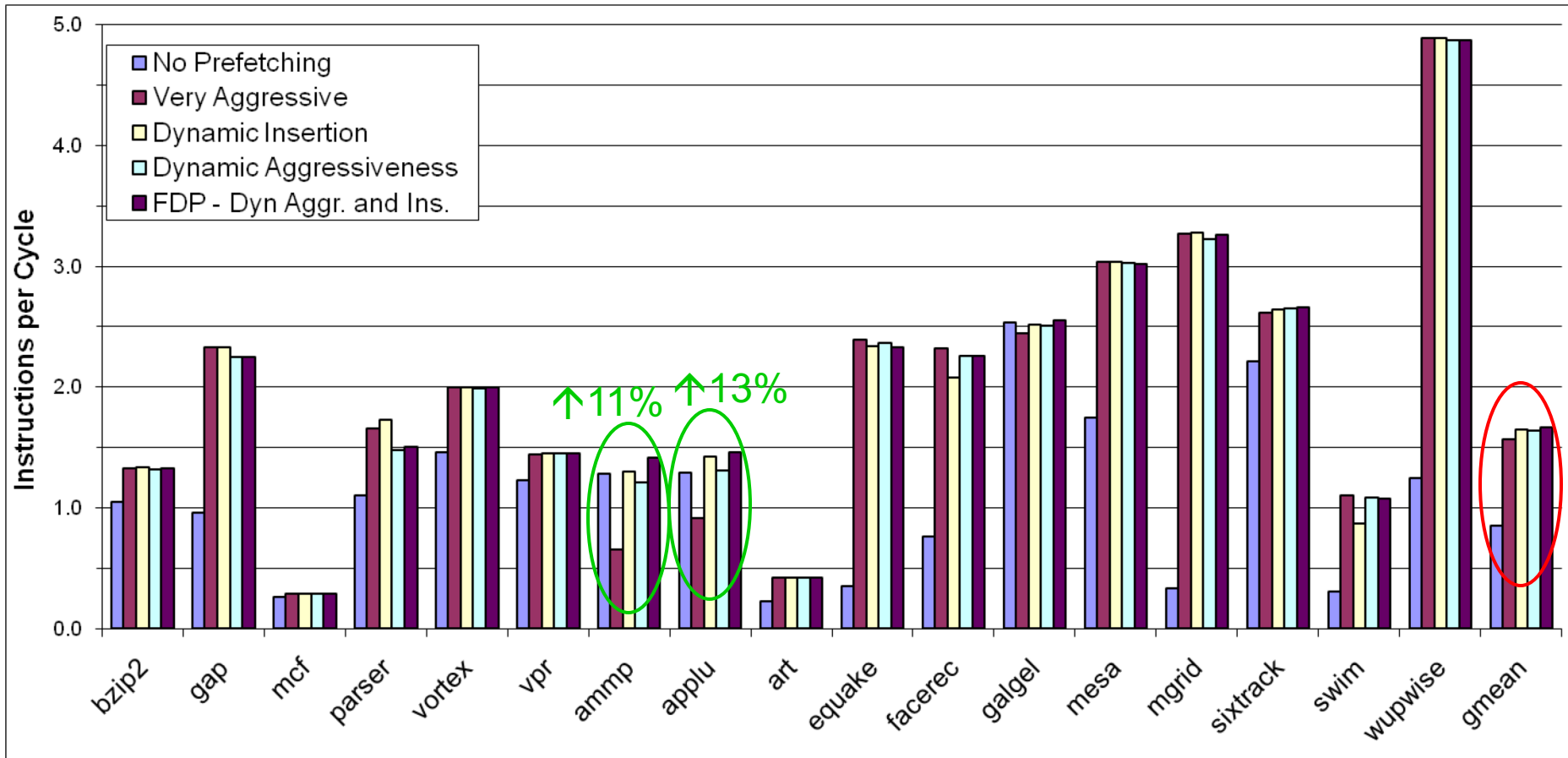
```
High Accuracy              Med Accuracy              Low Accuracy
    ↓       ↓                  ↓        ↓                 ↓         ↓
Not-Late   Late           Not-Poll  Polluting        Not-Poll   Decrease
    ↓       ↓                 ↓         ↓                ↓
Polluting  Increase        Late     Decrease         Not-Late
    ↓                        ↓                           ↓
Decrease                  Increase                   No Change
```

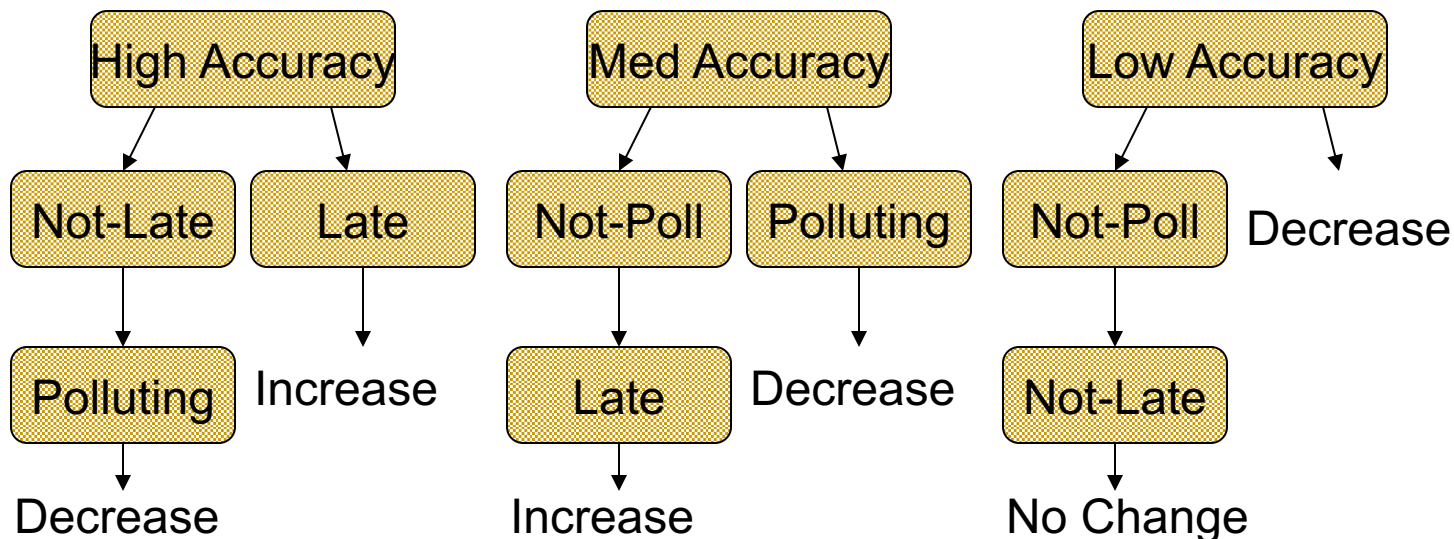# Feedback-Directed Prefetcher Throttling (II)

- 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)
One of the five papers nominated for the Best Paper Award by the Program Committee.

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), some addresses are more likely to be referenced next
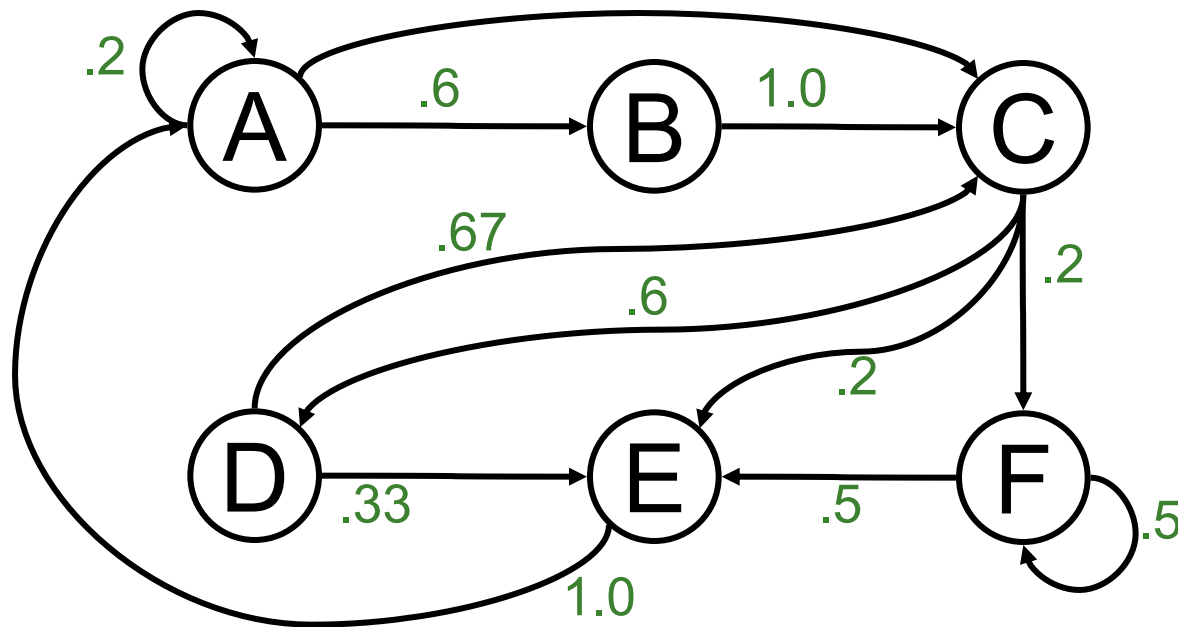
Markov Model

Address Correlation Based Prefetching (II)

Cache Block Addr	Prefetch	Confidence	....	Prefetch	Confidence
(tag)	Candidate 1		....	Candidate N	
....	....	......	....	....	......
			....		

Cache Block Addr →
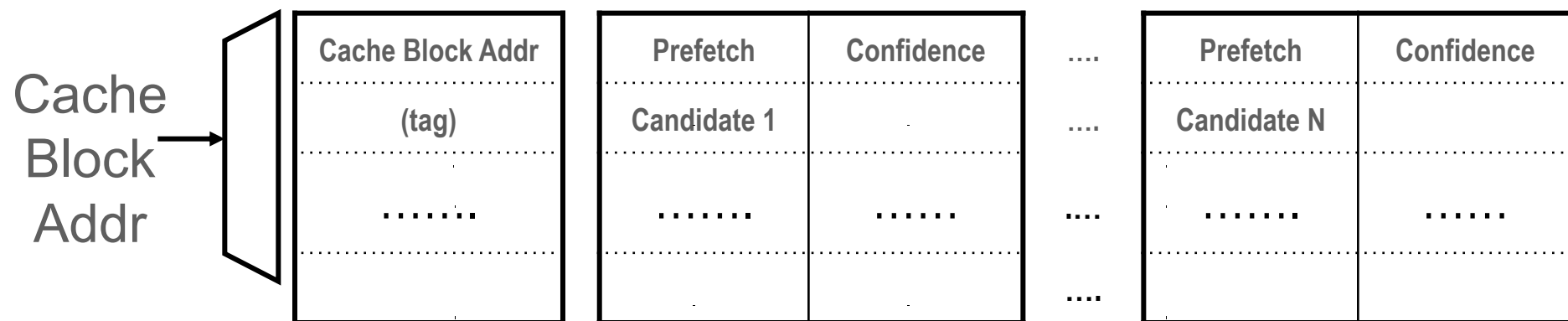
- Idea: Record the likely-next addresses (B, C, D) after seeing an address A
  - Next time A is accessed, prefetch B, C, D
  - A is said to be correlated with B, C, D
- Prefetch up to N next addresses to increase *coverage*
- Prefetch accuracy can be improved by using multiple addresses as key for the 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)
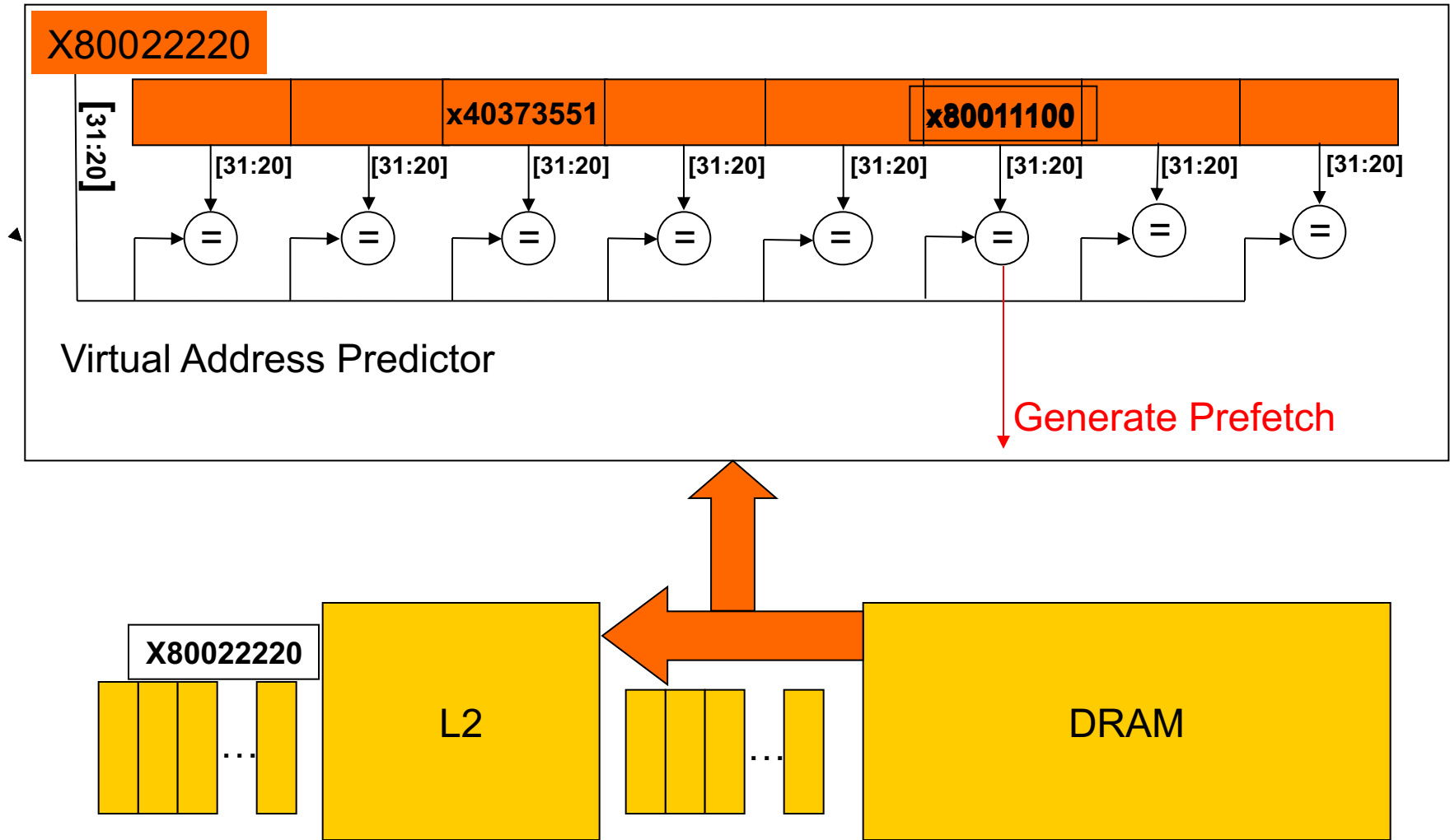
Making Content Directed Prefetching Efficient

- Hardware does not have enough information on pointers

- Software does (and can profile to get more information)

- Idea:
  - Compiler profiles/analyzes the code 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;
}
```
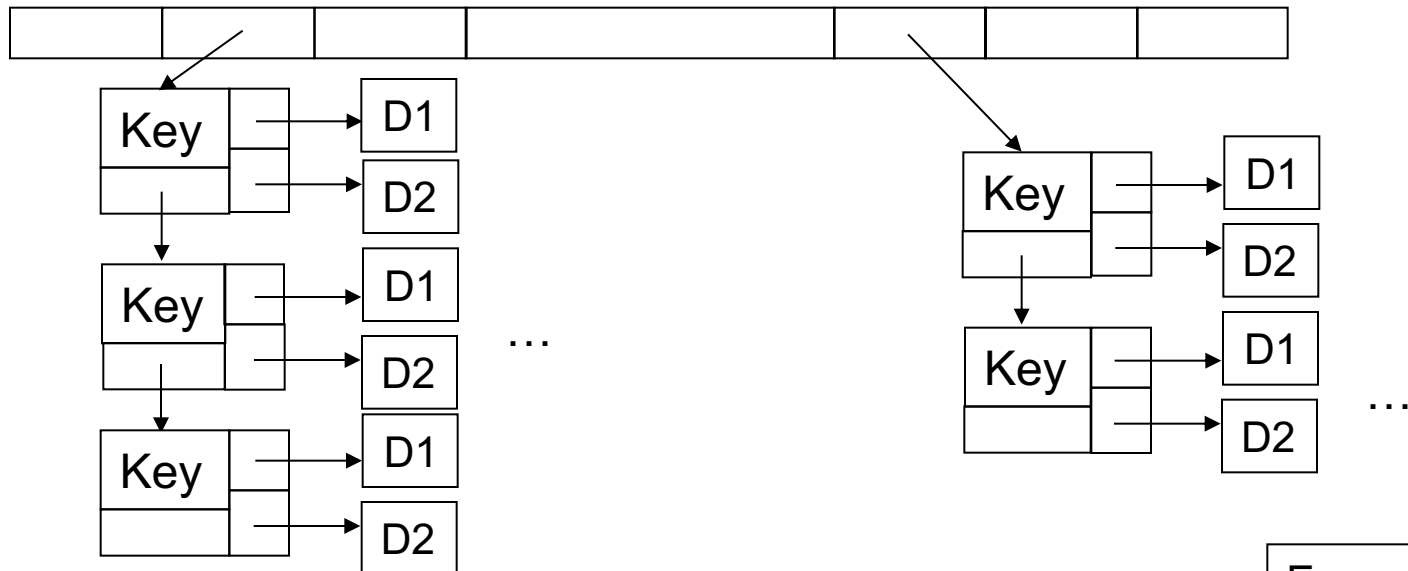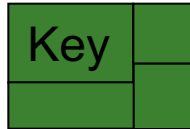

Example from mst

Shortcomings of CDP – An Example
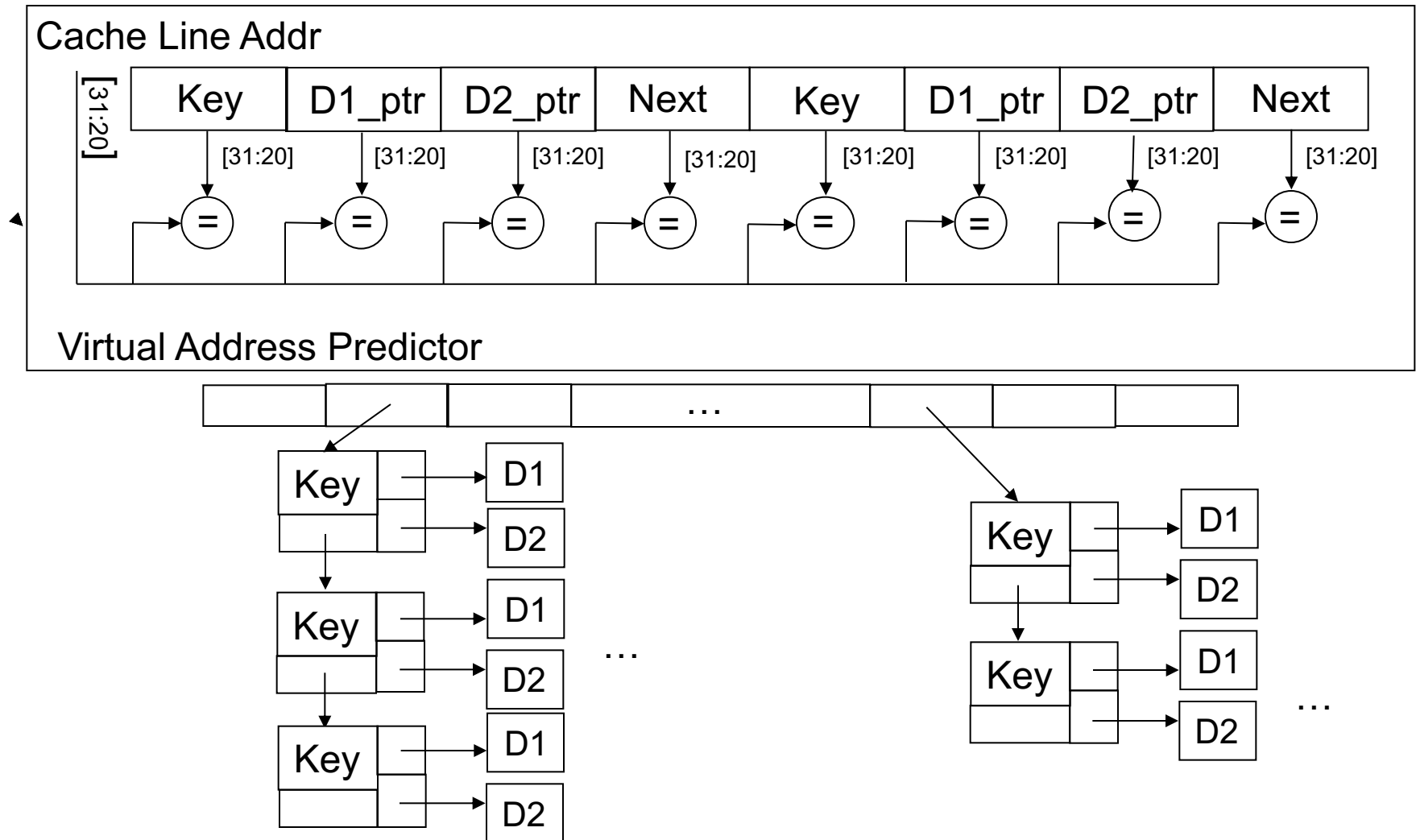
Shortcomings of CDP – An Example

```
HashLookup(int Key) {
    …
    for (node = head ; node -> Key != Key; node = node -> Next; ) ;
    if (node) return node -> D1;
}
```
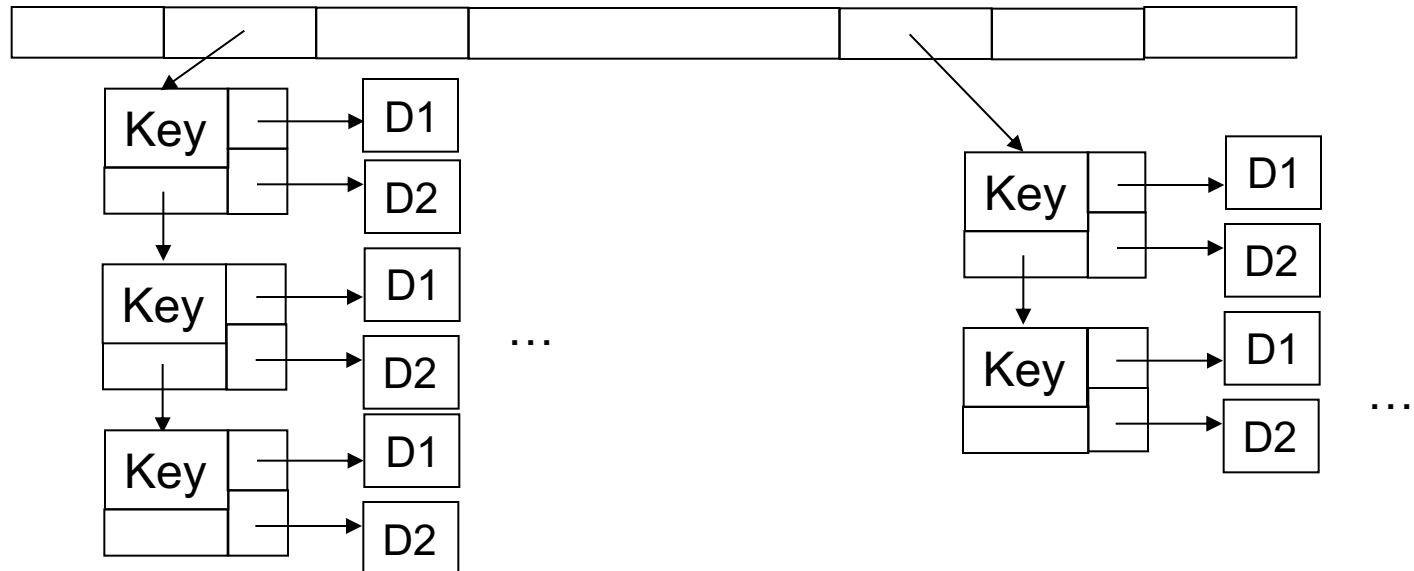
Overcoming the Shortcomings of CDP
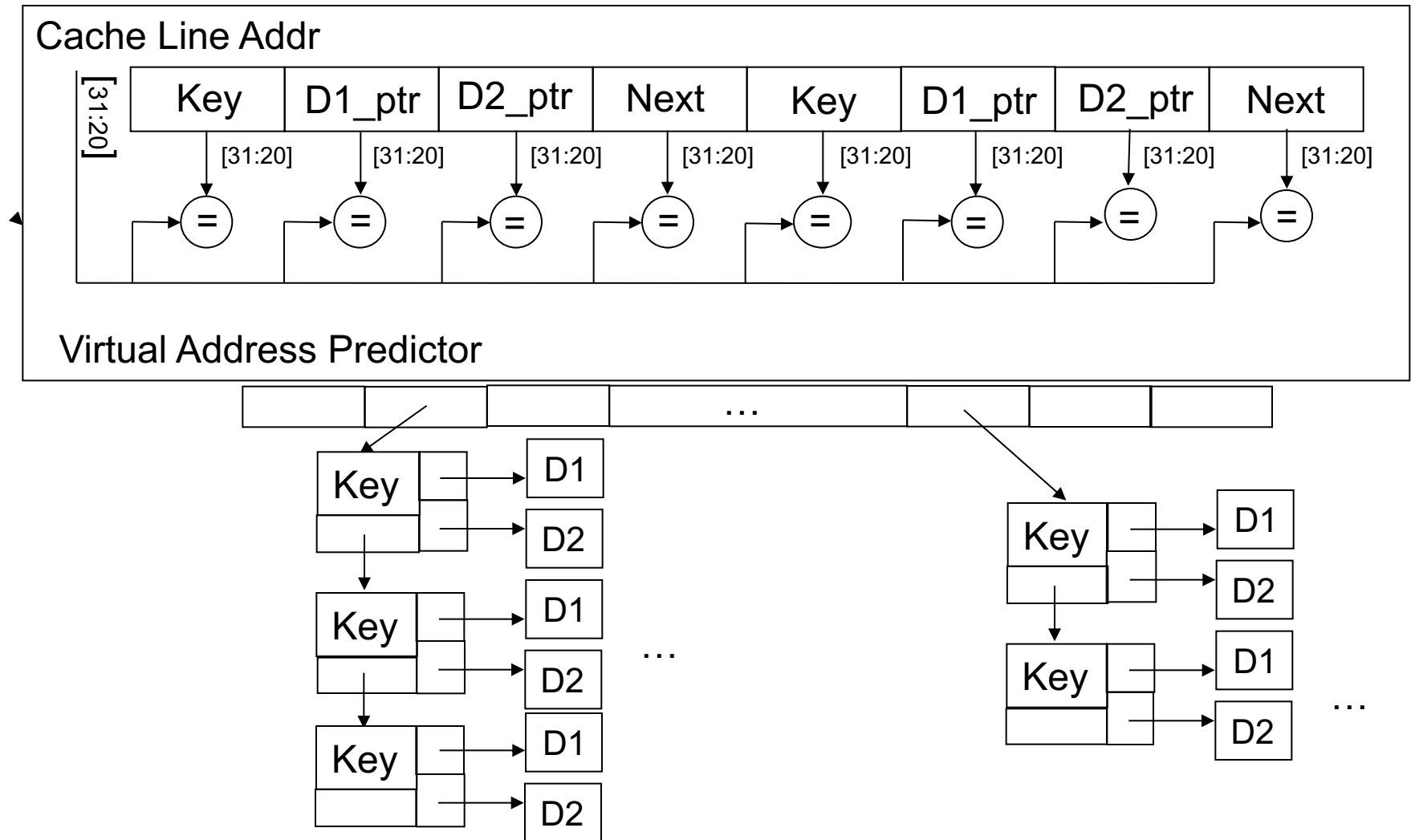
More on Content Directed Prefetching

- 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)
Best paper session. One of the three papers nominated for the Best Paper Award by the Program Committee.

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

Computer Architecture
Lecture 18: Prefetching

Prof. Onur Mutlu

ETH Zürich

Fall 2020

26 November 2020

We Did Not Cover
The Following Slides

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
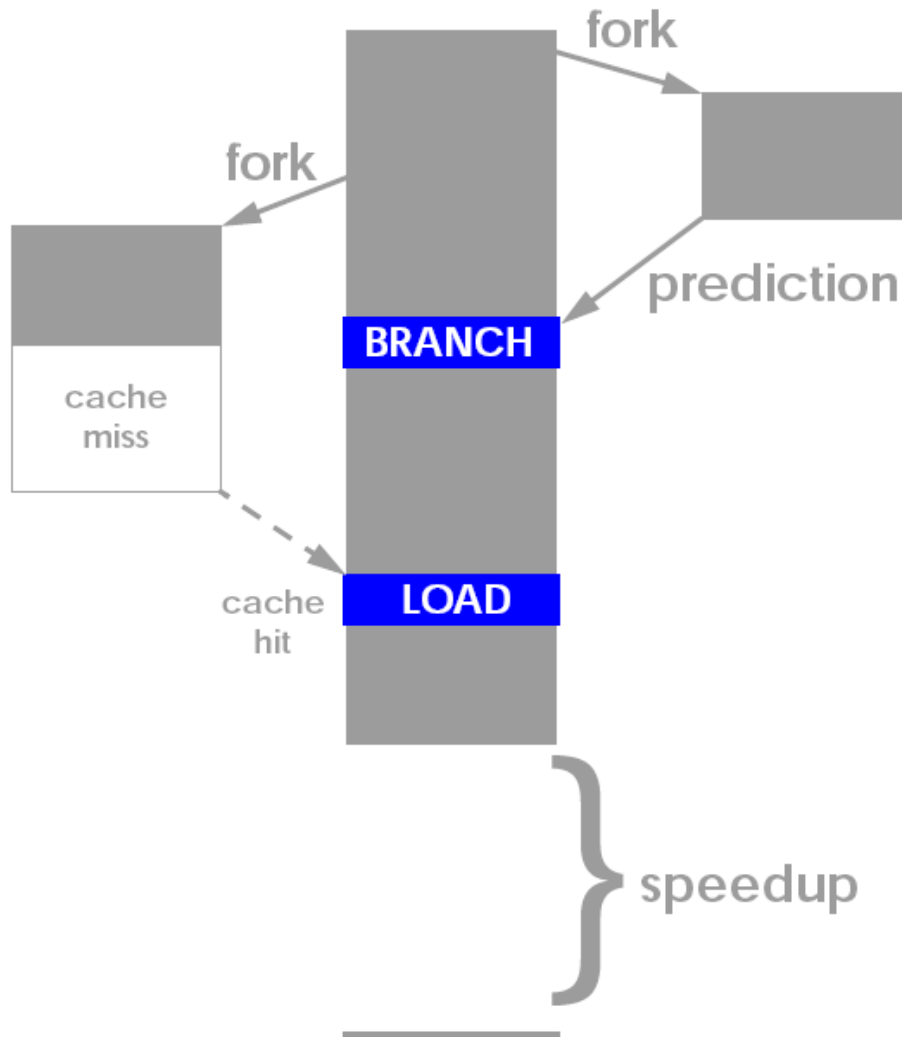
- Dubois and Song, "Assisted Execution," USC Tech Report 1998.

- 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
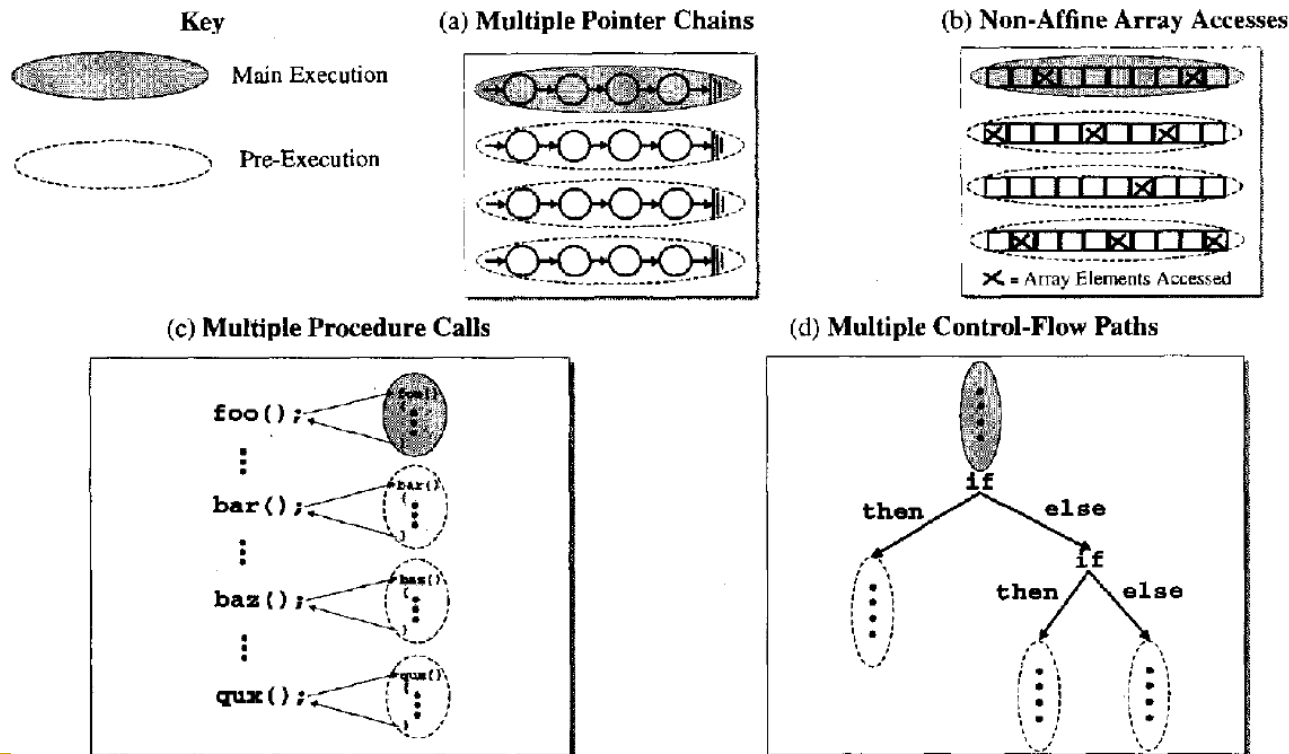
An Example

(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();
```

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

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.

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
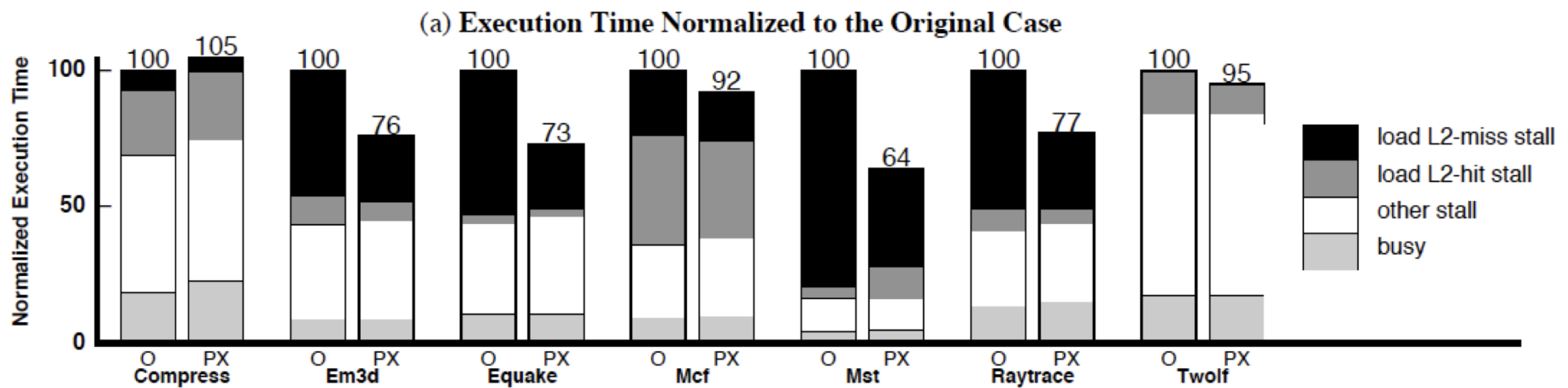
(a) Execution Time Normalized to the Original Case

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;          branch
2.    int ifrom = heap_tail;           misprediction
3.    int ito = ifrom/2;
4.    heap_tail++;                          cache miss
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;
    }
  }
```

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    cmptlt   $f0,$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:

Load 1 Miss

Load 2 Miss

| Compute | | Stall | Compute | | Stall | |

Miss 1

Miss 2

Runahead:

Load 1 Miss

Load 2 Miss

Load 1 Hit

Load 2 Hit

| Compute | Runahead | | | Compute | | |

Miss 1

Miss 2

Saved Cycles
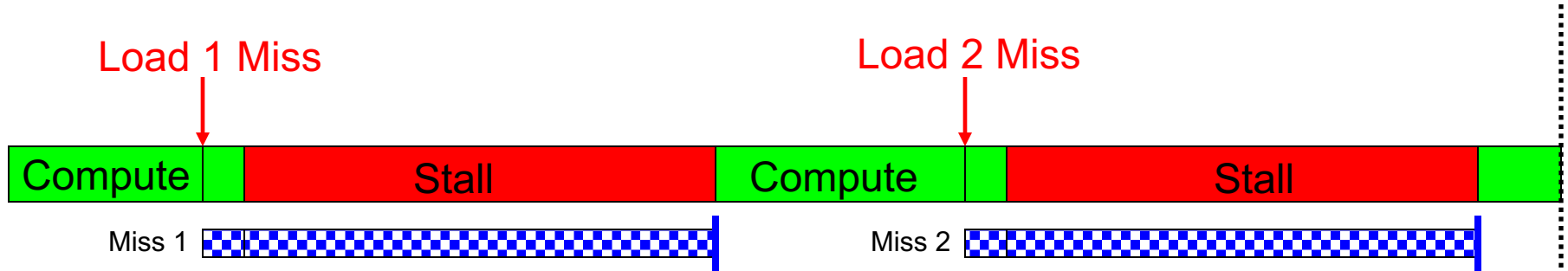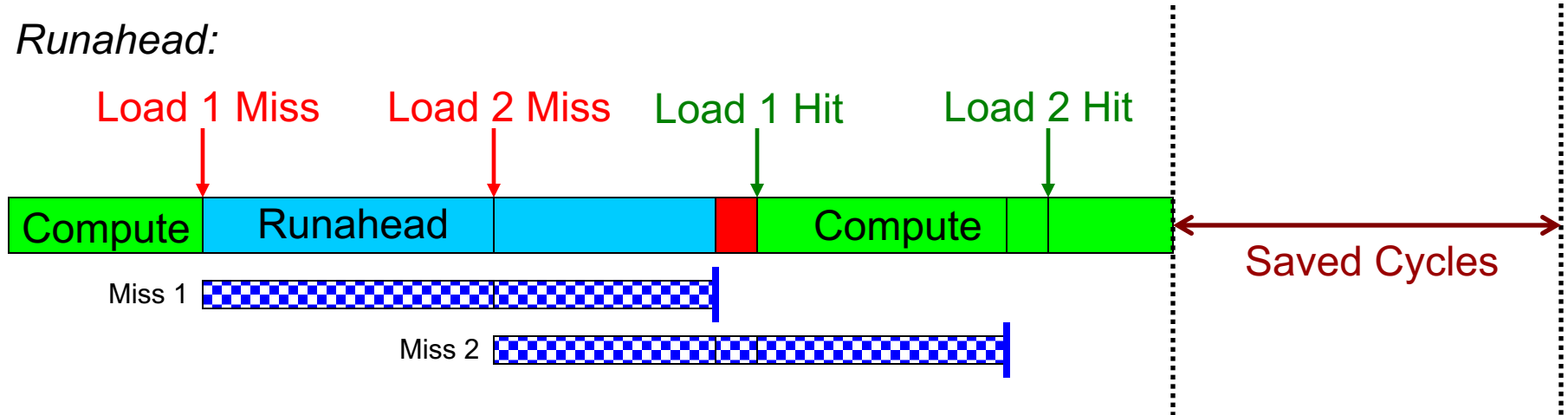
Runahead as an Execution-based Prefetcher

- 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?

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**), Anaheim, CA, February 2003. Slides (pdf)
  One of the 15 computer architecture papers of 2003 selected as Top Picks by IEEE Micro.

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

Effect of Runahead Execution in Sun ROCK

- Shailender Chaudhry talk, Aug 2008.

More on Runahead in SUN ROCK
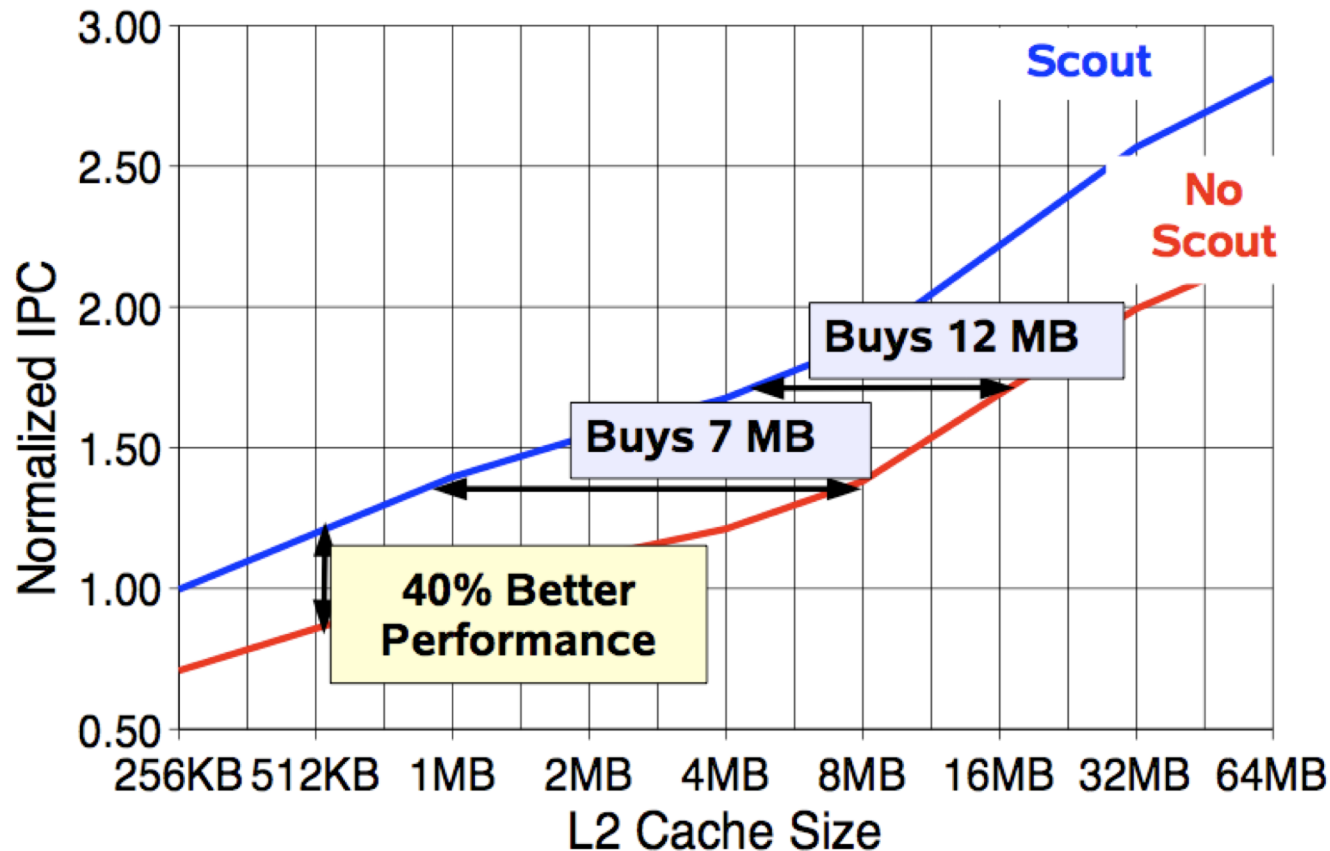
HIGH-PERFORMANCE THROUGHPUT COMPUTING

THROUGHPUT COMPUTING, ACHIEVED THROUGH MULTITHREADING AND MULTICORE TECHNOLOGY, CAN LEAD TO PERFORMANCE IMPROVEMENTS THAT ARE 10 TO 30× THOSE OF CONVENTIONAL PROCESSORS AND SYSTEMS. HOWEVER, SUCH SYSTEMS SHOULD ALSO OFFER GOOD SINGLE-THREAD PERFORMANCE. HERE, THE AUTHORS SHOW THAT HARDWARE SCOUTING INCREASES THE PERFORMANCE OF AN ALREADY ROBUST CORE BY UP TO 40 PERCENT FOR COMMERCIAL BENCHMARKS.

Chaudhry+, "High-Performance Throughput Computing," IEEE Micro 2005.

More on Runahead in SUN ROCK

Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun's ROCK Processor

Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson,
Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay
Sun Microsystems, Inc.
4180 Network Circle, Mailstop SCA18-211
Santa Clara, CA 95054, USA
{shailender.chaudhry, robert.cypher, magnus.ekman, martin.karlsson,
anders.landin, sherman.yip, haakan.zeffer, marc.tremblay}@sun.com

Chaudhry+, "Simultaneous Speculative Threading," ISCA 2009.

More Runahead Papers

Runahead Execution vs. Conventional Data Prefetching in the IBM POWER6 Microprocessor

Harold W. Cain   Priya Nagpurkar

IBM T.J. Watson Research Center
Yorktown Heights, NY
{tcain, pnagpurkar}@us.ibm.com

Cain+, "Runahead Execution vs. Conventional Data Prefetching in the IBM POWER6 Microprocessor," ISPASS 2010

Runahead Execution on IBM POWER6

Abstract

After many years of prefetching research, most commercially available systems support only two types of prefetching: software-directed prefetching and hardware-based prefetchers using simple sequential or stride-based prefetching algorithms. More sophisticated prefetching proposals, despite promises of improved performance, have not been adopted by industry. In this paper, we explore the efficacy of both hardware and software prefetching in the context of an IBM POWER6 commercial server. Using a variety of applications that have been compiled with an aggressively optimizing compiler to use software prefetching when appropriate, we perform the first study of a new runahead prefetching feature adopted by the POWER6 design, evaluating it in isolation and in conjunction with a conventional hardware-based sequential stream prefetcher and compiler-inserted software prefetching.

We find that the POWER6 implementation of runahead prefetching is quite effective on many of the memory intensive applications studied; in isolation it improves performance as much as 36% and on average 10%. However, it outperforms the hardware-based stream prefetcher on only two of the benchmarks studied, and in those by a small margin. When used in conjunction with the conventional prefetching mechanisms, the runahead feature adds an additional 6% on average, and 39% in the best case (GemsFDTD).

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 processsor 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

SAFARI

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

SAFARI

Some Examples

- **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]

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)
One of the five papers nominated for the Best Paper Award by the Program Committee.

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

On Bandwidth-Efficient 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)
  Best paper session. One of the three papers nominated for the Best Paper Award by the Program Committee.

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

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   [§]Intel Labs
University Park, PA 16802   Pittsburgh, PA 15213   Hillsboro, OR 97124
{adwait, onur, kandemir, das}@cse.psu.edu   onur@cmu.edu   {asit.k.mishra, ravishankar.iyer}@intel.com

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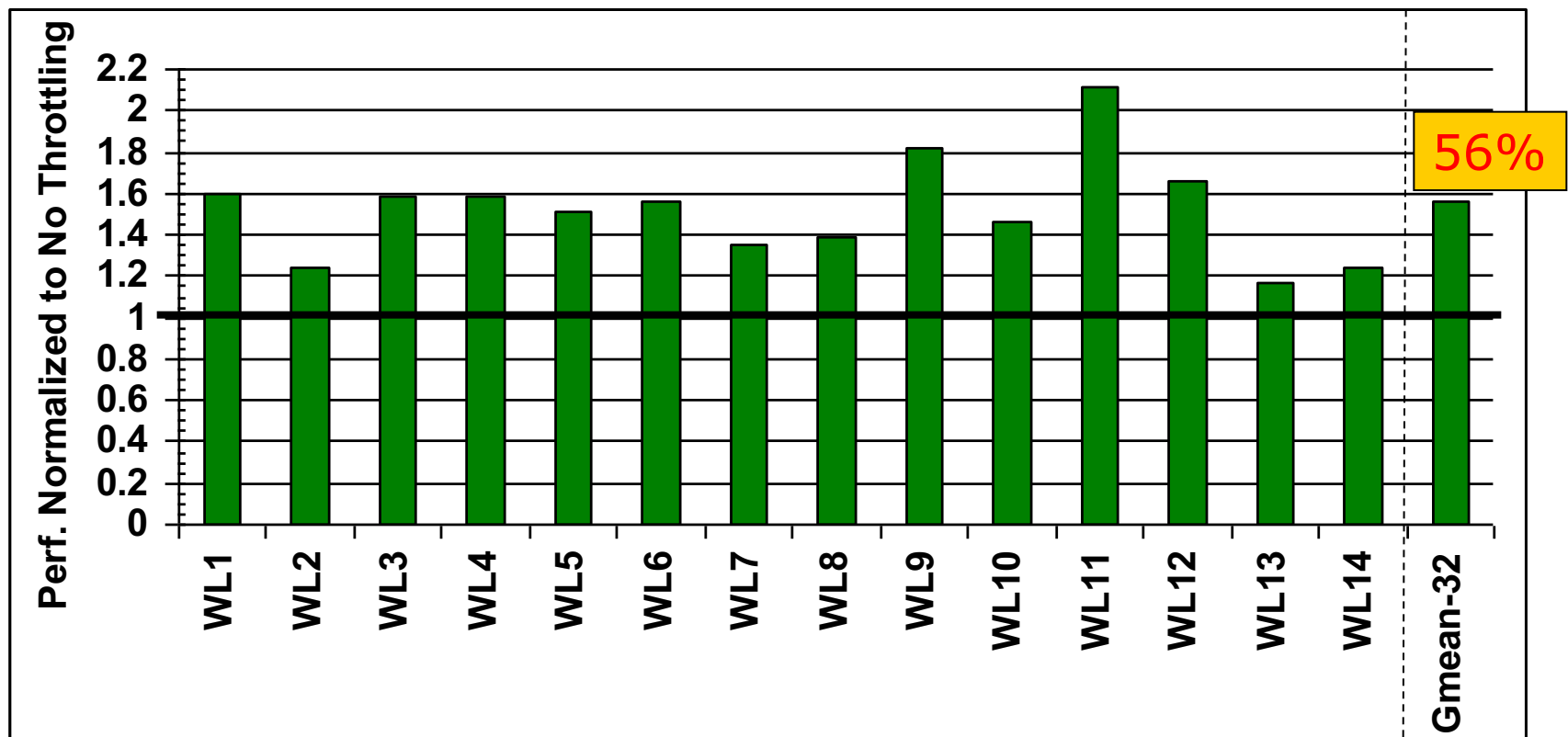
Exact workload combinations can be found in [Ebrahimi et al., MICRO 2009]

High Interference caused by Accurate Prefetchers

Legend:

Core0   Core 1   Miss / Dem 2   Core 3

Pref 1
Addr:C+64

Dem 2
Addr:A

Pref 3
Addr:D+64

Shared Cache

Dem 2   Dem 2   Pref 0   Dem 0

Dem X
Addr:Y
Demand Request

Memory Controller

Row Buffer
Hit

Dem 2
Addr:Y

DRAM

Requests Being Serviced	Pref 1 Addr:C	Pref 3 Addr:D
Row Buffers	Row: C to C+8K	Row: D to D+8K
	Bank 0	Bank 1

In a multi-core system, accurate prefetchers can cause significant interference with concurrently-executing applications

110

Shortcoming of Local Prefetcher Throttling

	Core 0	Core 1	Core 2	Core 3
	Prefetcher Degree: ~~2~~ 4	Prefetcher Degree: ~~2~~ 4		
	FDP Throttle Up	*FDP Throttle Up*		

Shared Cache
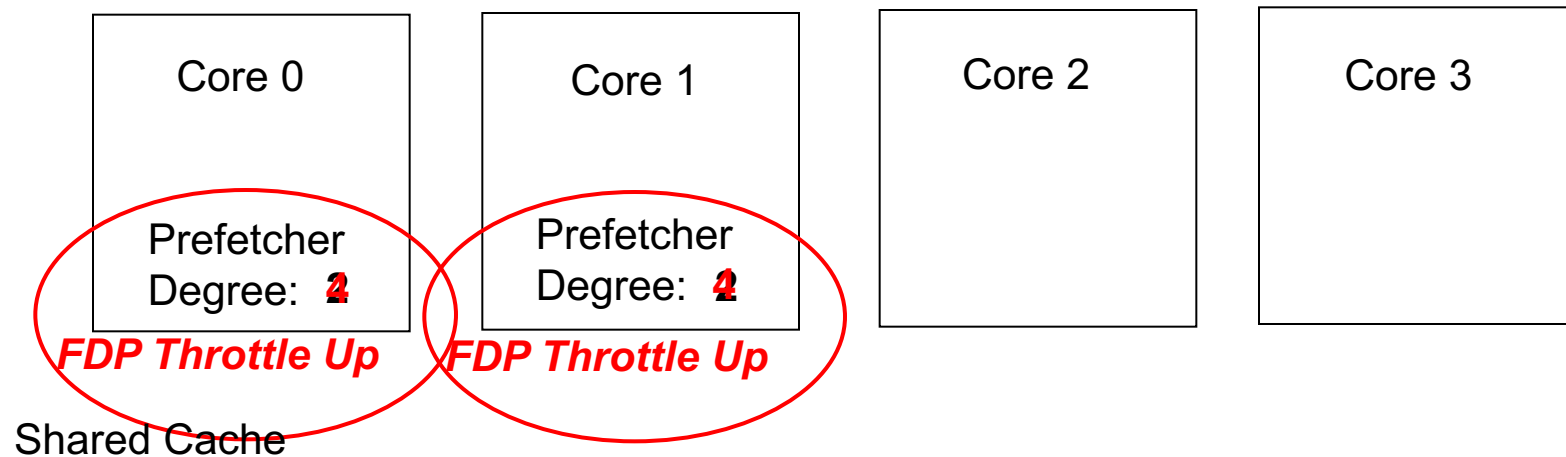
Set 0	~~Used_0~~ Pref 2	~~Used_0~~ Pref 2	~~Used_1~~ Pref 3	~~Used_1~~ Pref 3	Dem 2	Dem 2	Dem 3	Dem 3
Set 1	~~Used_0~~ Pref 3	~~Used_0~~ Pref 3	~~Used_1~~ Pref 2	~~Used_1~~ Pref 2	Dem 3	Dem 3	Dem 3	Dem 3
Set 2	~~Dem 0~~ Pref 2	~~Dem 0~~ Pref 2	~~Dem 0~~ Pref 2	~~Dem 0~~ Pref 2	~~Pref 3~~ Dem 1	~~Pref 3~~ Dem 1	~~Pref 3~~ Dem 1	~~Pref 3~~ Dem 1

**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

Legend:
- No Prefetching
- Pref. + No Throttling
- Feedback-Directed Prefetching
- HPAC
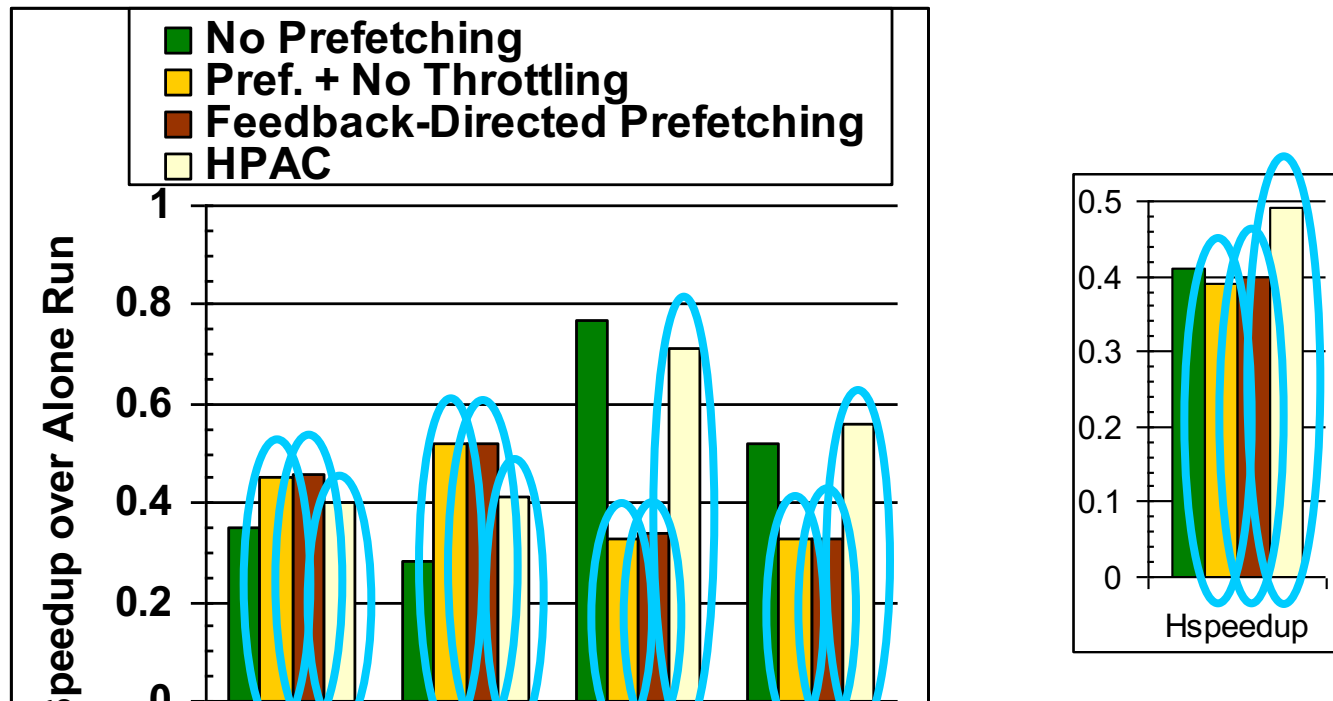
Our Approach: Use both *global* and per-core feedback to determine each prefetcher's aggressiveness

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: *Maximize the prefetching performance of core i independently*

Global Control: accepts or overrides decisions made by local control to improve *overall system performance*

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

Memory Controller

Bandwidth Feedback

Pref. i

Throttling Decision

Local Control

Core i

Final Throttling Decision

Accuracy

Local Throttling Decision

Global Control

Cache Pollution Feedback

Shared Cache
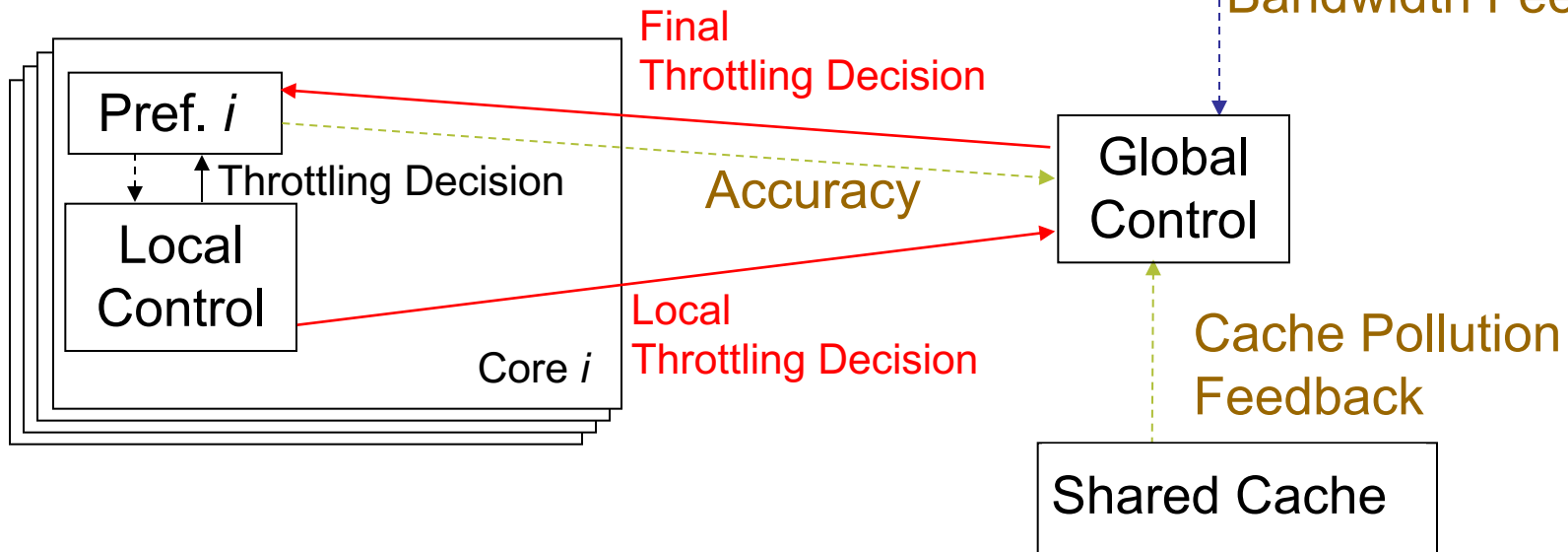
Hierarchical Prefetcher Throttling Example

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

Memory Controller

High BW (i)

High BWNO (i)

Pref. *i*

Local Control

Core *i*

Enforce Info **Throttle Down** Throttling Decision

Local Local **Throttle Up** Throttling Decision

High Acc (i)

Global Control

High Pol (i)

Pol. Filter *i*

Shared Cache
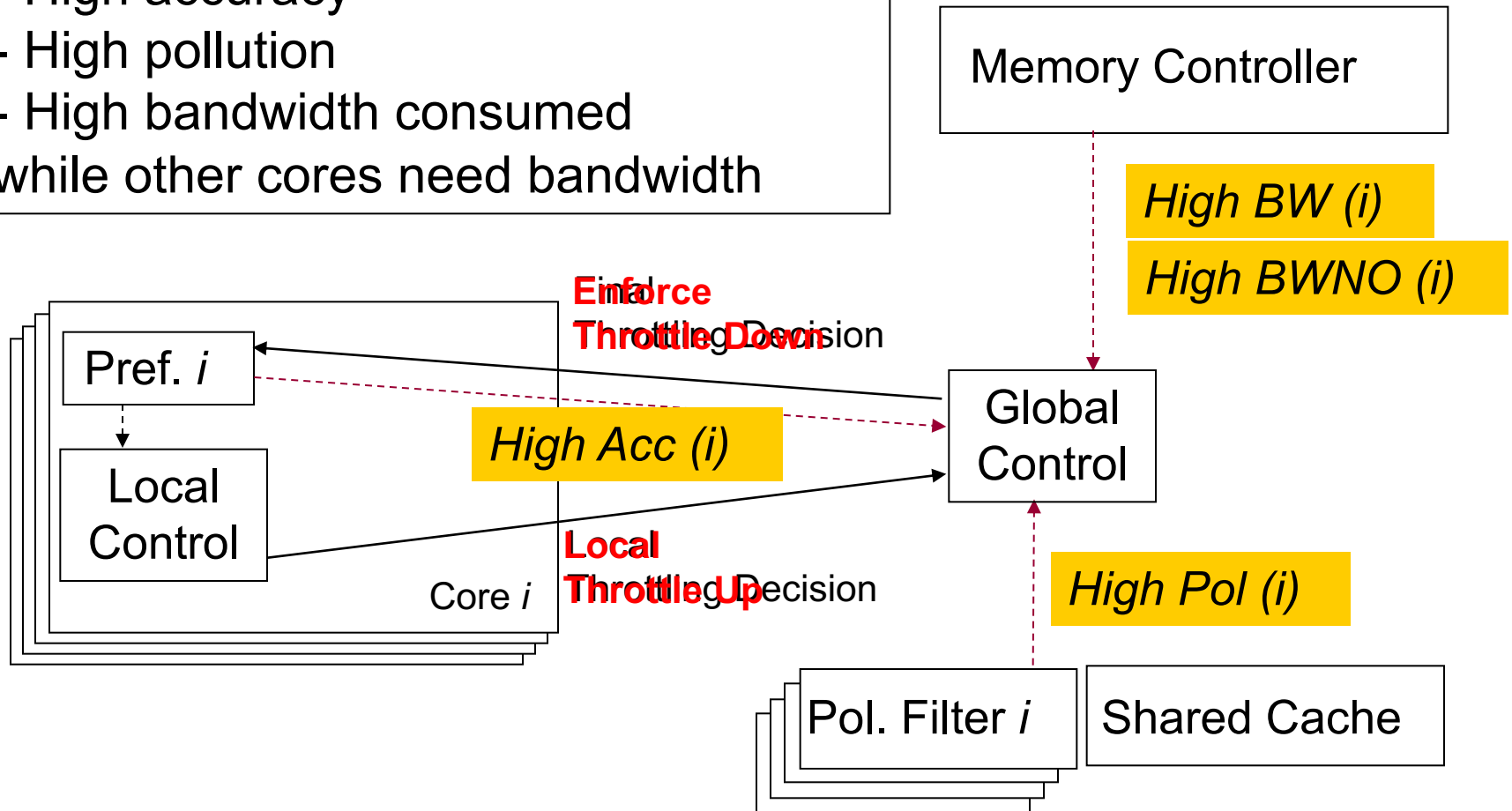
HPAC Control Policies

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

HPAC Evaluation

No Throttling

Feedback-Directed Prefetching (FDP)

Hierarchical Prefetcher Aggressiveness Control (HPAC)

Normalized to system with no prefetching

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}

Problems of Prefetch Handling

- **How to schedule *prefetches* vs *demands*?**
  - Demand-first: Always prioritizes demands over prefetch requests
  - Demand-prefetch-equal: Always treats them the same

Neither of these perform best

Neither take into account both:

1. Non-uniform access latency of DRAM systems

2. Usefulness of prefetches

When Prefetches are Useful

Stall —— Execution ▮

Demand-first

DRAM

Row B / **Row A** Row Buffer

~~Row-hit~~ **Row-conflict**

DRAM | Controller

Pref Row A	: X
Dem Row B	: Y
Pref Row A	: Z

2 row-conflicts, 1 row-hit

DRAM

Processor

Miss Y   Miss X   Miss Z

Processor needs Y, X, and Z

When Prefetches are Useful

Stall ——— Execution ▮

DRAM

| Row B | Row Buffer |

Row-conflict / Row-hit

DRAM Controller

Pref ...	: X
Dem Row B	: Y
Pref Row A	: Z

Processor needs Y, X, and Z

➢ **Demand-first**

2 row-conflicts, 1 row-hit

DRAM

Processor

Demand-pref-equal outperforms demand-first

➢ **Demand-pref-equal**

2 row-hits, 1 row-conflict

DRAM

Processor

Miss Y

Hit X Hit Z

Saved Cycles

When Prefetches are Useless

DRAM

Row A | Row Buffer

DRAM | Controller

Pref Row
Dem Row B | : Y
Pref Row A | : Z

Processor needs ONLY Y

> **Demand-first**

DRAM

Y   X   Z

Processor

Miss Y

Saved Cycles

Demand-first outperforms demand-pref-equal

> **Demand-pref-equal**

DRAM

X   Z   Y

Processor

Miss Y

Demand-first vs. Demand-pref-equal policy

Stream prefetcher enabled

Chart: IPC normalized to no prefetching

Legend:
- Demand-first (green)
- Demand-pref-equal (yellow)

Useless prefetches:
- Off-chip bandwidth
- Queue resources
- Cache Pollution

Benchmarks: galgel, ammp, art, milc, swim, libquantum, bwaves, leslie3d
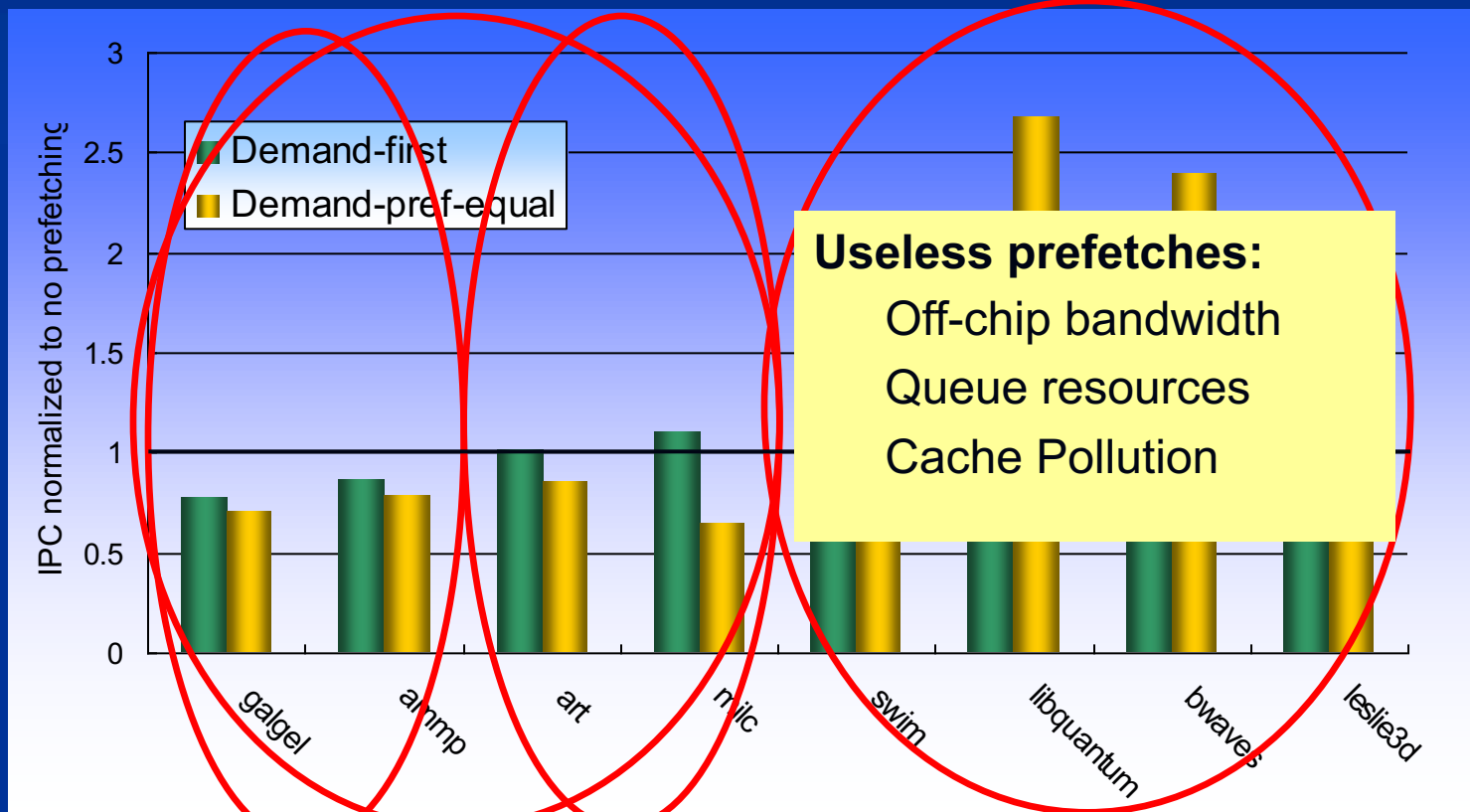
Goal 1: Adaptive Goal 2: Eliminate useless prefetches tch usefulness

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
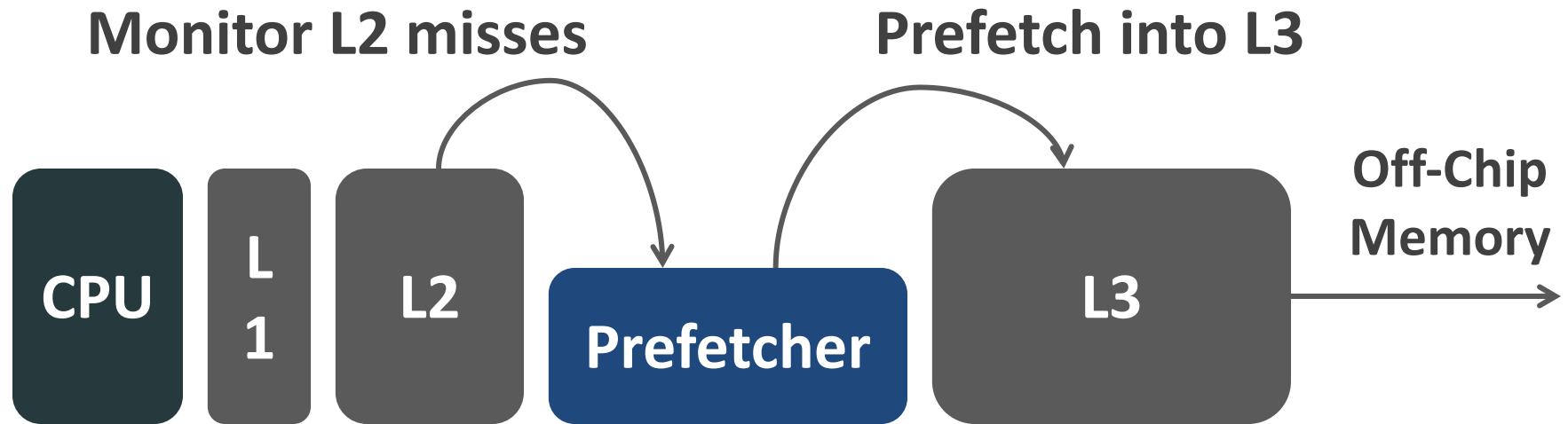
Caching Policies for Prefetched Blocks

Problem: Existing caching policies for prefetched blocks result in significant cache pollution

Cache Miss:

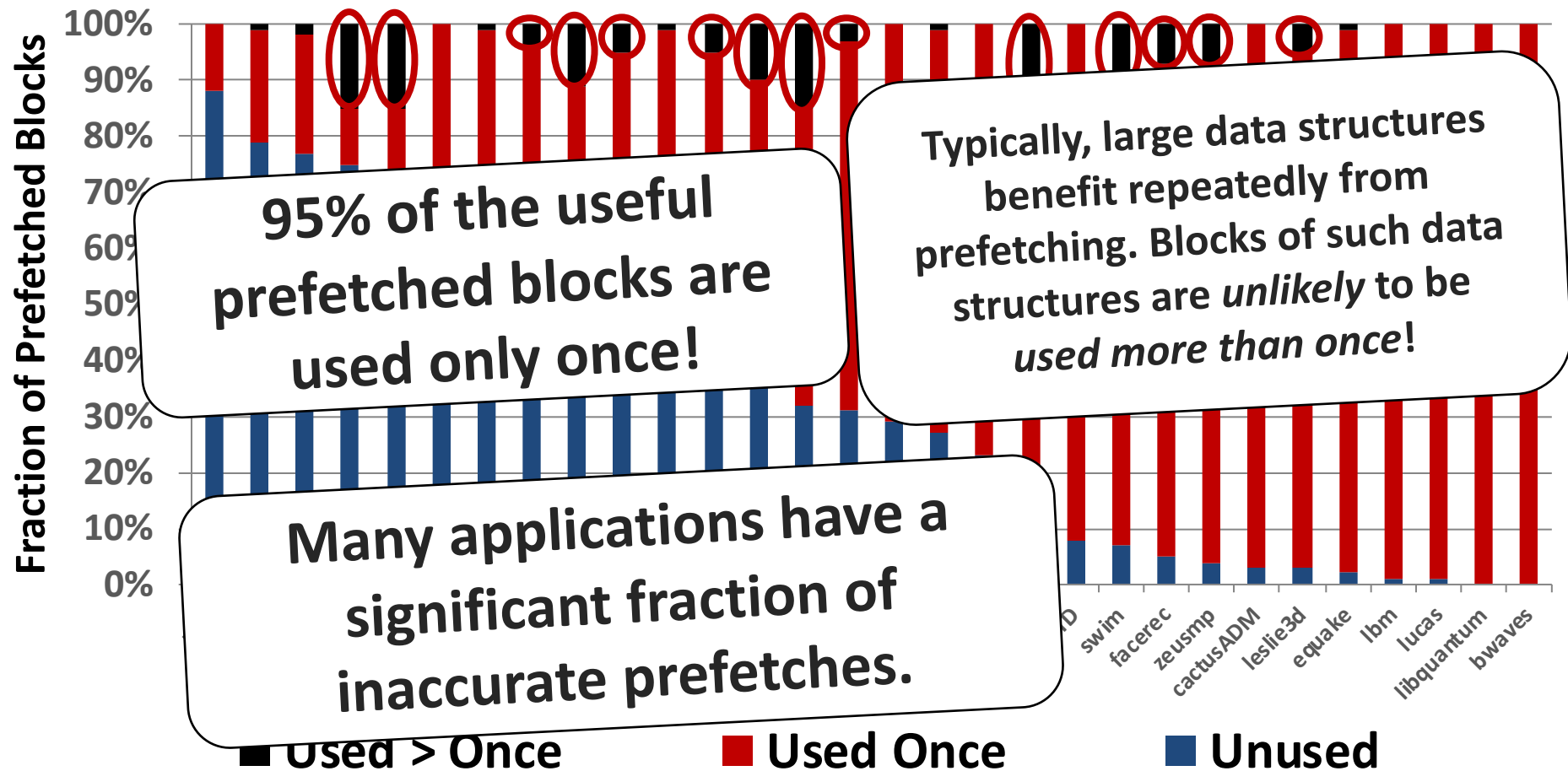Are these insertion and promotion policies good for prefetched blocks?

MRU | | | | | | | | | **LRU**

Cache Set
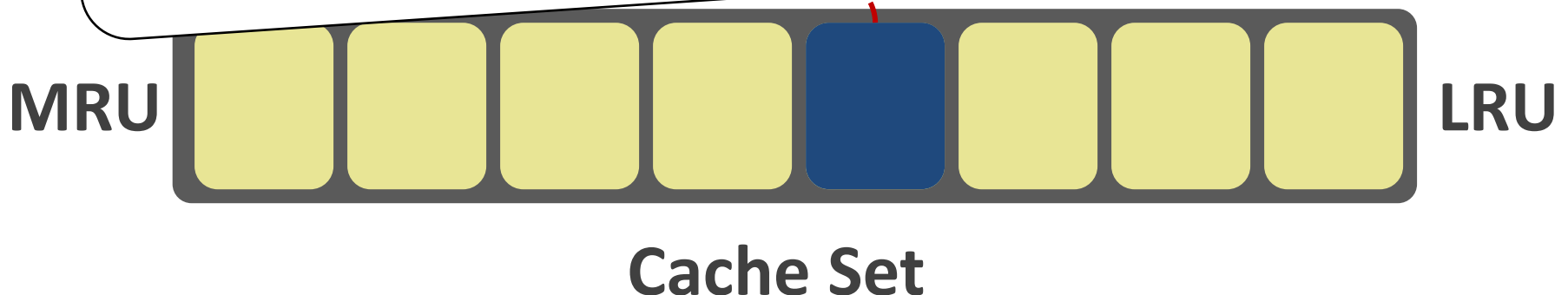
Prefetch Usage Experiment

Monitor L2 misses

Prefetch into L3

CPU

L1

L2

Prefetcher

L3

Off-Chip Memory

Classify prefetched blocks into three categories

1. **Blocks that are unused**

2. **Blocks that are used exactly once before evicted from cache**

3. **Blocks that are used more than once before evicted from cache**

Usage Distribution of Prefetched Blocks

95% of the useful prefetched blocks are used only once!

Typically, large data structures benefit repeatedly from prefetching. Blocks of such data structures are *unlikely* to be *used more than once*!

Many applications have a significant fraction of inaccurate prefetches.

■ Used > Once   ■ Used Once   ■ Unused

Promote to MRU

This is a **bad** policy. The block is unlikely to be reused in the cache.

This problem exists with state-of-the-art replacement policies (e.g., DRRIP, DIP)

M   U

Cache Set

Demotion of Prefetched Block

Demote to LRU

Ensures that the block is evicted from the cache quickly after it is used!

Only requires the cache to distinguish between prefetched blocks and demand-fetched blocks.

M

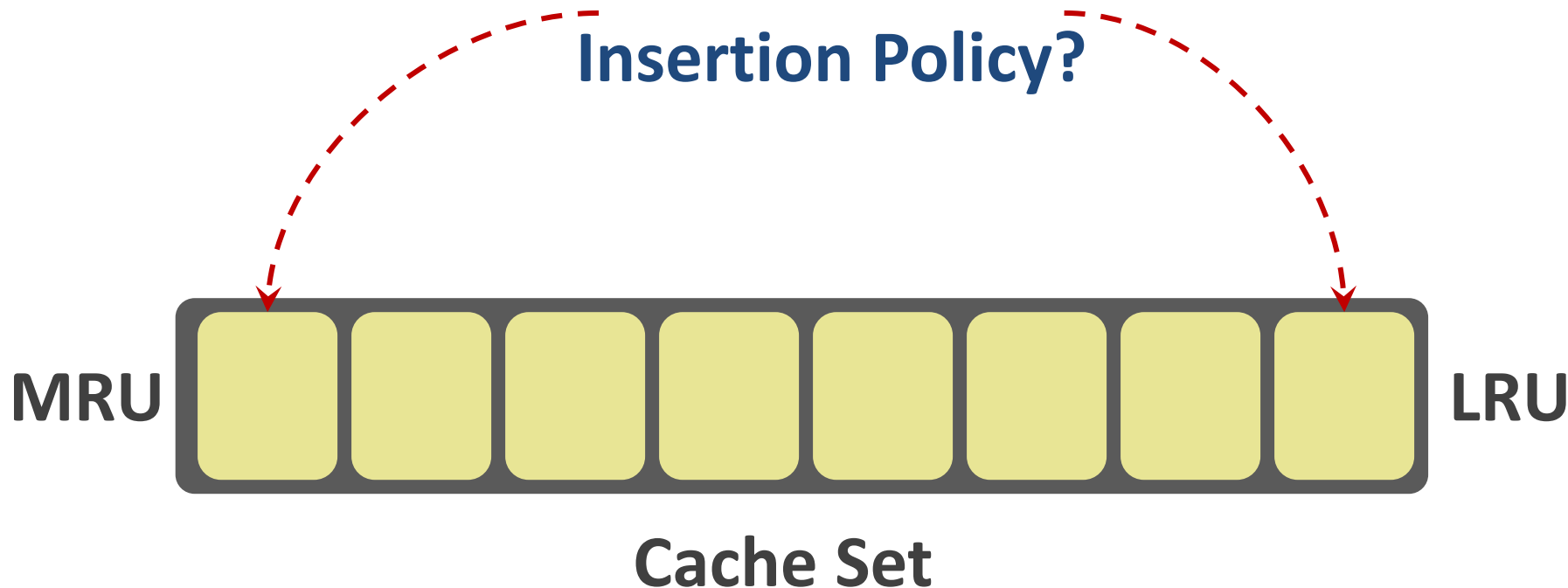U

Cache Set

Cache Insertion Policy for Prefetched Blocks

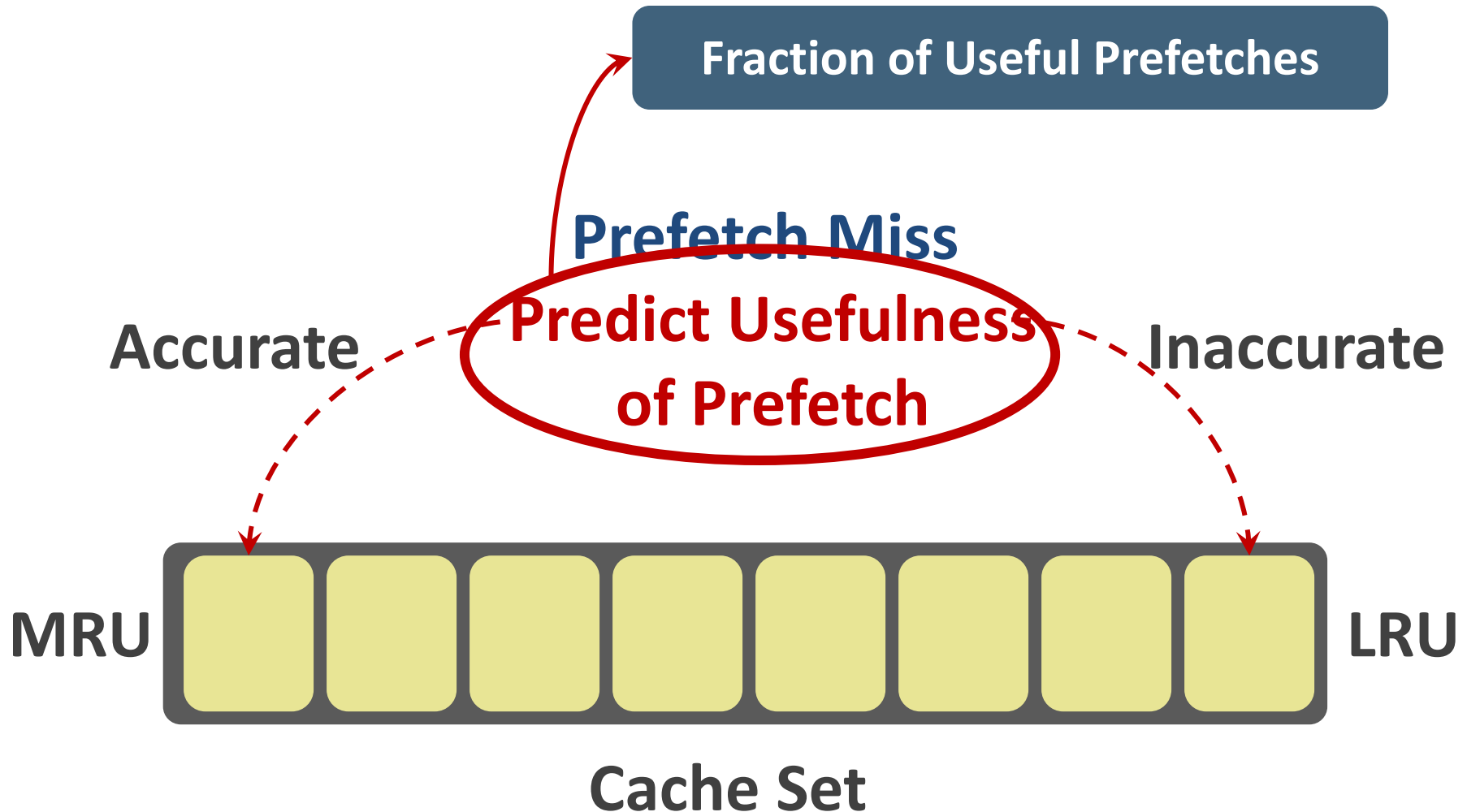**Good (Accurate prefetch)**
Bad (Inaccurate prefetch)

Good (Inaccurate prefetch)
Bad (accurate prefetch)

Prefetch Miss:
Insertion Policy?

MRU

Cache Set

LRU

Predicting Usefulness of Prefetch

Fraction of Useful Prefetches

Prefetch Miss

Predict Usefulness of Prefetch

Accurate

Inaccurate

MRU

LRU

Cache Set

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   [§]Intel Labs
University Park, PA 16802   Pittsburgh, PA 15213   Hillsboro, OR 97124
{adwait, onur, kandemir, das}@cse.psu.edu   onur@cmu.edu   {asit.k.mishra, ravishankar.iyer}@intel.com