

# Computer Architecture

## Lecture 16: Prefetching

Prof. Onur Mutlu

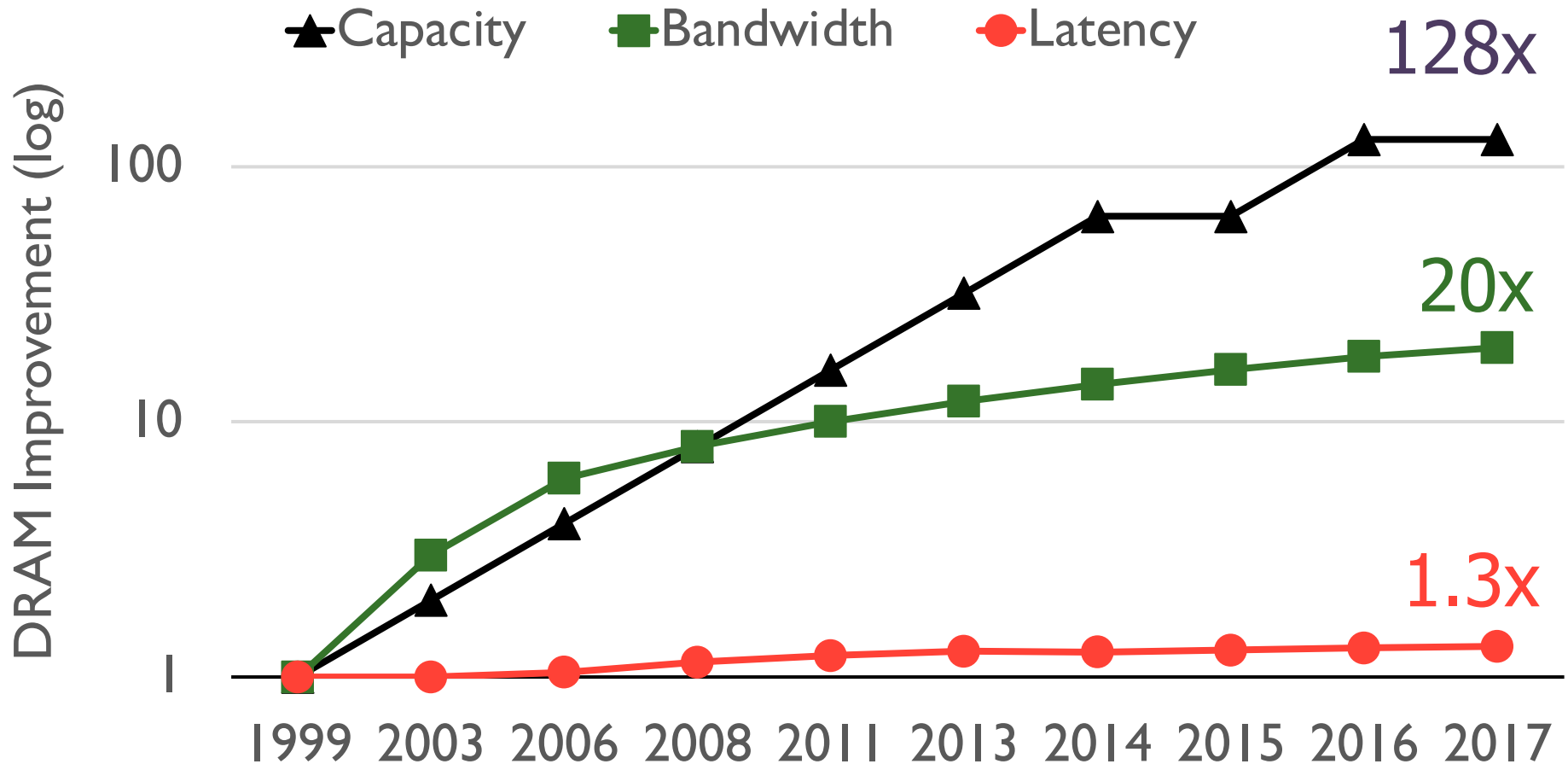
ETH Zürich

Fall 2022

18 November 2022

# The (Memory) Latency Problem

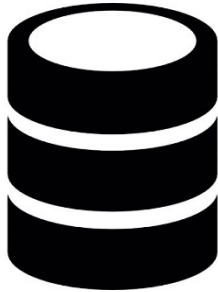
# Recall: Memory Latency Lags Behind



Memory latency remains almost constant

# DRAM Latency Is Critical for Performance

---



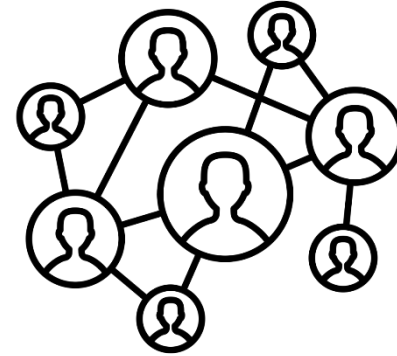
## In-memory Databases

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



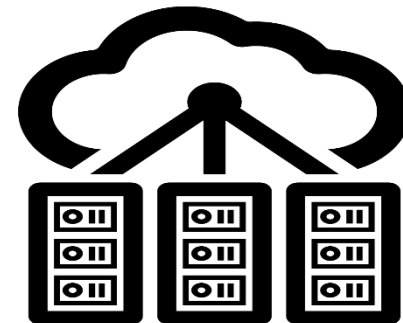
## In-Memory Data Analytics

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



## Graph/Tree Processing

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



## Datacenter Workloads

[Kanev+ (Google), ISCA'15]

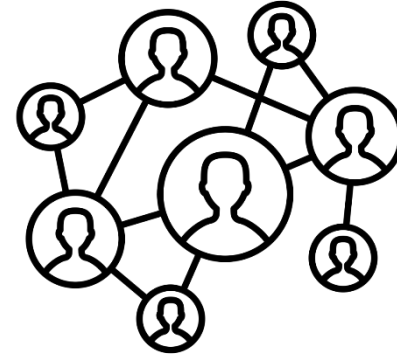


# DRAM Latency Is Critical for Performance

---



**In-memory Databases**



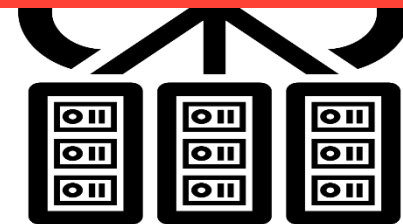
**Graph/Tree Processing**

Long memory latency → performance bottleneck



**In-Memory Data Analytics**

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



**Datacenter Workloads**

[Kanev+ (Google), ISCA'15]

# 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](#)]  
[[Source Code for Ramulator modeling Hybrid Memory Cube \(HMC\)](#)]

## Demystifying Complex Workload–DRAM Interactions: An Experimental Study

Saugata Ghose<sup>†</sup>

Tianshi Li<sup>†</sup>

Nastaran Hajinazar<sup>‡†</sup>

Damla Senol Cali<sup>†</sup>

Onur Mutlu<sup>§†</sup>

<sup>†</sup>Carnegie Mellon University

<sup>‡</sup>Simon Fraser University

<sup>§</sup>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 Lectures 8-9: Memory Latency
    - <https://www.youtube.com/watch?v=-jwkVSczPCw&list=PL5Q2soXY2Zi-cAls3cyauNzM7-74Eq31O&index=8>
    - <https://www.youtube.com/watch?v=pJ3mdIvMOY4&list=PL5Q2soXY2Zi-cAls3cyauNzM7-74Eq31O&index=9>
- 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 & Hiding

---

## ■ Caching

- ❑ <http://www.youtube.com/watch?v=mZ7CPJKzwfM>
- ❑ <http://www.youtube.com/watch?v=TsxQPLMXT60>
- ❑ [http://www.youtube.com/watch?v=OUk96\\_Bz708](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

# 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? Coherence?
- Involves predicting which address will be needed in the future
  - Works if programs have predictable miss address patterns



# Prefetching and Correctness

---

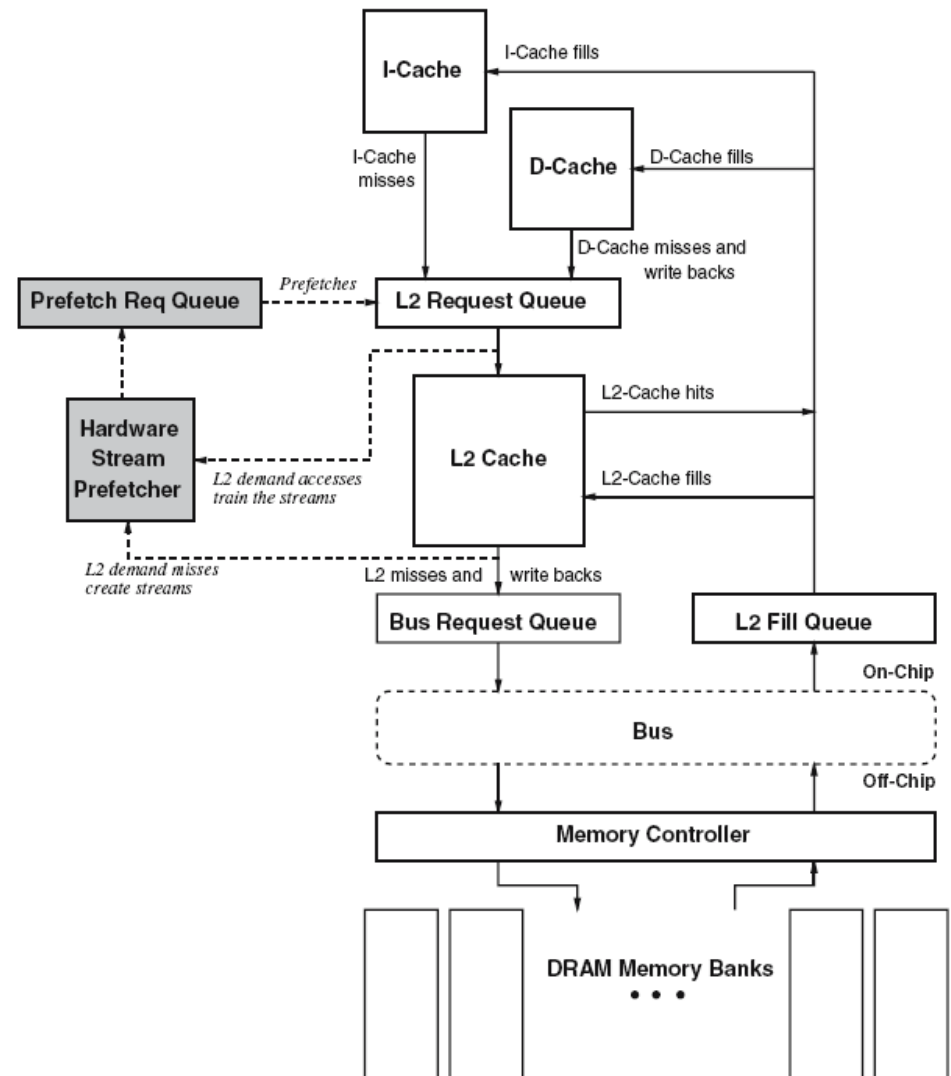
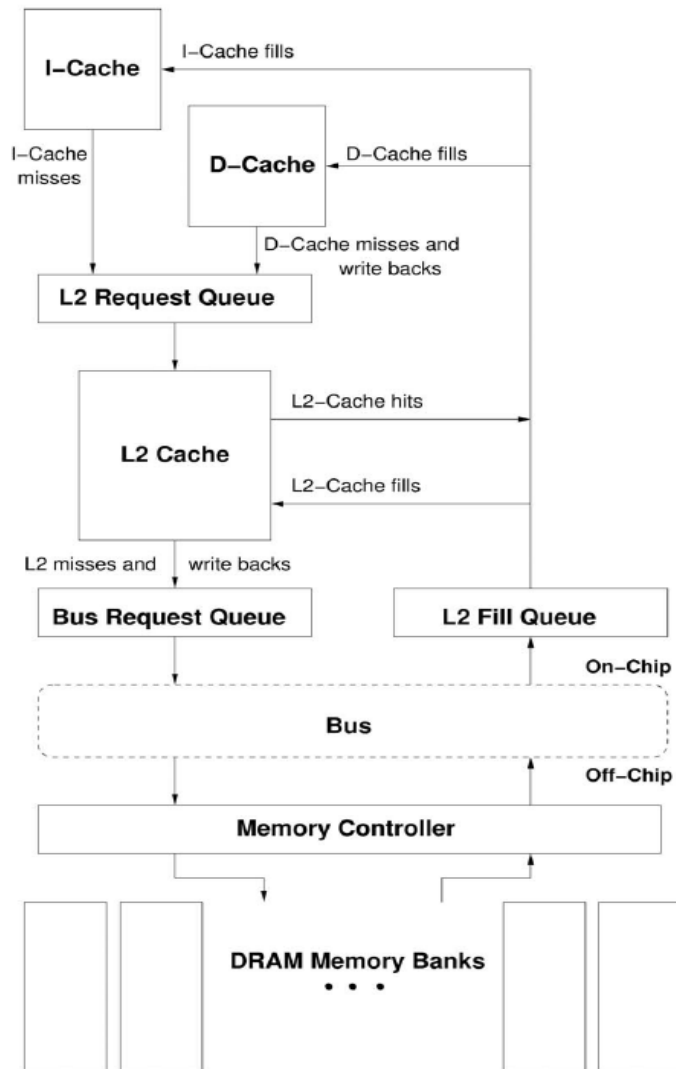
- Does a misprediction in prefetching affect correctness?
- No, prefetched data at a “mispredicted” address is simply not used
- There is no need for state recovery
  - In contrast to branch misprediction or value misprediction

# Basics

---

- In modern systems, prefetching is usually done at **cache block granularity**
- Prefetching is a technique that can reduce both
  - Miss rate
  - Miss latency
- Prefetching can be done by
  - Hardware
  - Compiler
  - Programmer
  - System

# How a HW Prefetcher Fits in the Memory System



# Prefetching: The Four Questions

---

## ■ What

- What addresses to prefetch (i.e., address prediction algorithm)

## ■ When

- When to initiate a prefetch request (early, late, on time)

## ■ Where

- Where to place the prefetched data (caches, separate buffer)
- Where to place the prefetcher (which level in memory hierarchy)

## ■ How

- How does the prefetcher operate and who operates it (software, hardware, execution/thread-based, cooperative, hybrid)

# Challenge 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/programmer'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 demand 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
  - Many Intel, AMD, IBM systems and more ...

# 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?

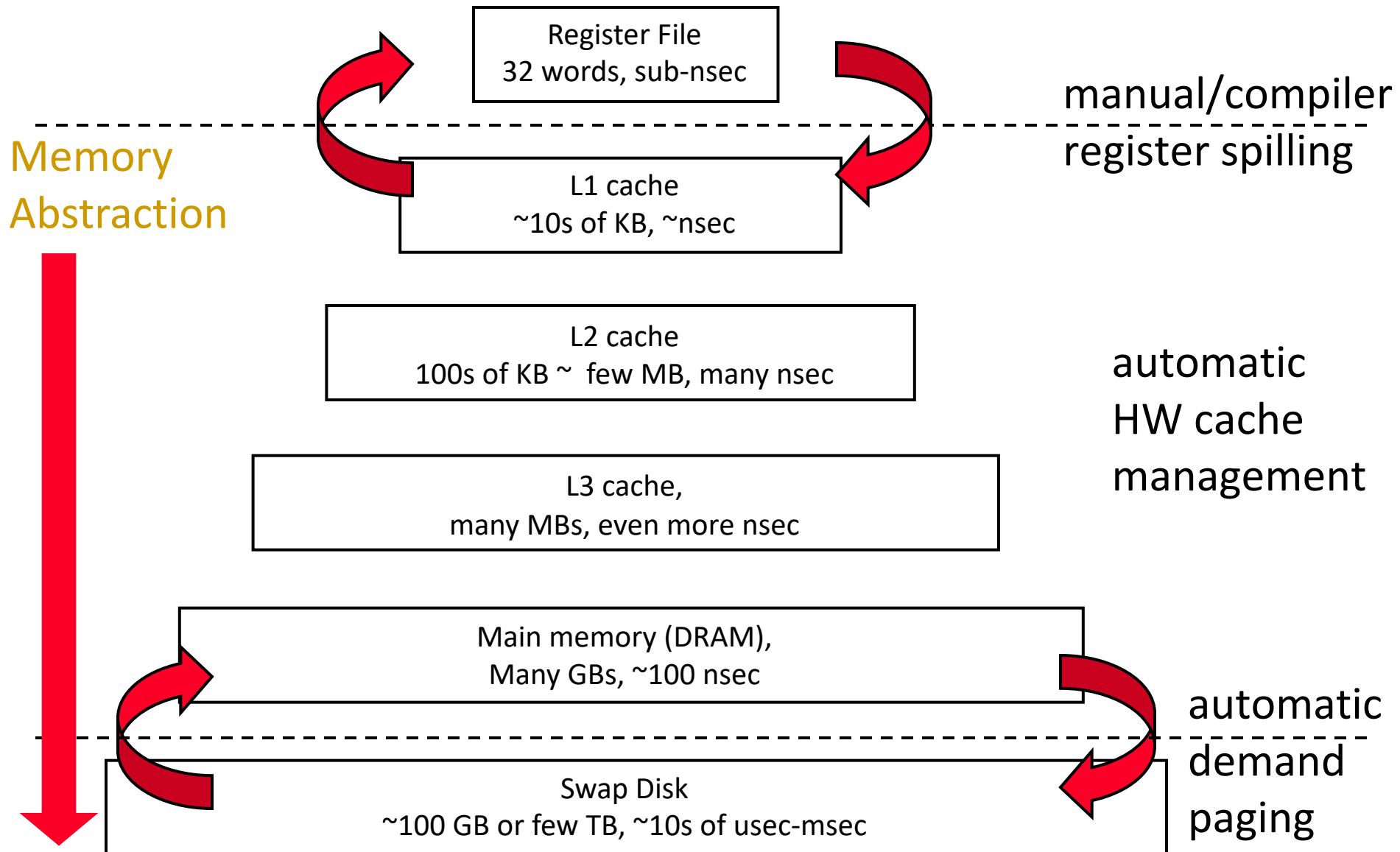


# Challenges in Prefetching: Where (III)

---

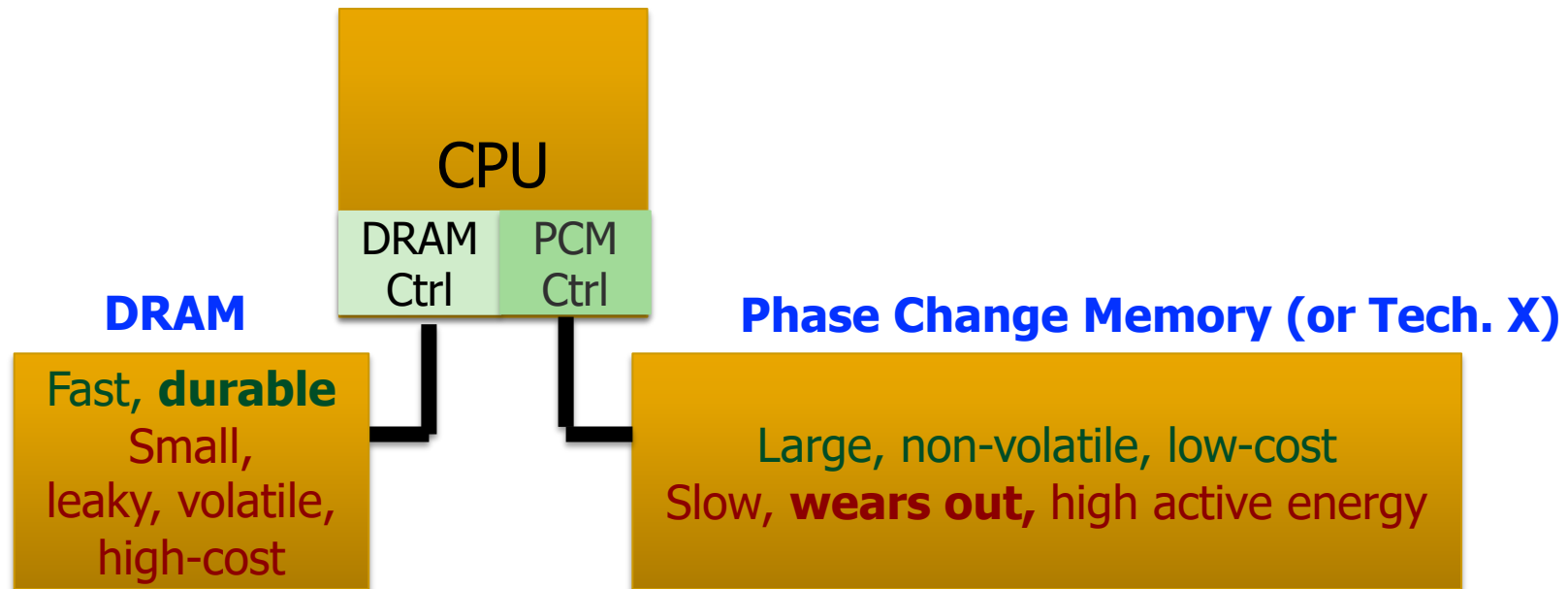
- **Where** to place the hardware prefetcher in the memory hierarchy?
  - ❑ In other words, what access patterns does the prefetcher see?
  - ❑ L1 hits and misses
  - ❑ L1 misses only
  - ❑ L2 misses only
- Seeing a more complete access pattern:
  - + Potentially better **accuracy** and **coverage**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

# Recall: A Modern Memory Hierarchy



# Recall: Hybrid Main Memory Extends the Hierarchy

---



Hardware/software manage data allocation & movement  
**to achieve the best of multiple technologies**

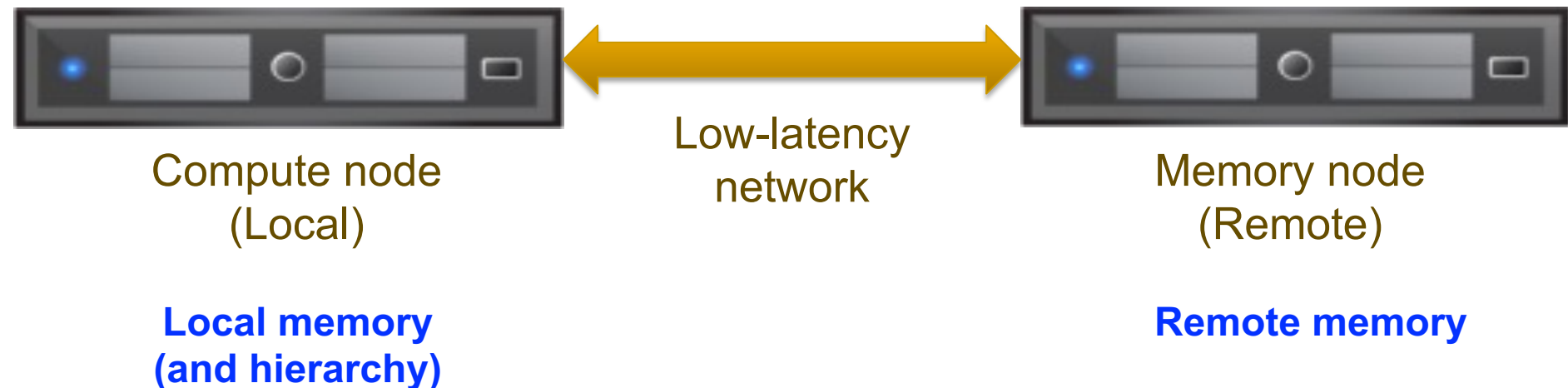
Meza+, "Enabling Efficient and Scalable Hybrid Memories," IEEE Comp. Arch. Letters, 2012.

Yoon+, "Row Buffer Locality Aware Caching Policies for Hybrid Memories," ICCD 2012 Best Paper Award.

# Recall: Remote Memory in Large Servers

---

- Memory hierarchy extends beyond a single server
- This enables even higher memory capacity
  - Needed to support modern data-intensive workloads



# 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
  - ❑ Specialized hardware monitors memory accesses
  - ❑ Memorizes, finds, learns address strides/patterns/correlations
  - ❑ 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

# 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
- Execution-based prefetching
- Prefetching performance
  - Coverage, accuracy, timeliness
  - Bandwidth consumption, cache pollution
- Prefetcher throttling
- Issues in multi-core

# Software Prefetching (I)

---

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

# X86 PREFETCH Instruction

## PREFETCHh—Prefetch Data Into Caches

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

### Description


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

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

microarchitecture  
dependent  
specification



different instructions  
for different cache  
levels





# Software Prefetching (II)

---

<pre>for (i=0; i&lt;N; i++) {     __prefetch(a[i+8]);     __prefetch(b[i+8]);     sum += a[i]*b[i]; }</pre>	<pre>while (p) {     __prefetch(p→next);     work(p→data);     p = p→next; }</pre>	<pre>while (p) {     __prefetch(p→next→next→next);     work(p→data);     p = p→next; }</pre>
---	--	--

Which one is better?

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

# Software Prefetching (III)

---

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

# Hardware Prefetching (I)

---

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

# Next-Line Prefetchers

---

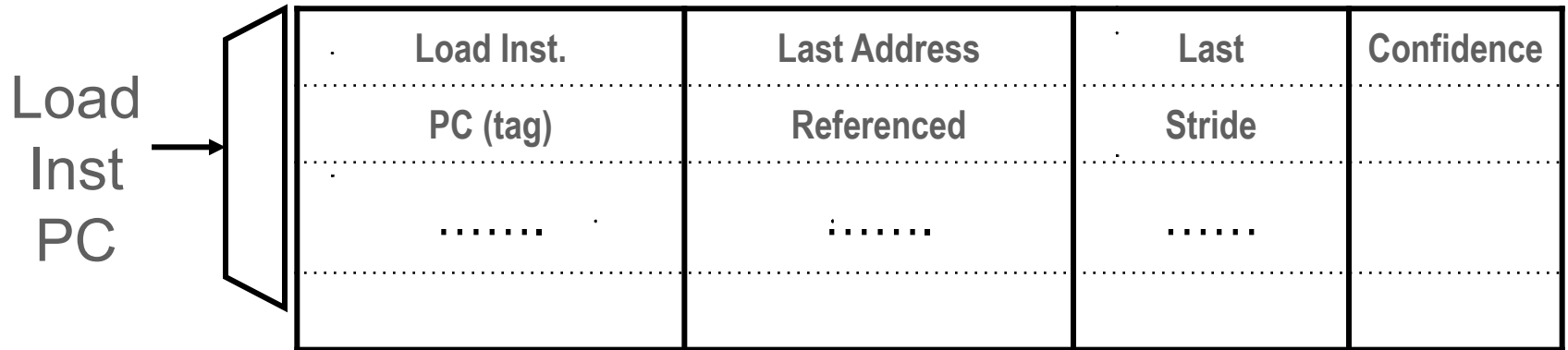
- Simplest form of hardware prefetching: always prefetch next N cache lines after a demand access (or a demand miss)
  - Next-line prefetcher (or next sequential prefetcher)
  - Tradeoffs:
    - + Simple to implement. No need for sophisticated pattern detection
    - + Works well for sequential/streaming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

# Stride Prefetchers

---

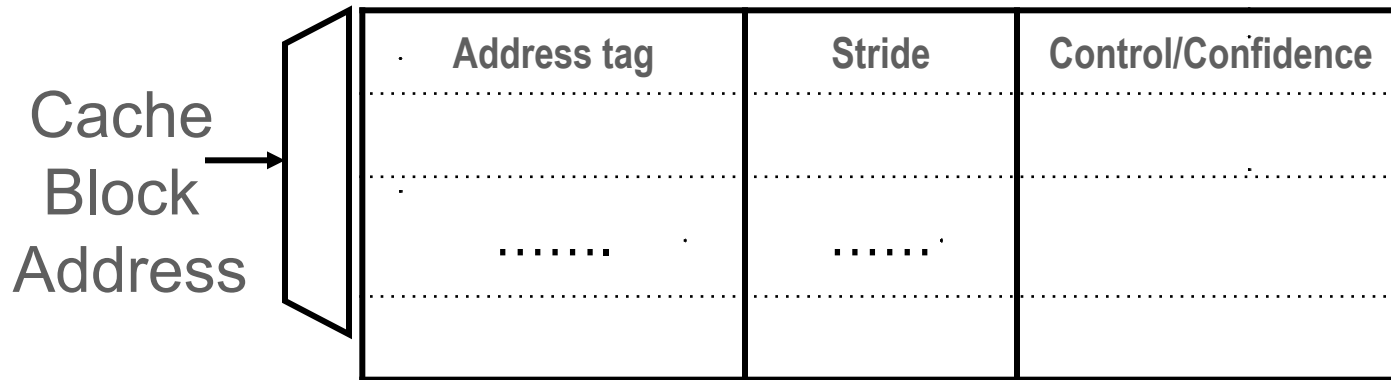
- Consider the following strided memory access pattern:
  - $A, A+N, A+2N, A+3N, A+4N\dots$
  - Stride =  $N$
- Idea: Record the stride between consecutive memory accesses; if stable, use it to predict next  $M$  memory accesses
- Two types
  - Stride determined on a per-instruction basis
  - Stride determined on a per-memory-region basis

# Instruction Based Stride Prefetching



- Each load/store instruction can lead to a memory access pattern with a different stride
  - Can only detect strides caused by each instruction
- Timeliness of prefetches can be an issue
  - Initiating the prefetch when the load is fetched the next time can be too **late**
  - Potential solution: Look ahead in the instruction stream

# Memory-Region Based Based Stride Prefetching



- Can detect strided memory access patterns that appear due to multiple instructions
  - $A, A+N, A+2N, A+3N, A+4N \dots$  where each access could be due to a different instruction
- **Stream prefetching (stream buffers)** is a special case of memory-region based stride prefetching where  $N = 1$

# Tradeoffs in Stream/Stride Prefetching

---

- Instruction based stride prefetching vs. memory region based 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



# Instruction-Based Stride Prefetching

## An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty

Jean-Loup Baer, Tien-Fu Chen  
Department of Computer Science and Engineering  
University of Washington  
Seattle, WA 98195

### Abstract

Conventional cache prefetching approaches can be either hardware-based, generally by using a one-block-lookahead technique, or compiler-directed, with insertions of non-blocking prefetch instructions. We introduce a new hardware scheme based on the prediction of the execution of the instruction stream and associated operand references. It consists of a reference prediction table and a look-ahead program counter and its associated logic. With this scheme, data with regular access patterns is preloaded, independently of the stride size, and preloading of data with irregular access patterns is prevented. We evaluate our design through trace driven simulation by comparing it with a pure data cache approach under three different memory access models. Our experiments show that this scheme is very effective for reducing the data access penalty for scientific programs and that it has moderate success for other applications.

### 1 Introduction

The time when peak processor performance will reach several hundred MIPS is not far away. Such instruction execution rates will have to be achieved through technological advances and enhanced architectural features. Superscalar or multifunctional unit CPU's will increase the raw computational speed. Efficient handling of vector data will be necessary to provide adequate performance for scientific programs. Memory latency will be reduced by cache hierarchies. Processors will have to be designed to support the synchronization and coherency effects of multiprocessing. Thus, we can safely envision that the processor chip will include several functional units, first-level instruction and data caches, and additional hardware support functions. In this paper, we propose the design of an on-chip hardware support function whose goal is to reduce the memory latency due to data cache misses. We will show how it can reduce the contribution of the on-chip data cache to the average number of clock cycles per instruction (CPI)[3].

The component of the CPI due to cache misses depends on two factors: miss ratio and memory latency. Its importance as a contributor to the overall CPI has been illustrated in recent papers [1, 5] where it is shown that the CPI contribution of first-level data caches can reach 2.5.

Current, and future, technology dictates that on-chip caches be small and most likely direct-mapped. Therefore, the small capacity and the lack of associativity will result in relatively high miss ratios. Moreover, pure demand fetching cannot prevent compulsory misses. Our goal is to avoid misses by preloading blocks before they are needed. Naturally, we won't always be successful, since we might preload the wrong block, fail to preload it in time, or displace a useful block. The technique that we present will, however, help in reducing the data cache CPI component.

Our notion of preloading is different from the conventional cache prefetching [11, 12] which associates a successor block to the block being currently referenced. Instead, the preloading technique that we propose is based on the prediction of the instruction stream execution and its associated operand references. Since we rely on instruction stream prediction, the target architecture must include a branch prediction table. The additional hardware support that we propose takes the form of a look-ahead program counter (LA-PC) and a reference prediction table and associated control (RPT). With the help of the LA-PC and the RPT, we generate concurrent cache loading instructions sufficiently ahead of the regular load instructions, so that the latter will result in cache hits. Although this design has some similarity with decoupled architectures [13], it is simpler since it requires significantly less control hardware and no compiler support.

The rest of the paper is organized as follows: Section 2 briefly reviews previous studies of cache prefetching. Section 3 introduces the basic idea and the supporting design. Section 4 explains the evaluation methodology. Section 5 reports on experiments. Section 6 contrasts our hardware-only design to a compiler solution. Concluding remarks are given in Section 7.

### 2 Background and Previous work

#### 2.1 Hardware-based prefetching

Standard caches use a demand fetching policy. As noted by Smith [12], cache prefetching, i.e., the loading of a block before it is going to be referenced, could be used. The pure local hardware management of caches imposes a one block look-ahead (OBL) policy i.e., upon referencing block  $i$ , the only potential prefetch is to block  $i + 1$ . Upon referencing block  $i$ , the options are: prefetch block  $i + 1$  unconditionally, only on a miss to block  $i$ , or if the prefetch has been successful in

# Instruction-Based Stride Prefetching

Doweck, “**Inside Intel® Core™ Microarchitecture and Smart Memory Access,**” Intel White Paper, 2006.

## Instruction Pointer-Based (IP) Prefetcher to Level 1 Data Cache

In addition to memory disambiguation, Intel Smart Memory Access includes advanced prefetchers. Just like their name suggests, prefetchers “prefetch” memory data before it’s requested, placing this data in cache for “just-in-time” execution. By increasing the number of loads that occur from cache versus main memory, prefetching reduces memory latency and improves performance.

The Intel Core microarchitecture includes in each processing core two prefetchers to the Level 1 data cache and the traditional prefetcher to the Level 1 instruction cache. In addition it includes two prefetchers associated with the Level 2 cache and shared between the cores. In total, there are eight prefetchers per dual core processor.

Of particular interest is the IP-based prefetcher that prefetches data to the Level 1 data cache. While the basic idea of IP-based prefetching isn’t new, Intel made some microarchitectural innovations to it for Intel Core microarchitecture.

The purpose of the IP prefetcher, as with any prefetcher, is to predict what memory addresses are going to be used by the program and deliver that data just in time. In order to improve the accuracy of the prediction, the IP prefetcher tags the history of each load using the Instruction Pointer (IP) of the load. For each load with an IP, the IP prefetcher builds a history and keeps it in the IP history array. Based on load history, the IP prefetcher tries to predict the address of the next load accordingly to a constant stride calculation (a fixed distance or “stride” between subsequent accesses to the same memory area). The IP prefetcher then generates a prefetch request with the predicted address and brings the resulting data to the Level 1 data cache.

Obviously, the structure of the IP history array is very important here for its ability to retain history information for each load. The history array in the Intel Core microarchitecture consists of following fields:

- 12 untranslated bits of last demand address
- 13 bits of last stride data (12 bits of positive or negative stride with the 13th bit the sign)
- 2 bits of history state machine
- 6 bits of last prefetched address—used to avoid redundant prefetch requests

Using this IP history array, it’s possible to detect iterating loads that exhibit a perfect stride access pattern ( $A_n - A_{n-1} = \text{Constant}$ ) and thus predict the address required for the next iteration. A prefetch request is then issued to the L1 cache. If the prefetch request hits the cache, the prefetch request is dropped. If it misses, the prefetch request propagates to the L2 cache or memory.

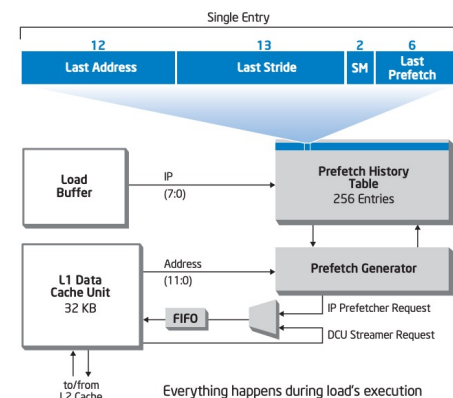


Figure 3: High level block diagram of the relevant parts in the Intel Core microarchitecture IP prefetcher system.

# Memory-Region-Based Stride Prefetching

## Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers

Norman P. Jouppi

Digital Equipment Corporation Western Research Lab

100 Hamilton Ave., Palo Alto, CA 94301

### Abstract

Projections of computer technology forecast processors with peak performance of 1,000 MIPS in the relatively near future. These processors could easily lose half or more of their performance in the memory hierarchy if the hierarchy design is based on conventional caching techniques. This paper presents hardware techniques to improve the performance of caches.

*Miss caching* places a small fully-associative cache between a cache and its refill path. Misses in the cache that hit in the miss cache have only a one cycle miss penalty, as opposed to a many cycle miss penalty without the miss cache. Small miss caches of 2 to 5 entries are shown to be very effective in removing mapping conflict misses in first-level direct-mapped caches.

*Victim caching* is an improvement to miss caching that loads the small fully-associative cache with the victim of a miss and not the requested line. Small victim caches of 1 to 5 entries are even more effective at removing conflict misses than miss caching.

*Stream buffers* prefetch cache lines starting at a cache miss address. The prefetched data is placed in the buffer and not in the cache. Stream buffers are useful in removing capacity and compulsory cache misses, as well as some instruction cache conflict misses. Stream buffers are more effective than previously investigated prefetch techniques at using the next slower level in the memory hierarchy when it is pipelined. An extension to the basic stream buffer, called *multi-way stream buffers*, is introduced. Multi-way stream buffers are useful for prefetching along multiple intertwined data reference streams.

Together, victim caches and stream buffers reduce the miss rate of the first level in the cache hierarchy by a factor of two to three on a set of six large benchmarks.

dous increases in miss cost. For example, a cache miss on a VAX 11/780 only costs 60% of the average instruction execution. Thus even if every instruction had a cache miss, the machine performance would slow down by only 60%! However, if a RISC machine like the WRL Titan [10] has a miss, the cost is almost ten instruction times. Moreover, these trends seem to be continuing, especially the increasing ratio of memory access time to machine cycle time. In the future a cache miss all the way to main memory on a superscalar machine executing two instructions per cycle could cost well over 100 instruction times! Even with careful application of well-known cache design techniques, machines with main memory latencies of over 100 instruction times can easily lose over half of their potential performance to the memory hierarchy. This makes both hardware and software research on advanced memory hierarchies increasingly important.

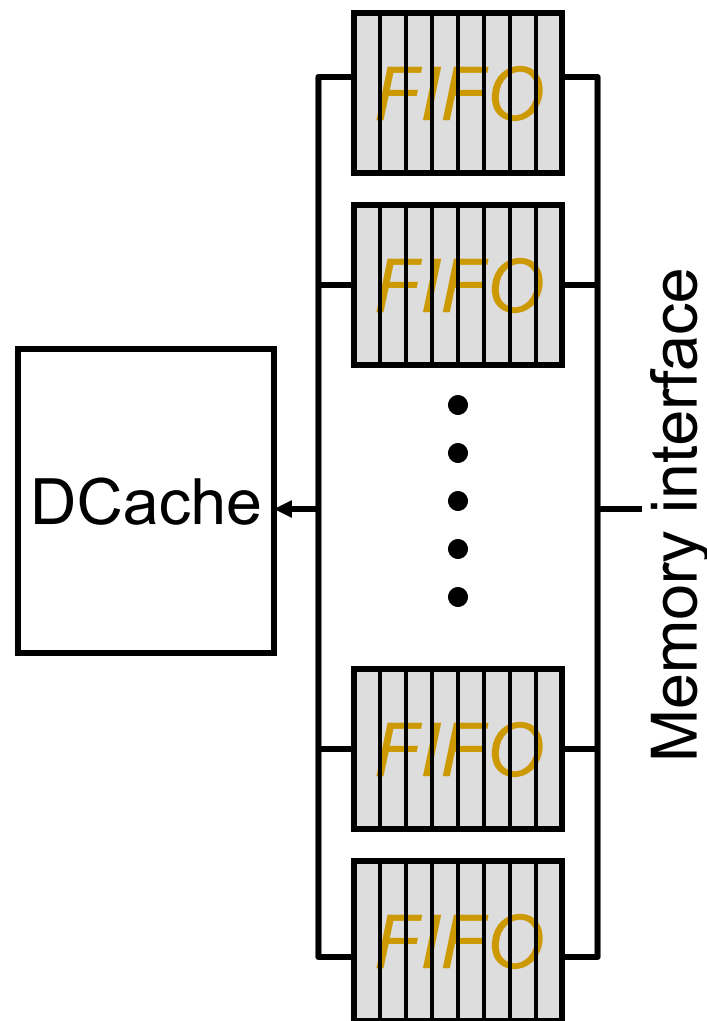
Machine	cycles per instr	cycle time (ns)	mem time (ns)	miss cost (cycles)	miss cost (instr)
VAX11/780	10.0	200	1200	6	.6
WRL Titan	1.4	45	540	12	8.6
?	0.5	4	280	70	140.0

Table 1-1: The increasing cost of cache misses

This paper investigates new hardware techniques for increasing the performance of the memory hierarchy. Section 2 describes a baseline design using conventional caching techniques. The large performance loss due to the memory hierarchy is a detailed motivation for the techniques discussed in the remainder of the paper. Techniques for reducing misses due to mapping conflicts (i.e., lack of associativity) are presented in Section 3. An

# Stream Buffers (Jouppi, ISCA 1990)

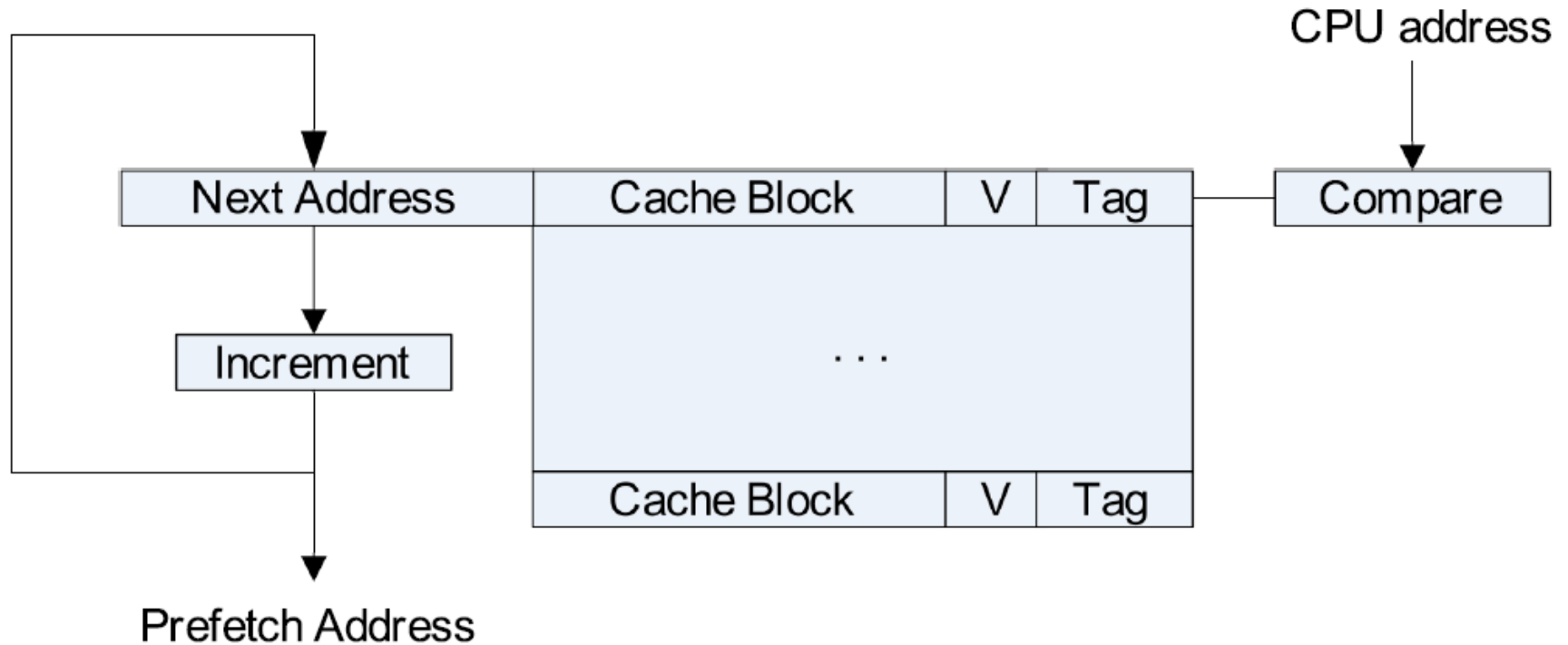
- Each stream buffer holds one stream of sequentially prefetched cache lines
- On a load miss check the head of all stream buffers for an address match
  - if hit, pop the entry from FIFO, update the cache with data
  - if not, allocate a new stream buffer to the new miss address (may have to replace 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



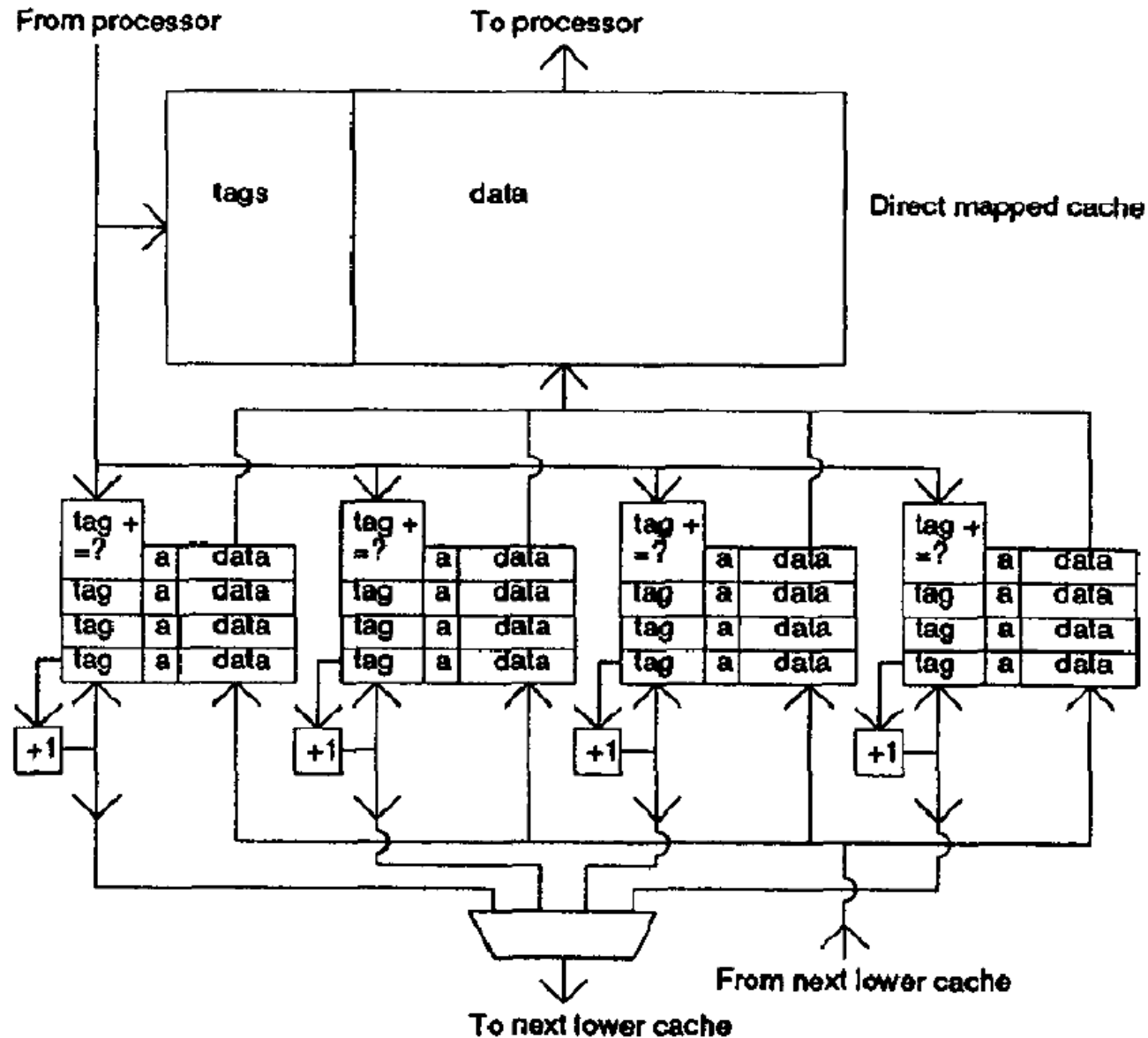
Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," ISCA 1990.

# Stream Buffer Design

---



# Stream Buffer Design





# Streaming Prefetcher in IBM POWER4

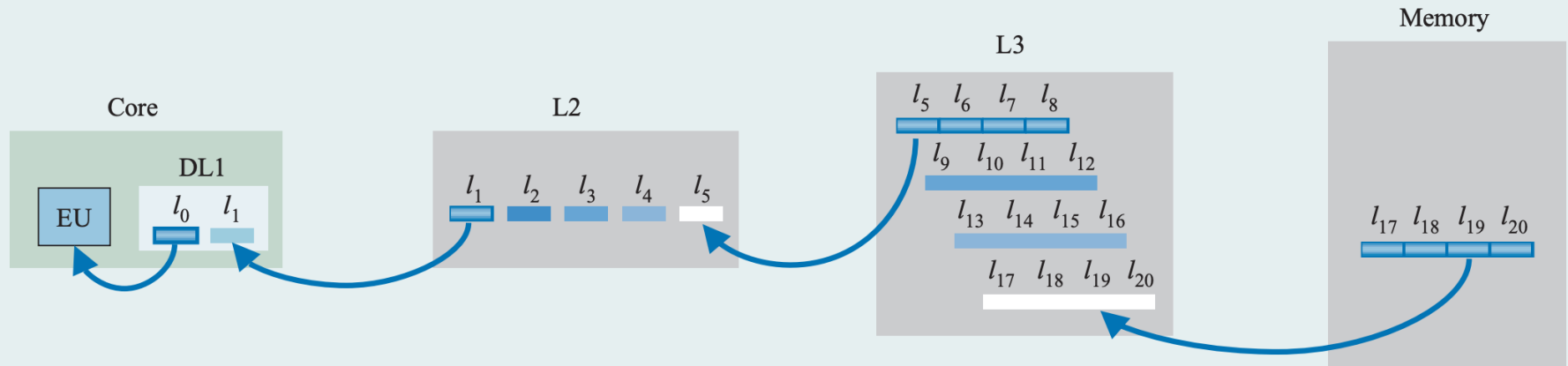


Figure 8

POWER4 hardware data prefetch.

## ***Hardware data prefetch***

POWER4 systems employ hardware to prefetch data transparently to software into the L1 data cache. When load instructions miss sequential cache lines, either ascending or descending, the prefetch engine initiates accesses to the following cache lines before being referenced by load instructions. In order to ensure that the data will be in the L1 data cache, data is prefetched into the L2 from the L3 and into the L3 from memory. **Figure 8** shows the sequence of prefetch operations. Eight such streams per processor are supported.

# A Recommended Paper: Stream Prefetching

## Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers

Norman P. Jouppi

Digital Equipment Corporation Western Research Lab

100 Hamilton Ave., Palo Alto, CA 94301

### Abstract

Projections of computer technology forecast processors with peak performance of 1,000 MIPS in the relatively near future. These processors could easily lose half or more of their performance in the memory hierarchy if the hierarchy design is based on conventional caching techniques. This paper presents hardware techniques to improve the performance of caches.

*Miss caching* places a small fully-associative cache between a cache and its refill path. Misses in the cache that hit in the miss cache have only a one cycle miss penalty, as opposed to a many cycle miss penalty without the miss cache. Small miss caches of 2 to 5 entries are shown to be very effective in removing mapping conflict misses in first-level direct-mapped caches.

*Victim caching* is an improvement to miss caching that loads the small fully-associative cache with the victim of a miss and not the requested line. Small victim caches of 1 to 5 entries are even more effective at removing conflict misses than miss caching.

*Stream buffers* prefetch cache lines starting at a cache miss address. The prefetched data is placed in the buffer and not in the cache. Stream buffers are useful in removing capacity and compulsory cache misses, as well as some instruction cache conflict misses. Stream buffers are more effective than previously investigated prefetch techniques at using the next slower level in the memory hierarchy when it is pipelined. An extension to the basic stream buffer, called *multi-way stream buffers*, is introduced. Multi-way stream buffers are useful for prefetching along multiple intertwined data reference streams.

Together, victim caches and stream buffers reduce the miss rate of the first level in the cache hierarchy by a factor of two to three on a set of six large benchmarks.

dous increases in miss cost. For example, a cache miss on a VAX 11/780 only costs 60% of the average instruction execution. Thus even if every instruction had a cache miss, the machine performance would slow down by only 60%! However, if a RISC machine like the WRL Titan [10] has a miss, the cost is almost ten instruction times. Moreover, these trends seem to be continuing, especially the increasing ratio of memory access time to machine cycle time. In the future a cache miss all the way to main memory on a superscalar machine executing two instructions per cycle could cost well over 100 instruction times! Even with careful application of well-known cache design techniques, machines with main memory latencies of over 100 instruction times can easily lose over half of their potential performance to the memory hierarchy. This makes both hardware and software research on advanced memory hierarchies increasingly important.

Machine	cycles per instr	cycle time (ns)	mem time (ns)	miss cost (cycles)	miss cost (instr)
VAX11/780	10.0	200	1200	6	.6
WRL Titan	1.4	45	540	12	8.6
?	0.5	4	280	70	140.0

Table 1-1: The increasing cost of cache misses

This paper investigates new hardware techniques for increasing the performance of the memory hierarchy. Section 2 describes a baseline design using conventional caching techniques. The large performance loss due to the memory hierarchy is a detailed motivation for the techniques discussed in the remainder of the paper. Techniques for reducing misses due to mapping conflicts (i.e., lack of associativity) are presented in Section 3. An



# 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.

# Pentium4-Like Locality Based Prefetcher [Srinath+, HPCA 2007]

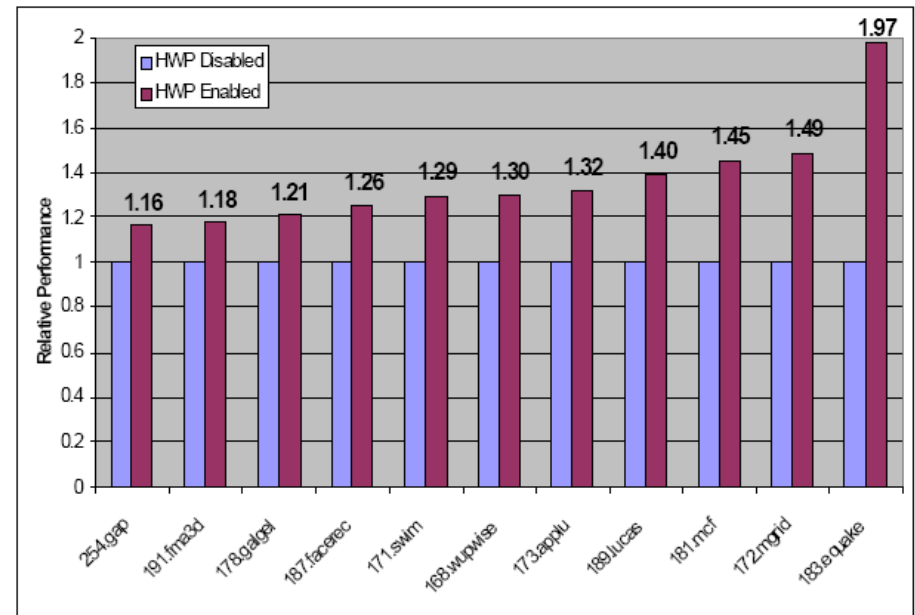
---

- 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).

# Effects 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 well in real life

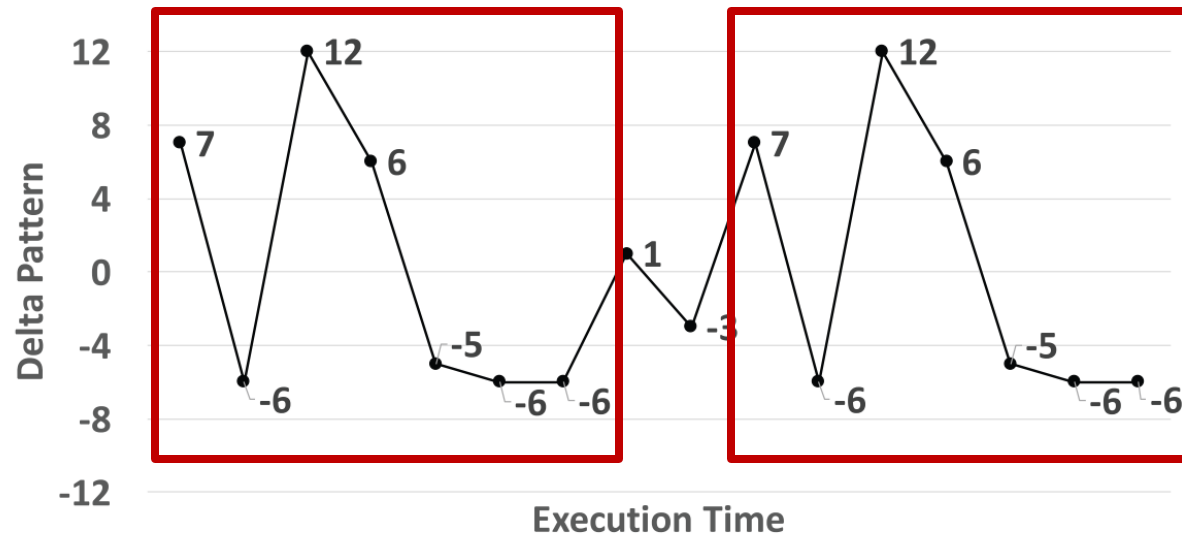
- Single-core systems
- Boggs et al., "The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology", Intel Technology Journal, Feb 2004.

# What About More Complex Access Patterns?

---

- Simple regular patterns
  - Stride, stream prefetchers do well
- Complex regular patterns
  - E.g., multiple regular strides
  - +1, +2, +3, +1, +2, +3, +1, +2, +3, ...
- Irregular patterns
  - Linked data structure traversals
  - Indirect array accesses
  - Random accesses
  - Multiple data structures accessed concurrently
  - ...

# Multi-Stride Detection in Modern Prefetchers



GemsFDTD

Complex but predictable set of strides

## Path Confidence based Lookahead Prefetching

Jinchun Kim\*, Seth H. Pugsley†, Paul V. Gratz\*, A. L. Narasimha Reddy\*, Chris Wilkerson† and Zeshan Chishti†

\*Texas A& M University

cienlux@tamu.edu, pgratz@gratz1.com, reddy@tamu.edu

†Intel Labs

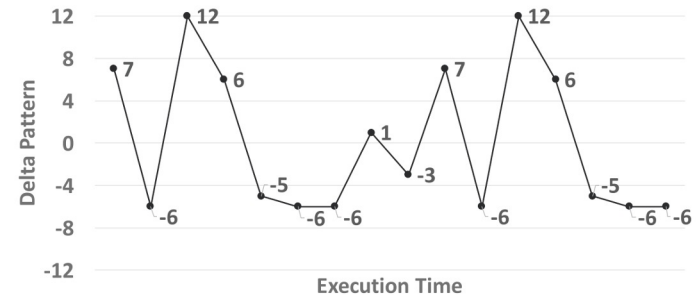
{seth.h.pugsley, chris.wilkerson, zeshan.a.chishti}@intel.com

# Path Confidence Based Lookahead Prefetching

## ■ Key Idea:

- Given a **history/signature/pattern of strides**, learn and predict what stride might come next

- $\{7, -6, 12\} \rightarrow 6, \{-6, 12, 6\} \rightarrow -5, \dots$



- **Bootstrap** prediction to generate new predictions, until the cascaded path confidence drops below a **threshold**

	History of Strides	Prediction	Prediction Confidence	Path Confidence	
Pass 1	{7,-6,12}	6	85%	85%	Bootstrap

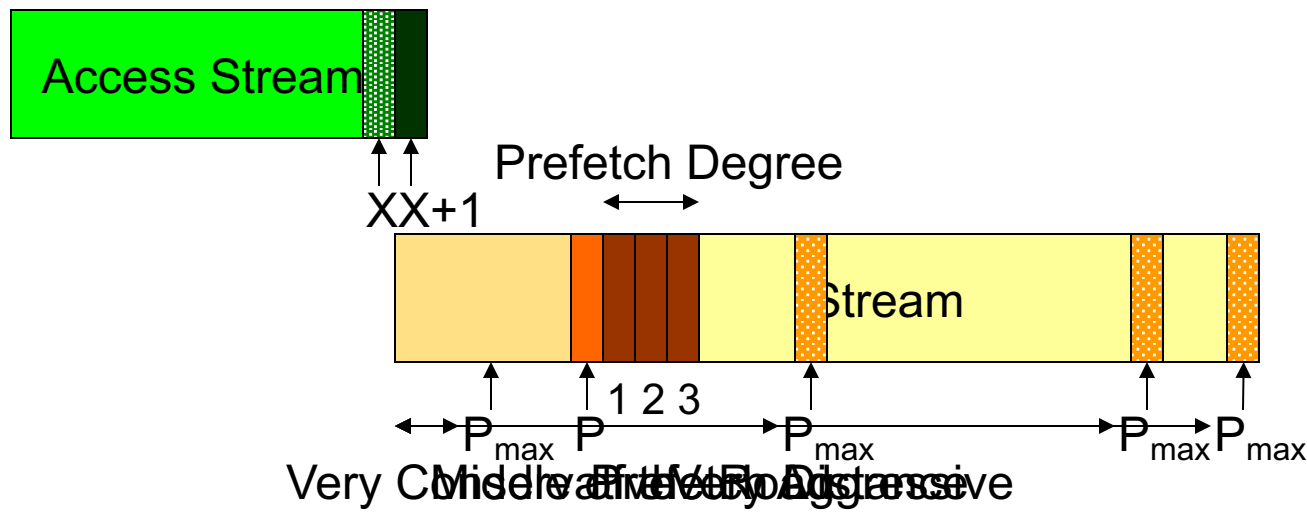
# Prefetcher Performance (I)

---

- **Accuracy** (used prefetches / sent prefetches)
- **Coverage** (prefetched misses / all misses)
- **Timeliness** (on-time prefetches / used prefetches)
  
- **Bandwidth consumption**
  - Memory bandwidth consumed with prefetcher / without prefetcher
  - Good news: **Can utilize idle bus bandwidth (if available)**
  
- **Cache pollution**
  - Extra demand misses due to prefetch placement in cache
  - More difficult to quantify but affects performance

# Prefetcher Performance (II)

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



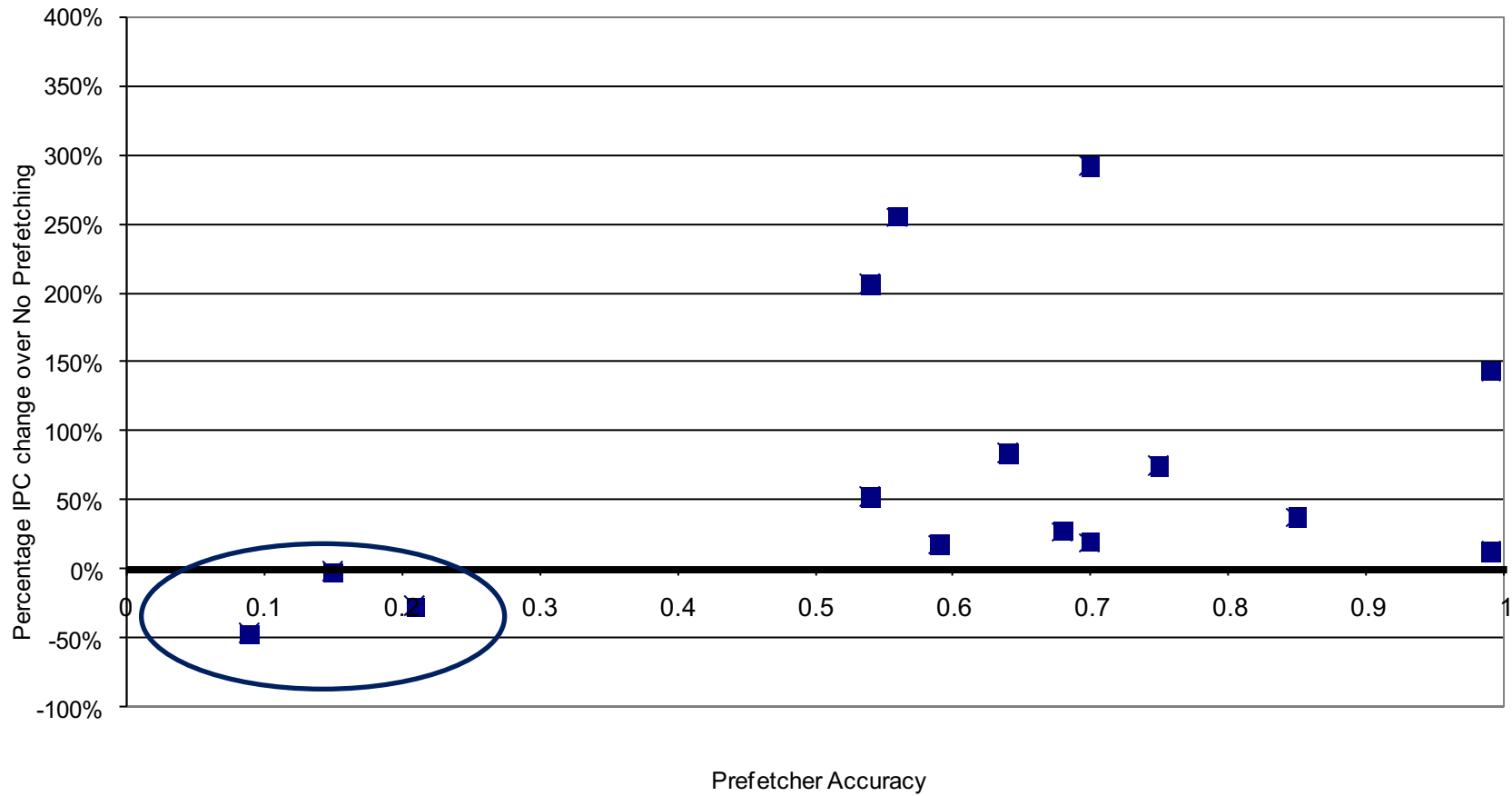


# Prefetcher Performance (III)

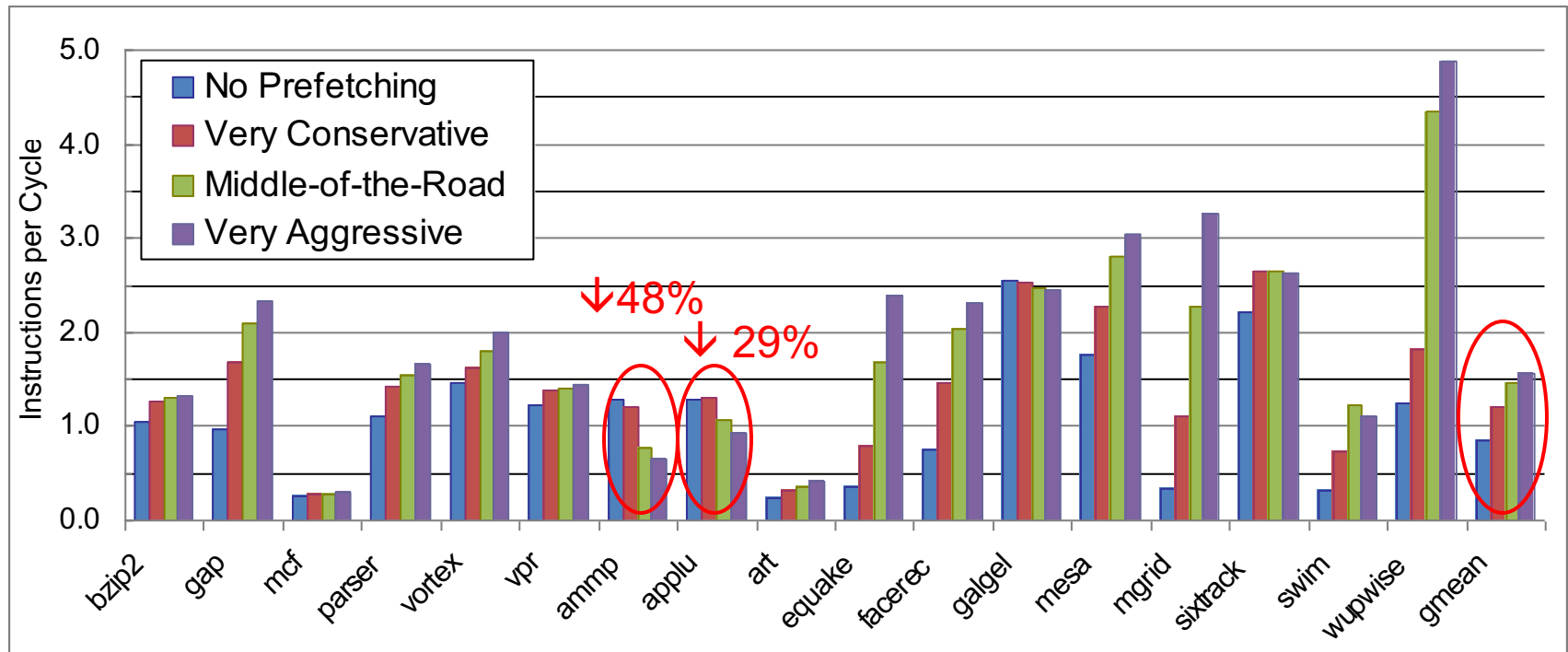
---

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

# Prefetcher Performance (IV)



# Prefetcher Performance (V)

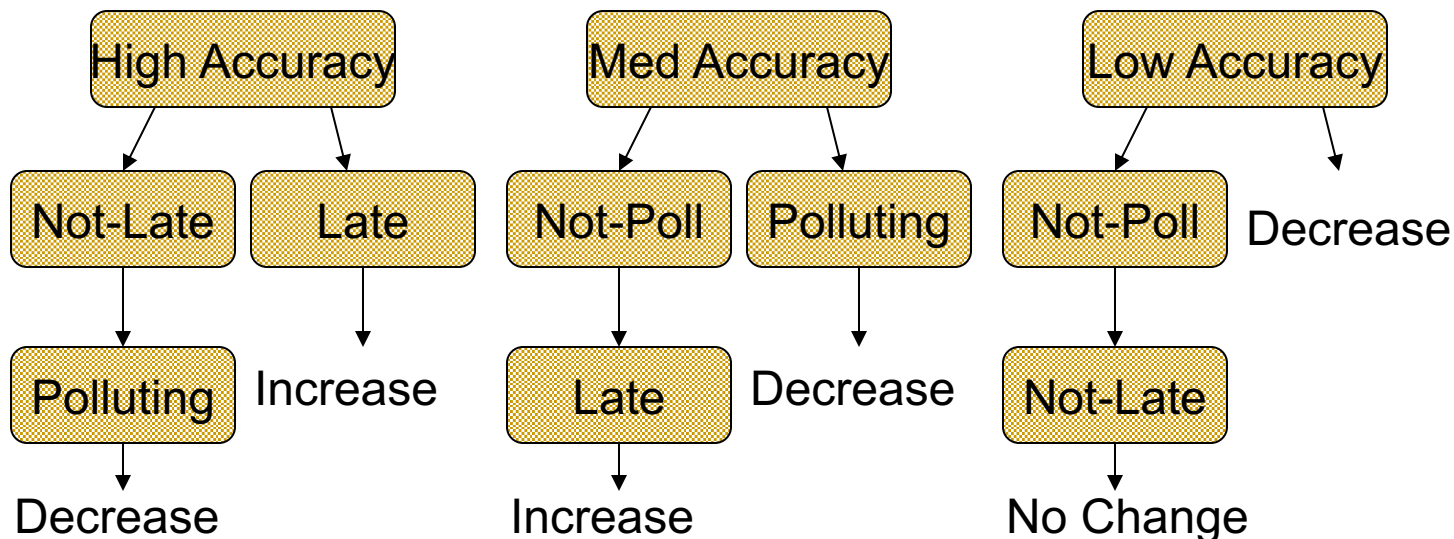


- Srinath et al., “Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers”, HPCA 2007.

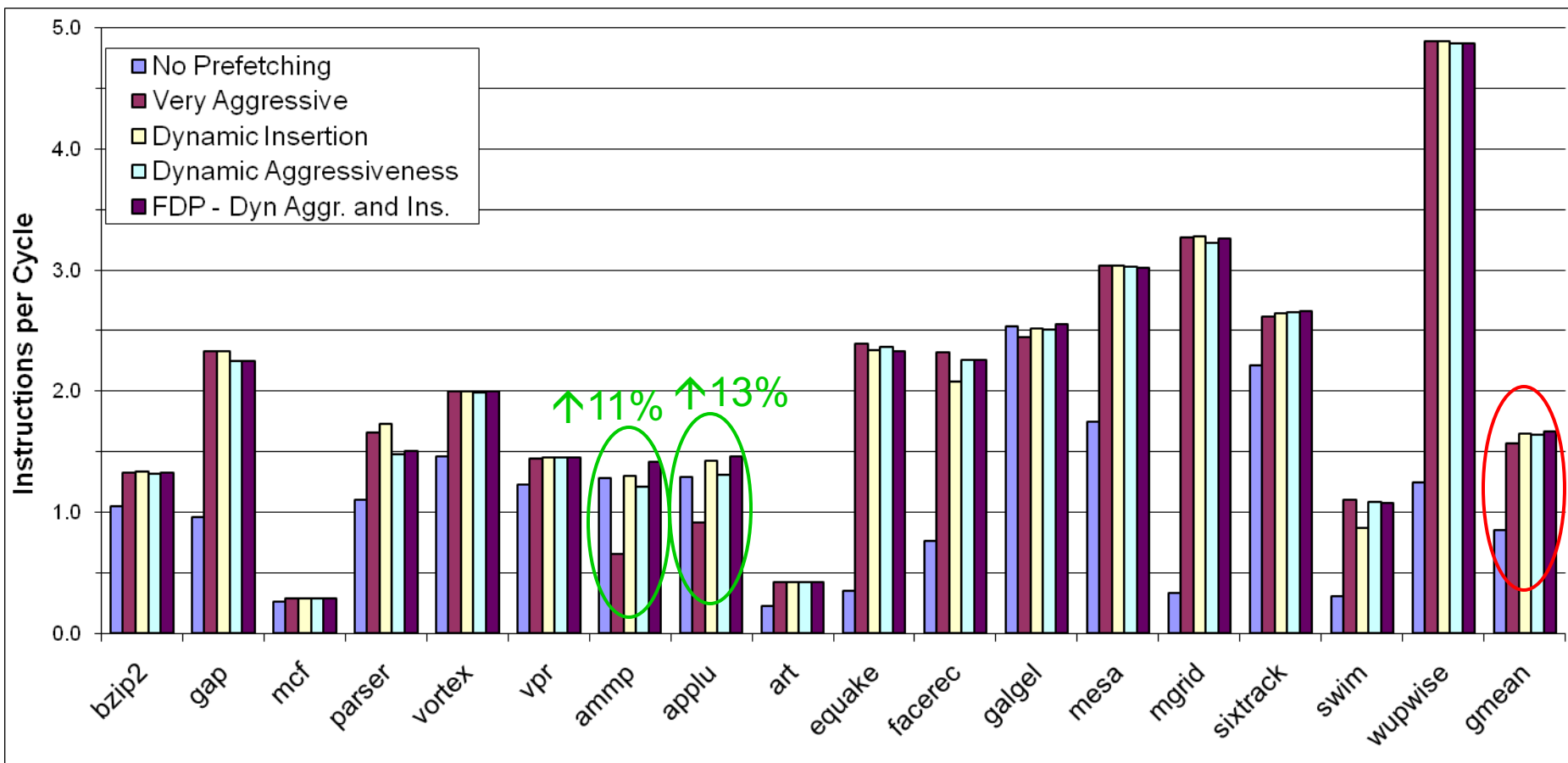
# Feedback-Directed Prefetcher Throttling (I)

## ■ Idea:

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



# Feedback-Directed Prefetcher Throttling (II)



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

# Feedback-Directed Prefetcher Throttling (III)

---

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

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

# 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<sup>†‡</sup> Onur Mutlu<sup>§</sup> Hyesoon Kim<sup>‡</sup> Yale N. Patt<sup>‡</sup>

<sup>†</sup>Microsoft  
ssri@microsoft.com

<sup>§</sup>Microsoft Research  
onur@microsoft.com

<sup>‡</sup>Department of Electrical and Computer Engineering  
The University of Texas at Austin  
{santhosh, hyesoon, patt}@ece.utexas.edu

# Coordinated Prefetching in Multi-Core Systems

---

- 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<sup>†</sup> Onur Mutlu<sup>§</sup> Chang Joo Lee<sup>†</sup> Yale N. Patt<sup>†</sup>

<sup>†</sup>Department of Electrical and Computer Engineering  
The University of Texas at Austin  
{ebrahimi, cjlee, patt}@ece.utexas.edu

<sup>§</sup>Computer Architecture Laboratory (CALCM)  
Carnegie Mellon University  
onur@cmu.edu



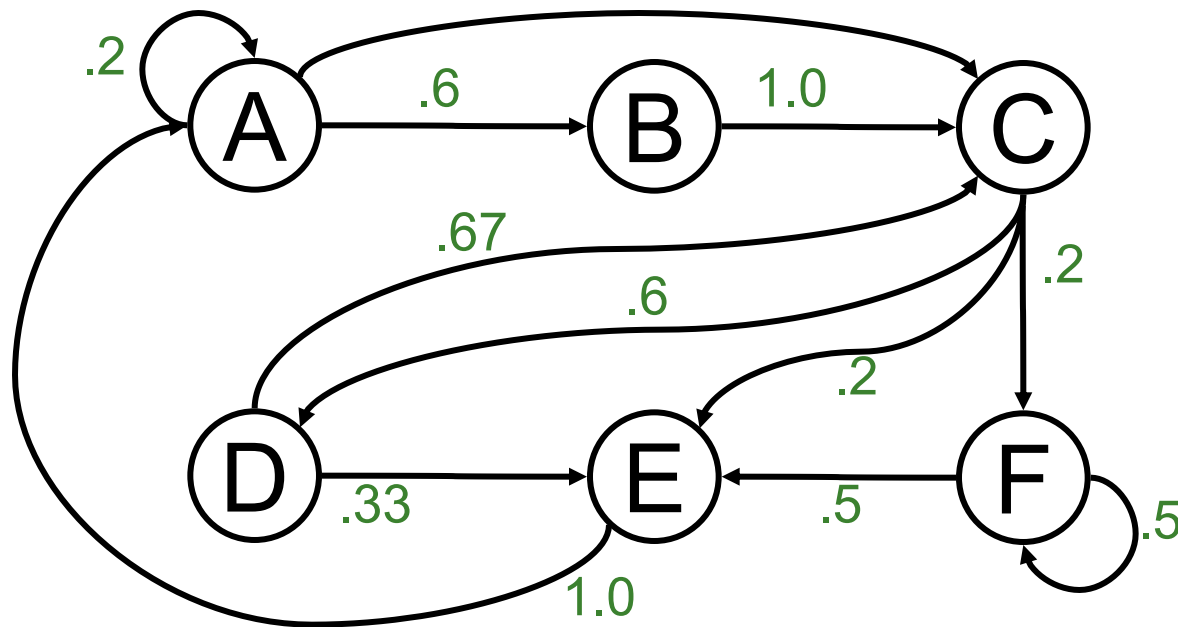
# How to Prefetch More Irregular Access Patterns?

---

- 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

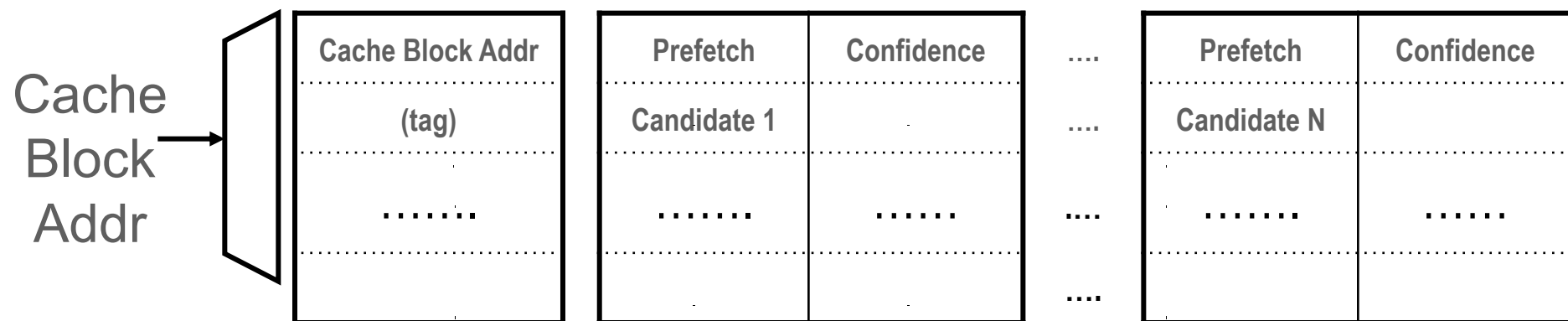
# Address Correlation Based Prefetching (I)

- 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)



- 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:

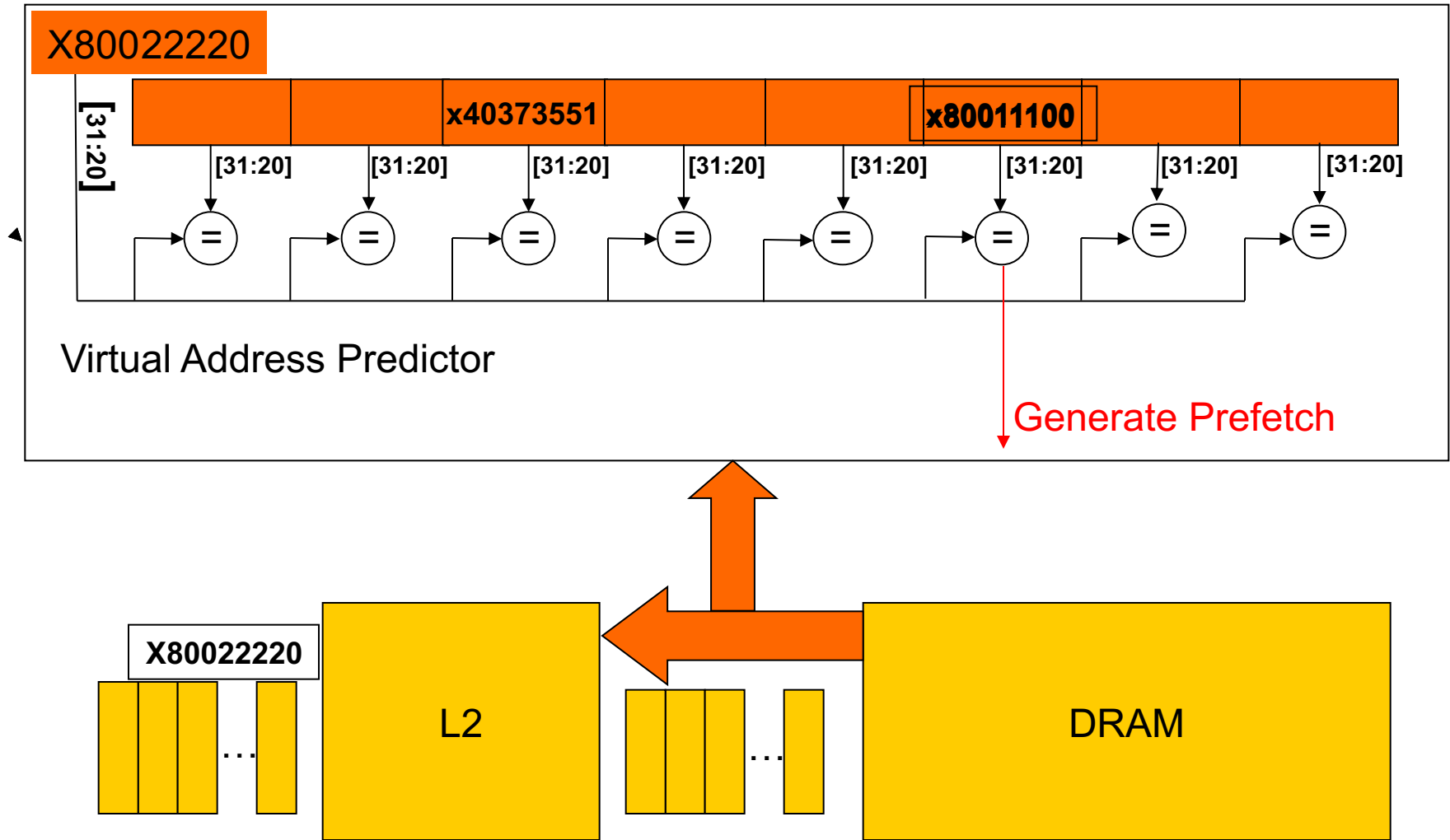
- ❑ **Correlation table** needs to be very large for high coverage
  - Recording every miss address and its subsequent miss addresses is infeasible
- ❑ **Can have low timeliness**: Lookahead is limited since a prefetch for the next access/miss is initiated right after previous
- ❑ Can consume a lot of **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) {
```

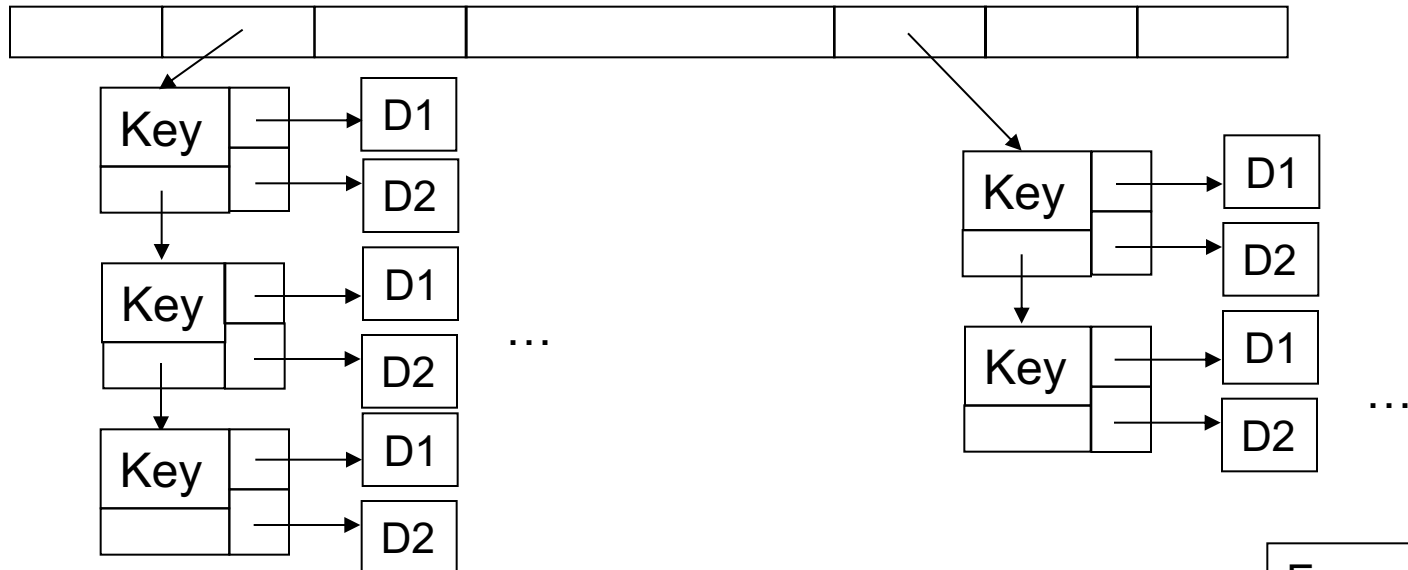
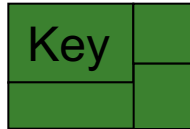
```
...
```

```
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

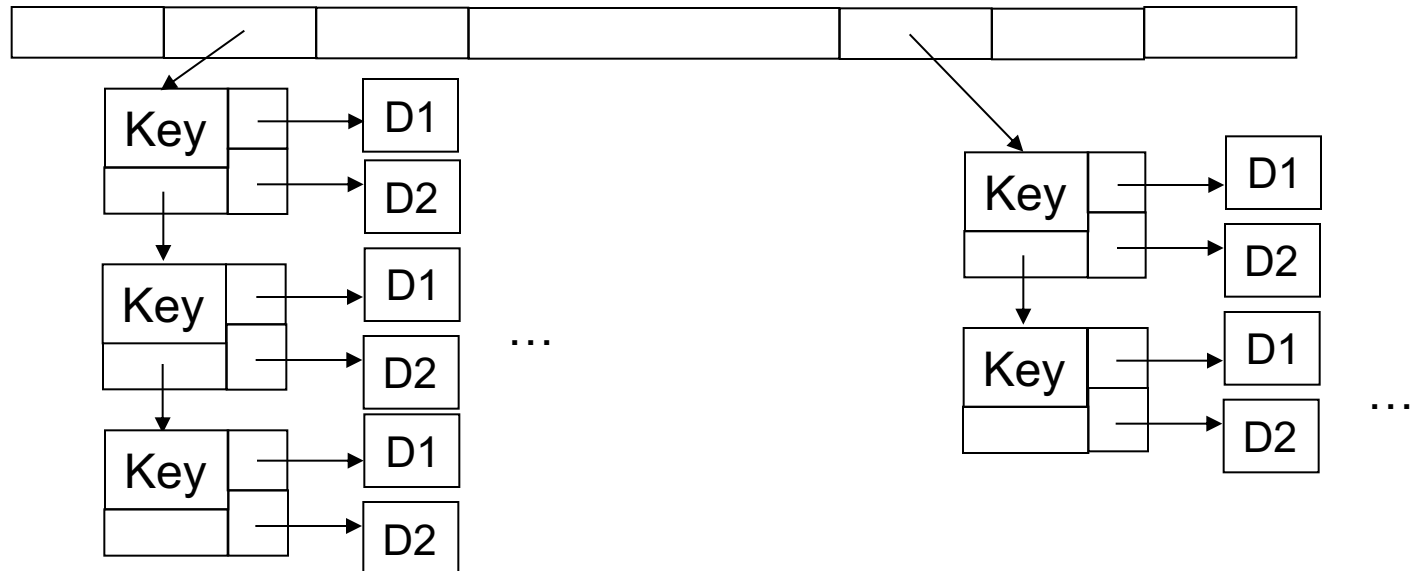




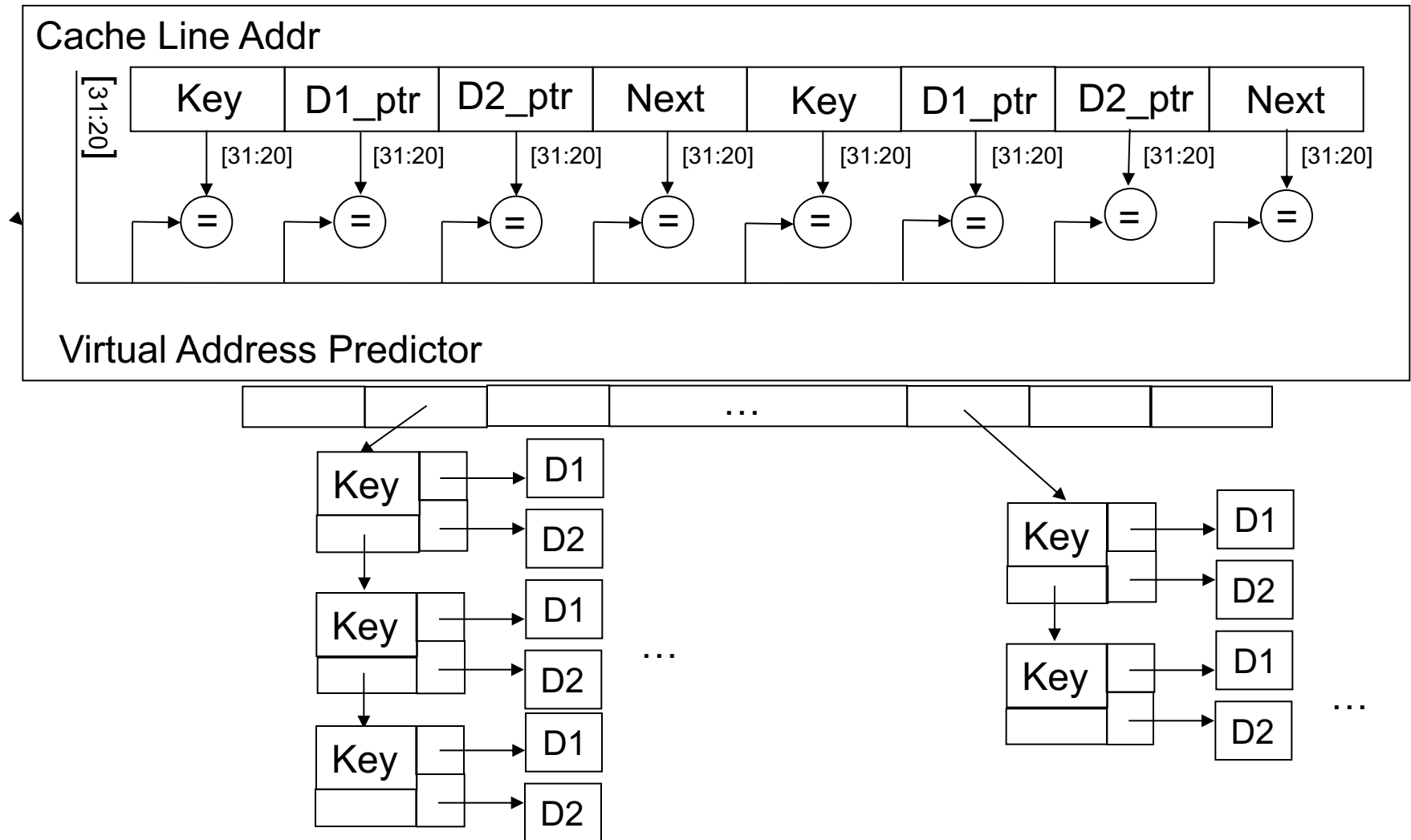
# Shortcomings of CDP – An Example

---

```
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<sup>†</sup>   Onur Mutlu<sup>§</sup>   Yale N. Patt<sup>†</sup>

<sup>†</sup>Department of Electrical and Computer Engineering  
The University of Texas at Austin  
{ebrahimi, patt}@ece.utexas.edu

<sup>§</sup>Computer Architecture Laboratory (CALCM)  
Carnegie Mellon University  
onur@cmu.edu

# Hybrid Hardware Prefetchers

---

- Idea: Use multiple prefetchers to cover many memory access patterns
- + Better prefetch coverage
- + Potentially better timeliness
- More complexity (many design & optimization decisions)
- More bandwidth-intensive
- Prefetchers interfere with each other (contention, pollution)
  - Need to manage accesses from each prefetcher

# Multi-Core Issues in Prefetching

# Real Systems: Prefetching in Multi-Core

---

- Prefetching shared data
  - Coherence misses
- Prefetching efficiency is a lot more important
  - Bus bandwidth more precious
  - Cache space more valuable
- One cores' prefetches interfere with other cores' requests
  - Cache conflicts at multiple levels
  - Bus contention at multiple levels
  - DRAM bank, rank, channel, row buffer contention
  - ...

# Bandwidth-Efficient Hybrid Prefetchers

---

- 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<sup>†</sup>   Onur Mutlu<sup>§</sup>   Yale N. Patt<sup>†</sup>

<sup>†</sup>Department of Electrical and Computer Engineering  
The University of Texas at Austin  
{ebrahimi, patt}@ece.utexas.edu

<sup>§</sup>Computer Architecture Laboratory (CALCM)  
Carnegie Mellon University  
onur@cmu.edu



# Coordinated Control of Prefetchers

---

- 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<sup>†</sup> Onur Mutlu<sup>§</sup> Chang Joo Lee<sup>†</sup> Yale N. Patt<sup>†</sup>

<sup>†</sup>Department of Electrical and Computer Engineering  
The University of Texas at Austin  
{ebrahimi, cjlee, patt}@ece.utexas.edu

<sup>§</sup>Computer Architecture Laboratory (CALCM)  
Carnegie Mellon University  
onur@cmu.edu

# Prefetching-Aware Shared Resource Management

---

- 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<sup>†</sup>   Chang Joo Lee<sup>†‡</sup>   Onur Mutlu<sup>§</sup>   Yale N. Patt<sup>†</sup>

<sup>†</sup>HPS Research Group  
The University of Texas at Austin  
{ebrahimi, patt}@hps.utexas.edu

<sup>‡</sup>Intel Corporation  
chang.joo.lee@intel.com

<sup>§</sup>Carnegie Mellon University  
onur@cmu.edu

# Prefetching-Aware DRAM Control (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<sup>†</sup>   Onur Mutlu<sup>§</sup>   Veynu Narasiman<sup>†</sup>   Yale N. Patt<sup>†</sup>

<sup>†</sup>Department of Electrical and Computer Engineering  
The University of Texas at Austin  
{cjlee, narasima, patt}@ece.utexas.edu

<sup>§</sup>Microsoft Research and Carnegie Mellon University  
onur@{microsoft.com,cmu.edu}

# Prefetching-Aware DRAM Control (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<sup>†</sup> Veynu Narasiman<sup>†</sup> Onur Mutlu<sup>§</sup> Yale N. Patt<sup>†</sup>

<sup>†</sup>Department of Electrical and Computer Engineering  
The University of Texas at Austin  
{cjlee, narasima, patt}@ece.utexas.edu

<sup>§</sup>Computer Architecture Laboratory (CALCM)  
Carnegie Mellon University  
onur@cmu.edu

# Prefetching-Aware Cache Management

---

- 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<sup>†</sup>   Onur Kayiran<sup>‡</sup>   Asit K. Mishra<sup>§</sup>   Mahmut T. Kandemir<sup>†</sup>

Onur Mutlu<sup>‡</sup>   Ravishankar Iyer<sup>§</sup>   Chita R. Das<sup>†</sup>

<sup>†</sup>The Pennsylvania State University  
University Park, PA 16802

<sup>‡</sup>Carnegie Mellon University  
Pittsburgh, PA 15213

<sup>§</sup>Intel Labs  
Hillsboro, OR 97124

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

# Another Example Prefetcher: Self-Optimizing Prefetcher

# Pythia: A Self-Optimizing Prefetcher

Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu,  
**"Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning"**  
*Proceedings of the 54th International Symposium on Microarchitecture (MICRO)*, Virtual, October 2021.

[[Slides \(pptx\)](#) ([pdf](#))]

[[Short Talk Slides \(pptx\)](#) ([pdf](#))]

[[Lightning Talk Slides \(pptx\)](#) ([pdf](#))]

[[Talk Video](#) (20 minutes)]

[[Lightning Talk Video](#) (1.5 minutes)]

[[Pythia Source Code](#) (Officially Artifact Evaluated with All Badges)]

[[arXiv version](#)]

***Officially artifact evaluated as available, reusable and reproducible.***



## Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning

Rahul Bera<sup>1</sup>   Konstantinos Kanellopoulos<sup>1</sup>   Anant V. Nori<sup>2</sup>   Taha Shahroodi<sup>3,1</sup>  
Sreenivas Subramoney<sup>2</sup>   Onur Mutlu<sup>1</sup>

<sup>1</sup>ETH Zürich   <sup>2</sup>Processor Architecture Research Labs, Intel Labs   <sup>3</sup>TU Delft

<https://arxiv.org/pdf/2109.12021.pdf>



1

Mainly use **one**  
program context info.  
for prediction



2

Lack **inherent system**  
**awareness**



3

Lack **in-silicon**  
**customizability**



Why do prefetchers  
not perform well?





## Pythia

Autonomously learns to prefetch using **multiple program context information** and **system-level feedback**

Can be **customized in silicon** to change program context information or prefetching objective on the fly



# Basics of Reinforcement Learning (RL)

---

- Algorithmic approach to learn to take an **action** in a given **situation** to maximize a numerical **reward**

Agent

Environment

- Agent stores **Q-values** for *every* state-action pair
  - **Expected return** for taking an action in a state
  - Given a state, selects action that provides **highest** Q-value

# Brief Overview of Pythia

---

Pythia formulates prefetching as a **reinforcement learning** problem

-----

# What is State?

- **$k$ -dimensional** vector of features

$$S \equiv \{\phi_S^1, \phi_S^2, \dots, \phi_S^k\}$$

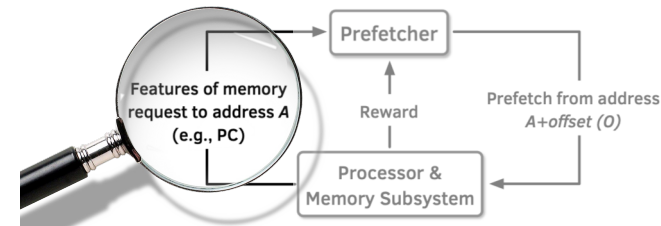
- Feature = control-flow + data-flow

- **Control-flow examples**

- PC
- Branch PC
- Last-3 PCs, ...

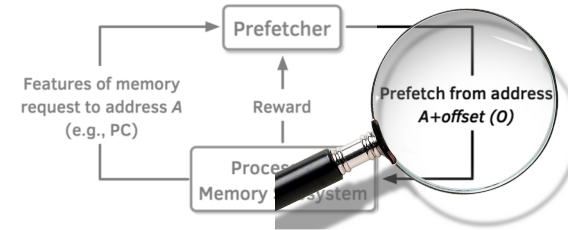
- **Data-flow examples**

- Cacheline address
- Physical page number
- Delta between two cacheline addresses
- Last 4 deltas, ...



# What is Action?

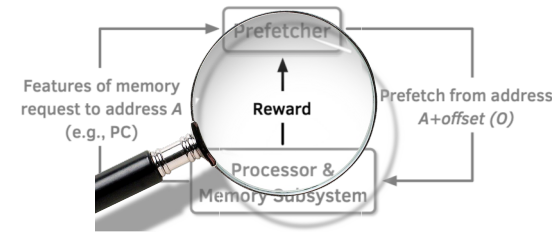
Given a demand access to address A  
the action is to **select prefetch offset “0”**



- **Action-space**: 127 actions in the range  $[-63, +63]$ 
  - For a machine with 4KB page and 64B cacheline
- Upper and lower limits ensure prefetches do not cross **physical page boundary**
- A **zero offset** means **no prefetch** is generated
- We further **prune** action-space by design-space exploration

# What is Reward?

- Defines the **objective** of Pythia
- Encapsulates two metrics:
  - **Prefetch usefulness** (e.g., accurate, late, out-of-page, ...)
  - **System-level feedback** (e.g., mem. b/w usage, cache pollution, energy, ...)
- We demonstrate Pythia with **memory bandwidth usage** as the system-level feedback in the paper



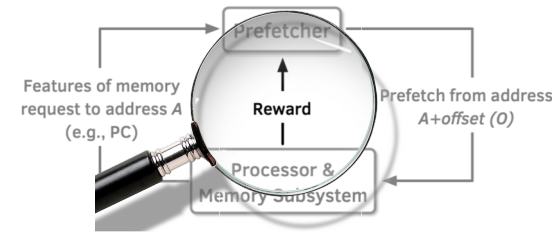
# What is Reward?

- **Seven** distinct reward levels

- *Accurate and timely* ( $R_{AT}$ )
- *Accurate but late* ( $R_{AL}$ )
- *Loss of coverage* ( $R_{CL}$ )
- *Inaccurate*
  - With low memory b/w usage ( $R_{IN-L}$ )
  - With high memory b/w usage ( $R_{IN-H}$ )
- *No-prefetch*
  - With low memory b/w usage ( $R_{NP-L}$ )
  - With high memory b/w usage ( $R_{NP-H}$ )

- Values are set at design time via **automatic design-space exploration**

- Can be **customized** further in silicon for higher performance





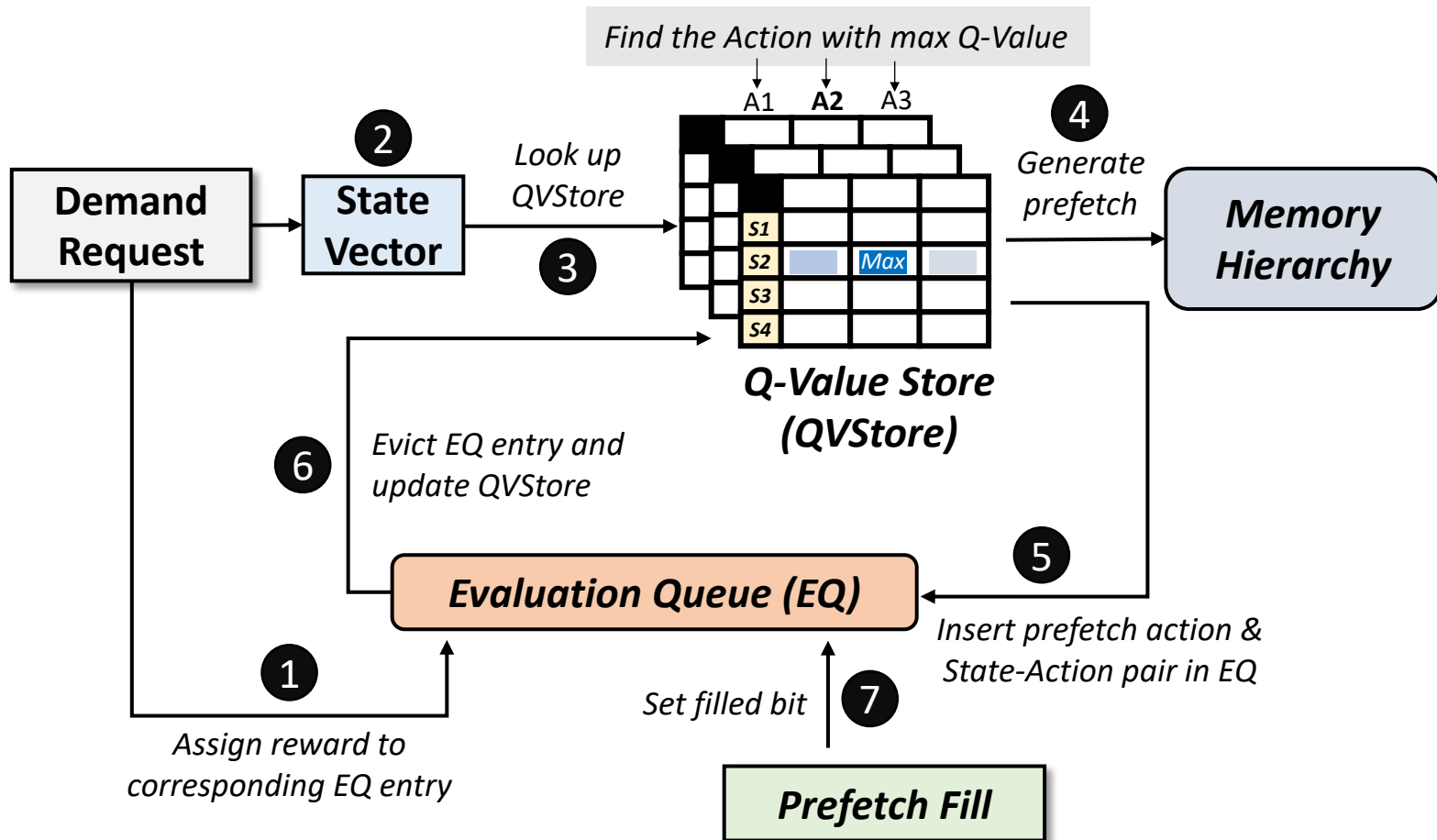
# Basic Pythia Configuration

---

- Derived from **automatic design-space exploration**
- **State:** 2 features
  - PC+Delta
  - Sequence of last-4 deltas
- **Actions:** 16 prefetch offsets
  - Ranging between -6 to +32. Including 0.
- **Rewards:**
  - $R_{AT} = +20$ ;  $R_{AL} = +12$ ;  $R_{NP-H} = -2$ ;  $R_{NP-L} = -4$ ;
  - $R_{IN-H} = -14$ ;  $R_{IN-L} = -8$ ;  $R_{CL} = -12$

# More Detailed Pythia Overview

- **Q-Value Store**: Records Q-values for *all* state-action pairs
- **Evaluation Queue**: A FIFO queue of recently-taken actions

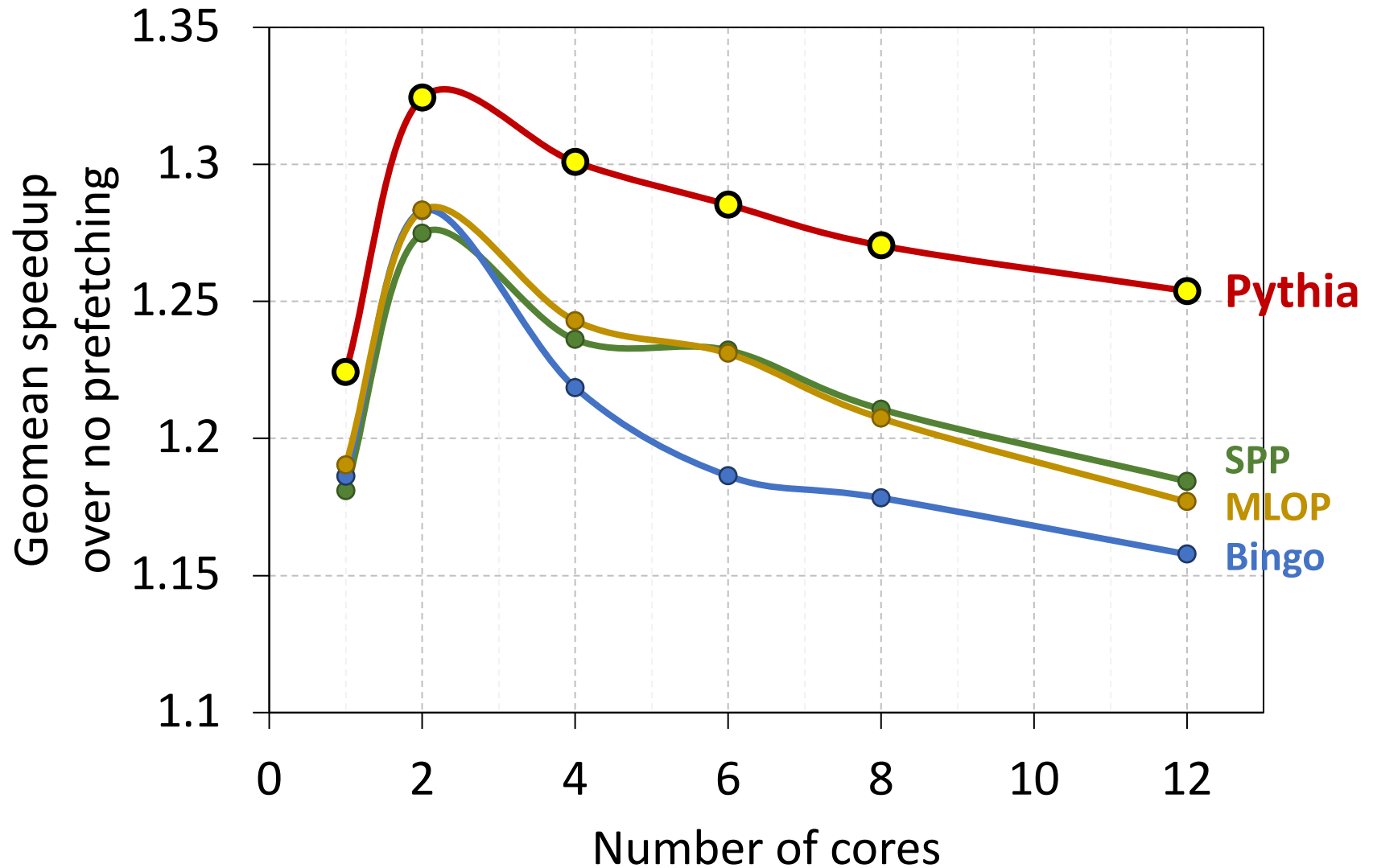


# Simulation Methodology

---

- **Champsim** [3] trace-driven simulator
- **150** single-core memory-intensive workload traces
  - SPEC CPU2006 and CPU2017
  - PARSEC 2.1
  - Ligra
  - Cloudsuite
- Homogeneous and heterogeneous multi-core mixes
- **Five** state-of-the-art prefetchers
  - SPP [Kim+, MICRO'16]
  - Bingo [Bakhshalipour+, HPCA'19]
  - MLOP [Shakerinava+, 3<sup>rd</sup> Prefetching Championship, 2019 ]
  - SPP+DSPatch [Bera+, MICRO'19]
  - SPP+PPF [Bhatia+, ISCA'20]

# Performance with Varying Core Count



# Performance with Varying Core Count

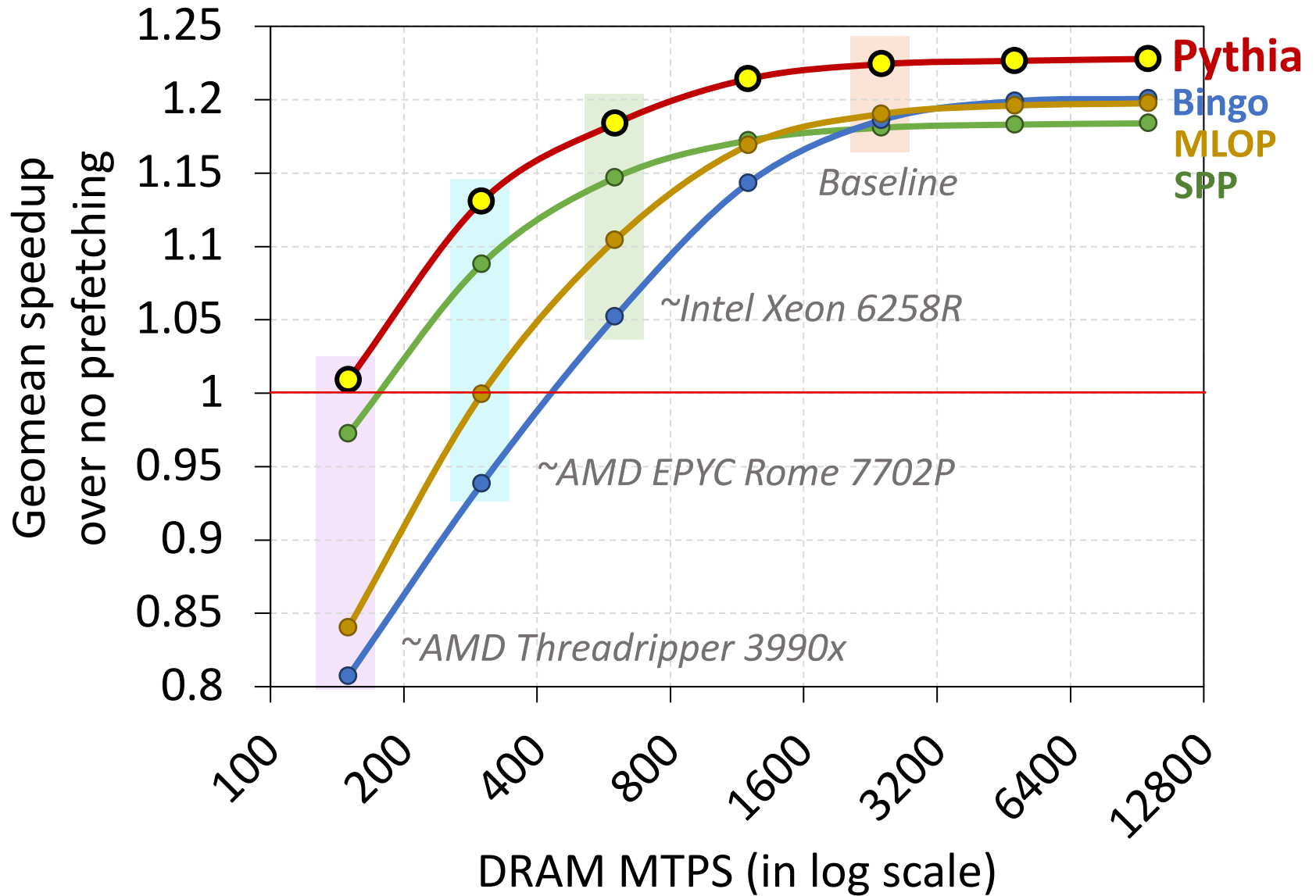


The graph shows performance on the y-axis (ranging from 1.1 to 1.35) against the number of cores on the x-axis (ranging from 0 to 12). Pythia (red line) starts at ~1.28 at 2 cores, peaks at ~1.32 at 4 cores, and then declines. Other models (blue, green, orange lines) show lower performance, with a green line showing a 3.4% gain at 2 cores and a 1.7% gain at 8 cores. A vertical green line at 12 cores is labeled 'SDD'.

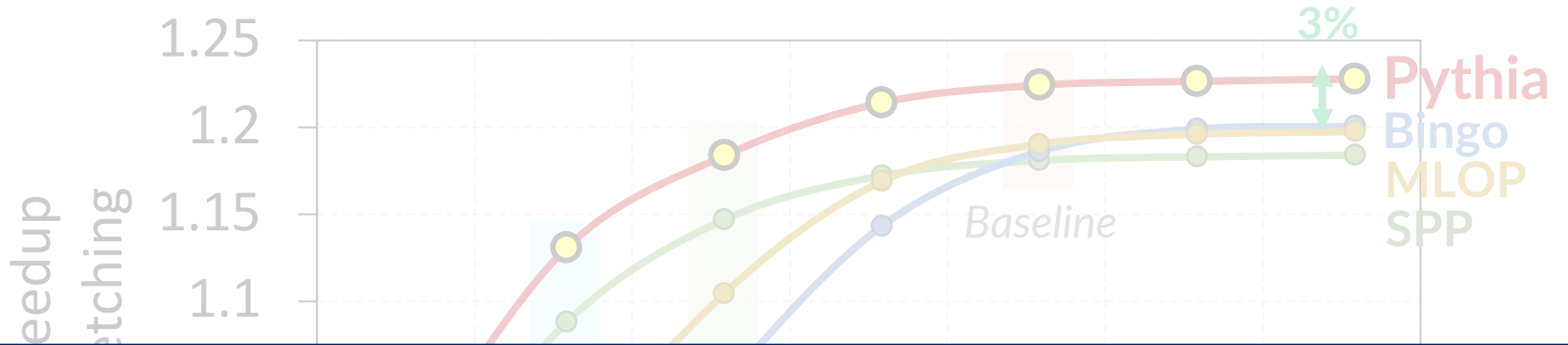
1. Pythia consistently provides the highest performance in **all core configurations**

2. Pythia's gain **increases with core count**

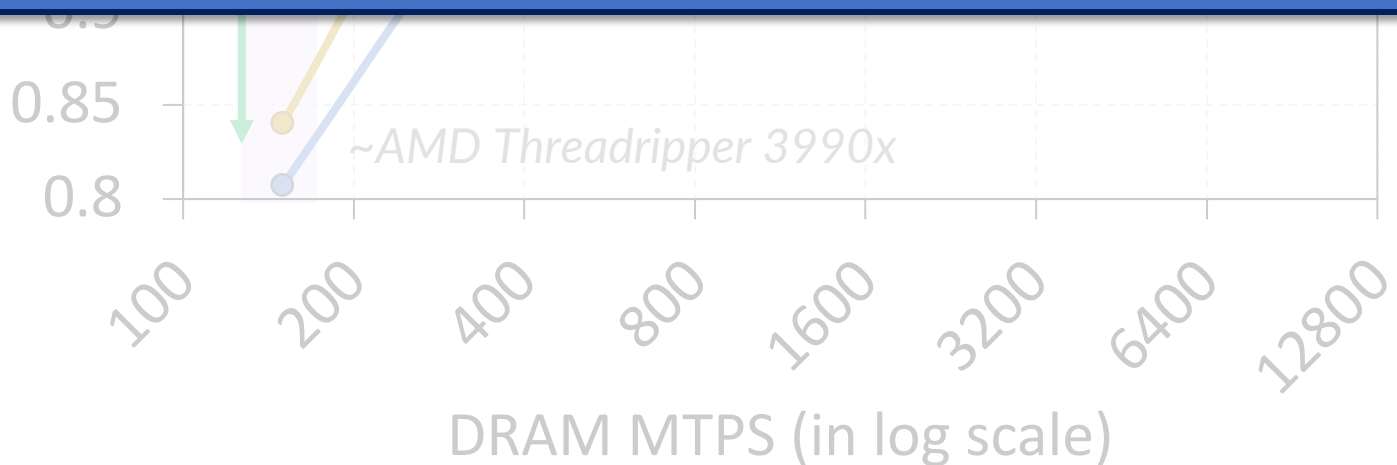
# Performance with Varying DRAM Bandwidth



# Performance with Varying DRAM Bandwidth



**Pythia outperforms prior best prefetchers for a wide range of DRAM bandwidth configurations**



# Pythia is Open Source



<https://github.com/CMU-SAFARI/Pythia>

- MICRO'21 **artifact evaluated**
- **Champsim source** code + **Chisel** modeling code
- **All traces** used for evaluation

The screenshot shows the GitHub repository for CMU-SAFARI/Pythia. The repository is public and has 3 unwatchers, 7 stars, and 2 forks. The main navigation bar includes links to Code, Issues, Pull requests, Actions, Projects, Wiki, Security, Insights, and Settings. The repository is currently on the master branch, with 1 branch and 5 tags. The commit history shows a recent update to the README by rahulbera 2 days ago, with 38 commits in total. The file list includes directories for branch, config, experiments, inc, prefetcher, replacement, scripts, src, tracer, and files for .gitignore, CITATION.cff, LICENSE, and LICENSE.champsim. The right sidebar contains an 'About' section describing Pythia as a customizable hardware prefetching framework, a link to the arXiv paper, and a list of related topics including machine-learning, reinforcement-learning, computer-architecture, prefetcher, microarchitecture, cache-replacement, branch-predictor, champsim-simulator, and champsim-tracer. Below the 'About' section are links to the README, View license, and Cite this repository.

File	Commit Message	Commit Date
branch	Initial commit for MICRO'21 artifact evaluation	2 months ago
config	Initial commit for MICRO'21 artifact evaluation	2 months ago
experiments	Added chart visualization in Excel template	2 months ago
inc	Updated README	6 days ago
prefetcher	Initial commit for MICRO'21 artifact evaluation	2 months ago
replacement	Initial commit for MICRO'21 artifact evaluation	2 months ago
scripts	Added md5 checksum for all artifact traces to verify download	2 months ago
src	Initial commit for MICRO'21 artifact evaluation	2 months ago
tracer	Initial commit for MICRO'21 artifact evaluation	2 months ago
.gitignore	Initial commit for MICRO'21 artifact evaluation	2 months ago
CITATION.cff	Added citation file	6 days ago
LICENSE	Updated LICENSE	2 months ago
LICENSE.champsim	Initial commit for MICRO'21 artifact evaluation	2 months ago



# Pythia Talk Video

## Steering Pythia's Objective via Reward Values

- Customizing reward values to make Pythia conservative towards p

**Strict** Pythia configuration



2 Server-class processors

Bandwidth-sensitive workloads



11:23 / 20:04 • Steering Pythia's Objective via Reward Values >

23



MICRO 2021 Conference Presentations

Pythia: A Customizable Prefetching Framework Using Reinforcement Learning - MICRO'21 Long Talk



Onur Mutlu Lectures

28.9K subscribers

Analytics

Edit video

22



Share



Download



Clip



Save



661 views 11 months ago

Talk: "Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning"  
Full Conference Talk at MICRO 2021 by Rahul Bera

# A Lot More in the Pythia Paper

Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu,  
**"Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning"**  
*Proceedings of the 54th International Symposium on Microarchitecture (MICRO)*, Virtual, October 2021.

[[Slides \(pptx\)](#) ([pdf](#))]

[[Short Talk Slides \(pptx\)](#) ([pdf](#))]

[[Lightning Talk Slides \(pptx\)](#) ([pdf](#))]

[[Talk Video](#) (20 minutes)]

[[Lightning Talk Video](#) (1.5 minutes)]

[[Pythia Source Code](#) (Officially Artifact Evaluated with All Badges)]

[[arXiv version](#)]

***Officially artifact evaluated as available, reusable and reproducible.***



## Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning

Rahul Bera<sup>1</sup>

Konstantinos Kanellopoulos<sup>1</sup>

Anant V. Nori<sup>2</sup>

Taha Shahroodi<sup>3,1</sup>

Sreenivas Subramoney<sup>2</sup>

Onur Mutlu<sup>1</sup>

<sup>1</sup>ETH Zürich

<sup>2</sup>Processor Architecture Research Labs, Intel Labs

<sup>3</sup>TU Delft

<https://arxiv.org/pdf/2109.12021.pdf>

# Learning-Based Off-Chip Load Predictors

---

- **Best Paper Award at MICRO 2022**



## **Hermes: Accelerating Long-Latency Load Requests via Perceptron-Based Off-Chip Load Prediction**

Rahul Bera<sup>1</sup>   Konstantinos Kanellopoulos<sup>1</sup>   Shankar Balachandran<sup>2</sup>   David Novo<sup>3</sup>  
Ataberk Olgun<sup>1</sup>   Mohammad Sadrosadati<sup>1</sup>   Onur Mutlu<sup>1</sup>

<sup>1</sup>ETH Zürich   <sup>2</sup>Intel Processor Architecture Research Lab   <sup>3</sup>LIRMM, Univ. Montpellier, CNRS

# Execution-Based Prefetching

# Execution-based Prefetchers (I)

---

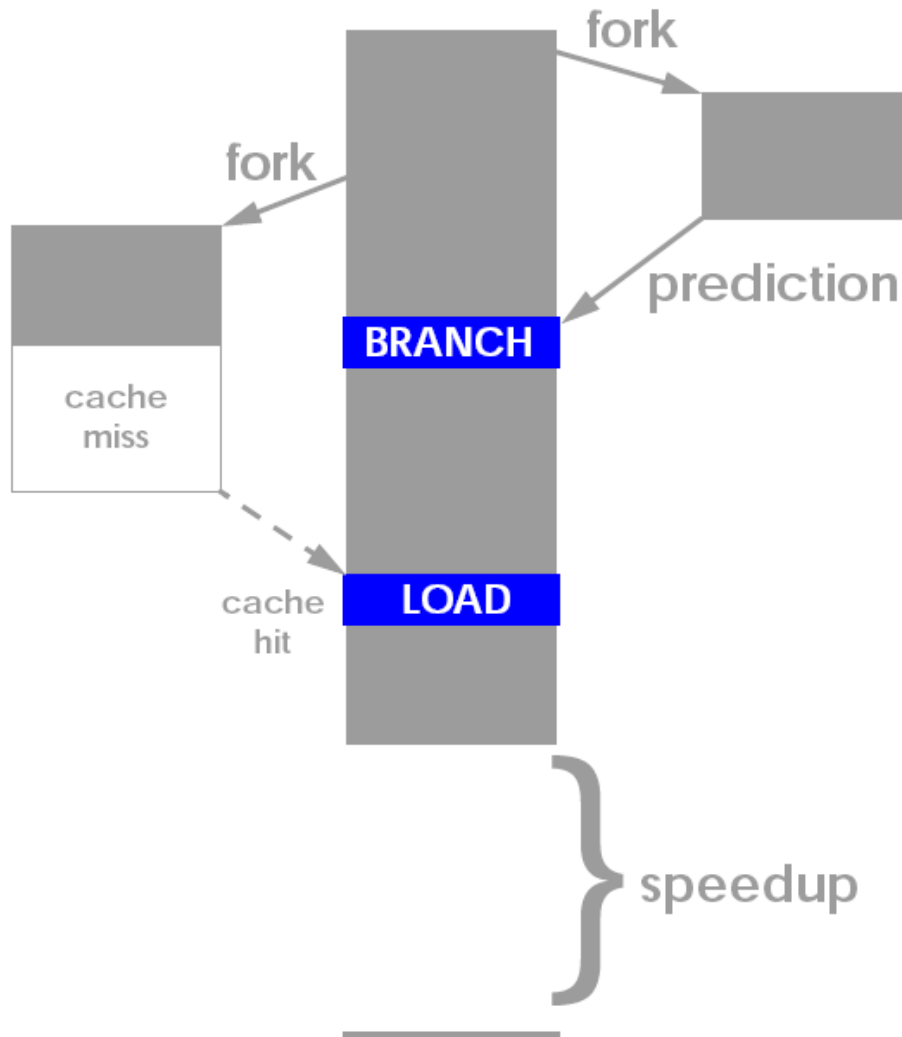
- Idea: Pre-execute a piece of the (pruned) program solely for prefetching data
  - Only need to distill pieces that lead to cache misses
- **Speculative thread:** Pre-executed program piece can be considered a “thread”
- Speculative thread can be executed
  - On a separate processor/core
  - On a separate 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 of the main thread
    - Performs only address generation computation, branch prediction, value prediction (to predict “unknown” values)
  - Purely speculative so there is no need for recovery of main program if the speculative thread is incorrect

# Thread-Based Pre-Execution



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

# Thread-Based Pre-Execution Issues

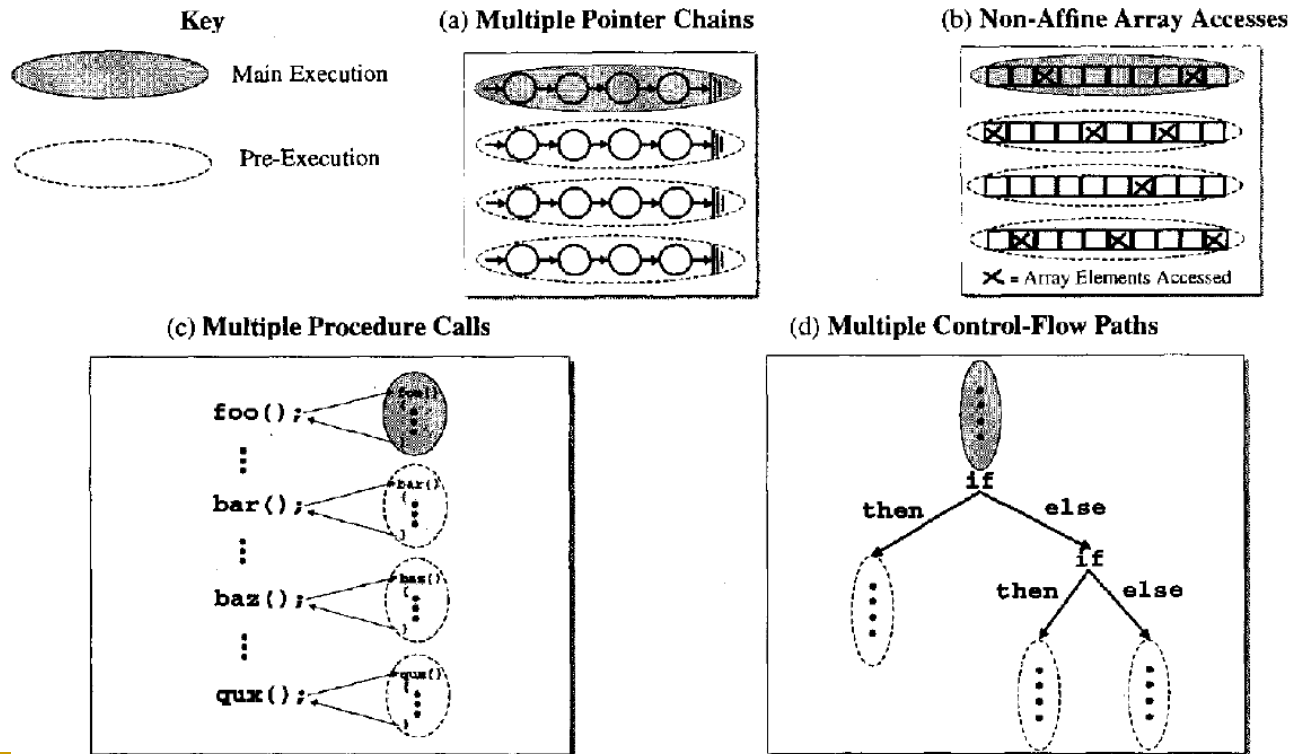
---

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



# Thread-Based Pre-Execution Issues

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



# An Example

## (a) Original Code

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

## (b) Code with Pre-Execution

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

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

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

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

# Example ISA Extensions

---

***Thread\_ID = PreExecute\_Start(Start\_PC, Max\_Insts):***

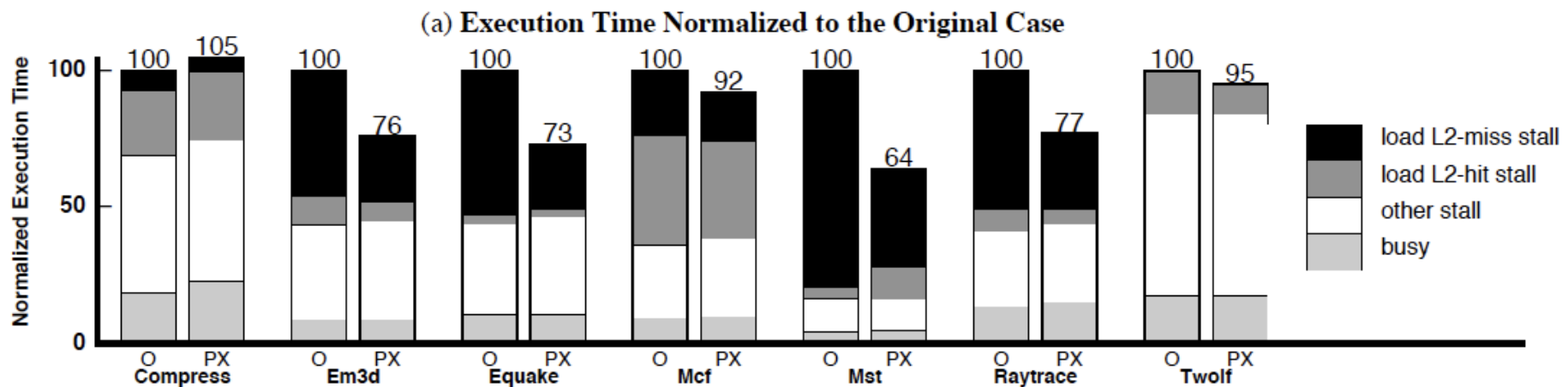
Request for an idle context to start pre-execution at *Start\_PC* and stop when *Max\_Insts* instructions have been executed; *Thread\_ID* holds either the identity of the pre-execution thread or -1 if there is no idle context. This instruction has effect only if it is executed by the main thread.

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

***PreExecute\_Cancel(Thread\_ID):*** Terminate the pre-execution thread with *Thread\_ID*. This instruction has effect only if it is executed by the main thread.

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

# Results on a Multithreaded Processor



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

# Problem Instructions

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

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

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

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

**branch misprediction** and **cache miss** annotations with arrows pointing to lines 6 and 7 respectively.

# 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  cmpltlt $f0,$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
```

# Runahead Execution



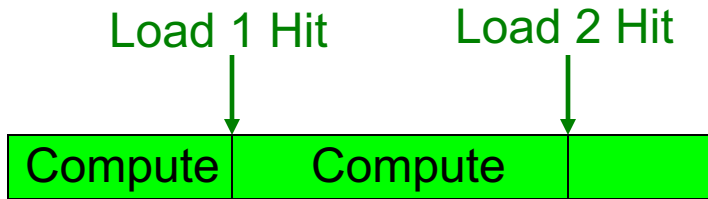
# Runahead Execution

---

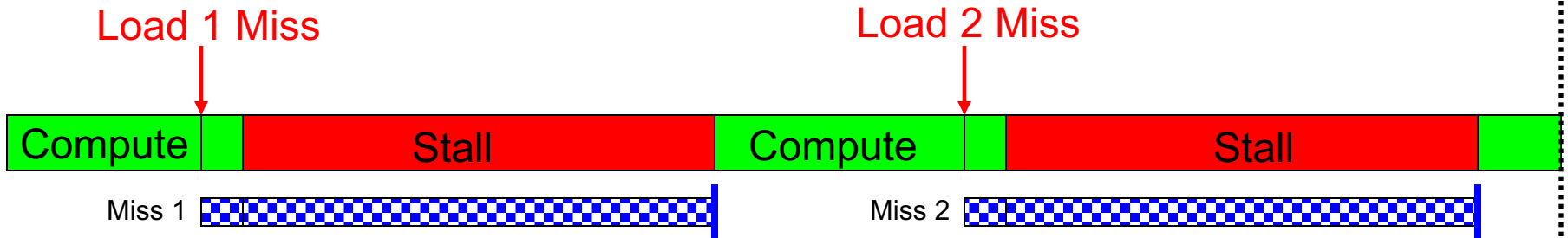
- A technique to obtain the memory-level parallelism benefits of a large instruction window
- When the oldest instruction is a long-latency cache miss:
  - **Checkpoint** architectural state and enter runahead mode
- In runahead mode:
  - **Speculatively pre-execute instructions**
  - **The purpose of pre-execution is to generate prefetches**
  - L2-miss dependent instructions are marked INV and dropped
- When the original miss returns:
  - **Restore checkpoint**, flush pipeline, resume normal execution
- Mutlu et al., “Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors,” HPCA 2003.

# Runahead Example

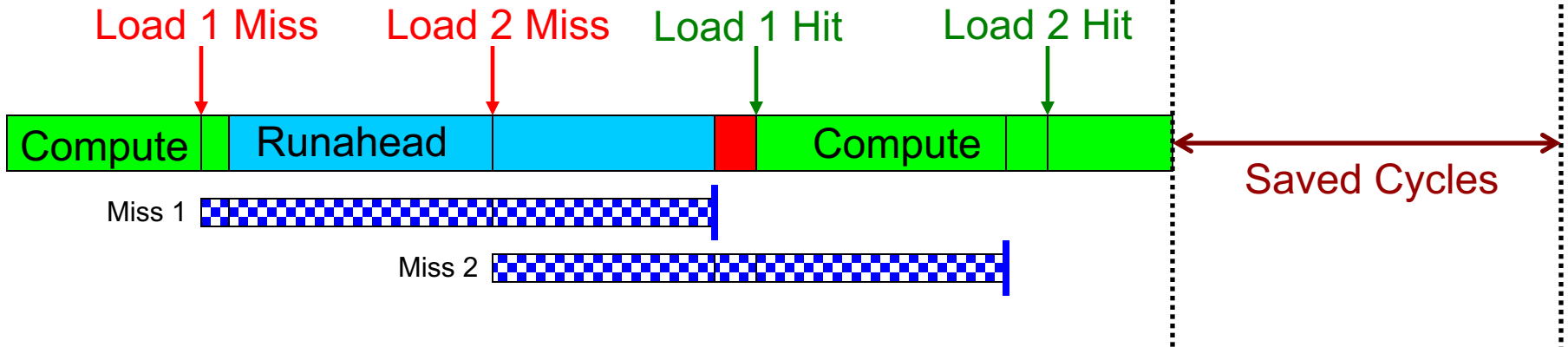
*Perfect Caches:*



*Small Window:*



*Runahead:*



# Benefits of Runahead Execution

---

Instead of stalling during an L2 cache miss:

- Pre-executed loads and stores independent of L2-miss instructions generate **very accurate data prefetches**:
    - For both regular and irregular access patterns
  - **Instructions on the predicted program path are prefetched** into the instruction cache and outer cache levels
  - **Hardware prefetcher and branch predictor tables are trained** using future access information
-

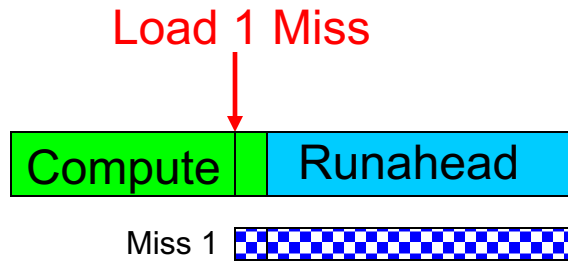
# Runahead Execution Mechanism

---

- Entry into runahead mode
    - Checkpoint architectural register state
  - Instruction processing in runahead mode
  - Exit from runahead mode
    - Restore architectural register state from checkpoint
-

# Instruction Processing in Runahead Mode

---

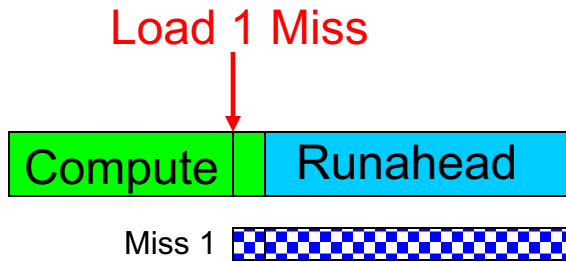


Runahead mode processing is the same as normal instruction processing, EXCEPT:

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

# L2-Miss Dependent Instructions

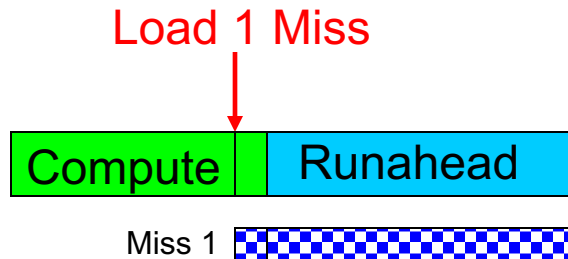
---



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

# Removal of Instructions from Window

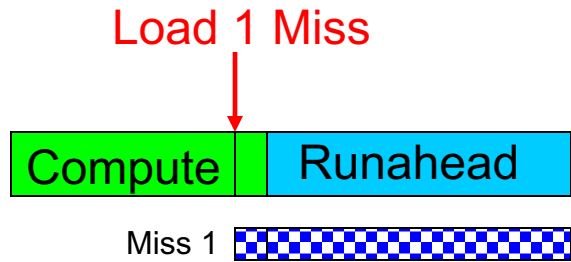
---



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

# Store/Load Handling in Runahead Mode

---

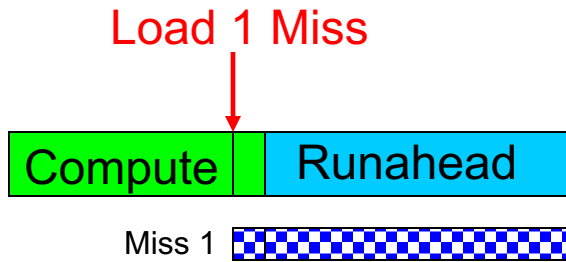


- A pseudo-retired store writes its data and INV status to a dedicated memory, called **runahead cache**.
  - Purpose: Data communication through memory in runahead mode.
  - A dependent load reads its data from the runahead cache.
  - Does not need to be always correct → Size of runahead cache is very small.
-



# Branch Handling in Runahead Mode

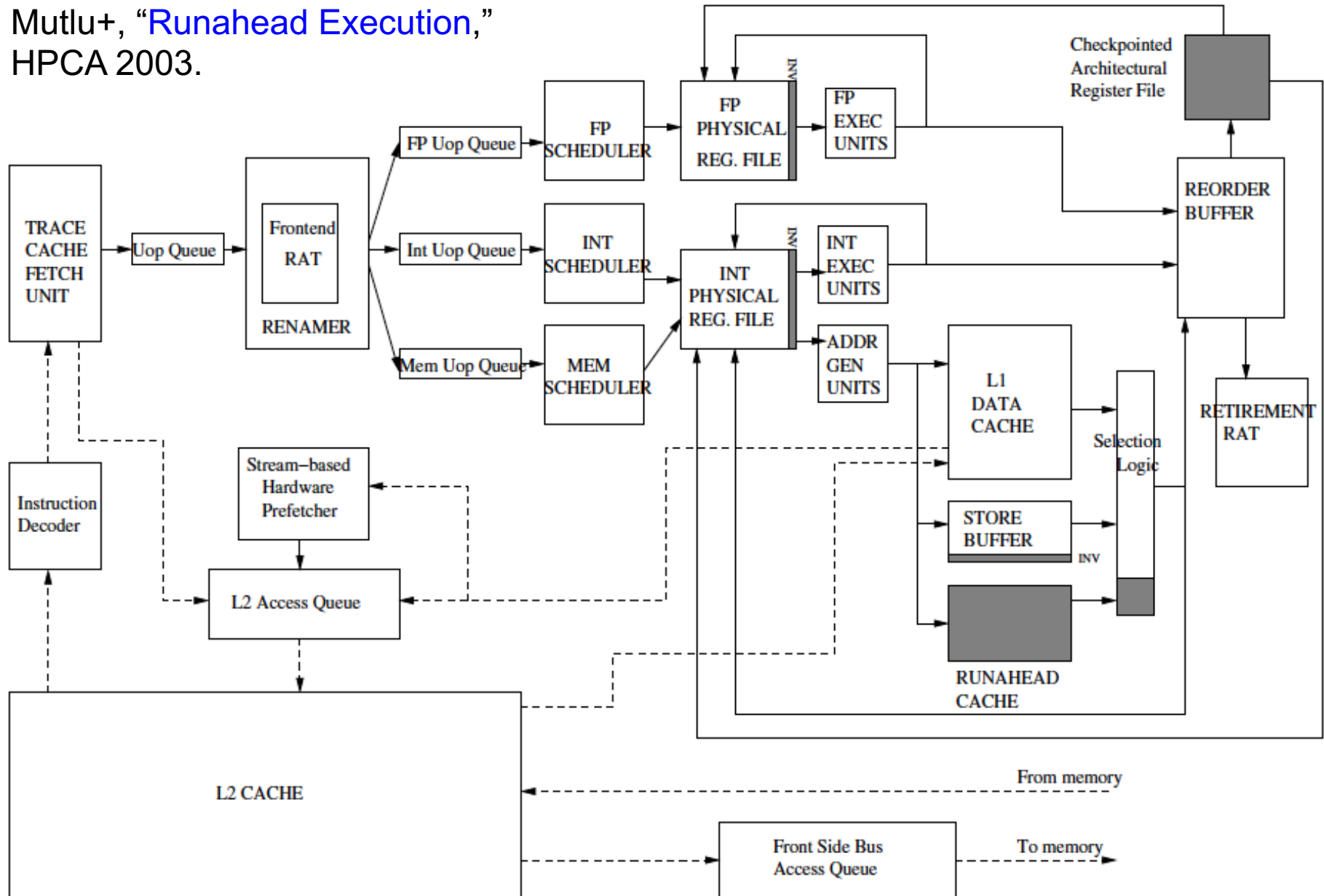
---



- **INV branches cannot be resolved.**
  - A mispredicted INV branch causes the processor to stay on the wrong program path until the end of runahead execution.
- **VALID branches are resolved and initiate recovery if mispredicted.**

# A Runahead Processor Diagram

Mutlu+, “Runahead Execution,”  
HPCA 2003.



# Runahead Execution Pros and Cons

---

## ■ Advantages:

- + Very accurate prefetches for data/instructions (all cache levels)
  - + Follows the program path
- + Simple to implement: most of the hardware is already built in
- + No waste of hardware context: uses the main thread context for prefetching
- + No need to construct a special-purpose pre-execution thread for prefetching

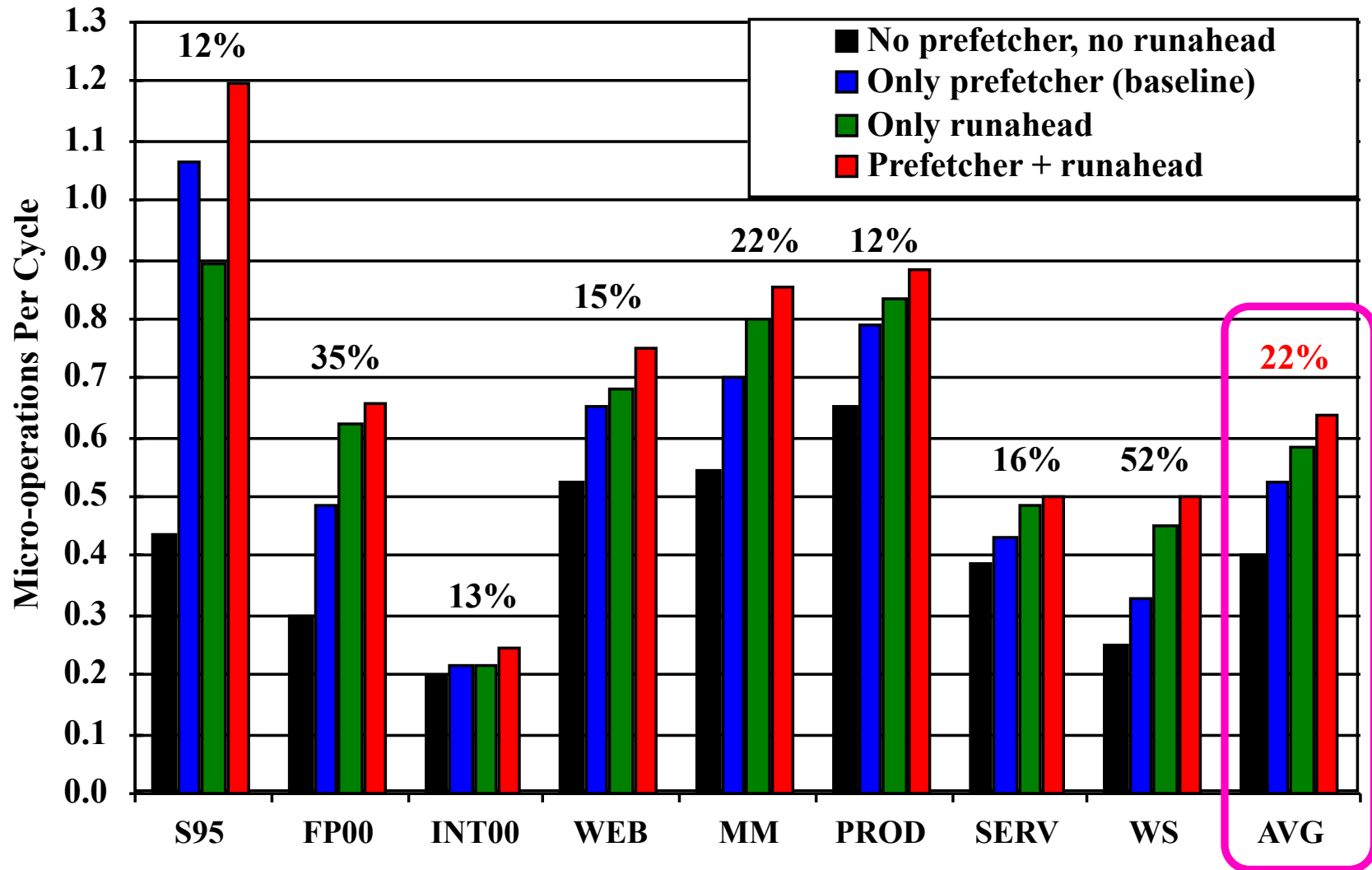
## ■ Disadvantages/Limitations

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

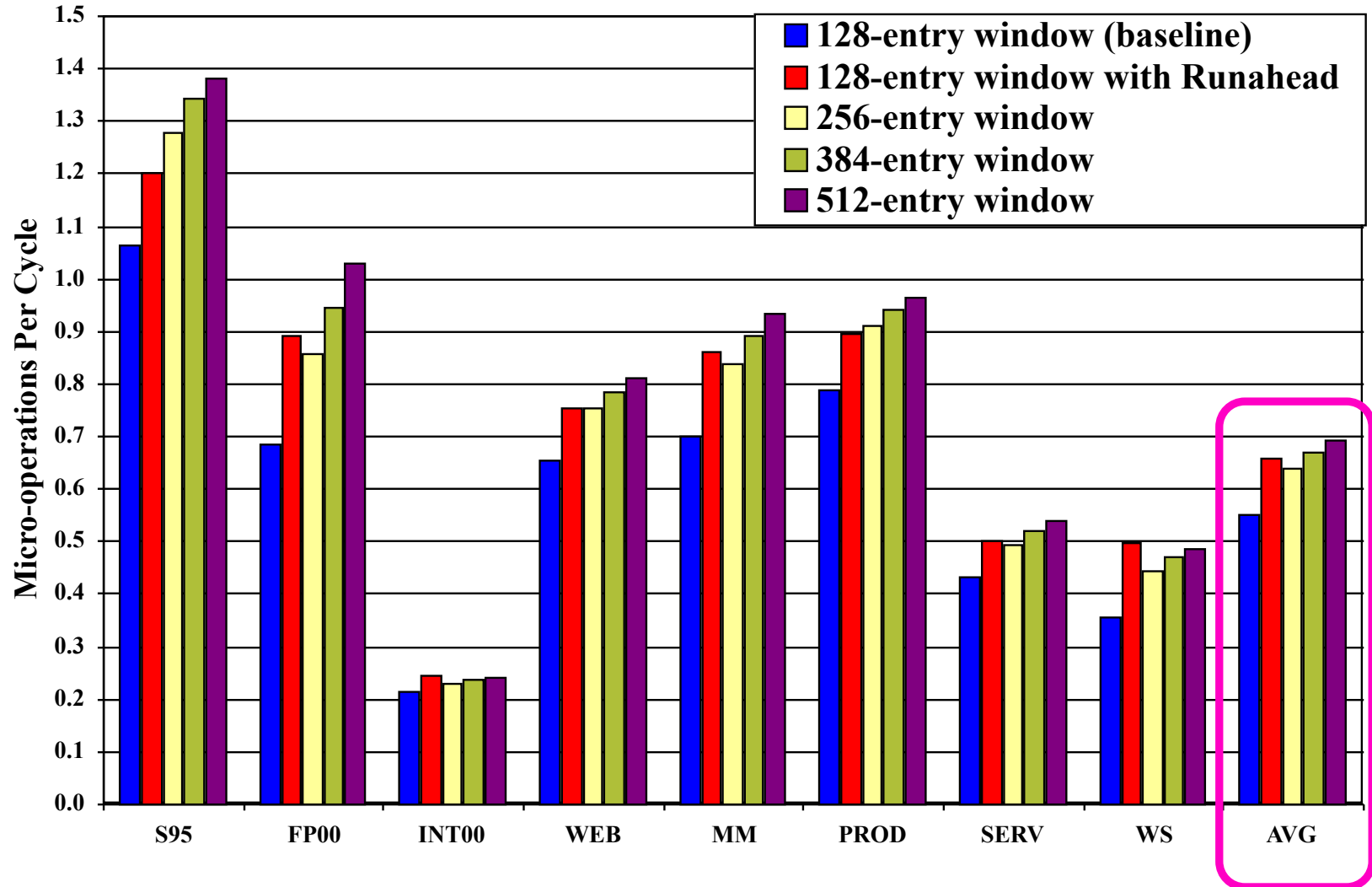
## ■ Implemented in Sun ROCK, IBM POWER6, NVIDIA Denver

---

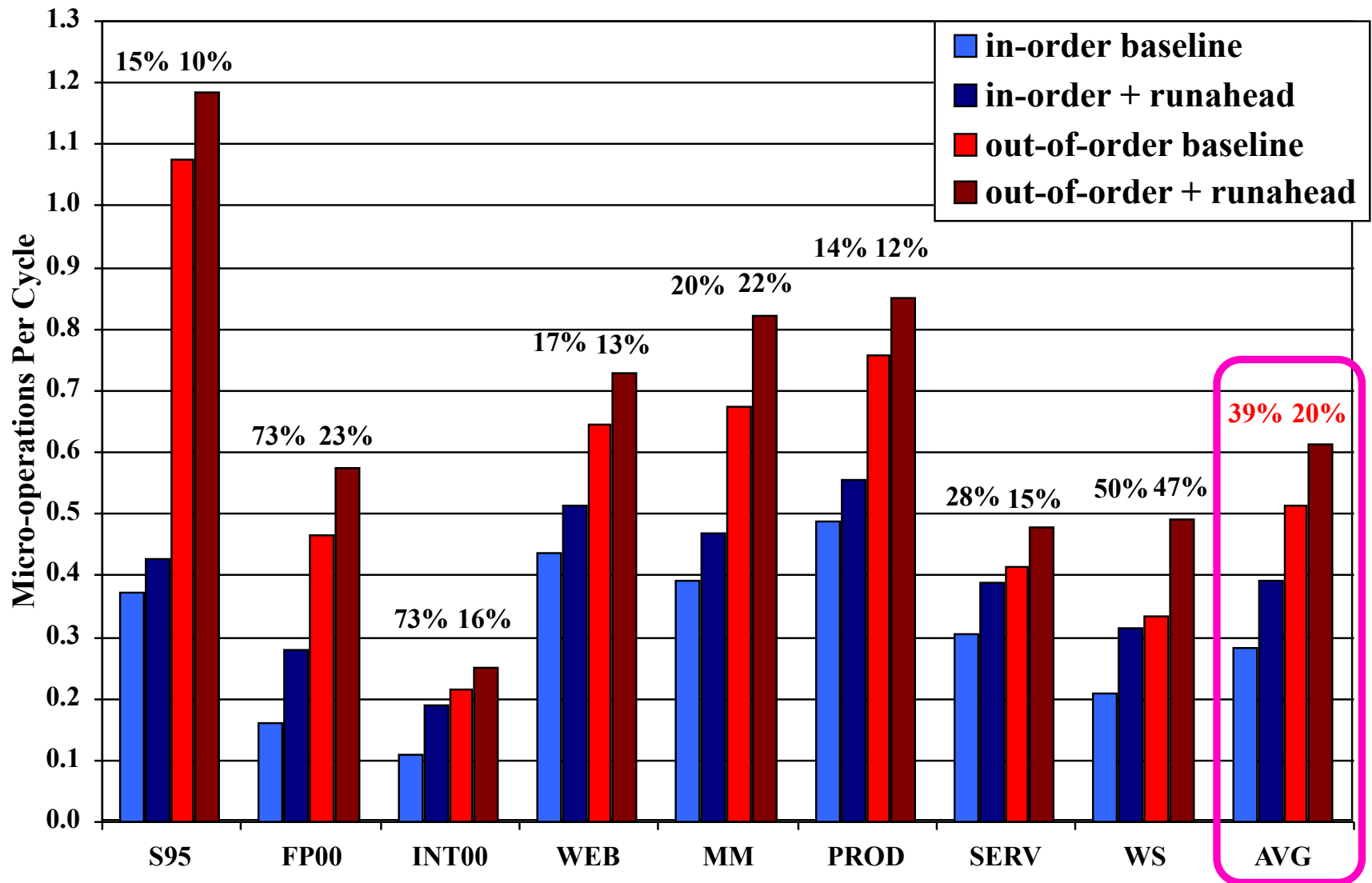
# Performance of Runahead Execution



# Runahead Execution vs. Large Windows



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



# More on Runahead Execution

---

- Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt,  
**"Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors"**

*Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 129-140, Anaheim, CA, February 2003. [Slides \(pdf\)](#)

***One of the 15 computer arch. papers of 2003 selected as Top Picks by IEEE Micro. HPCA Test of Time Award (awarded in 2021).***

[\[Lecture Slides \(pptx\) \(pdf\)\]](#)

[\[Lecture Video \(1 hr 54 mins\)\]](#)

[\[Retrospective HPCA Test of Time Award Talk Slides \(pptx\) \(pdf\)\]](#)

[\[Retrospective HPCA Test of Time Award Talk Video \(14 minutes\)\]](#)

## **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

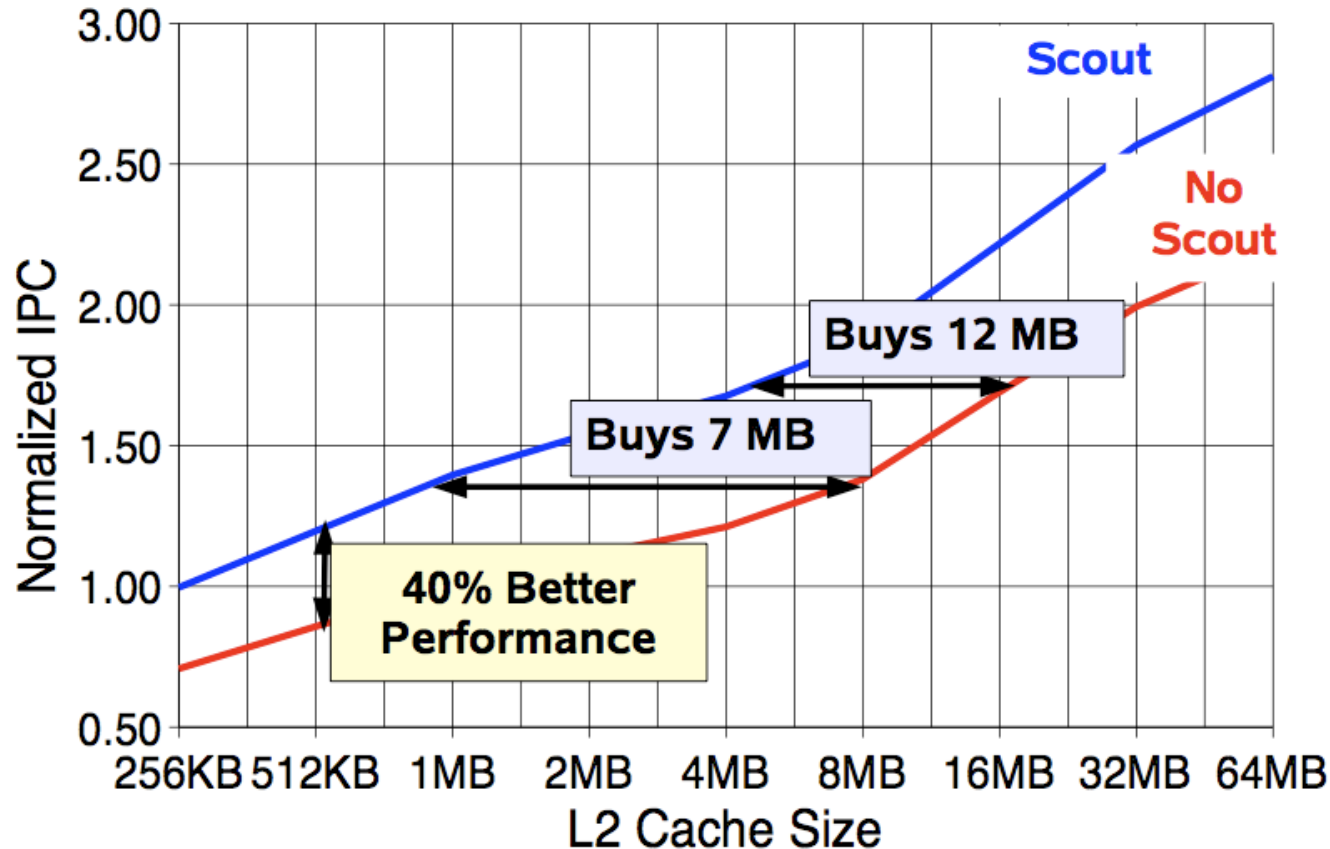
‡Desktop Platforms Group

Intel Corporation

chris.wilkerson@intel.com

# Effect of Runahead in Sun ROCK

- Shailender Chaudhry talk, Aug 2008.



Effective prefetching can both improve performance and reduce hardware cost



# 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.

# 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

# Runahead Execution in IBM POWER6

---

## **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 in 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).*

## DENVER: NVIDIA'S FIRST 64-BIT ARM PROCESSOR

NVIDIA'S FIRST 64-BIT ARM PROCESSOR, CODE-NAMED DENVER, LEVERAGES A HOST OF NEW TECHNOLOGIES, SUCH AS DYNAMIC CODE OPTIMIZATION, TO ENABLE HIGH-PERFORMANCE MOBILE COMPUTING. IMPLEMENTED IN A 28-NM PROCESS, THE DENVER CPU CAN ATTAIN CLOCK SPEEDS OF UP TO 2.5 GHZ. THIS ARTICLE OUTLINES THE DENVER ARCHITECTURE, DESCRIBES ITS TECHNOLOGICAL INNOVATIONS, AND PROVIDES RELEVANT COMPARISONS AGAINST COMPETING MOBILE PROCESSORS.

Boggs+, "[Denver: NVIDIA's First 64-Bit ARM Processor](#)," IEEE Micro 2015.

# Runahead Execution in NVIDIA Denver

Reducing the effects of long cache-miss penalties has been a major focus of the micro-architecture, using techniques like prefetching and run-ahead. An aggressive hardware prefetcher implementation detects L2 cache requests and tracks up to 32 streams, each with complex stride patterns.

Run-ahead uses the idle time that a CPU spends waiting on a long latency operation to discover cache and DTLB misses further down the instruction stream and generates prefetch requests for these misses.<sup>1</sup> These prefetch requests warm up the data cache and DTLB well before the actual execution of

the instructions that require the data. Run-ahead complements the hardware prefetcher because it's better at prefetching nonstrided streams, and it trains the hardware prefetcher faster than normal execution to yield a combined benefit of 13 percent on SPECint2000 and up to 60 percent on SPECfp2000.

The core includes a hardware prefetch unit that Boggs describes as “aggressive” in preloading the data cache but less aggressive in preloading the instruction cache. It also implements a “run-ahead” feature that continues to execute microcode speculatively after a data-cache miss; this execution can trigger additional cache misses that resolve in the shadow of the first miss. Once the data from the original miss returns, the results of this speculative execution are discarded and execution restarts with the bundle containing the original miss, but run-ahead can preload subsequent data into the cache, thus avoiding a string of time-wasting cache misses. These and other features help Denver outscore Cortex-A15 by more than 2.6x on a memory-read test even when both use the same SoC framework (Tegra K1).

Boggs+, “Denver: NVIDIA’s First 64-Bit ARM Processor,” IEEE Micro 2015.

Gwennap, “NVIDIA’s First CPU is a Winner,” MPR 2014.

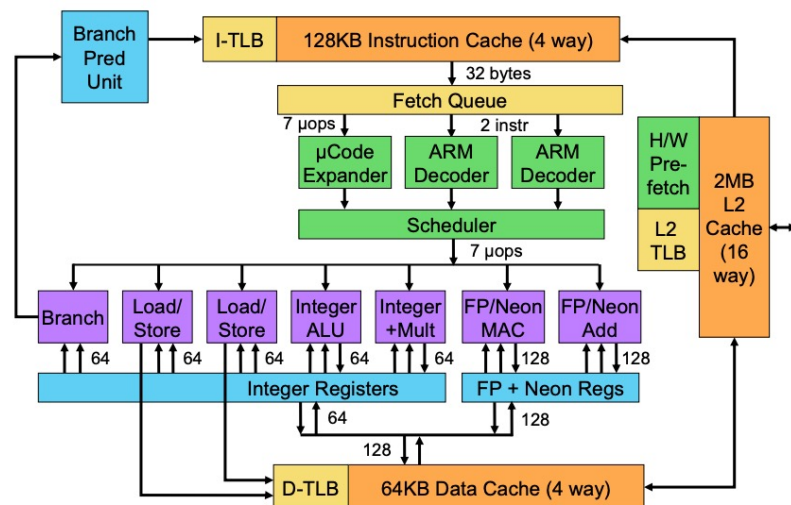


Figure 3. Denver CPU microarchitecture. This design combines a fairly

# Runahead Enhancements

# Runahead Enhancements

---

- Mutlu et al., “Techniques for Efficient Processing in Runahead Execution Engines,” ISCA 2005, IEEE Micro Top Picks 2006.
- Mutlu et al., “Address-Value Delta (AVD) Prediction,” MICRO 2005.
- Armstrong et al., “Wrong Path Events,” MICRO 2004.
- Mutlu et al., “An Analysis of the Performance Impact of Wrong-Path Memory References on Out-of-Order and Runahead Execution Processors,” IEEE TC 2005.



# Limitations of the Baseline Runahead Mechanism

---

## ■ Energy Inefficiency

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

## ■ Ineffectiveness for pointer-intensive applications

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

## ■ Irresolvable branch mispredictions in runahead mode

- ❑ Cannot recover from a mispredicted L2-miss dependent branch
  - ❑ **Wrong Path Events** [MICRO'04]
  - ❑ **Wrong Path Memory Reference Analysis** [IEEE TC'05]
-

# More on Efficient Runahead Execution

---

- Onur Mutlu, Hyesoon Kim, and Yale N. Patt,  
**"Techniques for Efficient Processing in Runahead Execution Engines"**  
*Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, pages 370-381, Madison, WI, June 2005. Slides (ppt) Slides (pdf)  
***One of the 13 computer architecture papers of 2005 selected as Top Picks by IEEE Micro.***

## Techniques for Efficient Processing in Runahead Execution Engines

Onur Mutlu   Hyesoon Kim   Yale N. Patt

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

# More on Efficient Runahead Execution

---

- Onur Mutlu, Hyesoon Kim, and Yale N. Patt,  
["Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance"](#)

*IEEE Micro, Special Issue: Micro's Top Picks from Microarchitecture Conferences (MICRO TOP PICKS)*, Vol. 26, No. 1, pages 10-20, January/February 2006.

## EFFICIENT RUNAHEAD EXECUTION: POWER-EFFICIENT MEMORY LATENCY TOLERANCE

# More Effective Runahead Execution

---

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

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

Onur Mutlu   Hyesoon Kim   Yale N. Patt

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

# More on Efficient Runahead Execution

---

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

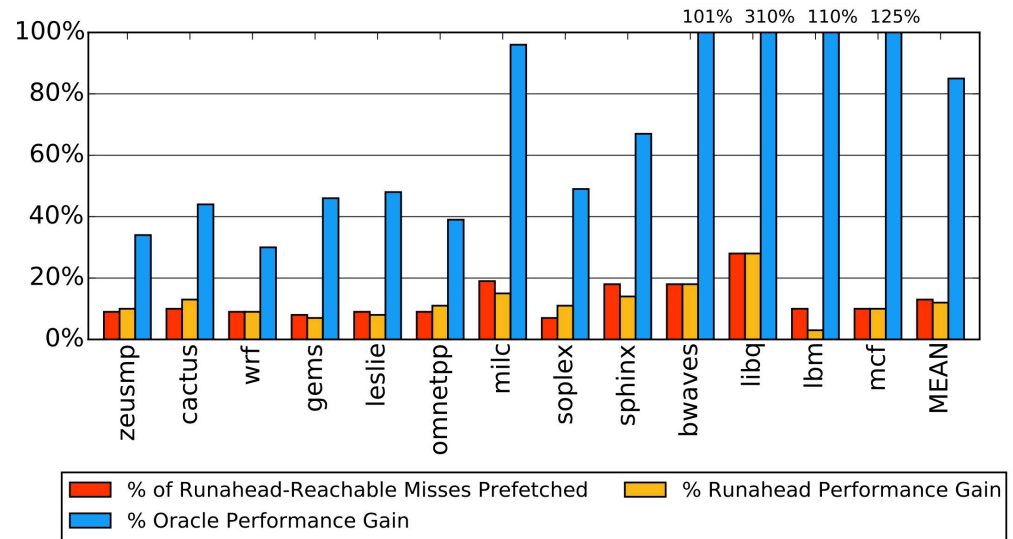
## Address-Value Delta (AVD) Prediction: A Hardware Technique for Efficiently Parallelizing Dependent Cache Misses

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

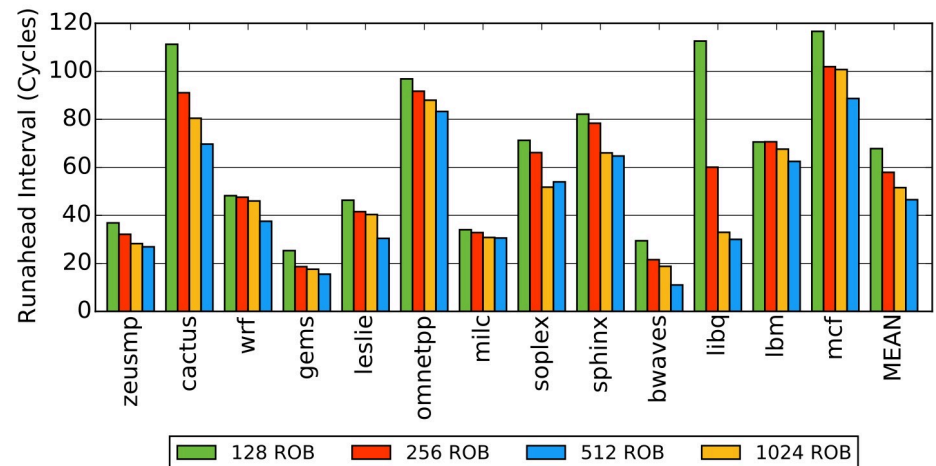
# Continuous Runahead

## ■ Key Observations:

- Runahead is prefetching only **13% of all runahead-reachable misses**



- Why? Because **runahead execution interval is very short (60 cycles on average)**



# Continuous Runahead

---

- **Key Idea:** Remove the limitation for short runahead interval
  - Identify chain of instructions that lead to a critical cache miss
  - Keep executing the chain of instructions in a loop in a special runahead hardware to keep on generating future misses
- **Key Results:**
  - 70% coverage of runahead-reachable misses (up from 13%)
  - 21.9% performance gain over best runahead implementation

# Continuous Runahead

---

- Milad Hashemi, Onur Mutlu, and Yale N. Patt,  
**"Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads"**  
*Proceedings of the 49th International Symposium on Microarchitecture (MICRO)*, Taipei, Taiwan, October 2016.  
[[Slides \(pptx\)](#)] [[pdf](#)] [[Lightning Session Slides \(pdf\)](#)] [[Poster \(pptx\)](#)] [[pdf](#)]  
***Best paper session.***

## Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads

Milad Hashemi\*, Onur Mutlu<sup>§</sup>, Yale N. Patt\*

\*The University of Texas at Austin    <sup>§</sup>ETH Zürich



# Runahead as an Execution-Based Prefetcher

# Runahead as an Execution-based Prefetcher

---

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

# Taking Advantage of Pure Speculation

---

- Runahead mode is purely speculative
- The goal is to find and generate cache misses that would otherwise stall execution later on
- How do we achieve this goal most efficiently and with the highest benefit?
- Idea: Find and execute only those instructions that will lead to cache misses (that cannot already be captured by the instruction window)
- How?

# Execution-based Prefetchers: Pros and Cons

---

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

# Looking to the Past

# At the Time... Early 2000s...

---

- Large focus on increasing the size of the window...
  - And, designing bigger, more complicated machines
- Runahead was a different way of thinking
  - Keep the OoO core simple and small
  - At the expense of some benefits (e.g., non-memory-related)
  - Use aggressive “automatic speculative execution” solely for prefetching
  - Synergistic with prefetching and branch prediction methods
- A lot of interesting and innovative ideas ensued...

# Important Precedent [Dundas & Mudge, ICS 1997]

---

## Improving Data Cache Performance by Pre-executing Instructions Under a Cache Miss

James Dundas and Trevor Mudge

Department of Electrical Engineering and Computer Science

The University of Michigan

Ann Arbor, Michigan 48109-2122

{dundas, tnm}@eecs.umich.edu

### Abstract

In this paper we propose and evaluate a technique that improves first level data cache performance by pre-executing future instructions under a data cache miss. We show that these pre-executed instructions can generate highly accurate data prefetches, particularly when the first level cache is small. The technique is referred to as *runahead* processing. The hardware required to implement runahead is modest, because, when a miss occurs, it makes use of an otherwise idle resource, the execution logic. The principal hardware cost is an extra register file. To measure the impact of runahead, we simulated a processor executing five integer Spec95 benchmarks. Our results show that runahead was able to significantly reduce data cache CPI for four of the five benchmarks. We also compared runahead to a simple form of prefetching, sequential prefetching, which would seem to be suitable for scientific benchmarks. We confirm this by enlarging the scope of our experiments to include a scientific benchmark. However, we show that runahead was also able to outperform sequential prefetching on the scientific benchmark. We also conduct studies that demonstrate that runahead can generate many useful prefetches for lines that show little spatial locality with the misses that initiate runahead episodes. Finally, we discuss some further enhancements of our baseline runahead prefetching scheme.

are allocated by the software. This hybrid hardware-software technique was presented in [8]. Their instruction stride table (IST) selectively generates cache miss initiated prefetches for accesses chosen beforehand by the compiler. This resulted in multiprocessor performance for scientific benchmarks comparable in some cases to software prefetching, with an instruction stride table as small as 4 entries. The IST concept was subsequently combined with the prefetch predicates of [2] in [9]. Another hardware prefetching scheme that avoids the need for significant amounts of hardware is the “wrong path” prefetching described in [10]. This actually prefetches instructions from the not-taken path, in the expectation that they will be executed during a later iteration.

Most prefetching techniques, software- or hardware-based, tend to perform poorly on an important class of applications having recursive data structures such as linked-lists. A software technique that overcomes this limitation was presented recently in [11], in which software prefetches were inserted at subroutine call sites that passed pointers as arguments. Another pointer-based approach was described in [12]. This approach uses pointers stored within the data structures to generate software prefetches.

The runahead prefetching approach presented in this paper is a hardware approach, that requires only a modest amount of hardware, because, when a miss occurs, it makes use of an otherwise

# An Inspiration [Glew, ASPLOS-WACI 1998]

## MLP yes! ILP no!

*Memory Level Parallelism, or why I no longer care about Instruction Level Parallelism*

Andrew Glew

Intel Microcomputer Research Labs and University of Wisconsin, Madison

**Problem Description:** It should be well known that processors are outstripping memory performance: specifically that memory latencies are not improving as fast as processor cycle time or IPC or memory bandwidth.

Thought experiment: imagine that a cache miss takes 10000 cycles to execute. For such a processor instruction level parallelism is useless, because most of the time is spent waiting for memory. Branch prediction is also less effective, since most branches can be determined with data already in registers or in the cache; branch prediction only helps for branches which depend on outstanding cache misses.

At the same time, pressures for reduced power consumption mount.

Given such trends, some computer architects in industry (although not Intel EPIC) are talking seriously about retreating from out-of-order superscalar processor architecture, and instead building simpler, faster, dumber, 1-wide in-order processors with high degrees of speculation. Sometimes this is proposed in combination with multiprocessing and multithreading: tolerate long memory latencies by switching to other processes or threads.

I propose something different: build narrow fast machines but use intelligent logic inside the CPU to increase the number of outstanding cache misses that can be generated from a single program.

**Solution:** First, change the mindset: MLP, Memory Level Parallelism, is what matters, not ILP, Instruction Level Parallelism.

By MLP I mean simply the number of outstanding cache misses that can be generated (by a single thread, task, or program) and executed in an overlapped manner. It does not matter what sort of execution engine generates the multiple outstanding cache misses. An out-of-order superscalar ILP CPU may generate multiple outstanding cache misses, but 1-wide processors can be just as effective.

Change the metrics: total execution time remains the overall goal, but instead of reporting IPC as an approximation to this, we must report MLP. Limit studies should be in terms of total number of non-overlapped cache misses on critical path.

Now do the research: Many present-day hot topics in computer architecture help ILP, but do not help MLP. As mentioned above, predicting branch directions for branches that can be determined from data already in the cache or in registers does not help MLP for extremely long latencies. Similarly, prefetching of data cache misses for array processing codes does not help MLP – it just

Instead, investigate microarchitectures that help MLP:

- (0) Trivial case – explicit multithreading, like SMT.
- (1) Slightly less trivial case – implicitly multithread single programs, either by compiler software on an MT machine, or by a hybrid, such as Wisconsin Multiscalar, or entirely in hardware, as in Intel's Dynamic Multi-Threading.
- (2) Build 1-wide processors that are as fast as possible: use circuit tricks, as well as logic tricks such as redundant encoding for numeric computation and memory addressing.
- (3) Allow the hardware dynamic scheduling mechanisms to use sequential algorithms implemented by this narrow, fast, processor, rather than limiting it to parallel algorithms implementable in associative logic.
- (4) Build very large instruction windows allowing speculation tens of thousands of instructions ahead. Avoid circuit speed issues by caching the instruction window. Remove small arbitrary limits on the number of cache misses outstanding allowed.
- (5) Further reduce the cost of very large instruction windows by throwing away anything that can be recomputed based on data in registers or cache.
- (6) Don't stall speculation because the oldest instruction in the machine is a cache miss. Let the front of the machine continue executing branches, forgetting data dependent on cache misses.
- (7) Parallelize linked data structure traversals by building skip lists in hardware – converting sequential data structures into parallel ones. Store these extra skip pointers in main memory.

Call such a processor microarchitecture a “super-non-blocking” microarchitecture.

**Justification:** The processor/memory trend is well known. Theoretically optimal cache studies show only limited headroom. Barring a revolution in memory technology, the Memory Wall is real, and getting closer. Multithreading and multiprocessing have some hope of tolerating memory latency, but only if there are parallel workloads. If single thread performance is still an issue, the only potentially MLP enhancing technologies are what I describe here, or data value prediction – and data value prediction seems to only do well for stuff that fits in the cache.

“Super-non-blocking” processors extends dynamic, out-of-order, execution to maximize MLP, but simplifies it by discarding superscalar ILP as unnecessary.



# Looking to the Future

# A Look into the Future...

---

- Microarchitecture (especially memory) is critically important
  - And, fun...
  - And, impactful...
- Runahead is a great example of harmonious industry-academia collaboration
- Fundamental problems will remain fundamental
  - And will require fundamental (and creative) solutions

# Citation for the Test of Time Award

---

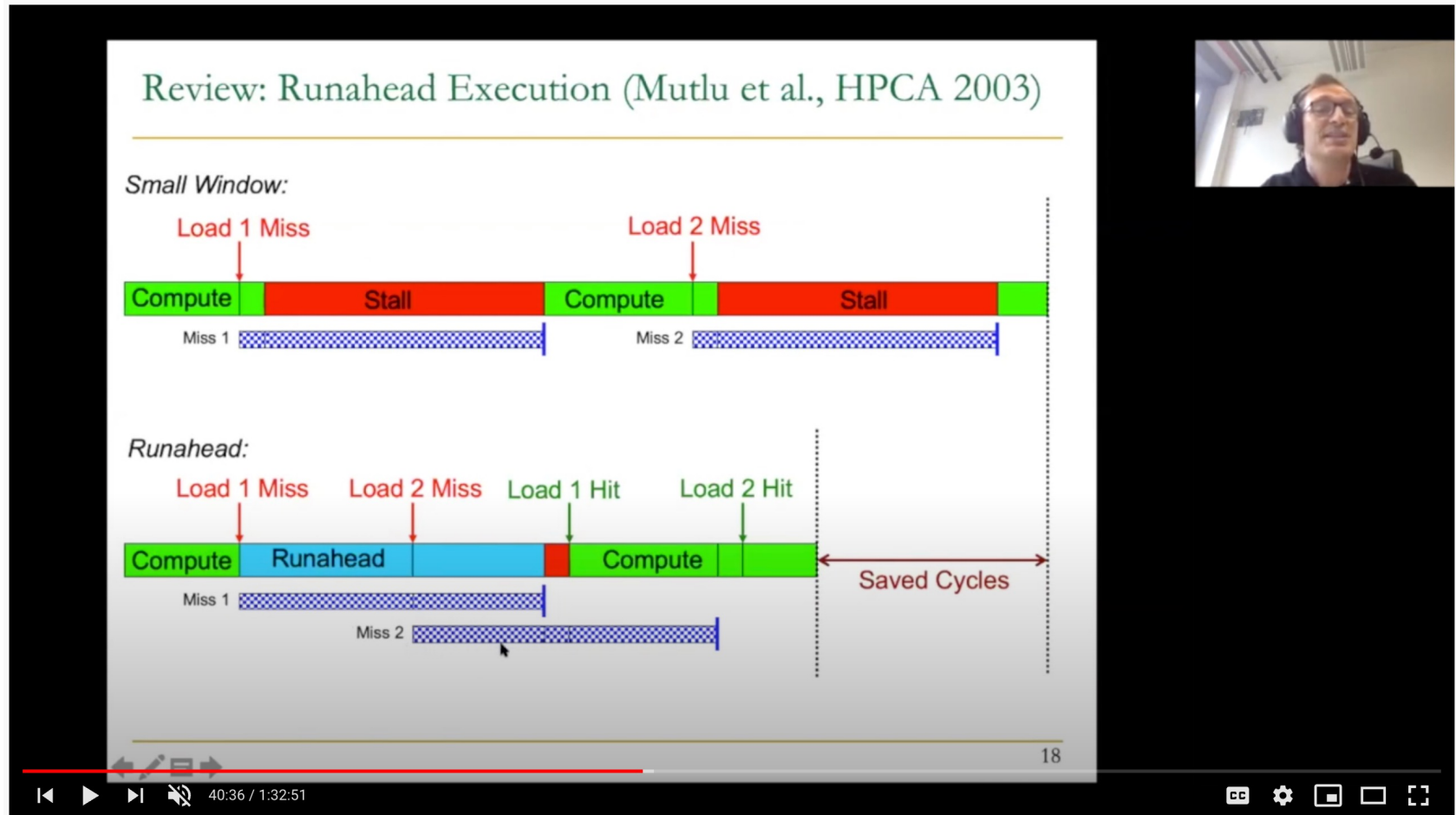
- Runahead Execution is a pioneering paper that opened up new avenues in dynamic prefetching.
- The basic idea of runahead execution effectively increases the instruction window very significantly, without having to increase physical resource size (e.g. the issue queue).
- This seminal paper spawned off a new area of ILP-enhancing microarchitecture research.
- This work has had strong industry impact as evidenced by IBM's POWER6 - Load Lookahead, NVIDIA Denver, and Sun ROCK's hardware scouting.

# More on Runahead Execution

---

- Lecture video from Fall 2020, Computer Architecture:
  - [https://www.youtube.com/watch?v=zPewo6IaJ\\_8](https://www.youtube.com/watch?v=zPewo6IaJ_8)
- Lecture video from Fall 2017, Computer Architecture:
  - <https://www.youtube.com/watch?v=Kj3relihGF4>
- Onur Mutlu,  
**"Efficient Runahead Execution Processors"**  
Ph.D. Dissertation, HPS Technical Report, TR-HPS-2006-007, July 2006. [Slides \(ppt\)](#)  
***Nominated for the ACM Doctoral Dissertation Award by the University of Texas at Austin.***

# More on Runahead Execution (I)



Computer Architecture - Lecture 19a: Execution-Based Prefetching (ETH Zürich, Fall 2020)

395 views • Nov 29, 2020

14 0 SHARE SAVE ...



Onur Mutlu Lectures  
16.5K subscribers

ANALYTICS

EDIT VIDEO

# More on Runahead Execution (II)

## Runahead Execution in NVIDIA Denver

Reducing the effects of long cache-miss penalties has been a major focus of the micro-architecture, using techniques like prefetching and run-ahead. An aggressive hardware prefetcher implementation detects L2 cache requests and tracks up to 32 streams, each with complex stride patterns.

Run-ahead uses the idle time that a CPU spends waiting on a long latency operation to discover cache and DTLB misses further down the instruction stream and generates prefetch requests for these misses.<sup>1</sup> These prefetch requests warm up the data cache and DTLB well before the actual execution of the instructions that require the data. Run-ahead complements the hardware prefetcher because it's better at prefetching nonstrided streams, and it trains the hardware prefetcher faster than normal execution to yield a combined benefit of 13 percent on SPECint2000 and up to 60 percent on SPECfp2000.

The core includes a hardware prefetch unit that Boggs describes as "aggressive" in preloading the data cache but less aggressive in preloading the instruction cache. It also implements a "run-ahead" feature that continues to execute microcode speculatively after a data-cache miss; this execution can trigger additional cache misses that resolve in the shadow of the first miss. Once the data from the original miss returns, the results of this speculative execution are discarded and execution restarts with the bundle containing the original miss, but run-ahead can preload subsequent data into the cache, thus avoiding a string of time-wasting cache misses. These and other features help Denver outscore Cortex-A15 by more than 2.6x on a memory-read test even when both use the same SoC framework (Tegra K1).

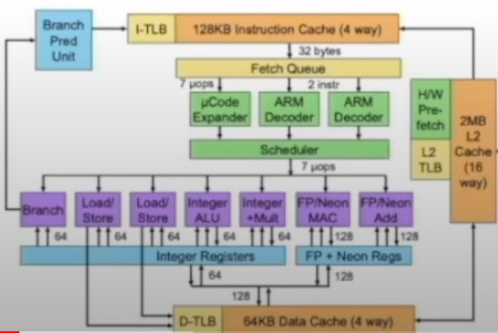


Figure 3. Denver CPU microarchitecture. This design combines a fairly

Boggs+, "Denver: NVIDIA's First 64-Bit ARM Processor," IEEE Micro 2015.

Gwennap, "NVIDIA's First CPU is a Winner," MPR 2014.

Onur Mutlu - Runahead Execution: A Short Retrospective (HPCA Test of Time Award Talk @ HPCA 2021)

1,162 views • Premiered Mar 6, 2021



Onur Mutlu Lectures  
16.5K subscribers

50 0 SHARE SAVE ...

ANALYTICS

EDIT VIDEO

# More Recommended Material on Prefetching

# Lectures on Prefetching (I)

## X86 PREFETCH Instruction

### PREFETCHh—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

Computer Architecture - Lecture 18: Prefetching (ETH Zürich, Fall 2020)

1,203 views • Nov 29, 2020

👍 26 🗨️ 0 ➦ SHARE ⚙️ SAVE ...



Onur Mutlu Lectures  
16.5K subscribers

ANALYTICS

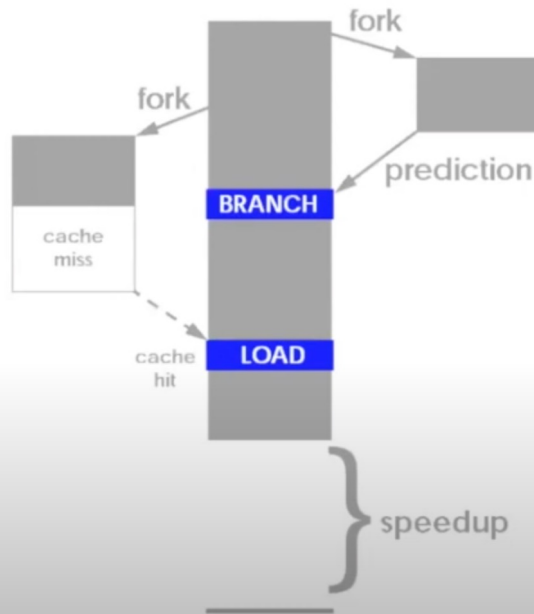
EDIT VIDEO

<https://www.youtube.com/watch?v=xZmDyj0g3Pw&list=PL5Q2soXY2Zi9xidylgBxUz7xRPS-wisBN&index=33>



# Lectures on Prefetching (II)

## 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.



7

Computer Architecture - Lecture 19a: Execution-Based Prefetching (ETH Zürich, Fall 2020)

424 views • Nov 29, 2020

16 0 SHARE SAVE ...



Onur Mutlu Lectures  
16.7K subscribers

ANALYTICS

EDIT VIDEO

# Lectures on Prefetching (III)

## Runahead Execution in NVIDIA Denver

Reducing the effects of long cache-miss penalties has been a major focus of the microarchitecture, using techniques like prefetching and run-ahead. An aggressive hardware prefetcher implementation detects L2 cache requests and tracks up to 32 streams, each with complex stride patterns.

Run-ahead uses the idle time that a CPU spends waiting on a long latency operation to discover cache and DTLB misses further down the instruction stream and generates prefetch requests for these misses.<sup>1</sup> These prefetch requests warm up the data cache and DTLB well before the actual execution of the instructions that require the data. Run-ahead complements the hardware prefetcher because it's better at prefetching nonstrided streams, and it trains the hardware prefetcher faster than normal execution to yield a combined benefit of 13 percent on SPECint2000 and up to 60 percent on SPECfp2000.

Boggs+, "Denver: NVIDIA's First 64-Bit ARM Processor," IEEE Micro 2015.

Gwennap, "NVIDIA's First CPU is a Winner," MPR 2014.

The core includes a hardware prefetch unit that Boggs describes as “aggressive” in preloading the data cache but less aggressive in preloading the instruction cache. It also implements a “run-ahead” feature that continues to execute microcode speculatively after a data-cache miss; this execution can trigger additional cache misses that resolve in the shadow of the first miss. Once the data from the original miss returns, the results of this speculative execution are discarded and execution restarts with the bundle containing the original miss, but run-ahead can preload subsequent data into the cache, thus avoiding a string of time-wasting cache misses. These and other features help Denver outscore Cortex-A15 by more than 2.6x on a memory-read test even when both use the same SoC framework (Tegra K1).

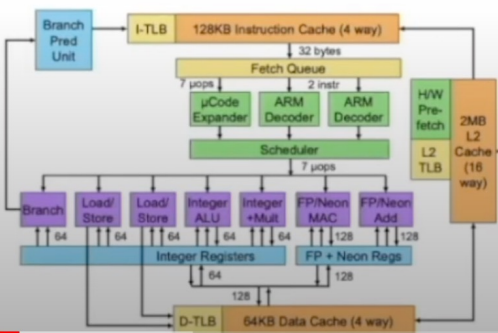
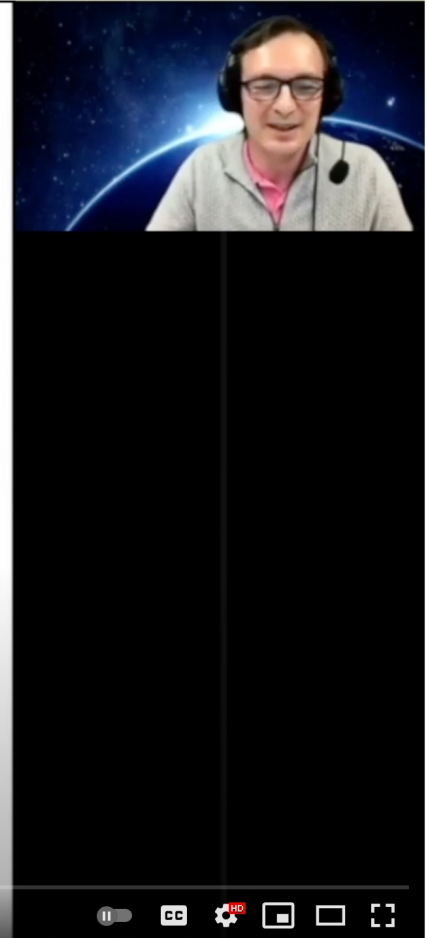


Figure 3. Denver CPU microarchitecture. This design combines a fairly



Onur Mutlu - Runahead Execution: A Short Retrospective (HPCA Test of Time Award Talk @ HPCA 2021)

1,162 views • Premiered Mar 6, 2021



**Onur Mutlu Lectures**  
16.5K subscribers

👍 50 🗨️ 0 ➡️ SHARE ≡+ SAVE ...

ANALYTICS

[EDIT VIDEO](#)

# Lectures on Prefetching (IV)

## 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?
    - 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

40

Lecture 25: Prefetching - Carnegie Mellon - Computer Architecture 2015 - Onur Mutlu

5,216 views • Apr 3, 2015

39 0 SHARE SAVE ...



Carnegie Mellon Computer Architecture  
23.3K subscribers

SUBSCRIBED



<https://www.youtube.com/watch?v=ibPL7T9iEwY&list=PL5PHm2jkkXmi5Cxxl7b3JCL1TWybTDtKq&index=29>

# Lectures on Prefetching (V)

**Address Correlation Based Prefetching (II)**

The diagram illustrates the Address Correlation Based Prefetching (II) mechanism. It shows a 'Cache Block Addr' pointing to a table with 'Cache Block Addr (tag)' and a list of addresses. To the right, there are two tables: 'Prefetch Candidate 1' and 'Confidence', and another 'Prefetch Candidate N' and 'Confidence' table. The 'Prefetch Candidate 1' table has a 'Candidate 1' row and a 'Confidence' row. The 'Prefetch Candidate N' table has a 'Candidate N' row and a 'Confidence' row. The 'Confidence' row in the 'Prefetch Candidate N' table contains a list of addresses.

- 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.

10

Lecture 26. More Prefetching and Emerging Memory Technologies - CMU - Comp. Arch. 2015 - Onur Mutlu

3,642 views • Apr 6, 2015

26 0 SHARE SAVE ...



Carnegie Mellon Computer Architecture  
23.3K subscribers

SUBSCRIBED





# Lectures on Prefetching

---

## ■ Computer Architecture, Fall 2020, Lecture 18

- ❑ Prefetching (ETH, Fall 2020)
- ❑ <https://www.youtube.com/watch?v=xZmDyj0g3Pw&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=33>

## ■ Computer Architecture, Fall 2020, Lecture 19a

- ❑ Execution-Based Prefetching (ETH, Fall 2020)
- ❑ [https://www.youtube.com/watch?v=zPewo6IaJ\\_8&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=34](https://www.youtube.com/watch?v=zPewo6IaJ_8&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=34)

## ■ Computer Architecture, Spring 2015, Lecture 25

- ❑ Prefetching (CMU, Spring 2015)
- ❑ <https://www.youtube.com/watch?v=ibPL7T9iEwY&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=29>

## ■ Computer Architecture, Spring 2015, Lecture 26

- ❑ More Prefetching (CMU, Spring 2015)
- ❑ <https://www.youtube.com/watch?v=TUFins4z6o4&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=30>

# Computer Architecture

## Lecture 16: Prefetching

Prof. Onur Mutlu

ETH Zürich

Fall 2022

18 November 2022

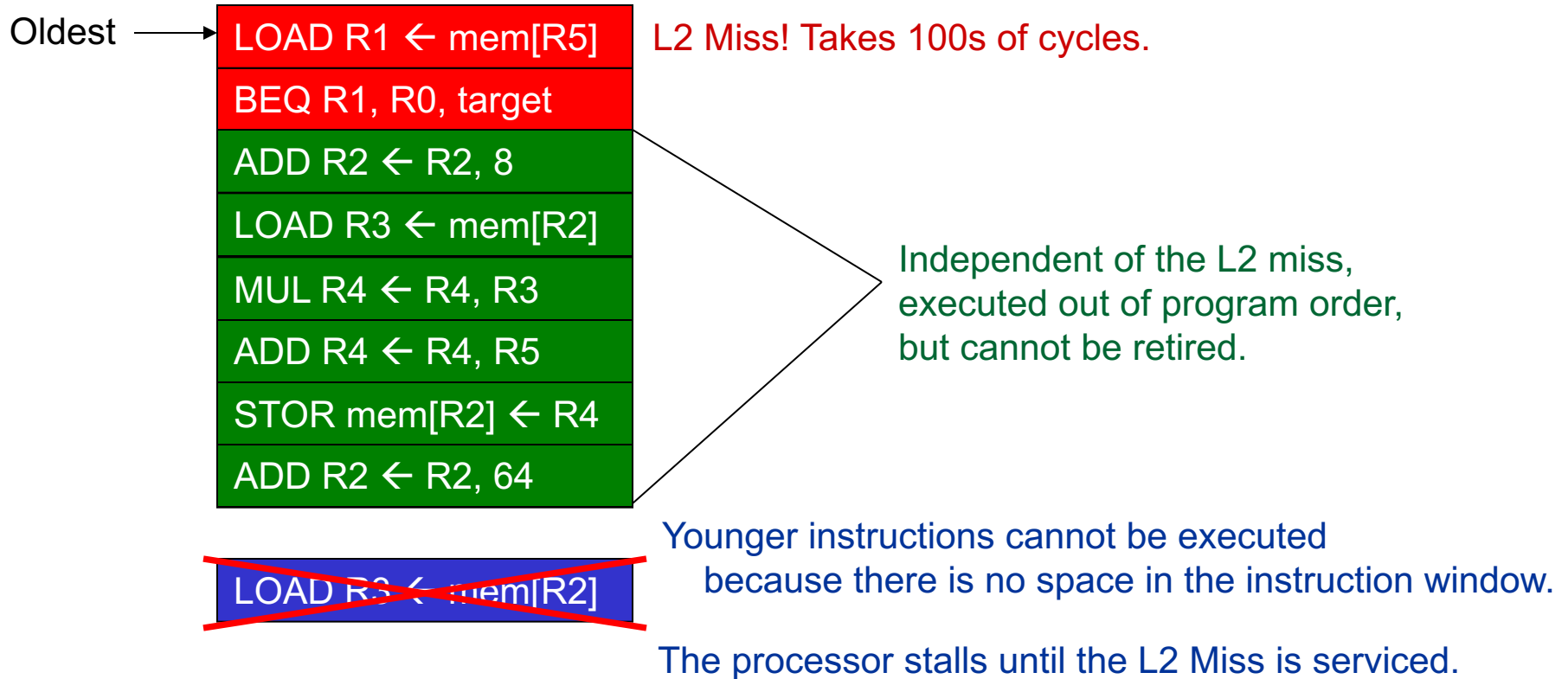
# Backup Slides

# Backup: Runahead Execution



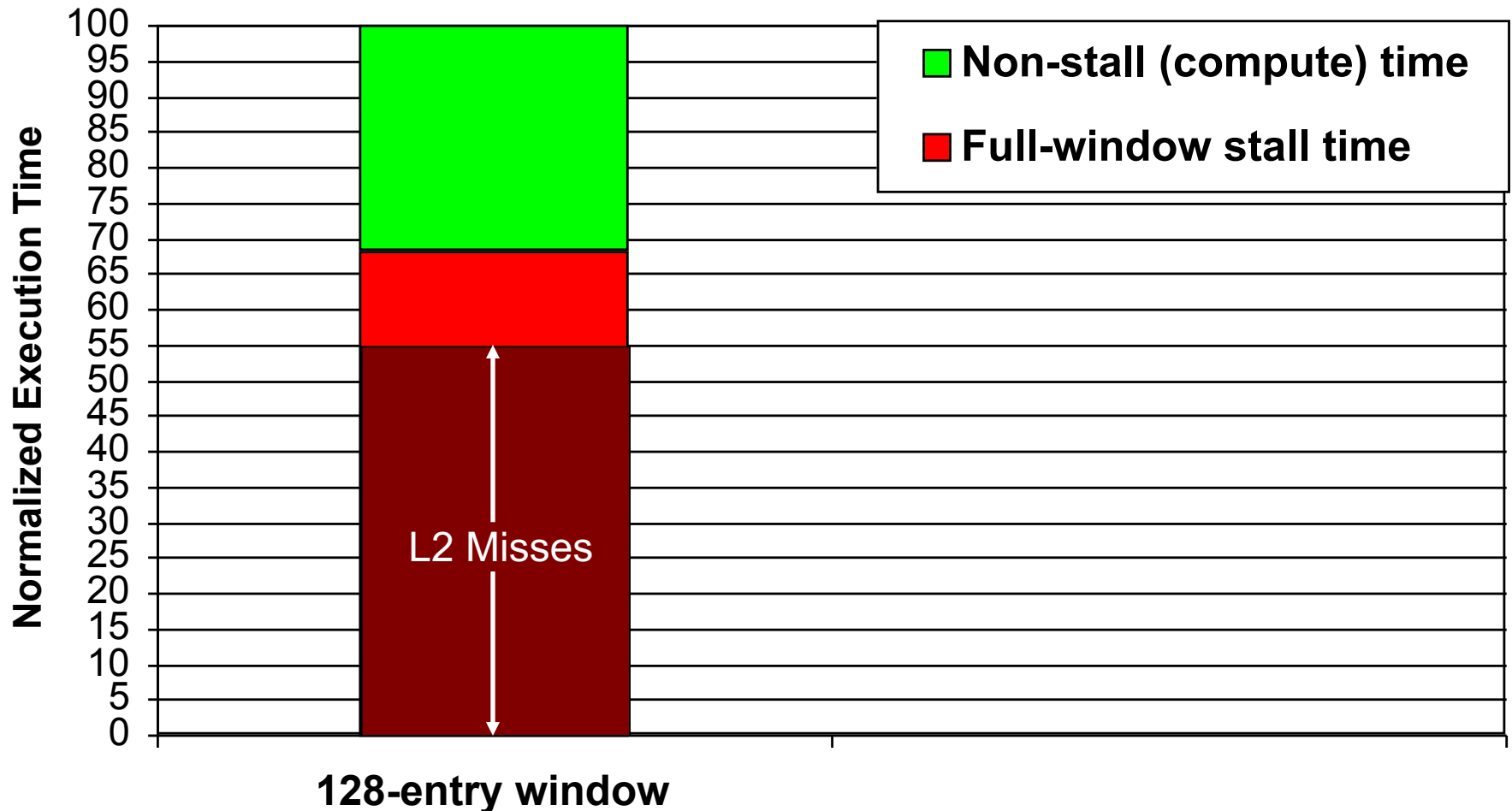
# Small Windows: Full-Window Stalls

8-entry instruction window:



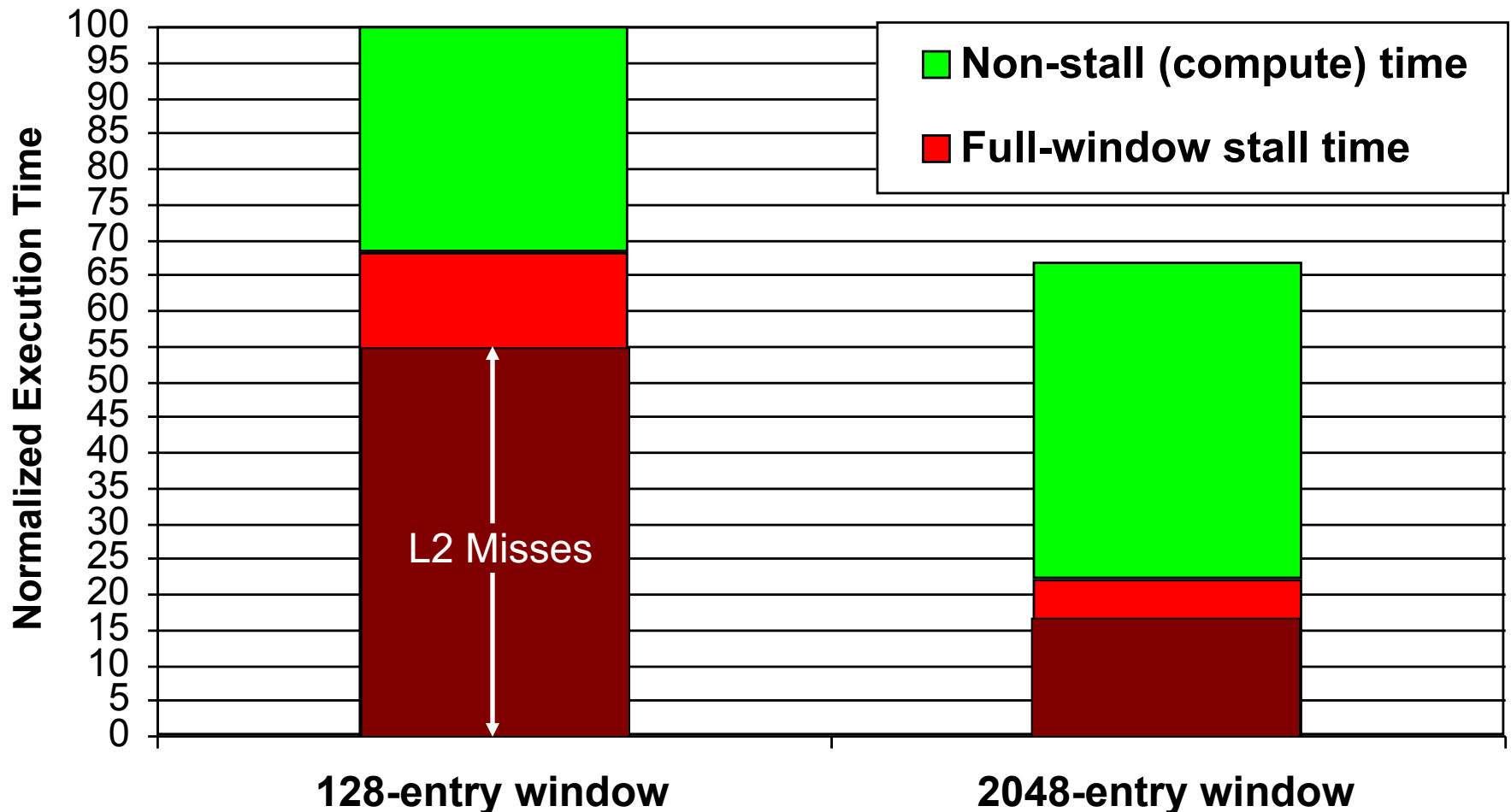
- Long-latency cache misses are responsible for most full-window stalls

# Impact of Long-Latency Cache Misses



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

# Impact of Long-Latency Cache Misses



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

# The Problem

---

- Out-of-order execution requires large instruction windows to tolerate today's main memory latencies
- As main memory latency increases, instruction window size should also increase to fully tolerate the memory latency
- Building a large instruction window is a challenging task if we would like to achieve
  - ❑ Low power/energy consumption (tag matching logic, load/store buffers)
  - ❑ Short cycle time (wakeup/select, regfile, bypass latencies)
  - ❑ Low design and verification complexity

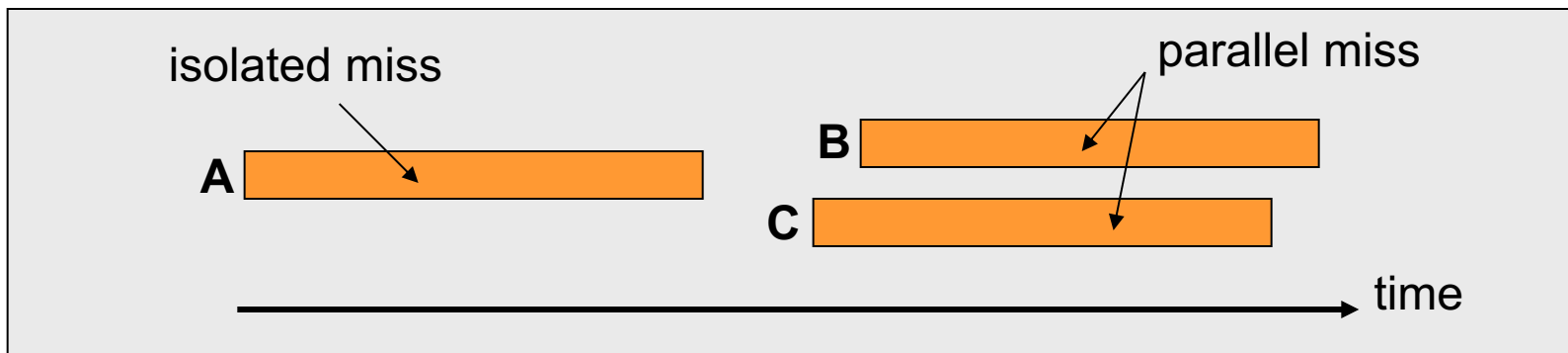
# Efficient Scaling of Instruction Window Size

---

- One of the major research issues in out of order execution
- How to achieve the benefits of a large window with a small one (or in a simpler way)?
- How do we efficiently tolerate memory latency with the machinery of out-of-order execution (and a small instruction window)?

# Memory Level Parallelism (MLP)

- Idea: Find and service multiple cache misses in parallel so that the processor stalls only once for all misses



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

# Runahead Execution

---

- A technique to obtain the memory-level parallelism benefits of a large instruction window
- When the oldest instruction is a long-latency cache miss:
  - Checkpoint architectural state and enter runahead mode
- In runahead mode:
  - Speculatively pre-execute instructions
  - The purpose of pre-execution is to generate prefetches
  - L2-miss dependent instructions are marked INV and dropped
- Runahead mode ends when the original miss returns
  - Checkpoint is restored and normal execution resumes
- Mutlu et al., “Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors,” HPCA 2003.

# Runahead vs. A (Real) Large Window

---

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



# Generalizing the Idea

---

- Runahead on different long-latency operations?

# Backup: Runahead Enhancements

# Readings

---

## ■ Required

- ❑ Mutlu et al., “Runahead Execution”, HPCA 2003, Top Picks 2003.

## ■ Recommended

- ❑ Mutlu et al., “Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance,” ISCA 2005, IEEE Micro Top Picks 2006.
- ❑ Mutlu et al., “Address-Value Delta (AVD) Prediction,” MICRO 2005.
- ❑ Armstrong et al., “Wrong Path Events,” MICRO 2004.

# Limitations of the Baseline Runahead Mechanism

---

## ■ Energy Inefficiency

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

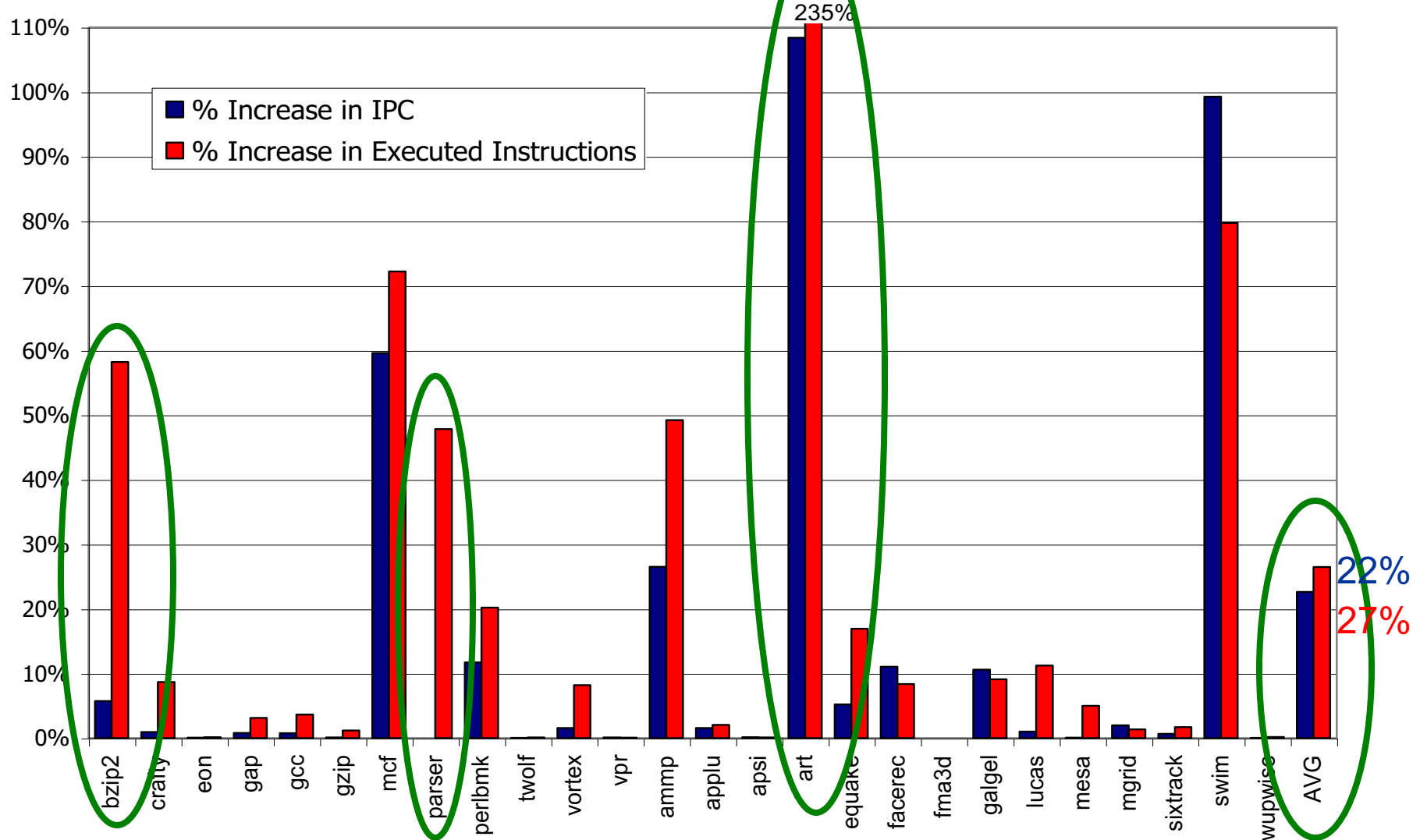
## ■ Ineffectiveness for pointer-intensive applications

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

## ■ Irresolvable branch mispredictions in runahead mode

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

# The Efficiency Problem



# Causes of Inefficiency

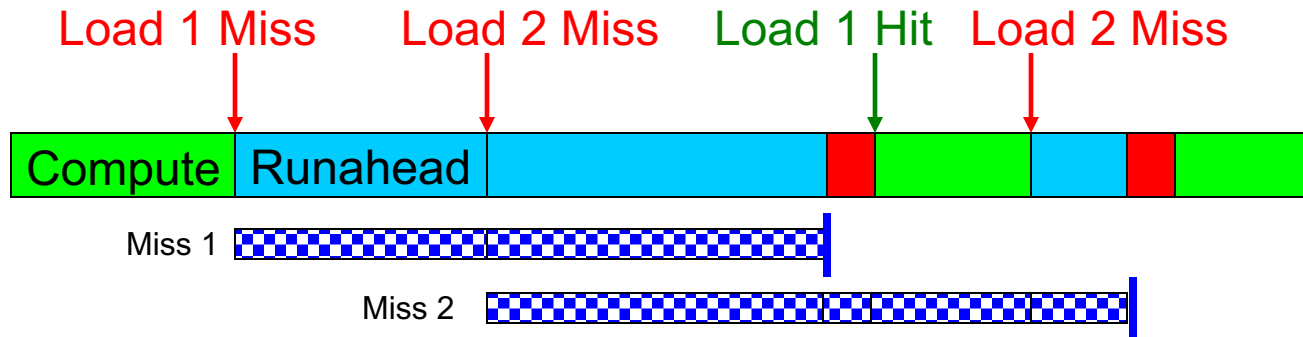
---

- Short runahead periods
  - Overlapping runahead periods
  - Useless runahead periods
  - Mutlu et al., “Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance,” ISCA 2005, IEEE Micro Top Picks 2006.
-

# Short Runahead Periods

---

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



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

# Overlapping Runahead Periods

---

- Two runahead periods that execute the same instructions



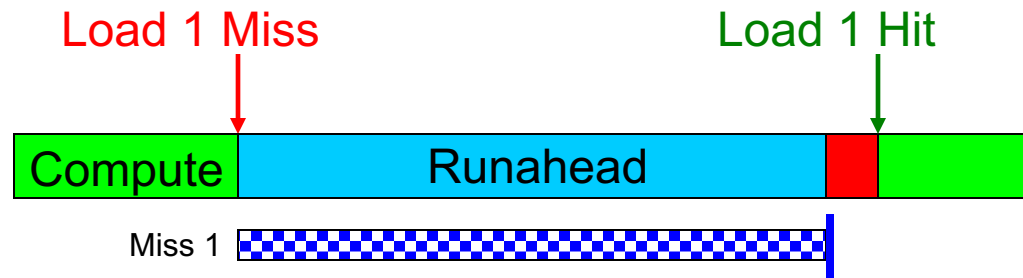
- Second period is inefficient
-



# Useless Runahead Periods

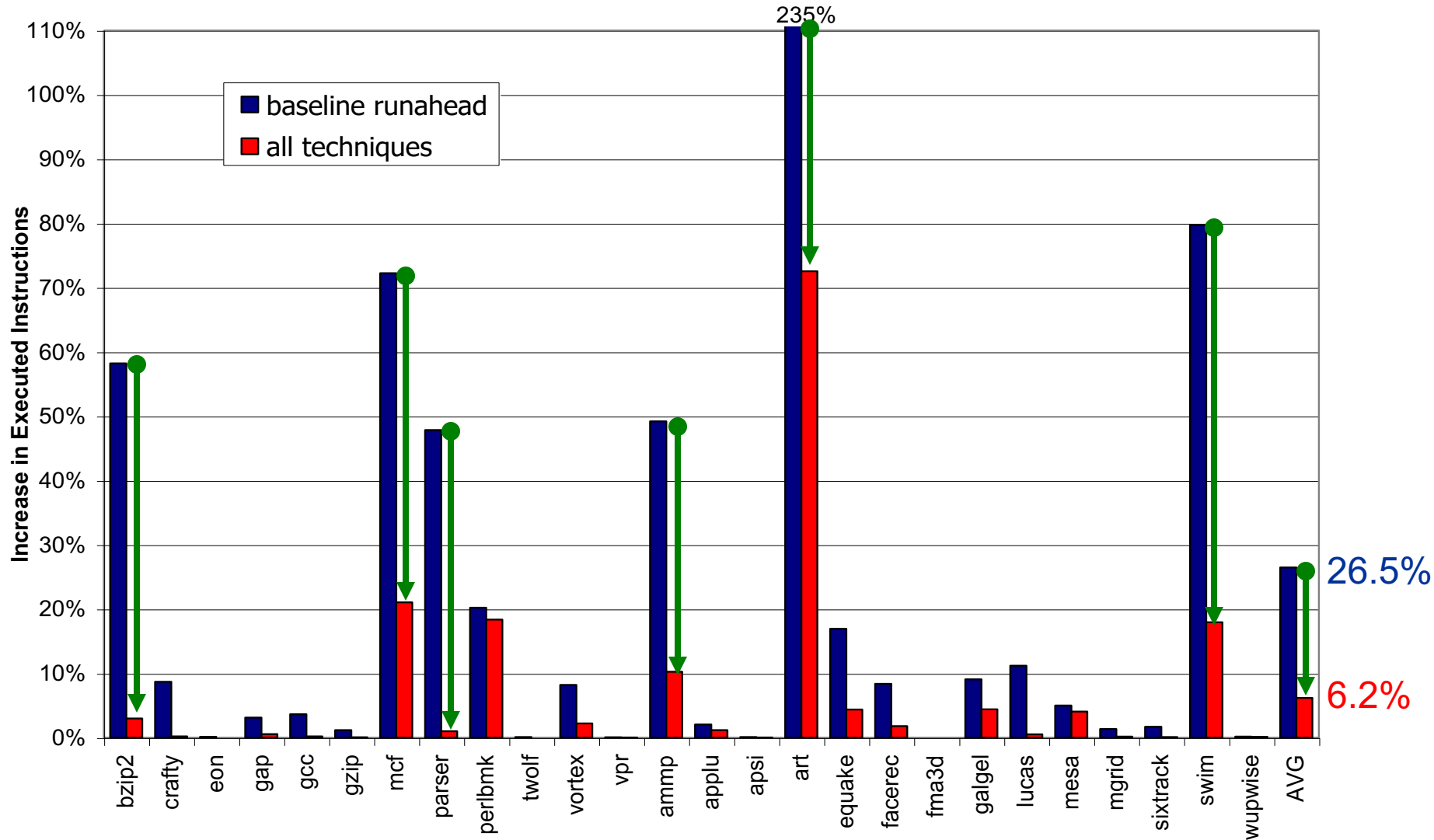
---

- Periods that do not result in prefetches for normal mode

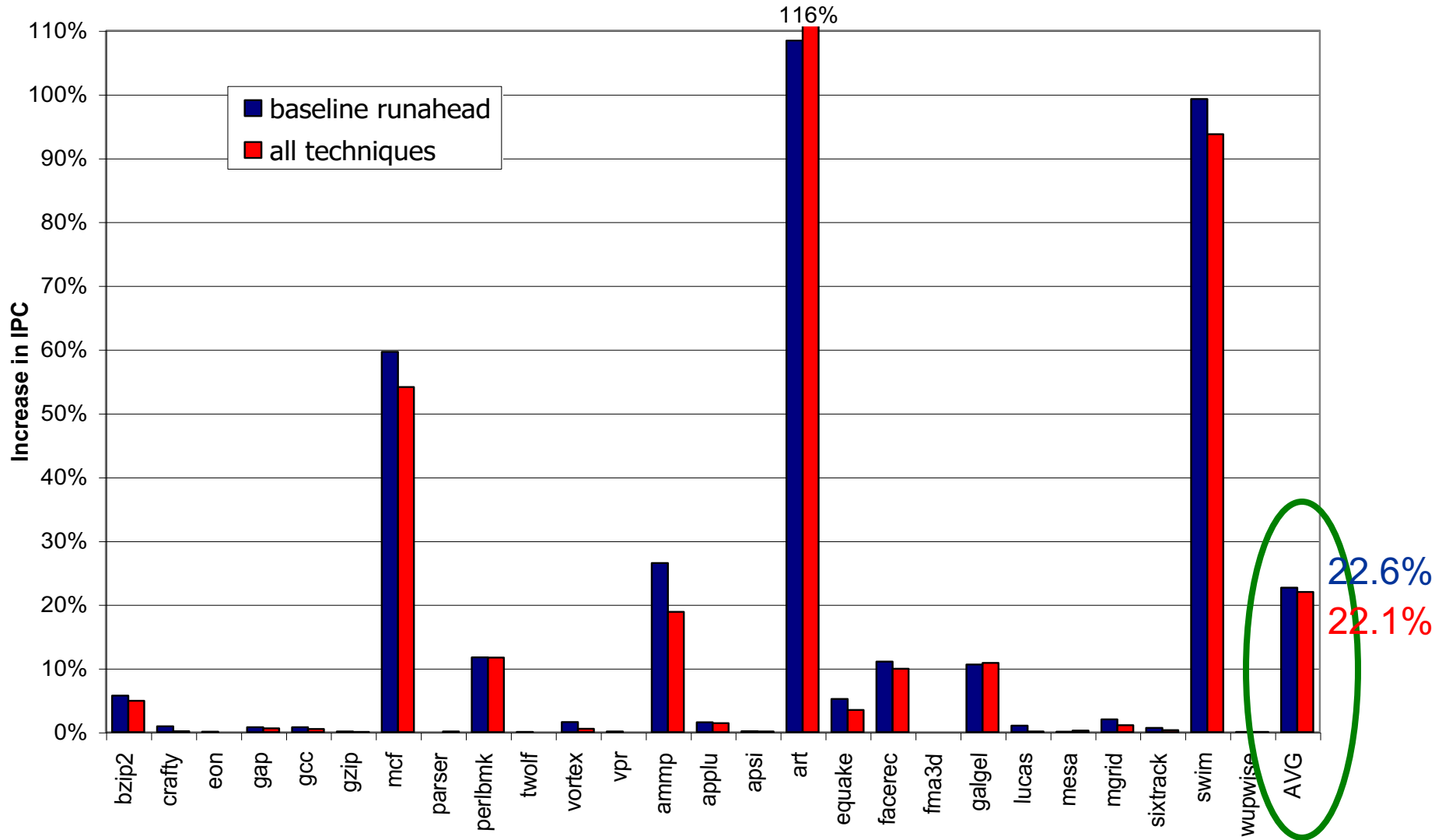


- They exist due to the lack of memory-level parallelism
  - Mechanism to eliminate useless periods:
    - Predict if a period will generate useful L2 misses
    - Estimate a period to be useful if it generated an L2 miss that cannot be captured by the instruction window
      - Useless period predictors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

# Overall Impact on Executed Instructions



# Overall Impact on IPC



# More on Efficient Runahead Execution

---

- Onur Mutlu, Hyesoon Kim, and Yale N. Patt,  
**"Techniques for Efficient Processing in Runahead Execution Engines"**  
*Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, pages 370-381, Madison, WI, June 2005. [Slides \(ppt\)](#) [Slides \(pdf\)](#)  
***One of the 13 computer architecture papers of 2005 selected as Top Picks by IEEE Micro.***

## Techniques for Efficient Processing in Runahead Execution Engines

Onur Mutlu   Hyesoon Kim   Yale N. Patt

Department of Electrical and Computer Engineering

University of Texas at Austin

{onur,hyesoon,patt}@ece.utexas.edu

# More on Efficient Runahead Execution

---

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

## EFFICIENT RUNAHEAD EXECUTION: POWER-EFFICIENT MEMORY LATENCY TOLERANCE

# Limitations of the Baseline Runahead Mechanism

---

## ■ Energy Inefficiency

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

## ■ Ineffectiveness for pointer-intensive applications

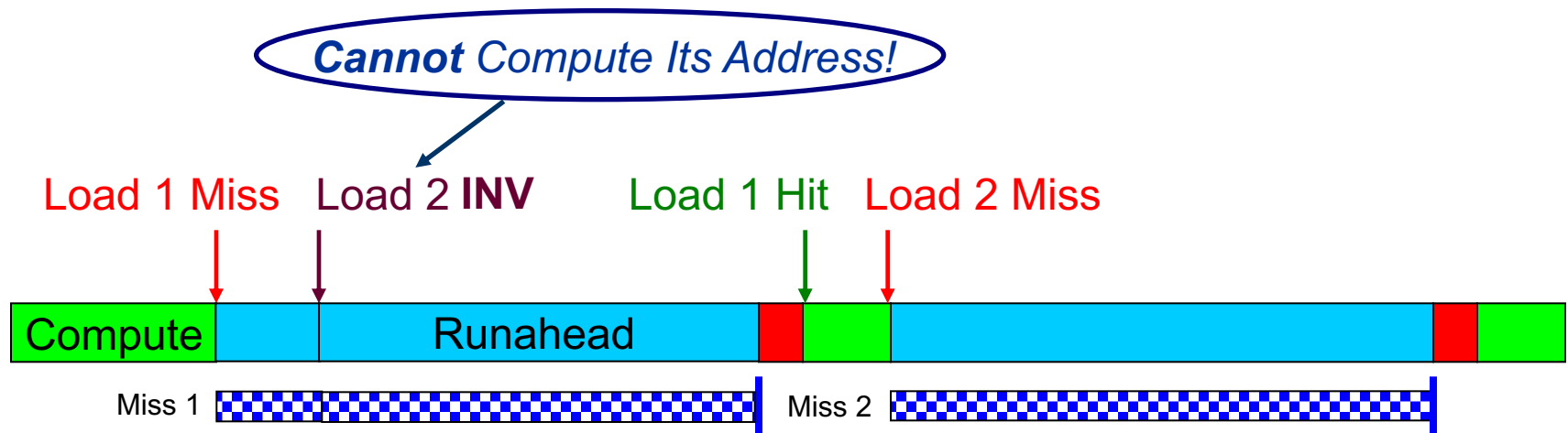
- ❑ Runahead cannot parallelize dependent L2 cache misses
- ❑ **Address-Value Delta (AVD) Prediction** [MICRO' 05]

## ■ Irresolvable branch mispredictions in runahead mode

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

# The Problem: Dependent Cache Misses

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



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

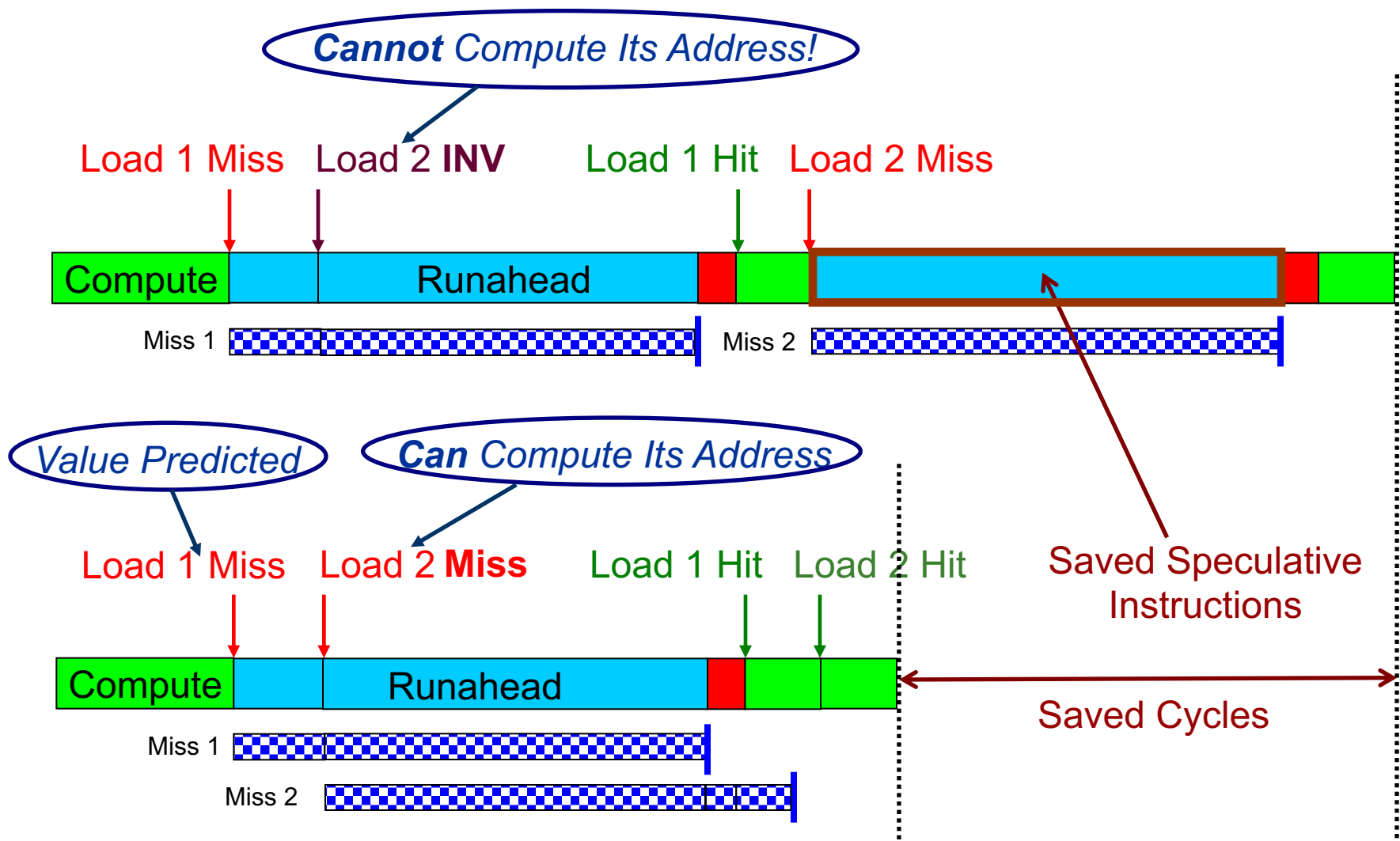
# Parallelizing Dependent Cache Misses

---

- **Idea:** Enable the parallelization of dependent L2 cache misses in runahead mode with a low-cost mechanism
  - **How:** Predict the values of L2-miss **address (pointer) loads**
    - **Address load:** loads an address into its destination register, which is later used to calculate the address of another load
    - as opposed to **data load**
  - **Read:**
    - Mutlu et al., “Address-Value Delta (AVD) Prediction,” MICRO 2005.
-



# Parallelizing Dependent Cache Misses



# More on AVD Prediction

---

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

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

Onur Mutlu   Hyesoon Kim   Yale N. Patt

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

# More on AVD Prediction (II)

---

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

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

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

# Even More on Runahead Execution

---

- Lecture video from Fall 2017
  - <https://www.youtube.com/watch?v=Kj3relihGF4>
- Onur Mutlu,  
**"Efficient Runahead Execution Processors"**  
Ph.D. Dissertation, HPS Technical Report, TR-HPS-2006-007, July 2006. [Slides \(ppt\)](#)  
***Nominated for the ACM Doctoral Dissertation Award by the University of Texas at Austin.***

# 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]



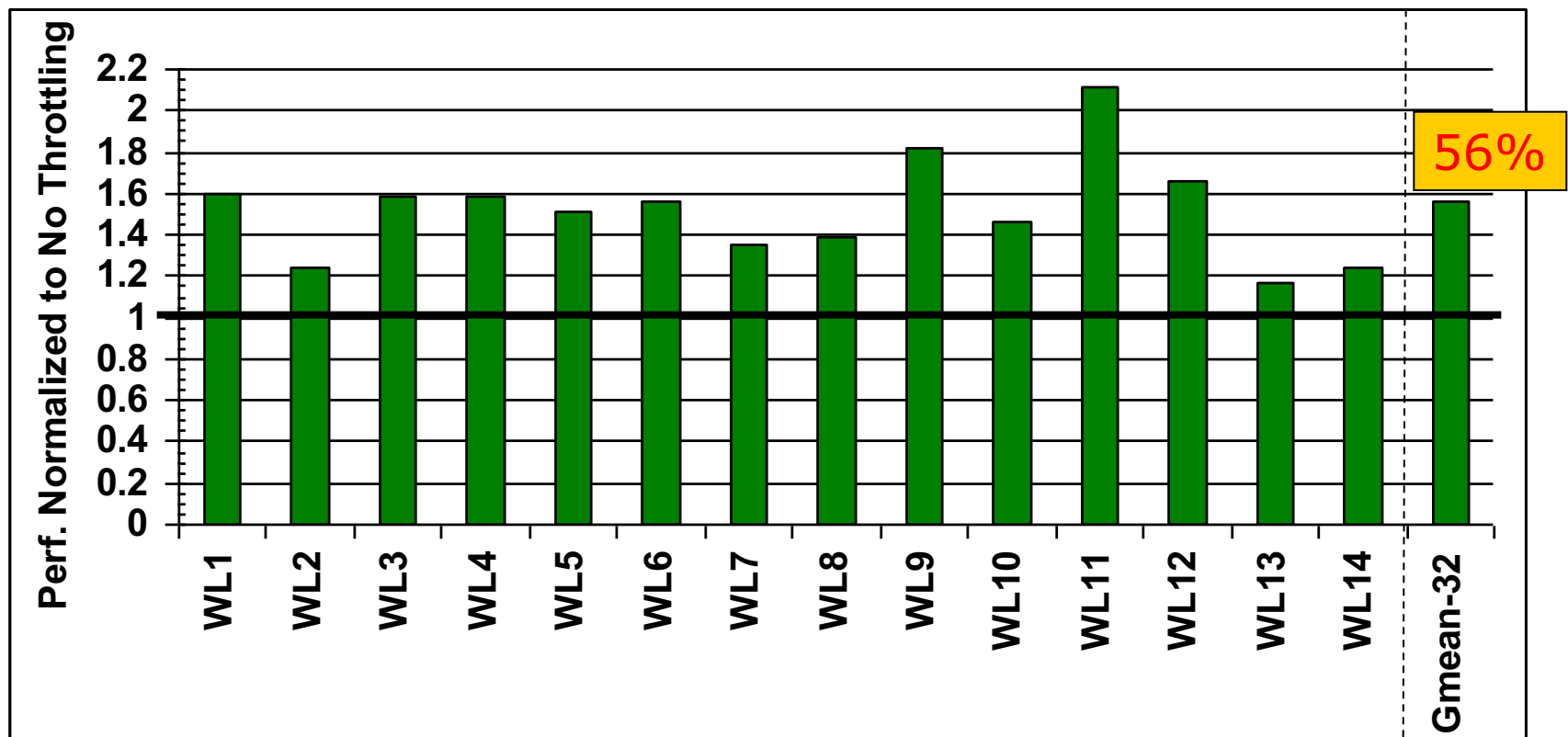
# Motivation

---

- 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

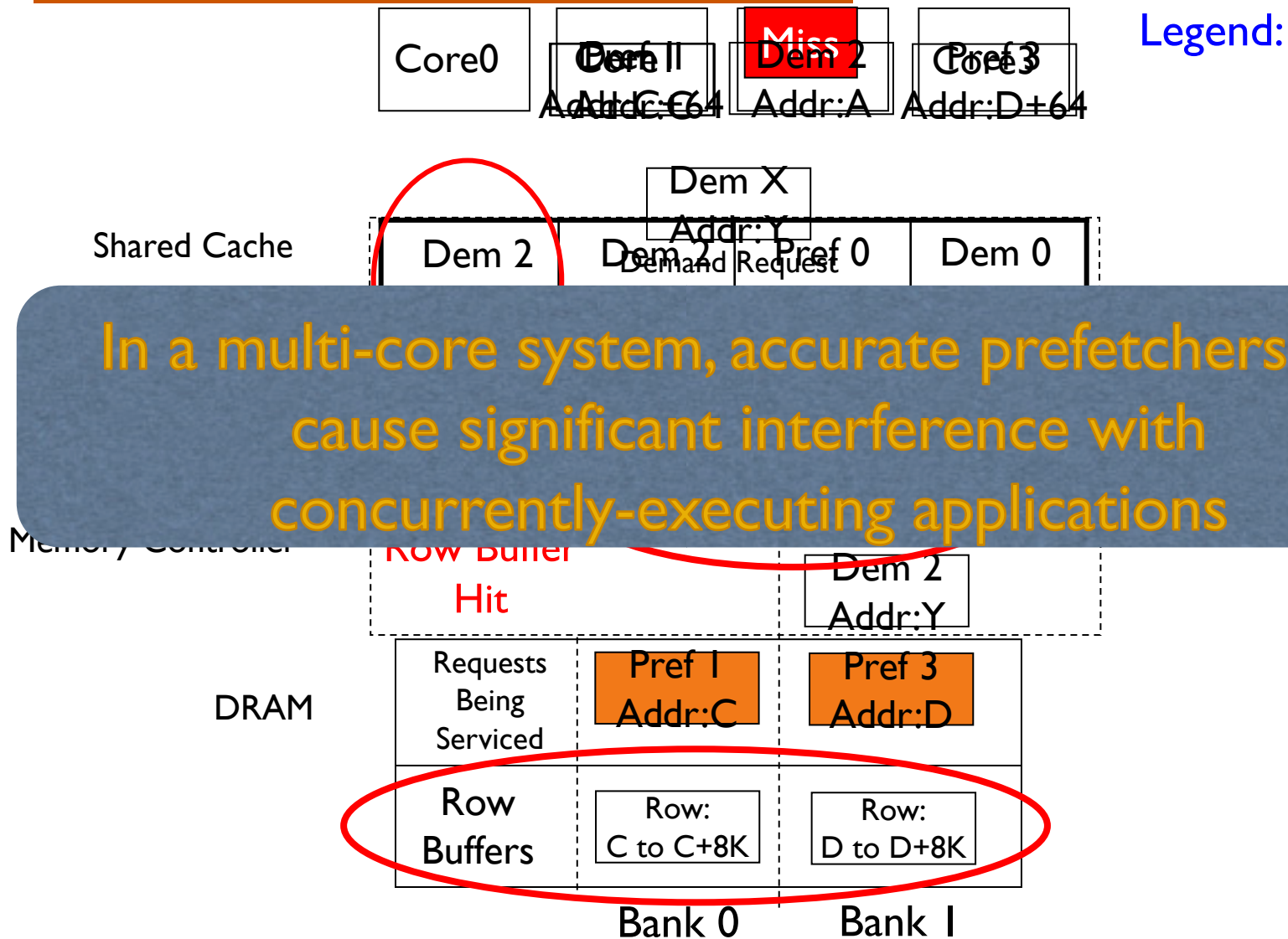
# Potential Performance

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

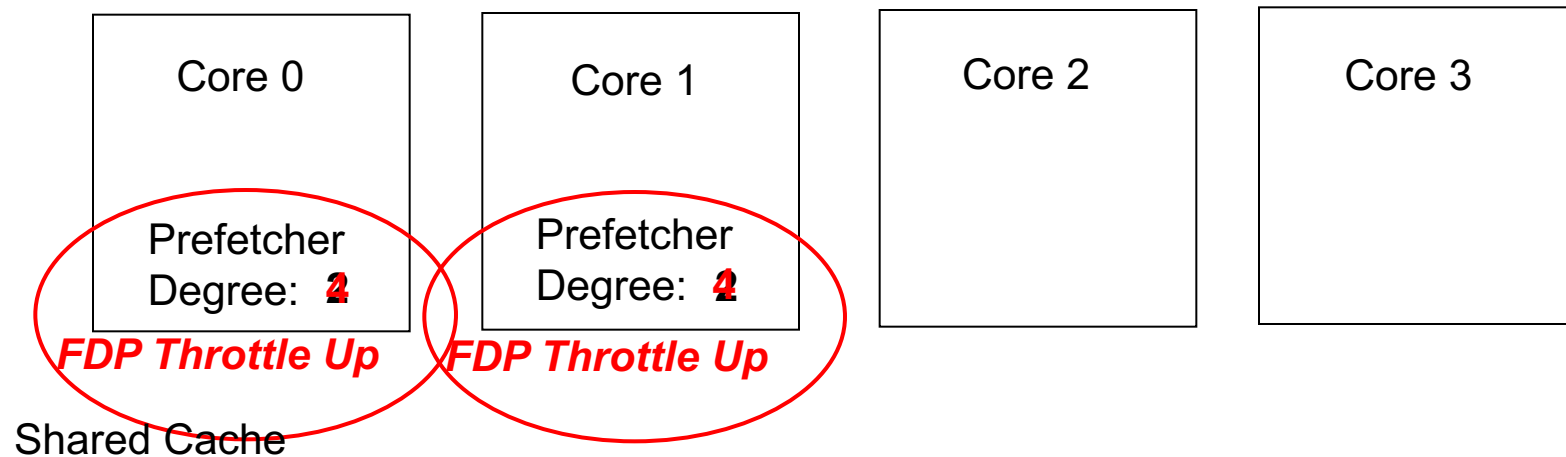


Exact workload combinations can be found in [Ebrahimi et al., MICRO 2009]

# High Interference caused by Accurate Prefetchers



# Shortcoming of Local Prefetcher Throttling

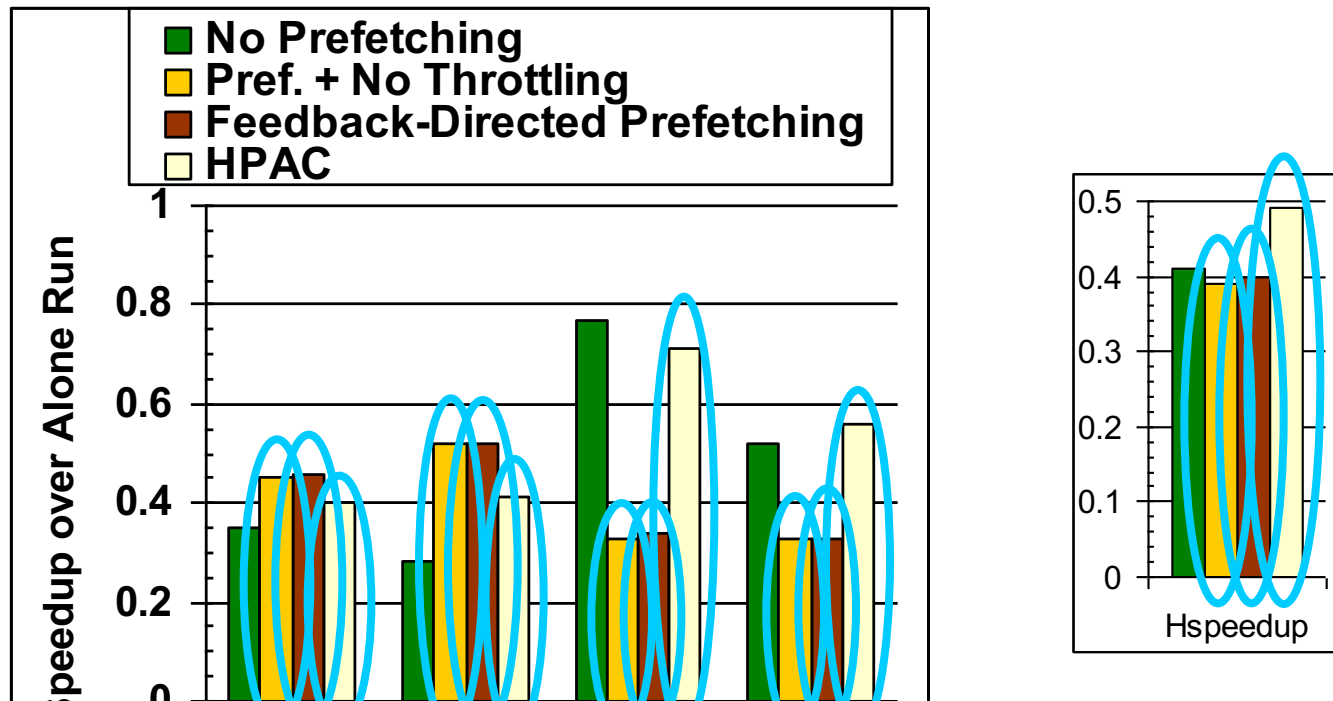


Set 0	<del>Dem 0</del>	<del>Dem 0</del>	<del>Dem 1</del>	<del>Dem 1</del>	Dem 2	Dem 2	Dem 3	Dem 3
Set 1	<del>Dem 0</del>	<del>Dem 0</del>	<del>Dem 1</del>	<del>Dem 1</del>	Dem 3	Dem 3	Dem 3	Dem 3
Set 2	<del>Dem 0</del>	<del>Dem 0</del>	<del>Dem 0</del>	<del>Dem 0</del>	<del>Dem 1</del>	<del>Dem 1</del>	<del>Dem 1</del>	<del>Dem 1</del>

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

# Shortcoming of Local-Only Prefetcher Control

4-core workload example: lbm\_06 + swim\_00 + crafty\_00 + bzip2\_00



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

# Prefetching in Multi-Core (II)

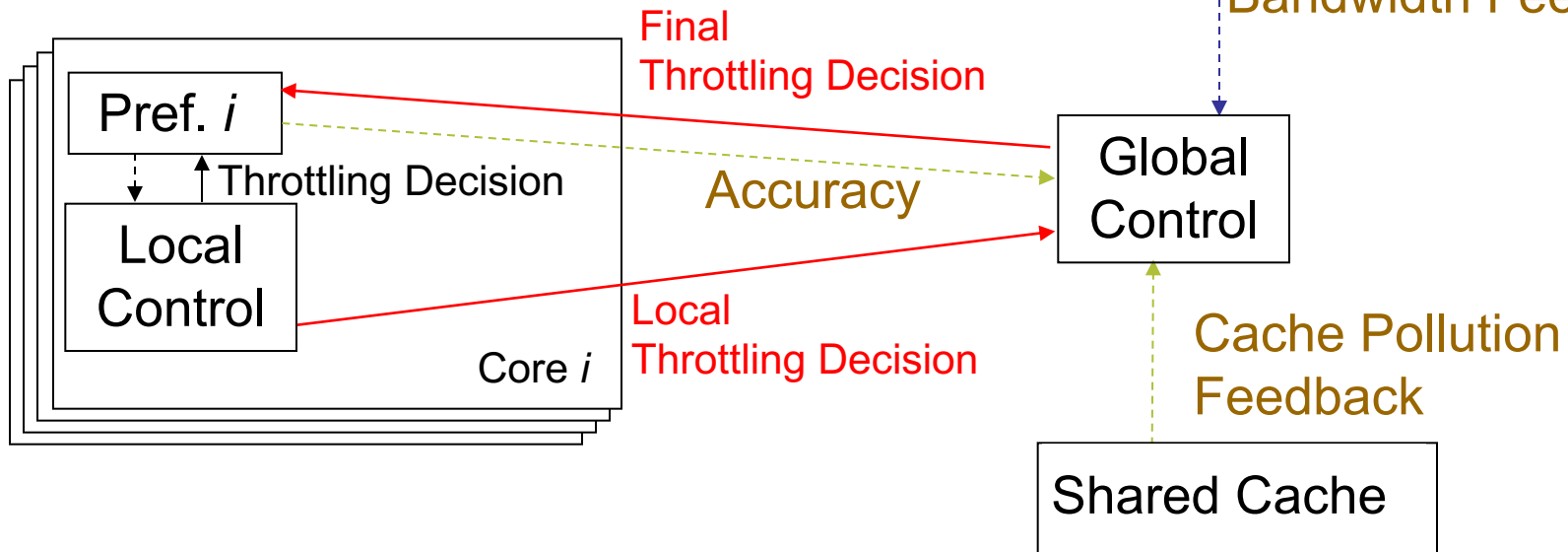
---

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

# Hierarchical Prefetcher Throttling

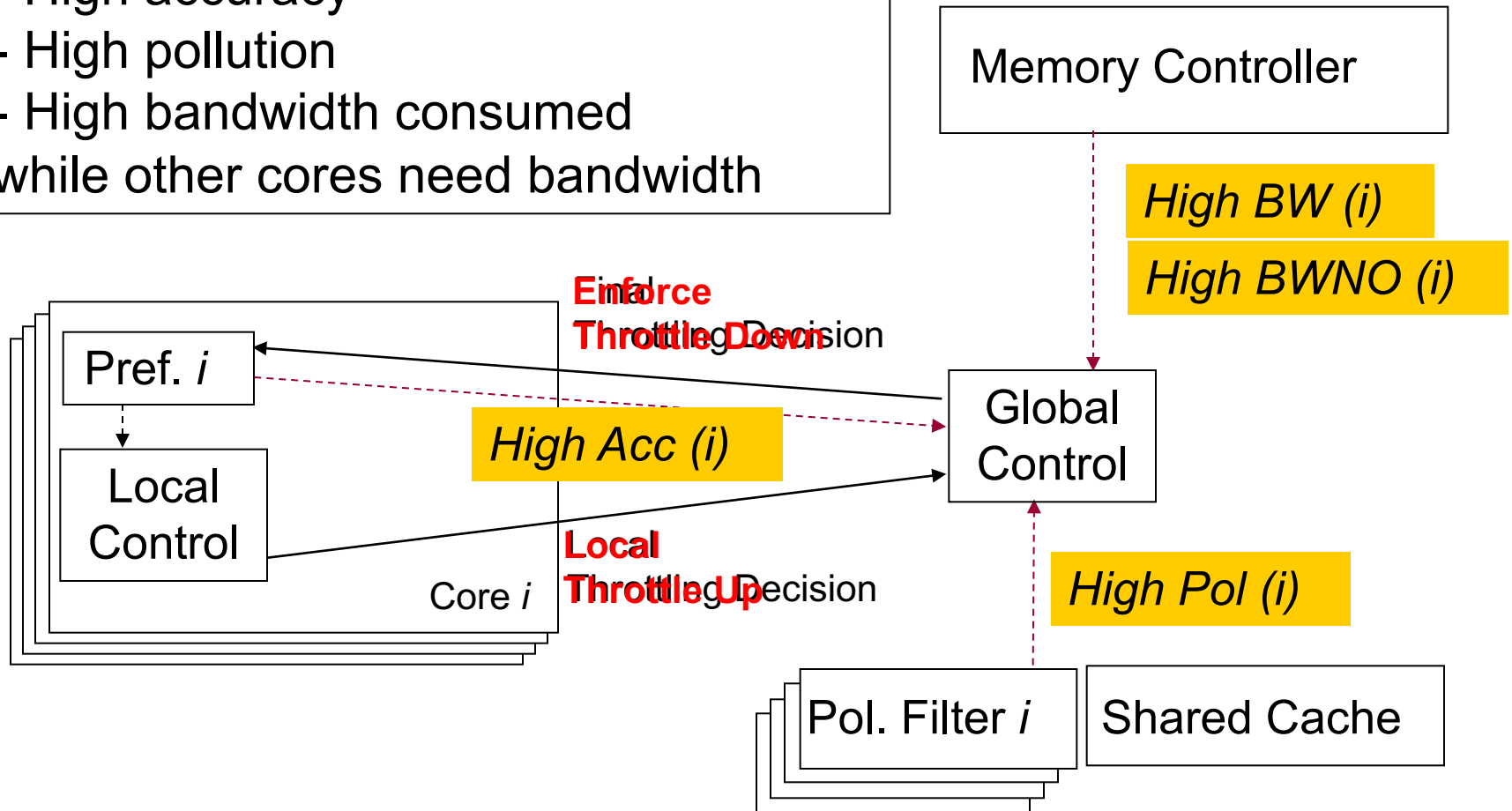
Global Control's goal: **accepts** or **overrides** the decisions made by local control to improve core  $i$  **independently** overall system performance

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



# Hierarchical Prefetcher Throttling Example

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

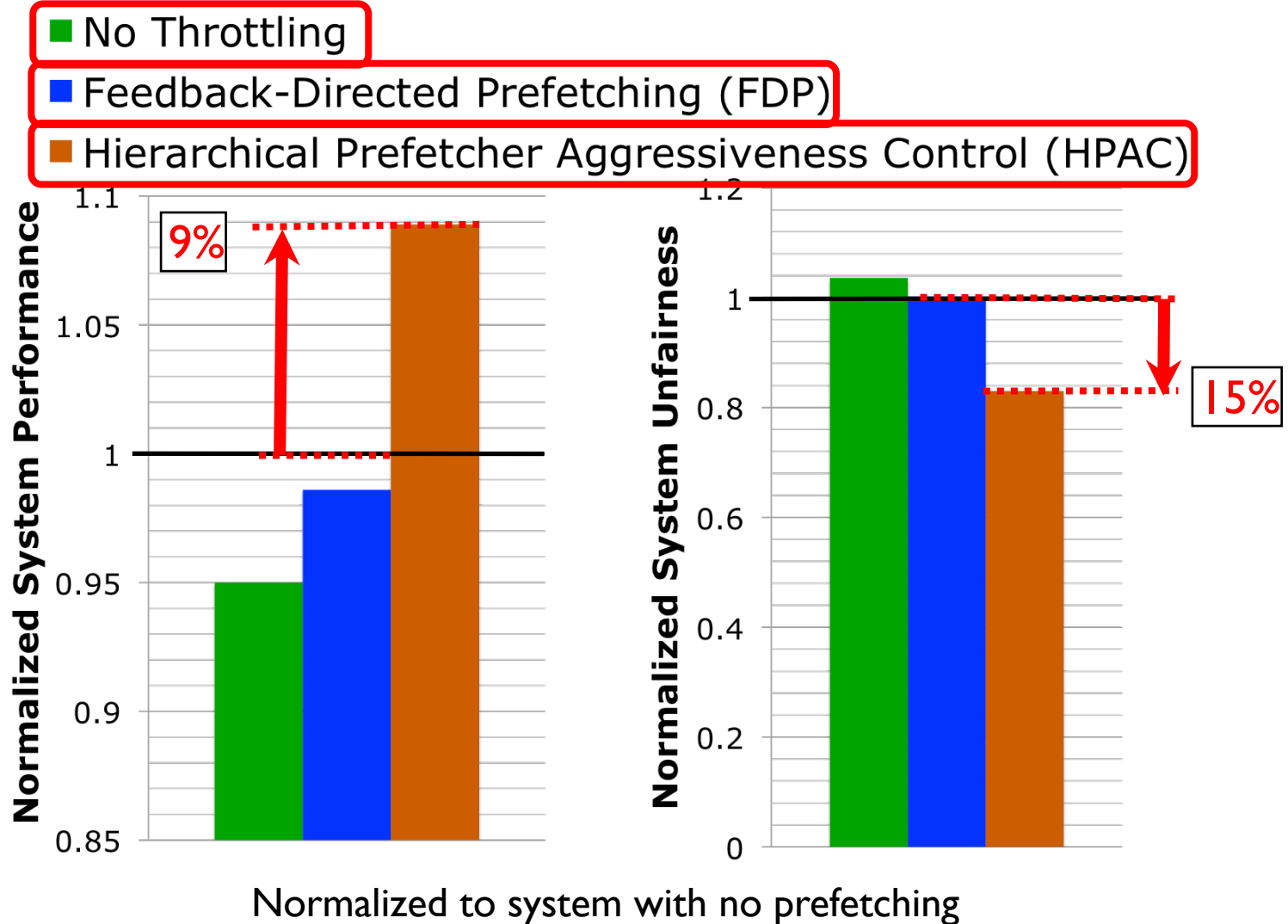




# HPAC Control Policies

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

# HPAC Evaluation



# More on Coordinated Prefetcher Control

---

- Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt,  
**"Coordinated Control of Multiple Prefetchers in Multi-Core Systems"**  
*Proceedings of the 42nd International Symposium on  
Microarchitecture (MICRO)*, pages 316-326, New York, NY, December  
2009. Slides (ppt)

## Coordinated Control of Multiple Prefetchers in Multi-Core Systems

Eiman Ebrahimi<sup>†</sup> Onur Mutlu<sup>§</sup> Chang Joo Lee<sup>†</sup> Yale N. Patt<sup>†</sup>

<sup>†</sup>Department of Electrical and Computer Engineering  
The University of Texas at Austin  
{ebrahimi, cjlee, patt}@ece.utexas.edu

<sup>§</sup>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<sup>†</sup>   Onur Mutlu<sup>§</sup>   Veynu Narasiman<sup>†</sup>   Yale N. Patt<sup>†</sup>

<sup>†</sup>Department of Electrical and Computer Engineering  
The University of Texas at Austin  
{cjlee, narasima, patt}@ece.utexas.edu

<sup>§</sup>Microsoft Research and Carnegie Mellon University  
onur@{microsoft.com,cmu.edu}

# Problems of Prefetch Handling

---

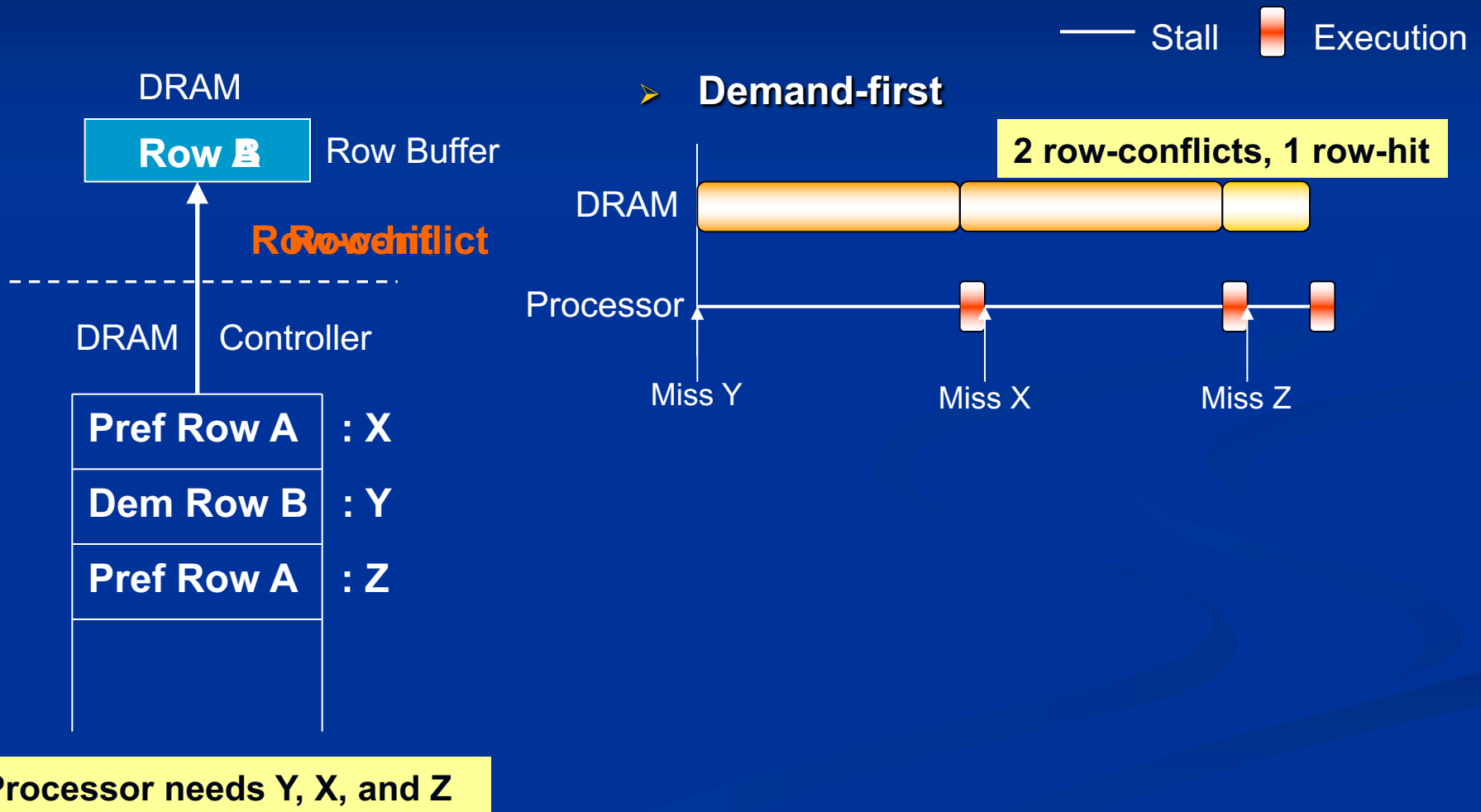
- **How to schedule *prefetches* vs *demands*?**
  - Demand-first: Always prioritizes demands over prefetch requests
  - Demand-prefetch-equal: Always treats them the same

**Neither of these perform best**

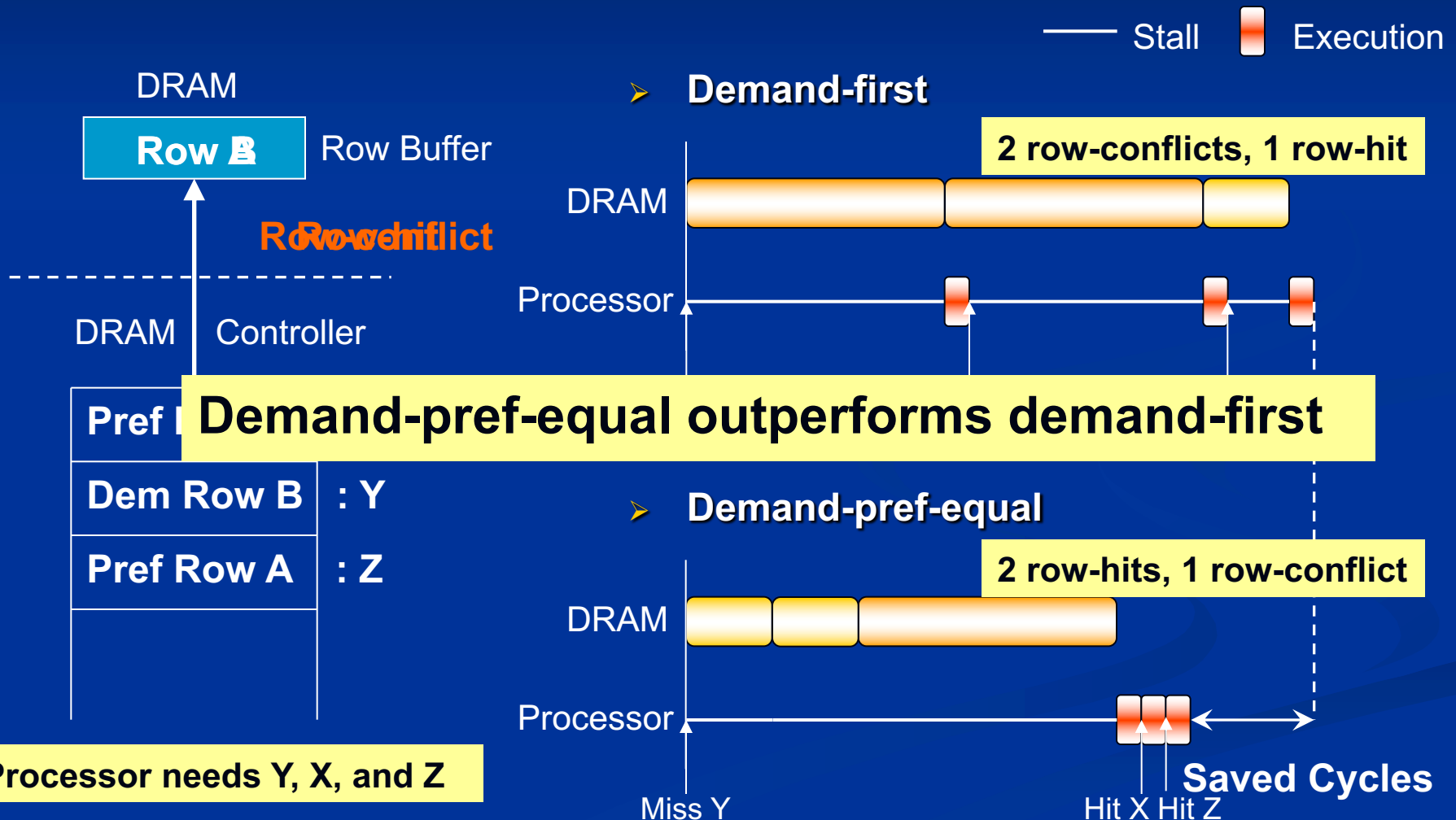
**Neither take into account both:**

- 1. Non-uniform access latency of DRAM systems**
- 2. Usefulness of prefetches**

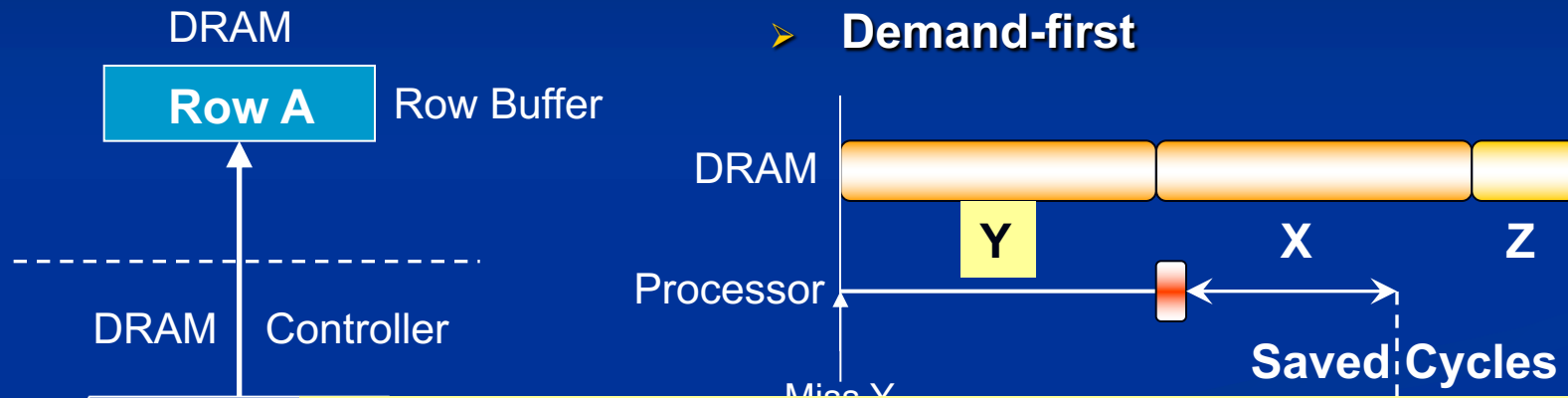
# When Prefetches are Useful



# When Prefetches are Useful

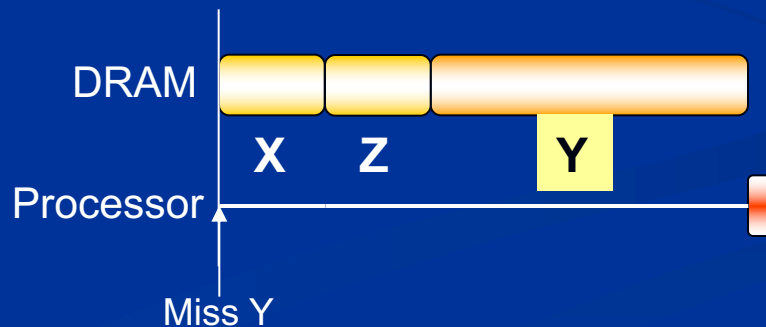


# When Prefetches are Useless



**Demand-first outperforms demand-pref-equal**

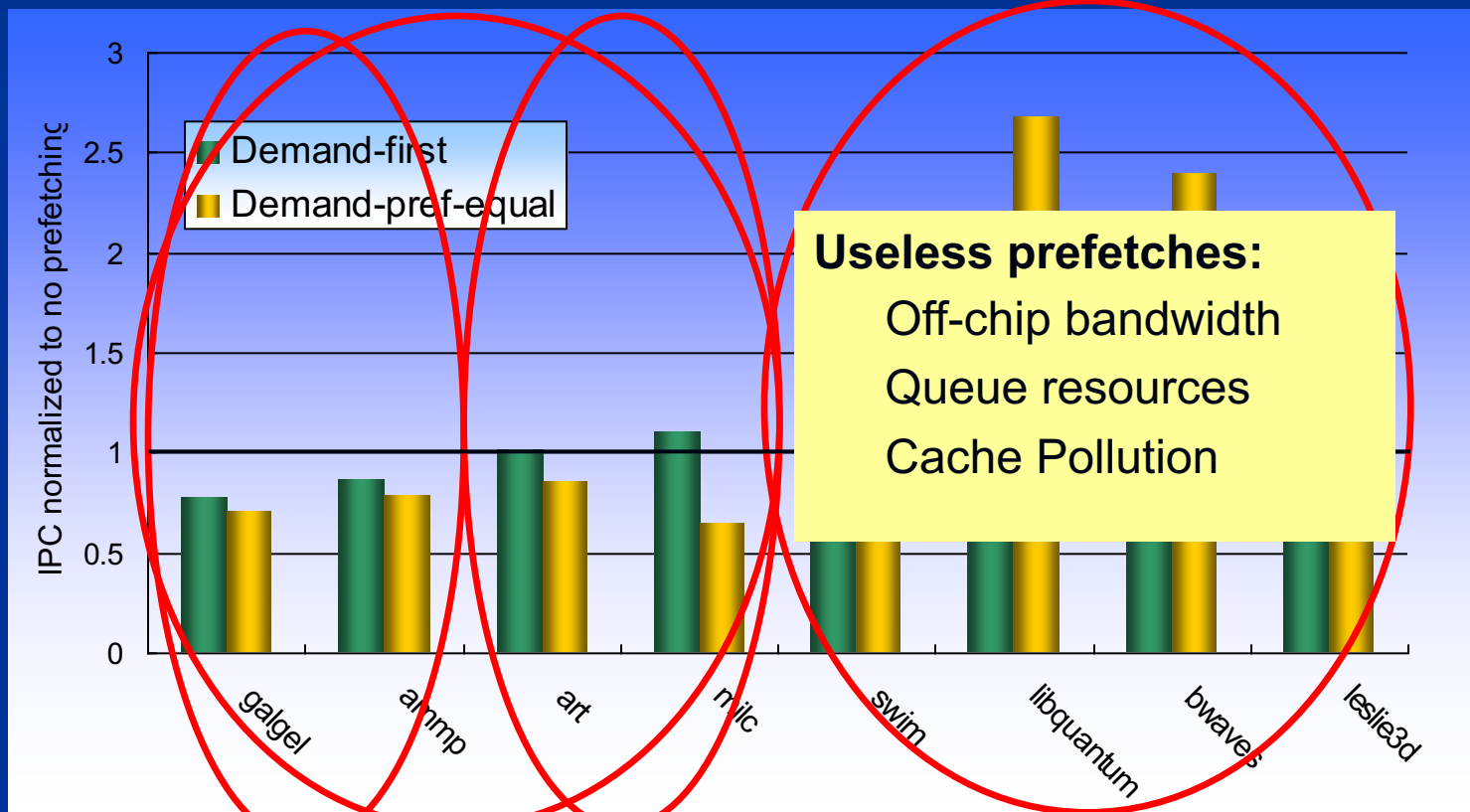
➤ **Demand-pref-equal**





# Demand-first vs. Demand-pref-equal policy

Stream prefetcher enabled



**Goal 1: Adaptive Goal 2: Eliminate useless prefetches**    **etch 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<sup>†</sup> Veynu Narasiman<sup>†</sup> Onur Mutlu<sup>§</sup> Yale N. Patt<sup>†</sup>

<sup>†</sup>Department of Electrical and Computer Engineering  
The University of Texas at Austin  
{cjlee, narasima, patt}@ece.utexas.edu

<sup>§</sup>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<sup>†</sup>   Chang Joo Lee<sup>†‡</sup>   Onur Mutlu<sup>§</sup>   Yale N. Patt<sup>†</sup>

<sup>†</sup>HPS Research Group  
The University of Texas at Austin  
{ebrahimi, patt}@hps.utexas.edu

<sup>‡</sup>Intel Corporation  
chang.joo.lee@intel.com

<sup>§</sup>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

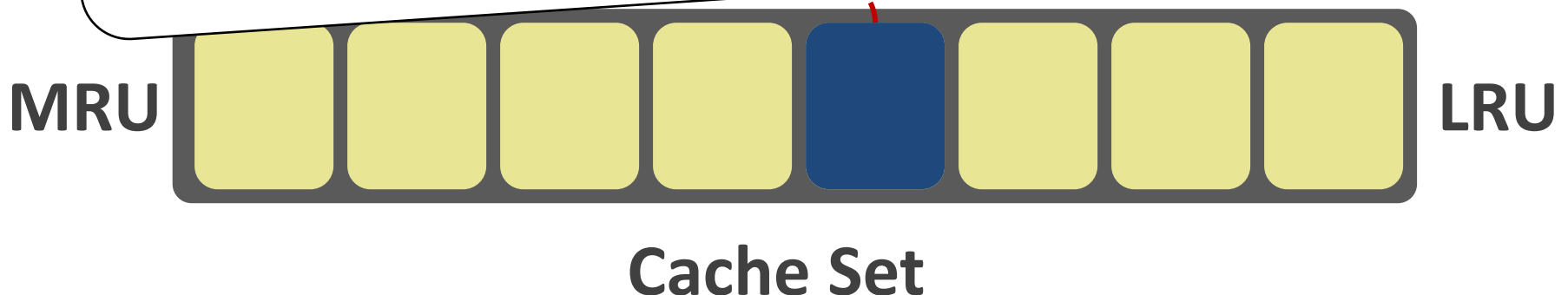
VIVEK SESHADRI, SAMIHAN YEDKAR, HONGYI XIN, and ONUR MUTLU,  
Carnegie Mellon University  
PHILLIP B. GIBBONS and MICHAEL A. KOZUCH, Intel Pittsburgh  
TODD C. MOWRY, Carnegie Mellon University

# Caching Policies for Prefetched Blocks

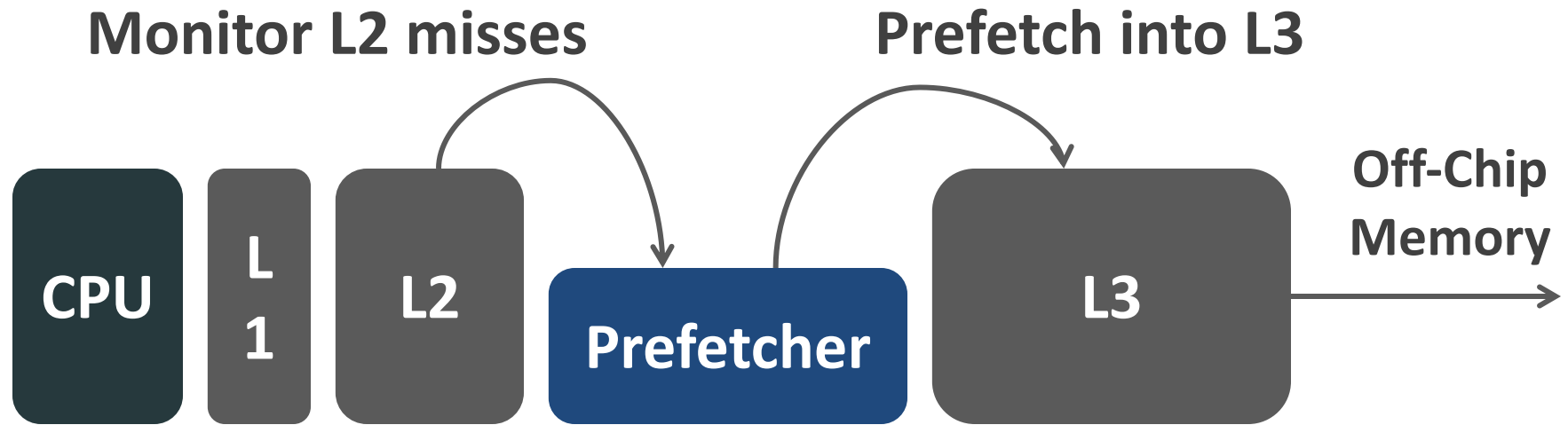
**Problem:** Existing caching policies for prefetched blocks result in significant cache pollution

Cache Miss:

Are these insertion and promotion policies good for prefetched blocks?



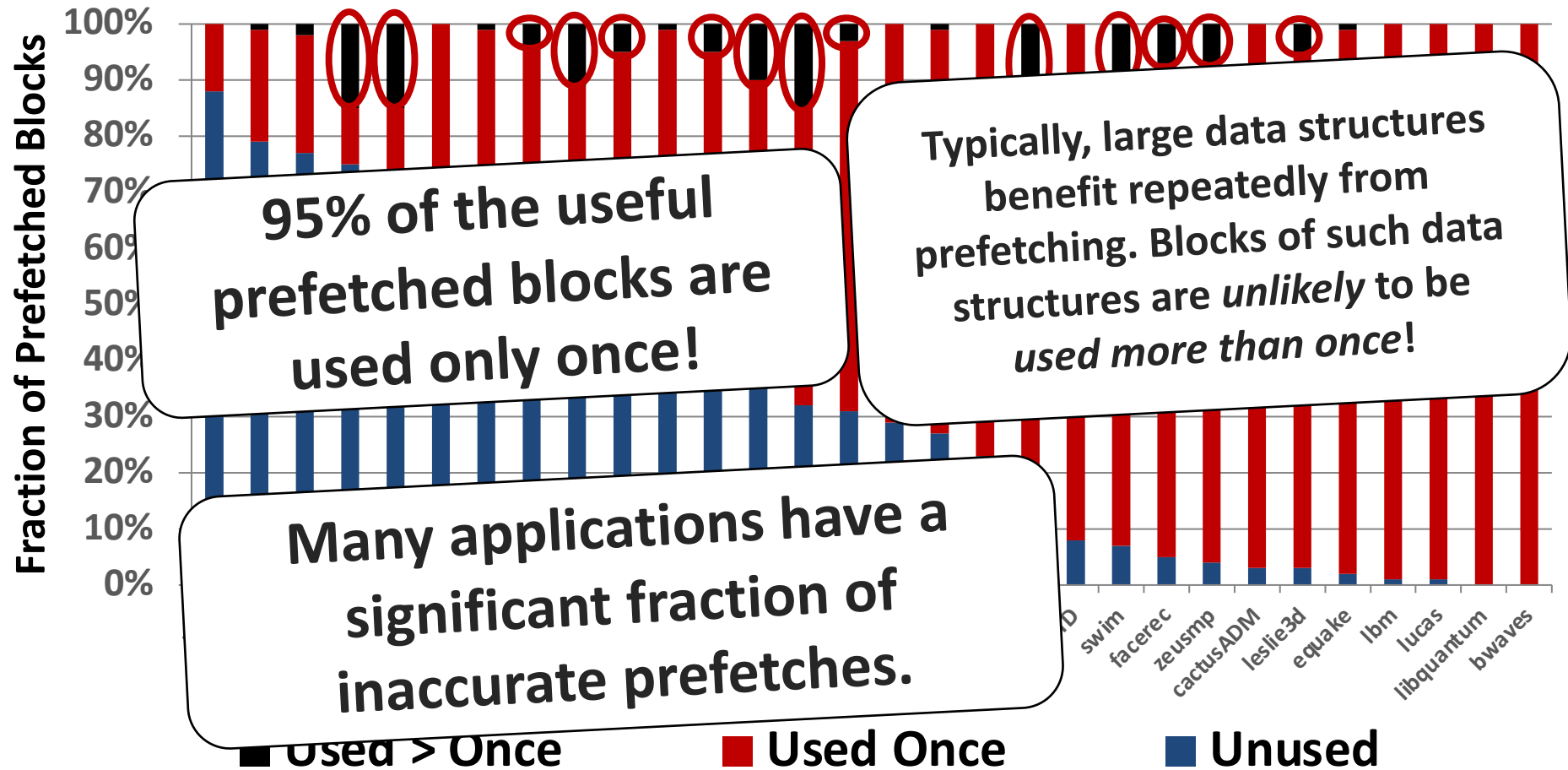
# Prefetch Usage Experiment



**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



# Shortcoming of Traditional Promotion Policy

Promote to MRU

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

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

Cache Set



# Demotion of Prefetched Block

Demote to LRU

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

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

Cache Set

# Cache Insertion Policy for Prefetched Blocks

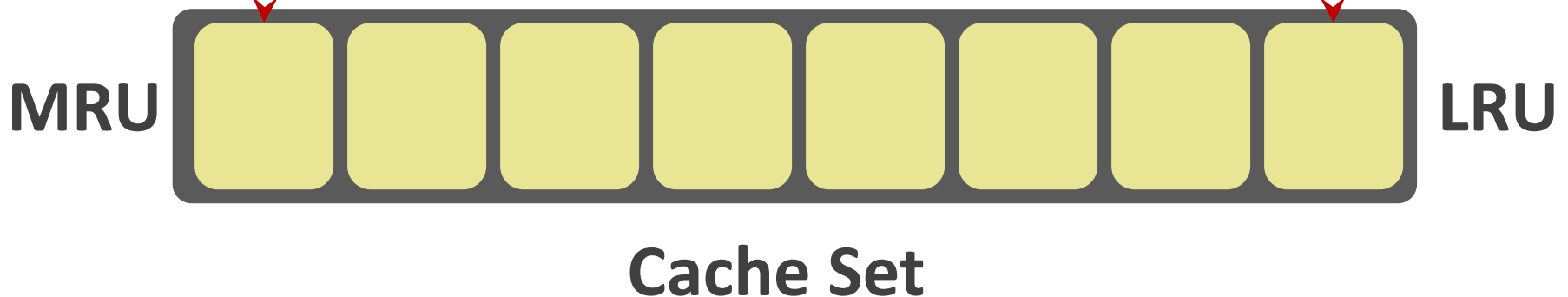
Good (Accurate prefetch)

Bad (Inaccurate prefetch)

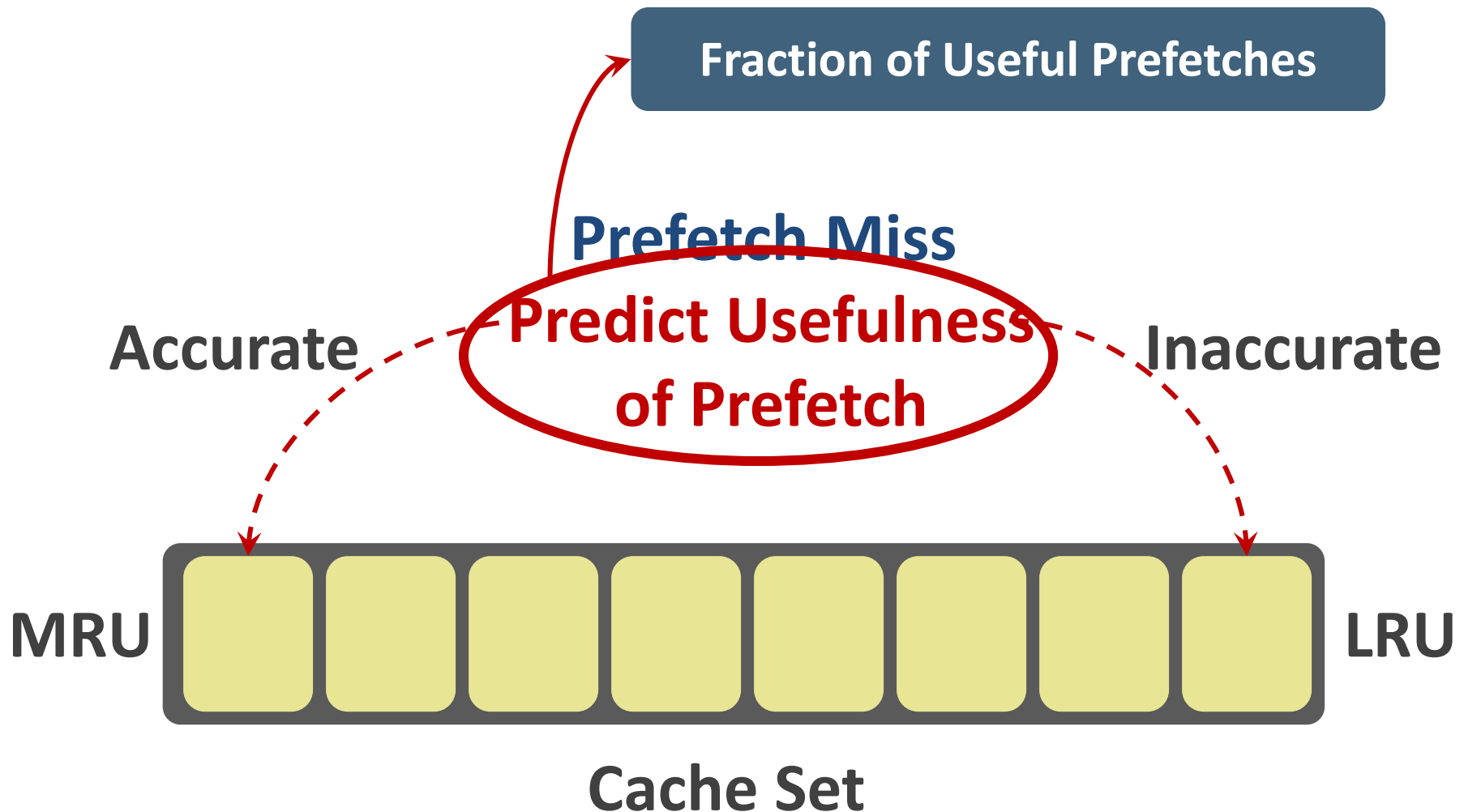
Good (Inaccurate prefetch)

Bad (accurate prefetch)

**Prefetch Miss:  
Insertion Policy?**



# Predicting Usefulness of Prefetch



# 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<sup>†</sup>   Onur Kayiran<sup>‡</sup>   Asit K. Mishra<sup>§</sup>   Mahmut T. Kandemir<sup>†</sup>  
Onur Mutlu<sup>‡</sup>   Ravishankar Iyer<sup>§</sup>   Chita R. Das<sup>†</sup>

<sup>†</sup>The Pennsylvania State University  
University Park, PA 16802

<sup>‡</sup>Carnegie Mellon University  
Pittsburgh, PA 15213

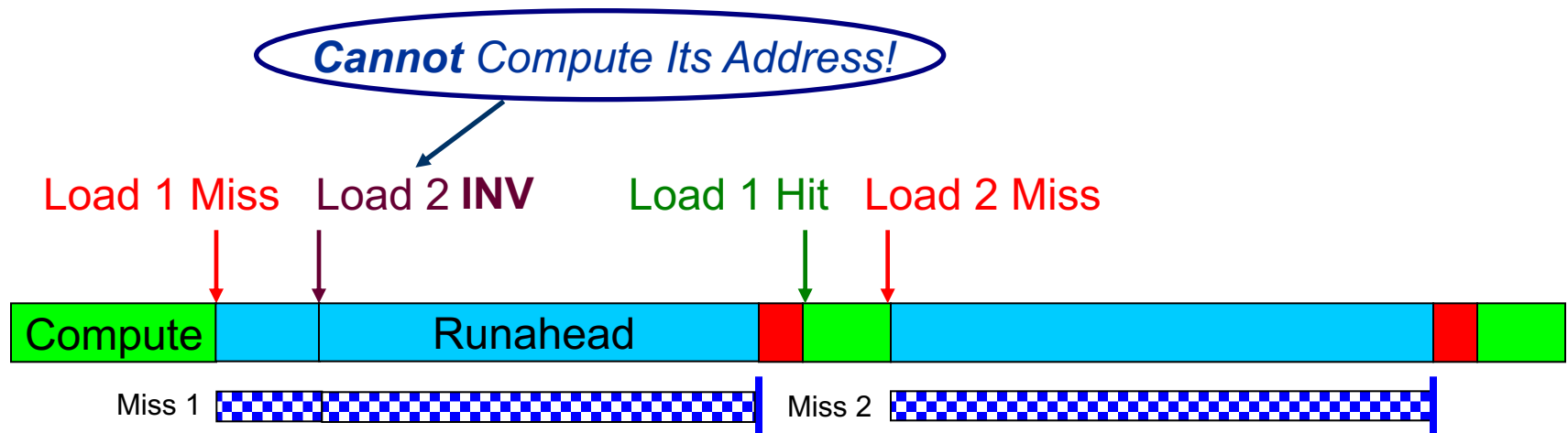
<sup>§</sup>Intel Labs  
Hillsboro, OR 97124

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

# Address-Value Delta Prediction

# The Problem: Dependent Cache Misses

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



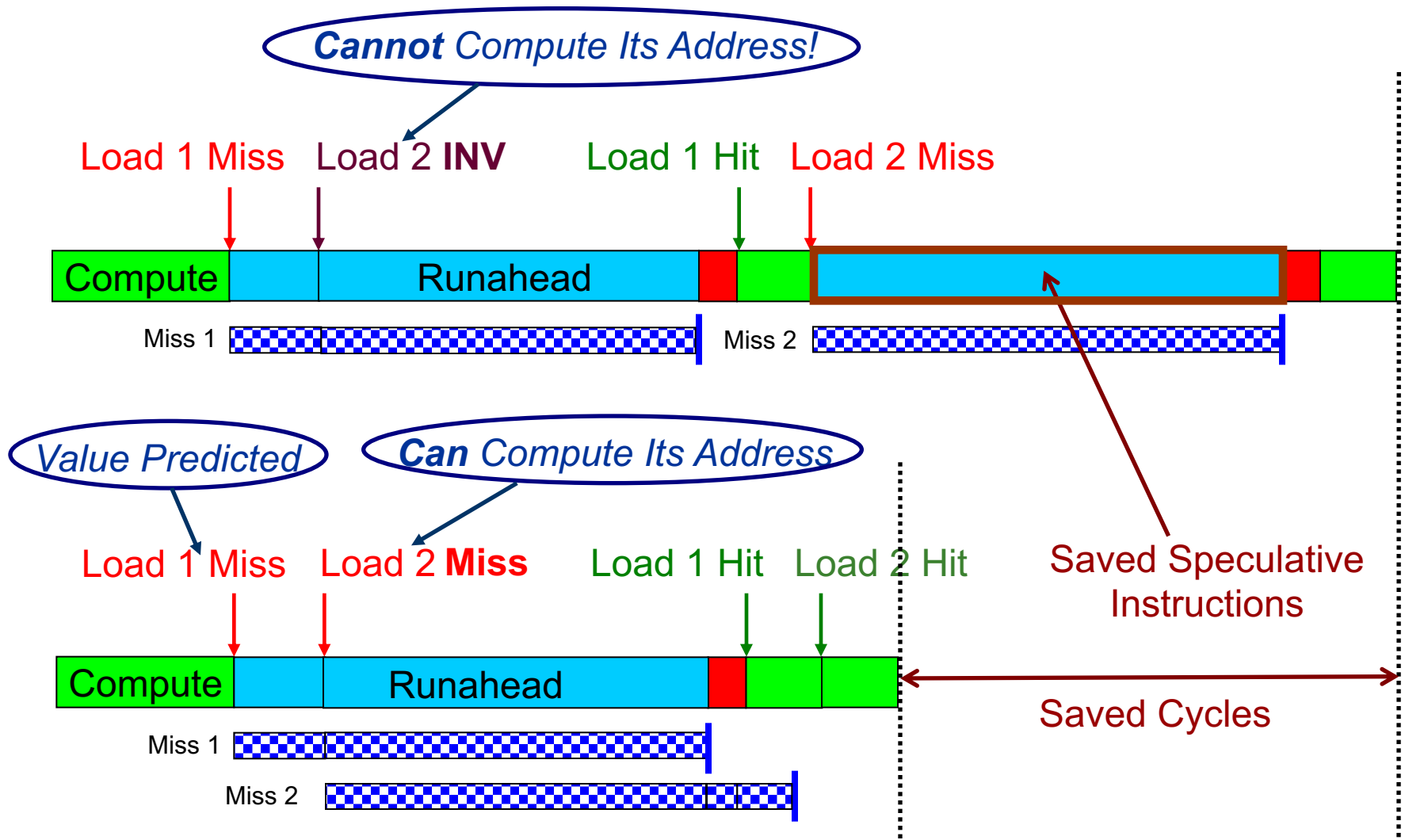
- Runahead execution cannot parallelize dependent misses
  - ❑ wasted opportunity to improve performance
  - ❑ wasted energy (useless pre-execution)
- Runahead performance would improve by 25% if this limitation were ideally overcome

# Parallelizing Dependent Cache Misses

---

- **Idea:** Enable the parallelization of dependent L2 cache misses in runahead mode with a low-cost mechanism
  - **How:** Predict the values of L2-miss **address (pointer) loads**
    - **Address load:** loads an address into its destination register, which is later used to calculate the address of another load
    - as opposed to **data load**
  - **Read:**
    - Mutlu et al., “Address-Value Delta (AVD) Prediction,” MICRO 2005.
-

# Parallelizing Dependent Cache Misses





# AVD Prediction [MICRO' 05]

---

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

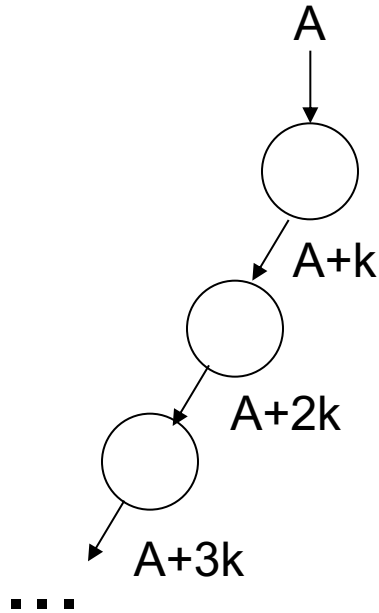
# Why Do Stable AVDs Occur?

---

- Regularity in the way data structures are
    - allocated in memory AND
    - traversed
  - Two types of loads can have stable AVDs
    - Traversal address loads
      - Produce addresses consumed by **address loads**
    - Leaf address loads
      - Produce addresses consumed by **data loads**
-

# Traversal Address Loads

Regularly-allocated linked list:



A **traversal address load** loads the pointer to next node:

**node = node→next**

$AVD = \text{Effective Addr} - \text{Data Value}$

Effective Addr	Data Value	AVD
<b>A</b>	<b>A+k</b>	<b>-k</b>
<b>A+k</b>	<b>A+2k</b>	<b>-k</b>
<b>A+2k</b>	<b>A+3k</b>	<b>-k</b>

Striding  
data value

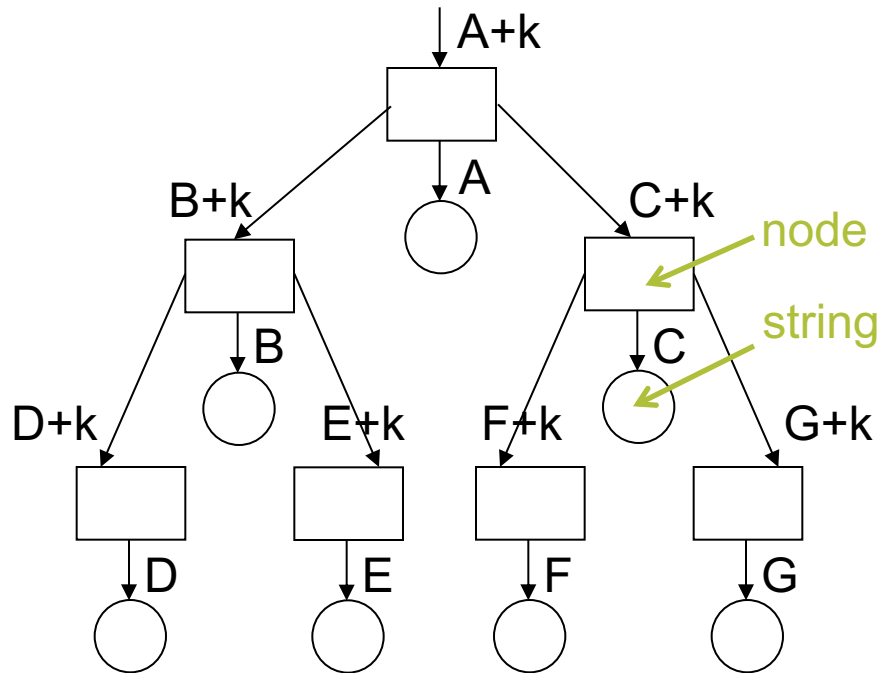
Stable AVD

# Leaf Address Loads

Sorted dictionary in **parser**:

**Nodes** point to **strings** (words)

String and node allocated consecutively



Dictionary looked up for an input word.

A **leaf address load** loads the pointer to the string of each node:

```
lookup (node, input) { // ...
```

```
    ptr_str = node → string;
```

```
    m = check_match(ptr_str, input);
```

```
    // ...
```

```
}
```

**AVD = Effective Addr – Data Value**

Effective Addr	Data Value	AVD
<b>A+k</b>	<b>A</b>	<b>k</b>
<b>C+k</b>	<b>C</b>	<b>k</b>
<b>F+k</b>	<b>F</b>	<b>k</b>

No stride! Stable AVD

# Identifying Address Loads in Hardware

---

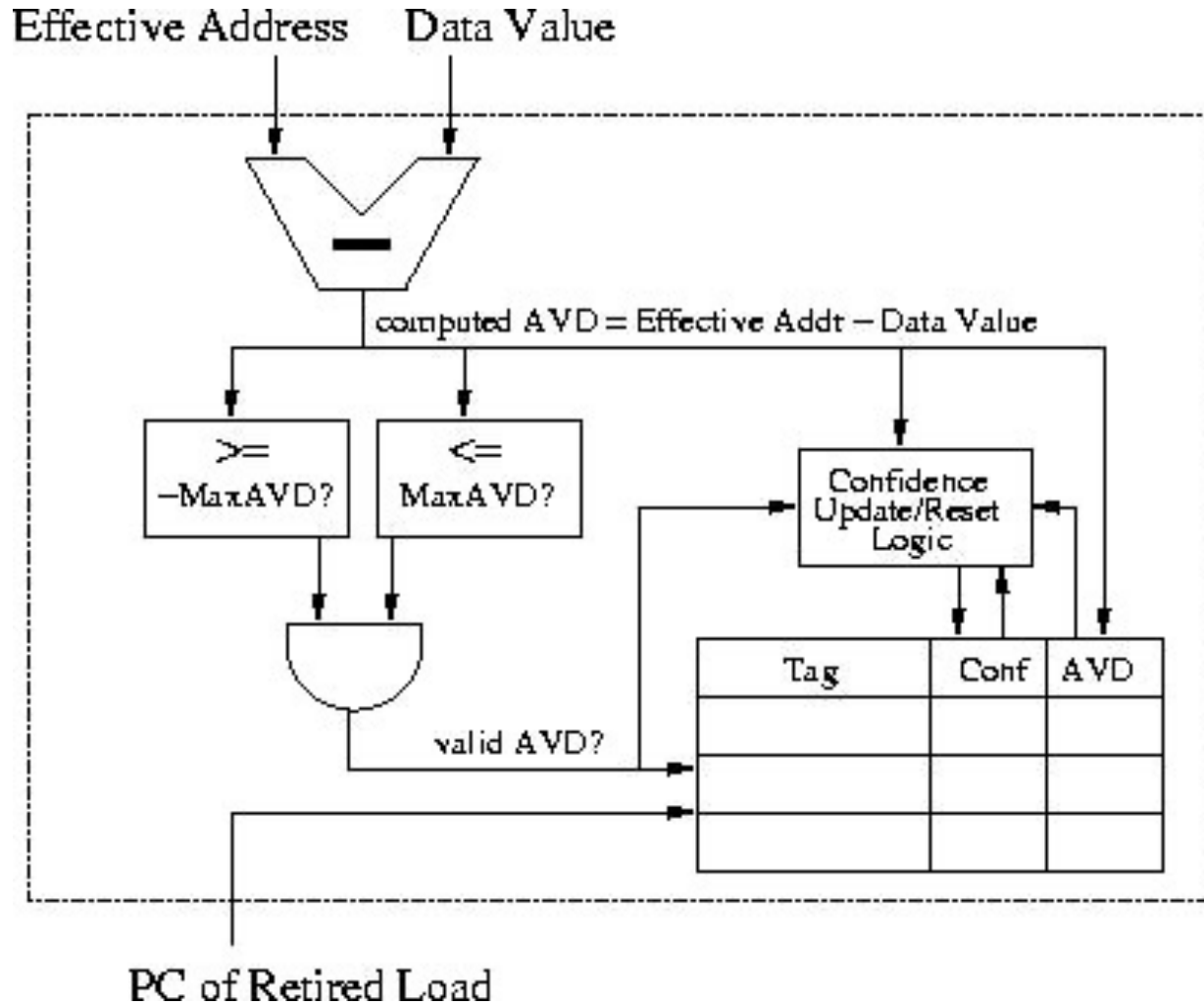
- Insight:
  - If the AVD is too large, the value that is loaded is likely **not** an address
- Only keep track of loads that satisfy:  
$$-\text{MaxAVD} \leq \text{AVD} \leq +\text{MaxAVD}$$
- This identification mechanism eliminates many loads from consideration for prediction
  - No need to value- predict the loads that will not generate addresses
  - Enables the predictor to be small

# An Implementable AVD Predictor

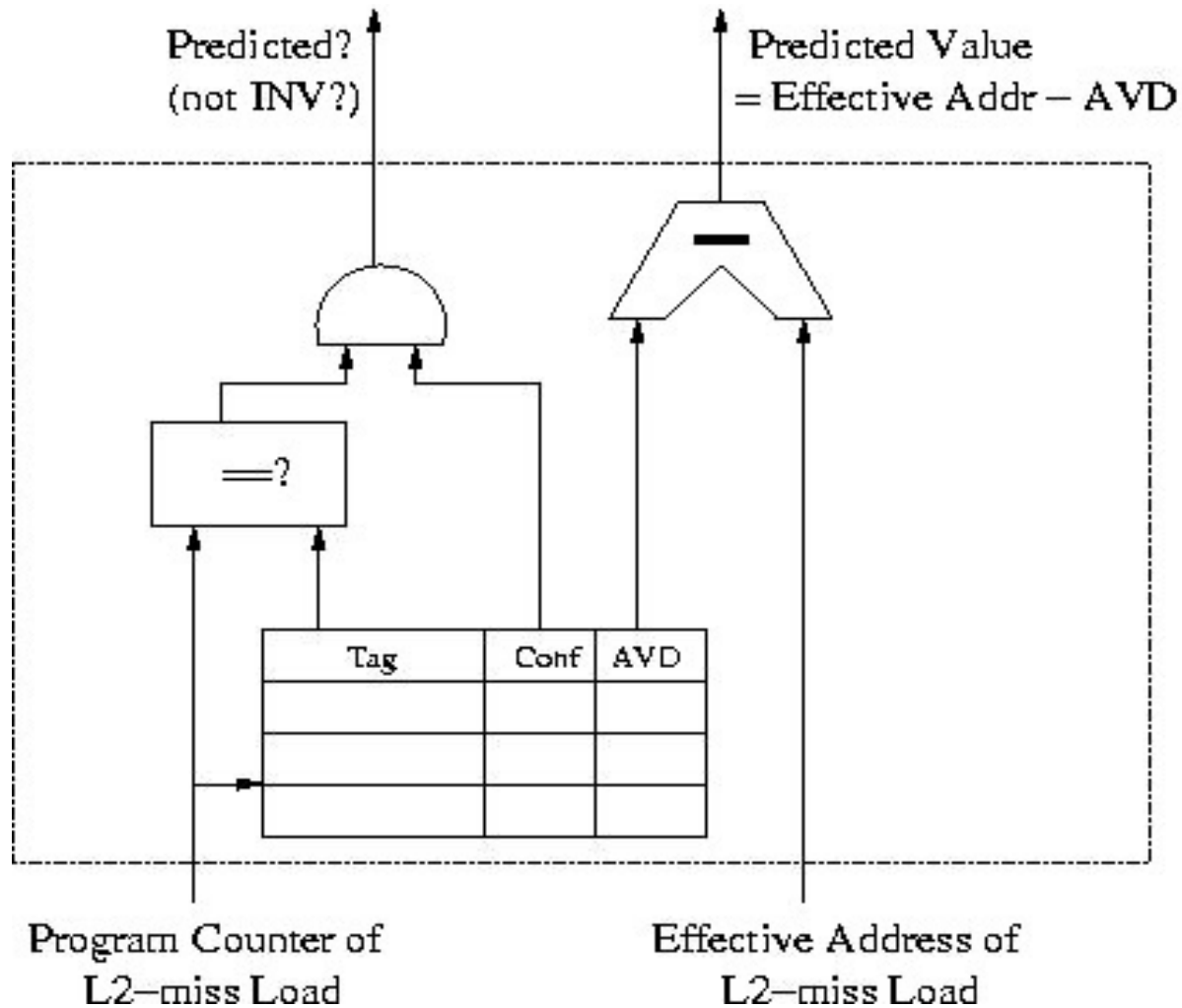
---

- Set-associative prediction table
- Prediction table entry consists of
  - Tag (Program Counter of the load)
  - Last AVD seen for the load
  - Confidence counter for the recorded AVD
- Updated when an address load is retired in normal mode
- Accessed when a load misses in L2 cache in runahead mode
- **Recovery-free:** No need to recover the state of the processor or the predictor on misprediction
  - **Runahead mode is purely speculative**

# AVD Update Logic

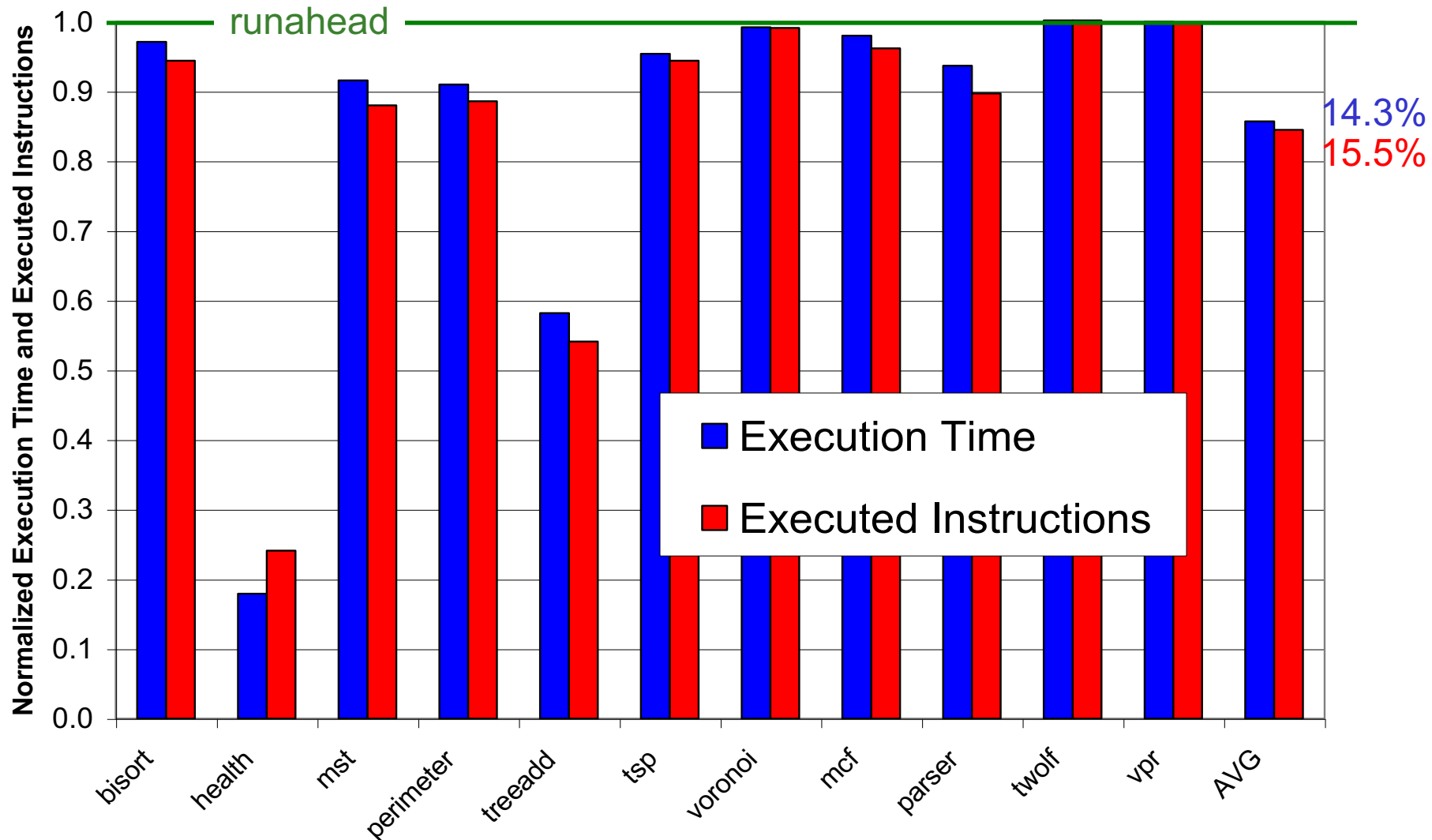


# AVD Prediction Logic





# Performance of AVD Prediction



# More on AVD Prediction

---

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

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

Onur Mutlu   Hyesoon Kim   Yale N. Patt

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

# More on AVD Prediction (II)

---

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

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

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

# Wrong Path Events

# An Observation and A Question

- In an out-of-order processor, some instructions are executed on the mispredicted path (wrong-path instructions).
- Is the behavior of wrong-path instructions different from the behavior of correct-path instructions?
  - If so, we can use the difference in behavior for early misprediction detection and recovery.

# What is a Wrong Path Event?

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

Wrong Path Event = WPE

**Probability (wrong path | WPE)  $\sim 1$**

# Why Does a WPE Occur?

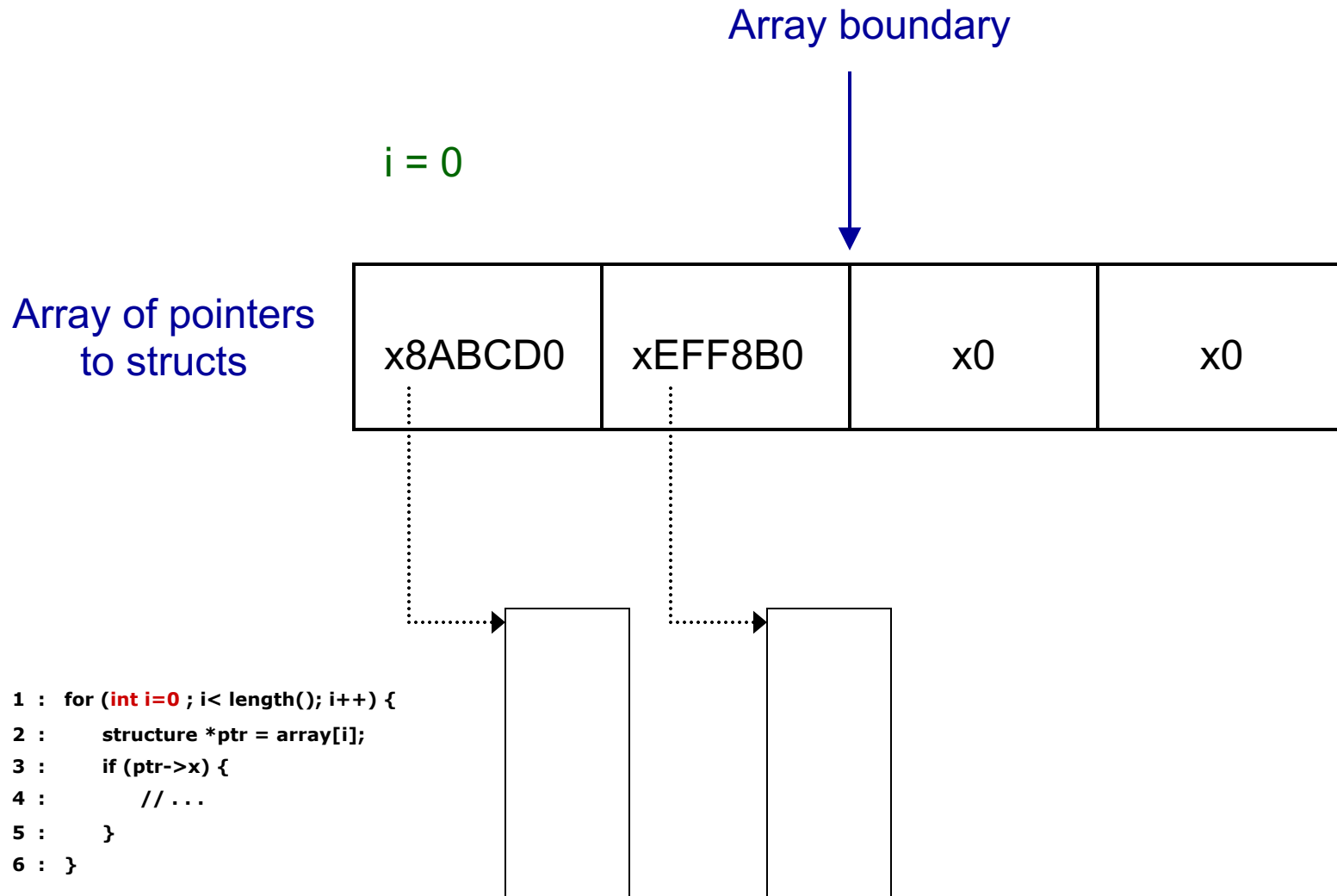
- A wrong-path instruction may be executed *before* the mispredicted branch is executed.
  - Because the mispredicted branch may be dependent on a long-latency instruction.
- The wrong-path instruction may consume a data value that is not properly initialized.

# WPE Example from *eon*: NULL pointer dereference

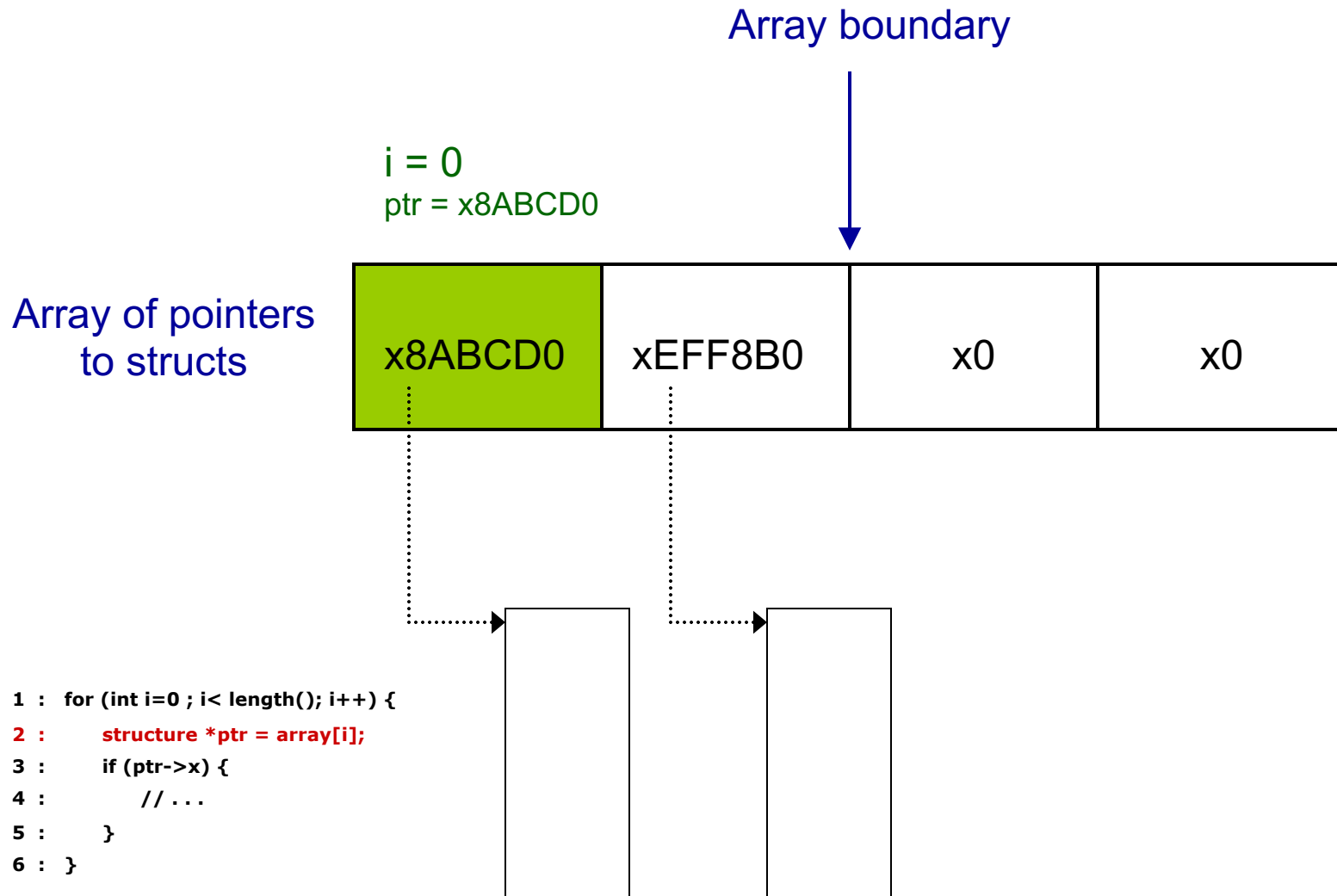
```
1 : for (int i=0 ; i< length(); i++) {  
2 :     structure *ptr = array[i];  
3 :     if (ptr->x) {  
4 :         // ...  
5 :     }  
6 : }
```



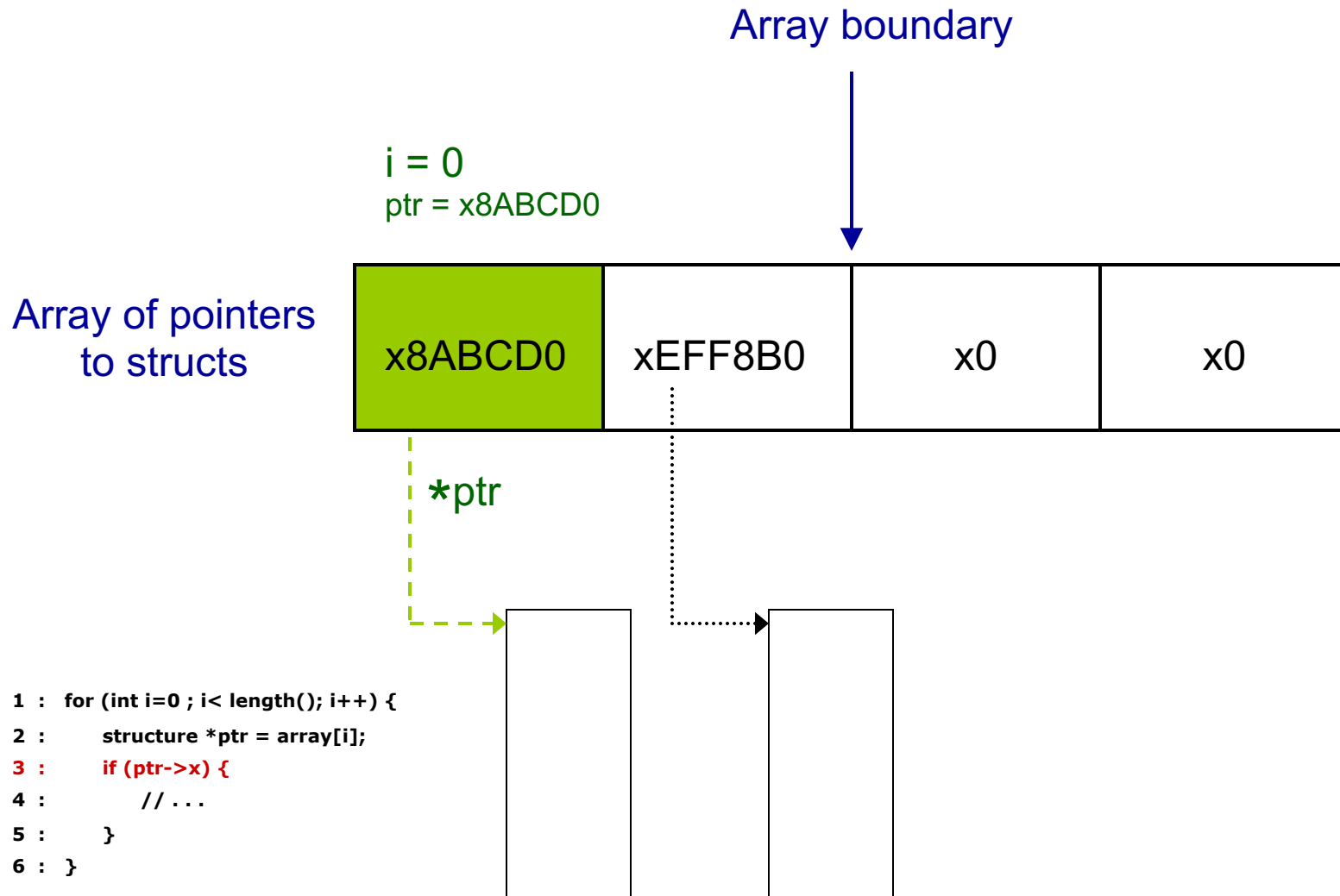
# Beginning of the loop



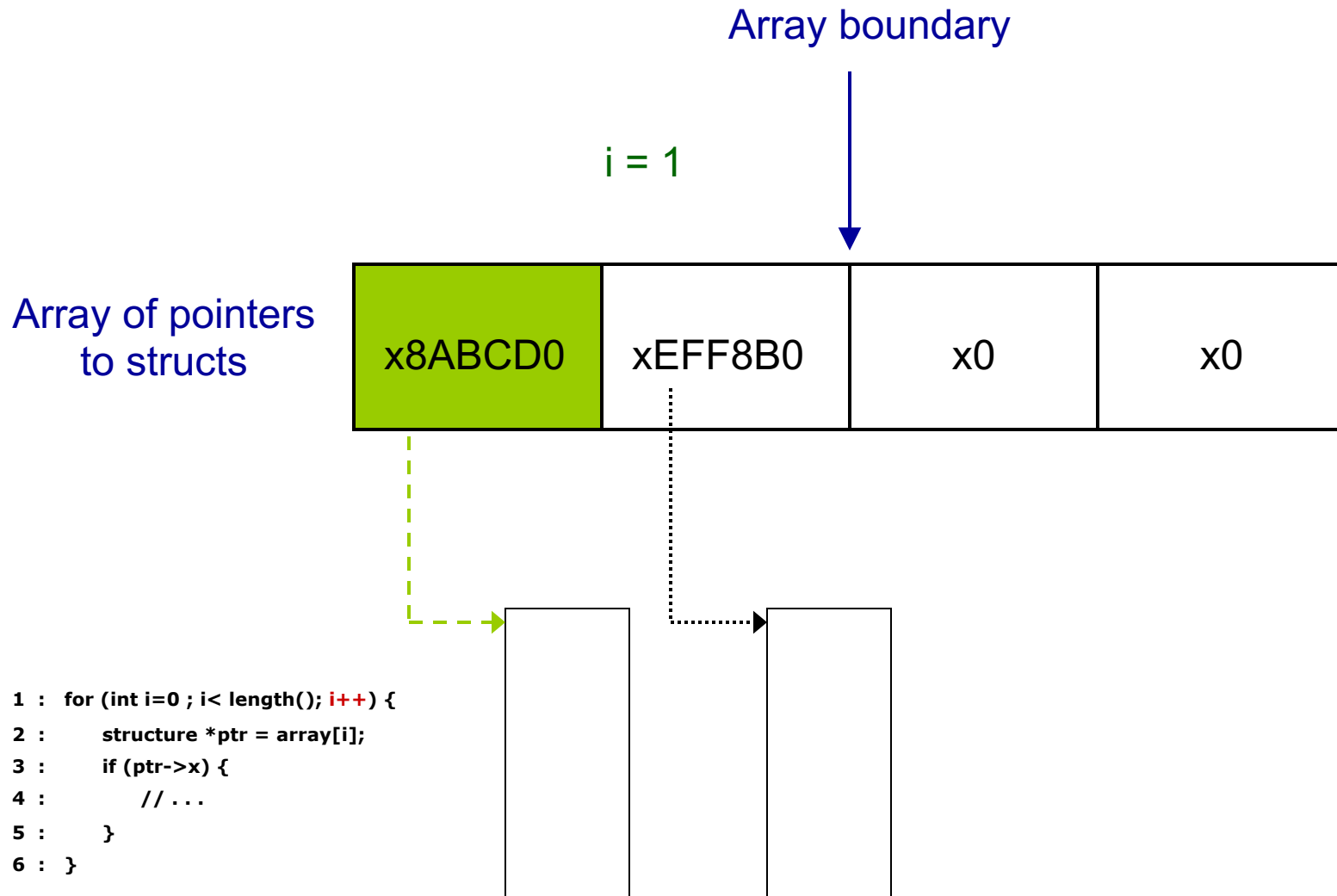
# First iteration



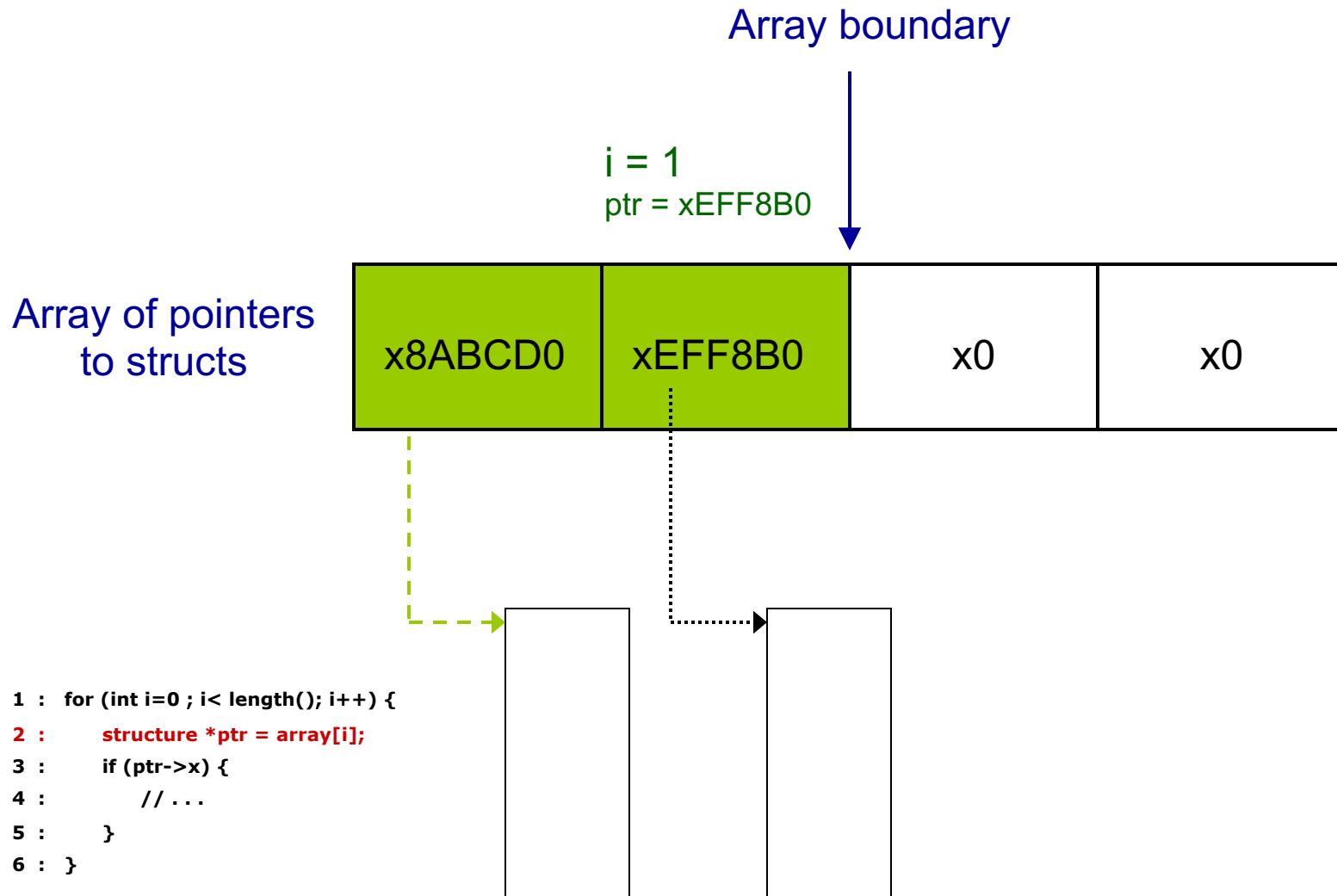
# First iteration



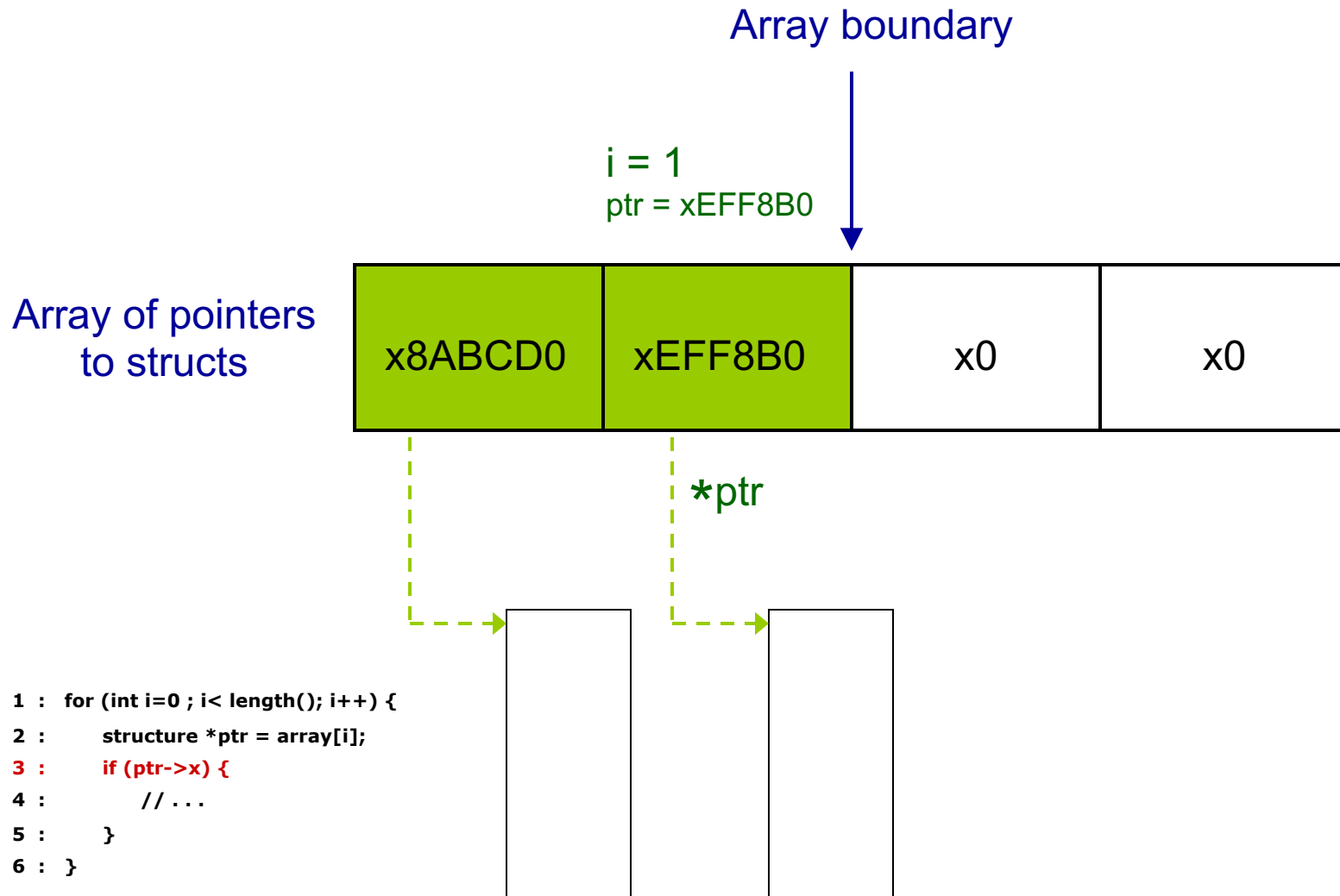
# Loop branch correctly predicted



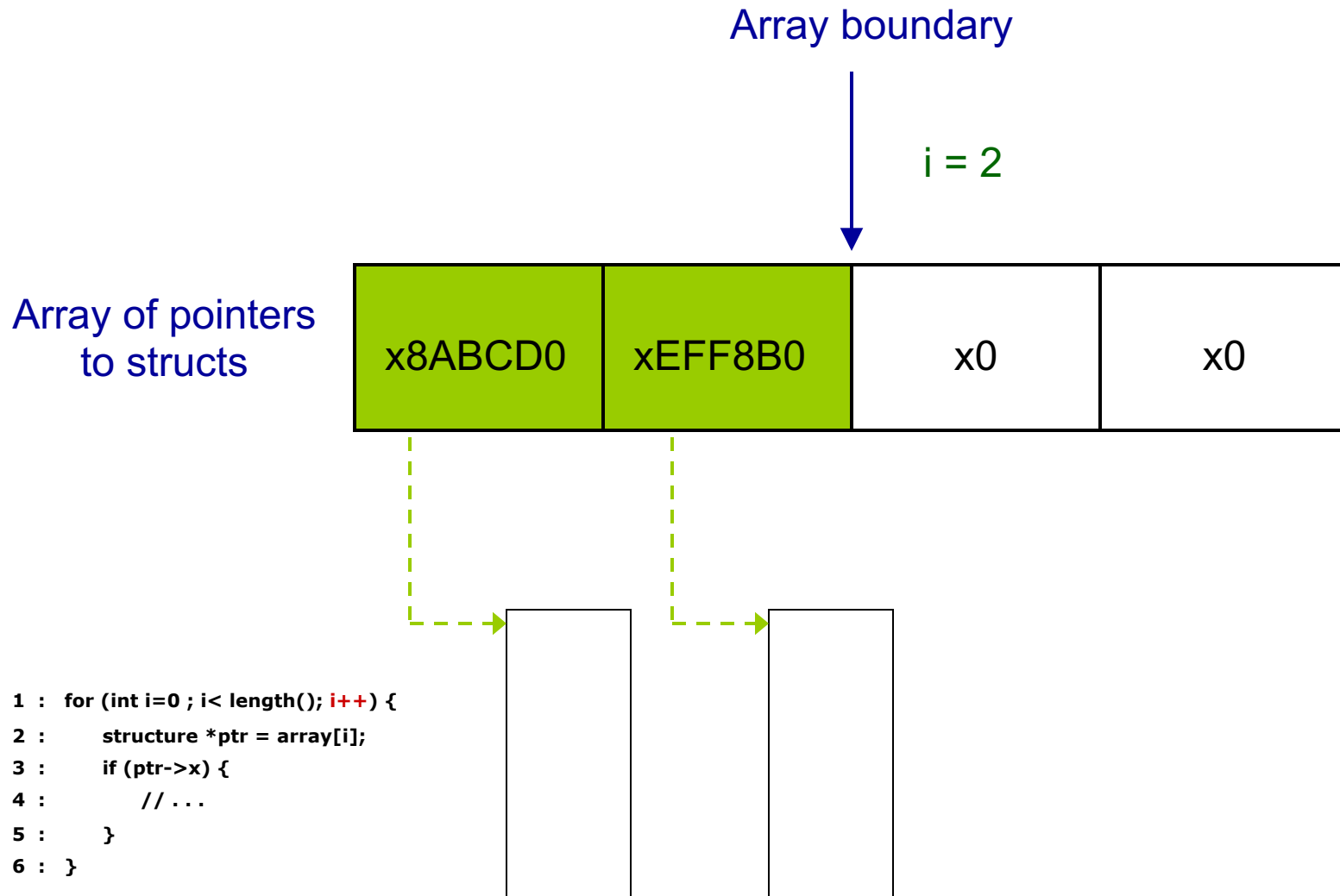
# Second iteration



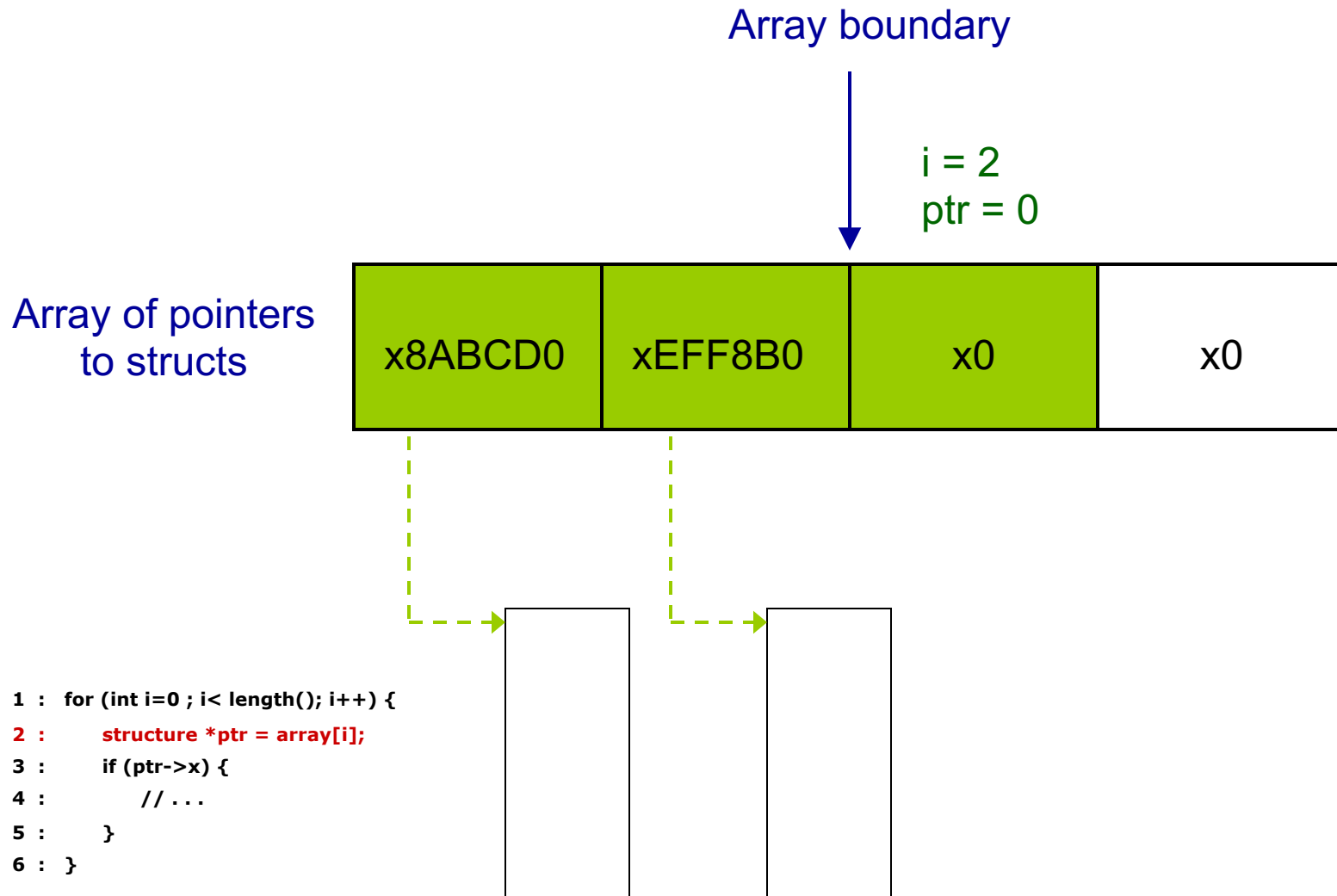
# Second iteration



# Loop exit branch mispredicted

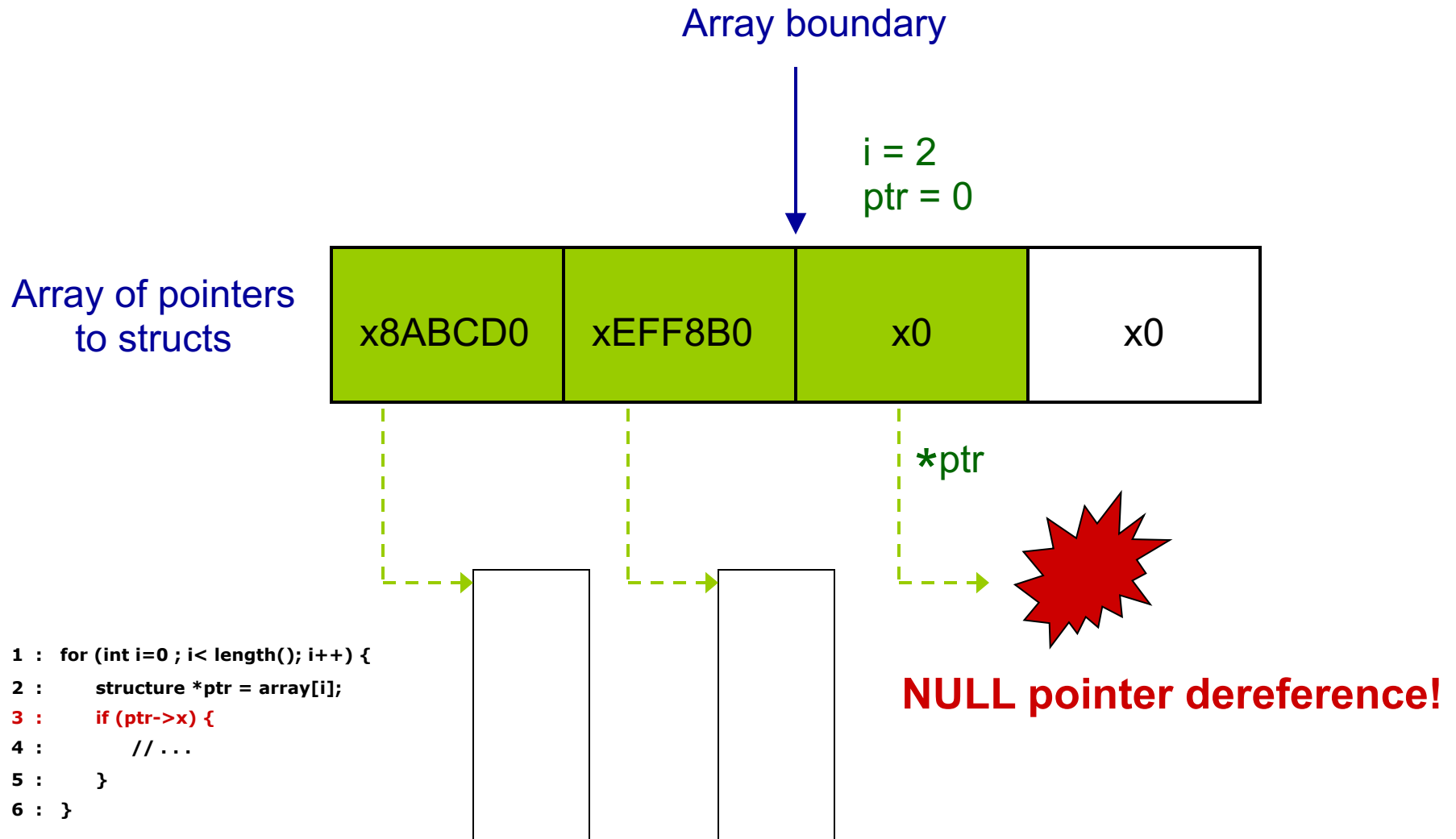


# Third iteration on wrong path





# Wrong Path Event



# Types of WPEs

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

# Types of WPEs (continued)

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

# Two Empirical Questions

1. How often do WPEs occur?
2. When do WPEs occur on the wrong path?

# More on Wrong Path Events

---

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

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

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

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

# Why Is This Important?

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

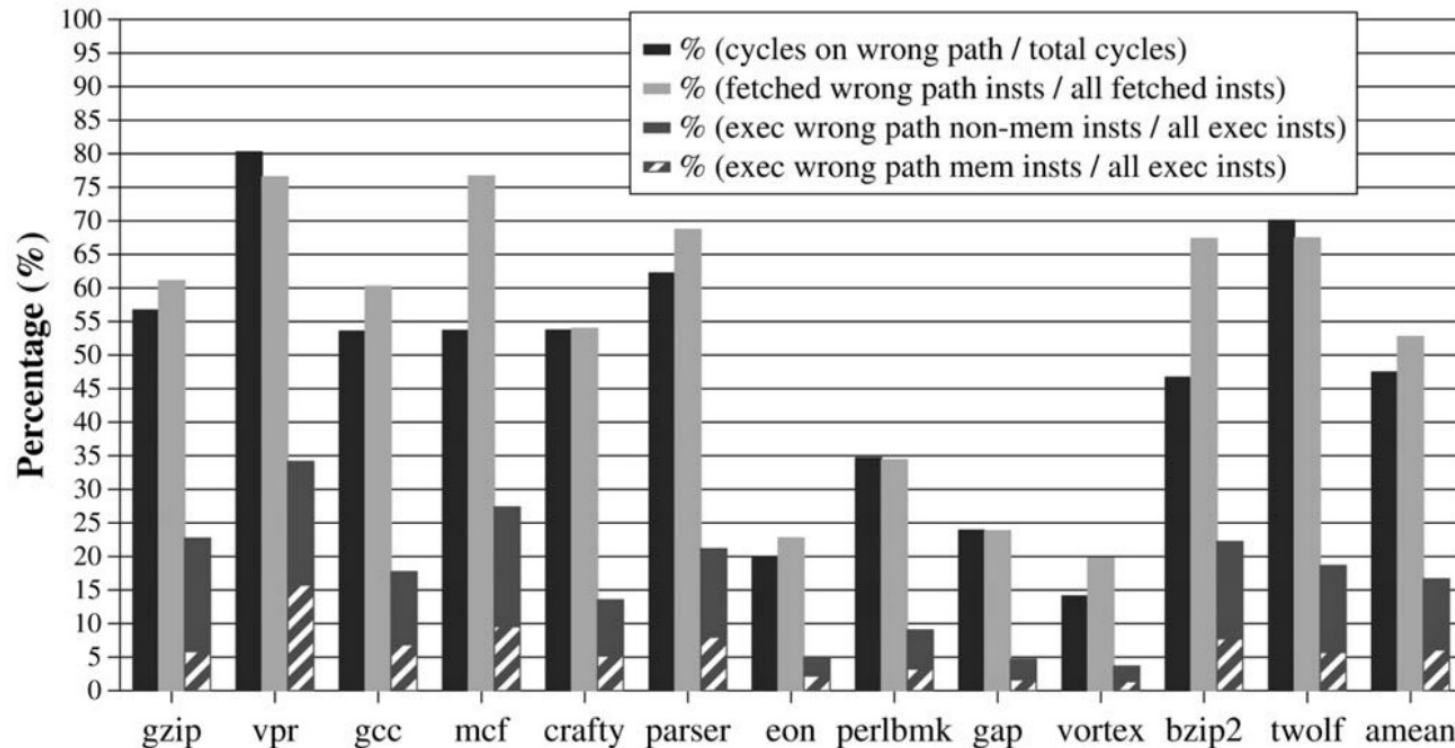


Fig. 1. Percentage of fetch cycles spent on the wrong path, percentage of instructions fetched on the wrong path, and percentage of instructions (memory and nonmemory) executed on the wrong path in the baseline processor for SPEC 2000 integer benchmarks.

# A Lot of Time Spent on The Wrong Path

- A runahead processor, much more so...

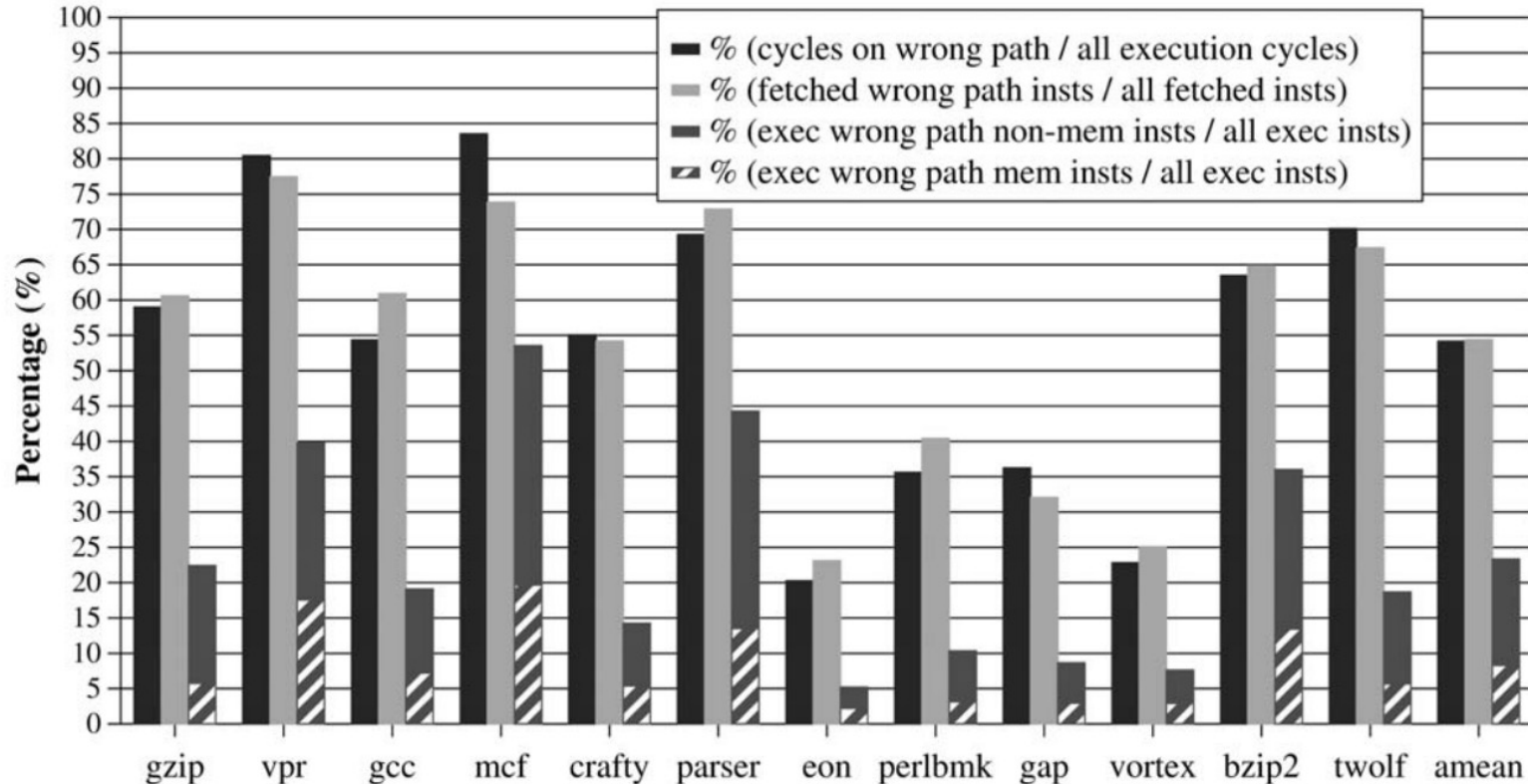


Fig. 20. Percentage of total cycles spent on the wrong path, percentage of instructions fetched on the wrong path, and percentage of instructions (memory and nonmemory) executed on the wrong path in the runahead processor.

# Is Wrong-Path Execution Useless/Useful/Harmful?

---

## 4 WRONG PATH: TO MODEL OR NOT TO MODEL

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

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



# Wrong Path Is Often Useful for Performance

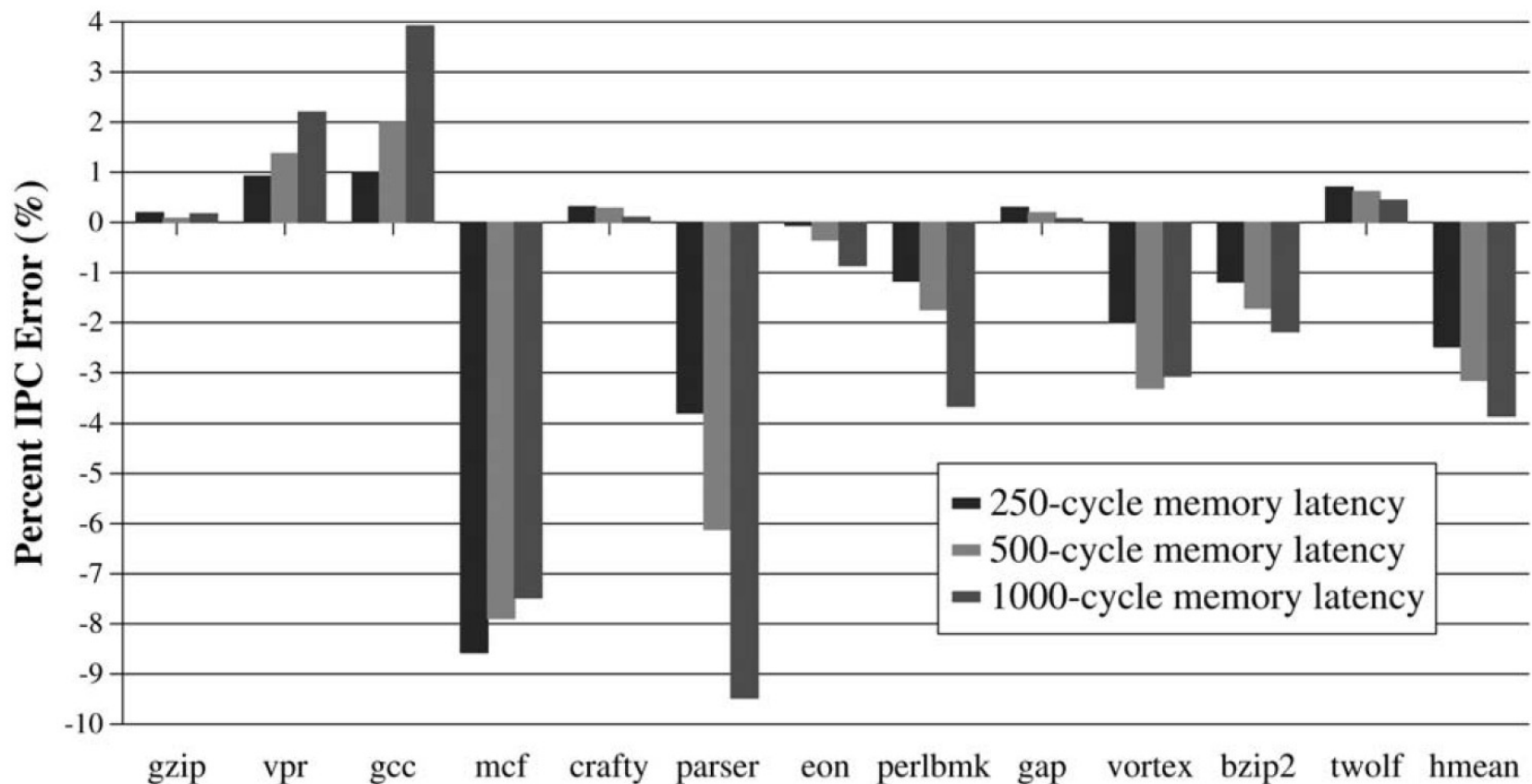


Fig. 7. Error in the IPC of the baseline processor with a stream prefetcher for three different memory latencies if wrong-path memory references are not simulated.

# More So In Runahead Execution

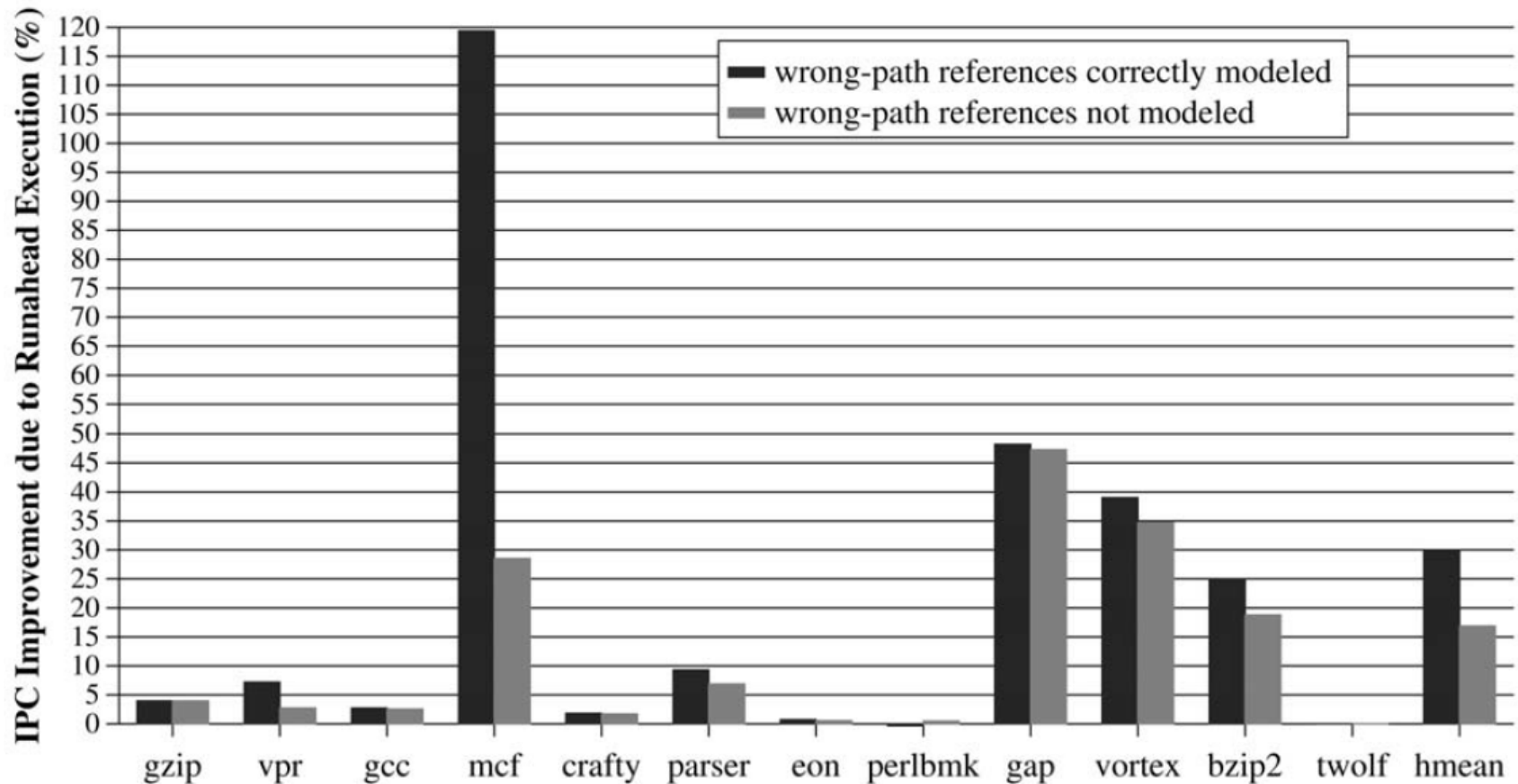


Fig. 19. IPC improvement of adding runahead execution to the baseline processor if wrong-path memory references are or are not modeled.

# Why is Wrong Path Useful? (I)

---

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

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

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

# Why is Wrong Path Useful? (II)

---

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

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

# Why is Wrong Path Useful? (III)

---

- Same data used in different control flow paths

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

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

# More on Wrong Path Execution (I)

---

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

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

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

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

# More on Wrong Path Execution (II)

---

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

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

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

# What If ...

---

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