# Computer Architecture

Lec 8: Prefetching

# Outline of Prefetching Issues

- Why prefetch? Why could/does it work?
- The four questions
  - What (to prefetch), when, where, how
- Software prefetching algorithms
- Hardware prefetching algorithms
- Execution-based prefetching techniques and algorithms
- Prefetching performance
  - Coverage, accuracy, timeliness
  - Bandwidth consumption, cache pollution
- …

# 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

# 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

# Some Predictable Address Access Patterns?

- Cache Block Addresses

A, A+1, A+2, A+3, A+4, …

B, B+42, B+84, B+126, …

C, C+2, C+5, C+9, C+11, C+14, C+18, C+20, C+23, C+27, …

X, Y, T, Q, R, S, X, Y, T, A, B, C, D, E, X, Y, T, F, G, H, X, Y, T, …

A+1, A+67, A+18, A+7, A+99, Z+1, Z+67, Z+18, Z+7, Z+99,
P+1, P+67, P+18, P+7, P+99, …

# 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)

- Which level of cache to prefetch into?
  - Memory to L4/L3/L2, memory to L1. Advantages/disadvantages?
  - L3 to L2? 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 (II)

- Where to place the hardware prefetcher in the memory hierarchy?
  - In other words, what access patterns does the prefetcher see?
  - L1 hits and misses
  - L1 misses only
  - L2 misses only

- Seeing a more complete access pattern:

  + Potentially better accuracy and coverage in prefetching

  -- Prefetcher needs to examine more requests (bandwidth intensive, more ports into the prefetcher?)

# Challenges in Prefetching: How

- Software prefetching
  - ISA provides prefetch instructions
  - Programmer or compiler inserts prefetch instructions into code
  - 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 prefetching
  - A "thread" is executed to prefetch data for the main program
  - Can be generated by either software/programmer or hardware

# Prefetcher Performance Metrics

- 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 exactly quantify, but affects performance

# Software Prefetching

# Software Prefetching (I)

```
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

# Software Prefetching (II)

- 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

# X86 PREFETCH Instruction

## PREFETCH*h*—Prefetch Data Into Caches

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

# Hardware Prefetching

# Hardware Prefetching

- 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?

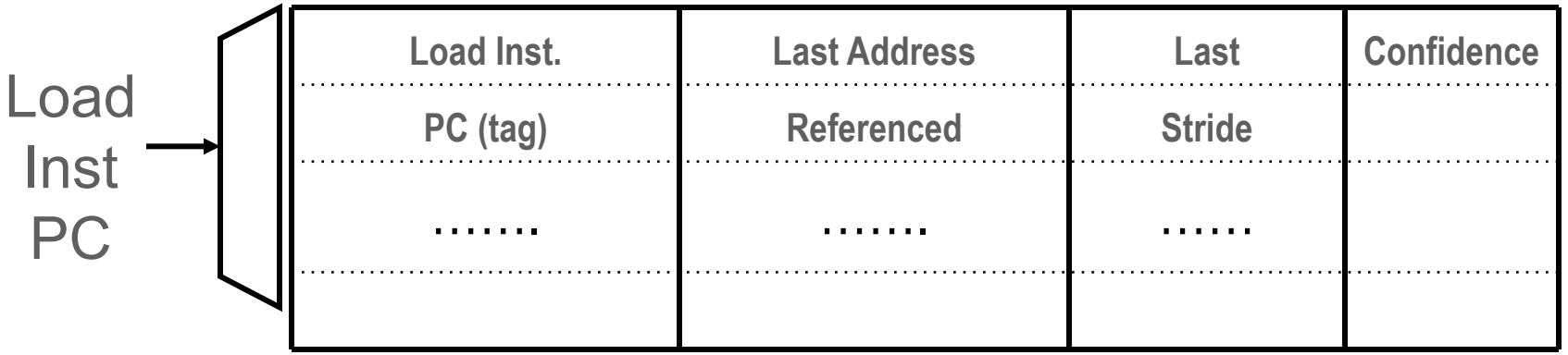
# Stride Prefetchers

- Consider the following strided memory access pattern:
  - A, A+N, A+2N, A+3N, A+4N…
  - 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

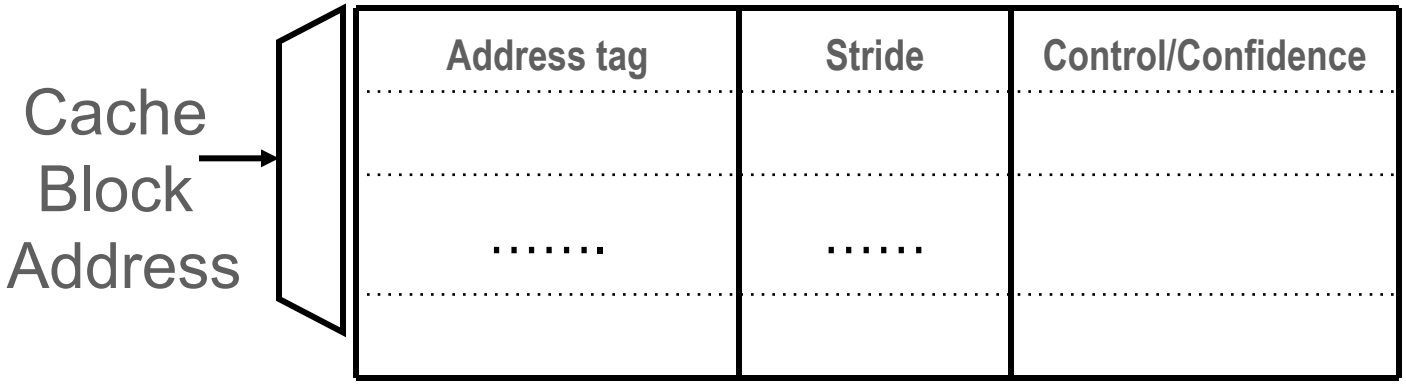
# Instruction-Based Stride Prefetching




- Each load/store instruction can lead to a memory access pattern with a different stride
  - Can 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 … 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

# 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 is 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

Baer & Chen, "An effective on-chip preloading scheme to reduce data access penalty," SC 1991.

# 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 ($A_n - A_{n-1}$ = 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.

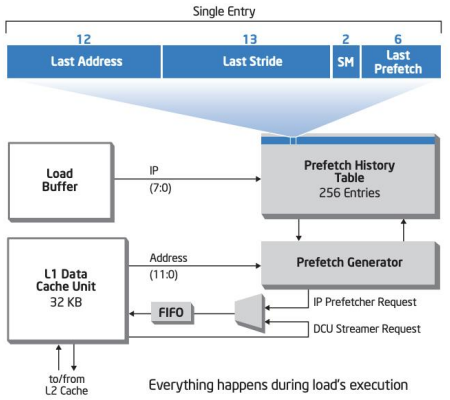



**Figure 3:** High level block diagram of the relevant parts in the Intel Core microarchitecture IP prefetcher system.

https://www.all-electronics.de/wp-content/uploads/migrated/document/196371/413ei0507-intel-sma.pdf

# Memory-Region-Based 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

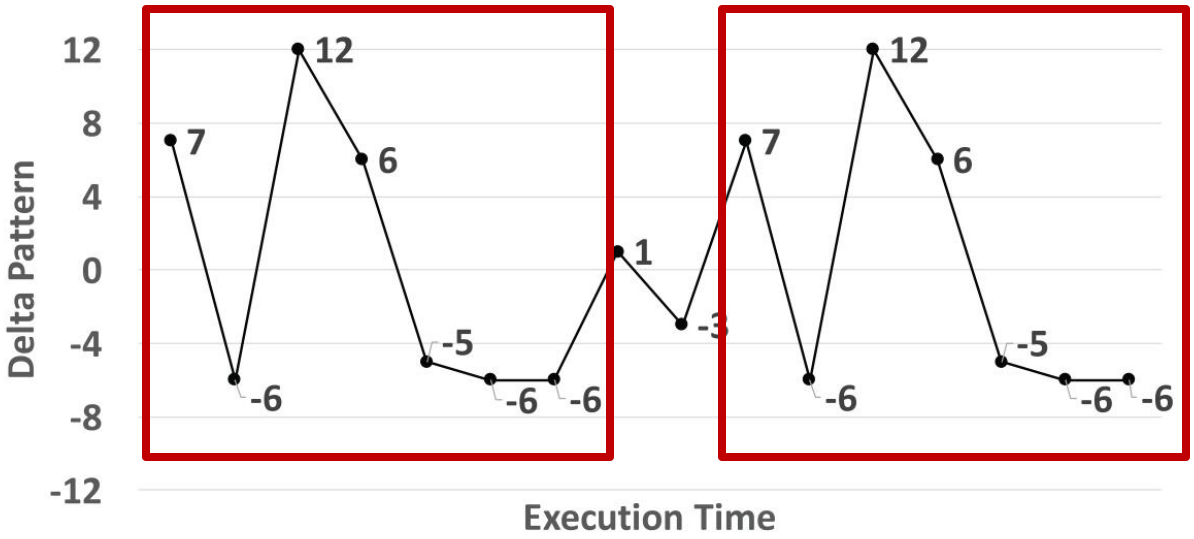
This paper investigates new hardware techniques for increasing the performance of the memory hierarchy. Section 2 describes a baseline design using conventional caching techniques. The large performance loss due to the memory hierarchy is a detailed motivation for the techniques discussed in the remainder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

Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," ISCA 1990.

# What About More Complex Access Patterns?

- Simple regular patterns
  - Stride 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

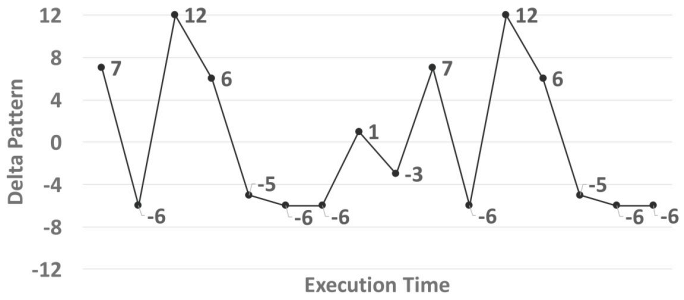
Jinchun Kim*, Seth H. Pugsley†, Paul V. Gratz*, A. L. Narasimha Reddy*, Chris Wilkerson† and Zeshan Chishti†

*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, record and predict what stride might come next
    - {7,-6,12} → 6, {-6,12,6} → -5, …




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

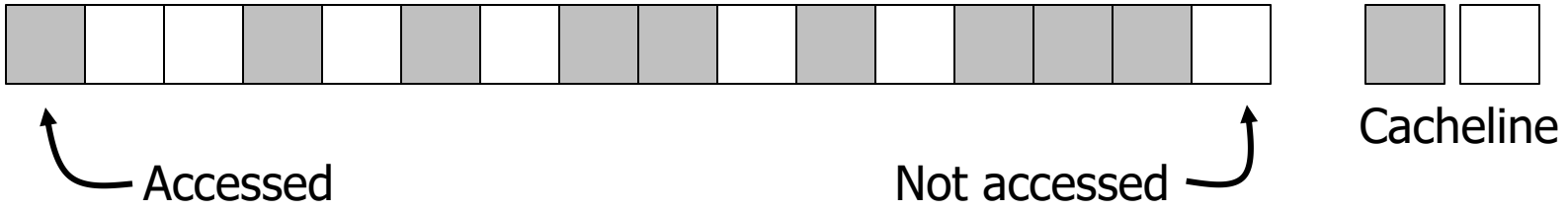
| | History of Strides | Prediction | Prediction Confidence | Path Confidence | |
|---|---|---|---|---|---|
| Pass 1 | {7,-6,12} | 6 | 85% | 85% | Bootstrap |

# Spatial Memory Streaming (SMS)

- What happens when strides are not repeating?
  - Out-of-order memory accesses
  - Linked data structure traversals
  - …
- Key Idea:
  - Observe the access pattern not as a sequence of consecutive strides, but as a bit-pattern over a large spatial region (e.g., physical page)
  - Record and learn pattern repetitions

# Spatial Memory Streaming (SMS)

$PC_x$ accesses page $P_1$ with following bit-pattern




$PC_x$ accesses page $P_2$ with the same bit-pattern

Pattern learning: $PC_x$ →

# Spatial Memory Streaming (SMS)

**Spatial Memory Streaming**

Stephen Somogyi, Thomas F. Wenisch,
Anastassia Ailamaki, Babak Falsafi and Andreas Moshovos†
*http://www.ece.cmu.edu/~stems*

Computer Architecture Laboratory (CALCM)
Carnegie Mellon University

†Dept. of Electrical & Computer Engineering
University of Toronto

# Self-Optimizing Memory Prefetchers

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**

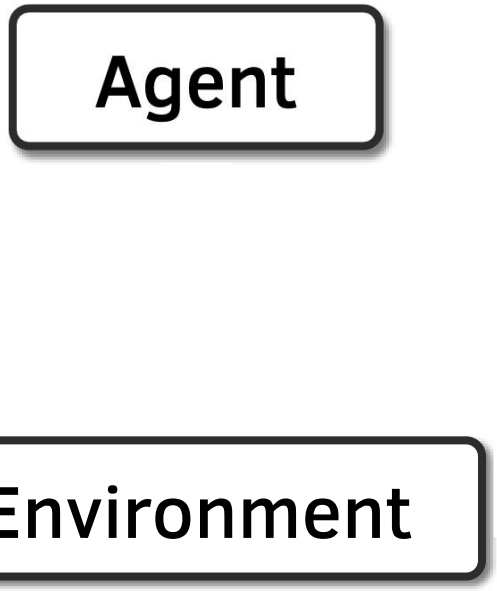
Rahul Bera[1] Konstantinos Kanellopoulos[1] Anant V. Nori[2] Taha Shahroodi[3,1]
Sreenivas Subramoney[2] Onur Mutlu[1]

[1]ETH Zürich [2]Processor Architecture Research Labs, Intel Labs [3]TU Delft

**https://arxiv.org/pdf/2109.12021.pdf**

# Basics of Reinforcement Learning (RL)

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




- 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

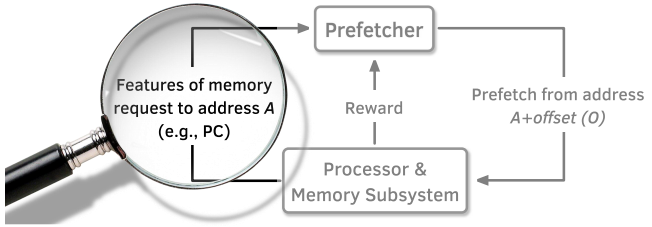
**Pythia formulates prefetching as a reinforcement learning problem**

# What is State?

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

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

- Feature = control-flow + data-flow

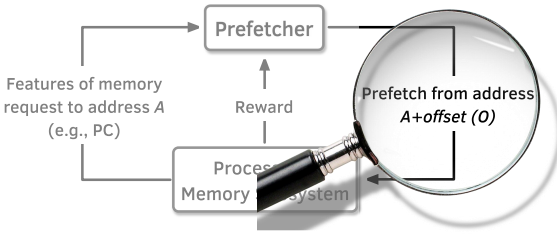



- **Control-flow examples**
  - PC
  - Branch PC
  - Last-3 PCs, …
- **Data-flow examples**
  - Cacheline address
  - Physical page number
  - Delta between two cacheline addresses
  - Last 4 deltas, …

# What is Action?

Given a demand access to address A

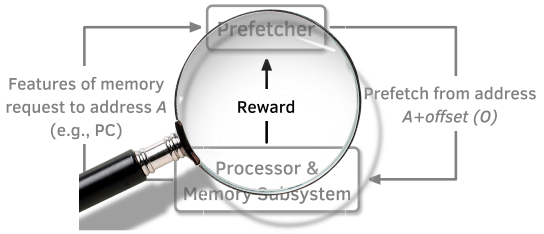
the action is to **select prefetch offset "O"**




- **Action-space**: 127 actions in the range [-63, +63]
  - For a machine with 4KB page and 64B cacheline
- 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

# Basic Pythia Configuration

- Derived from **automatic design-space exploration**

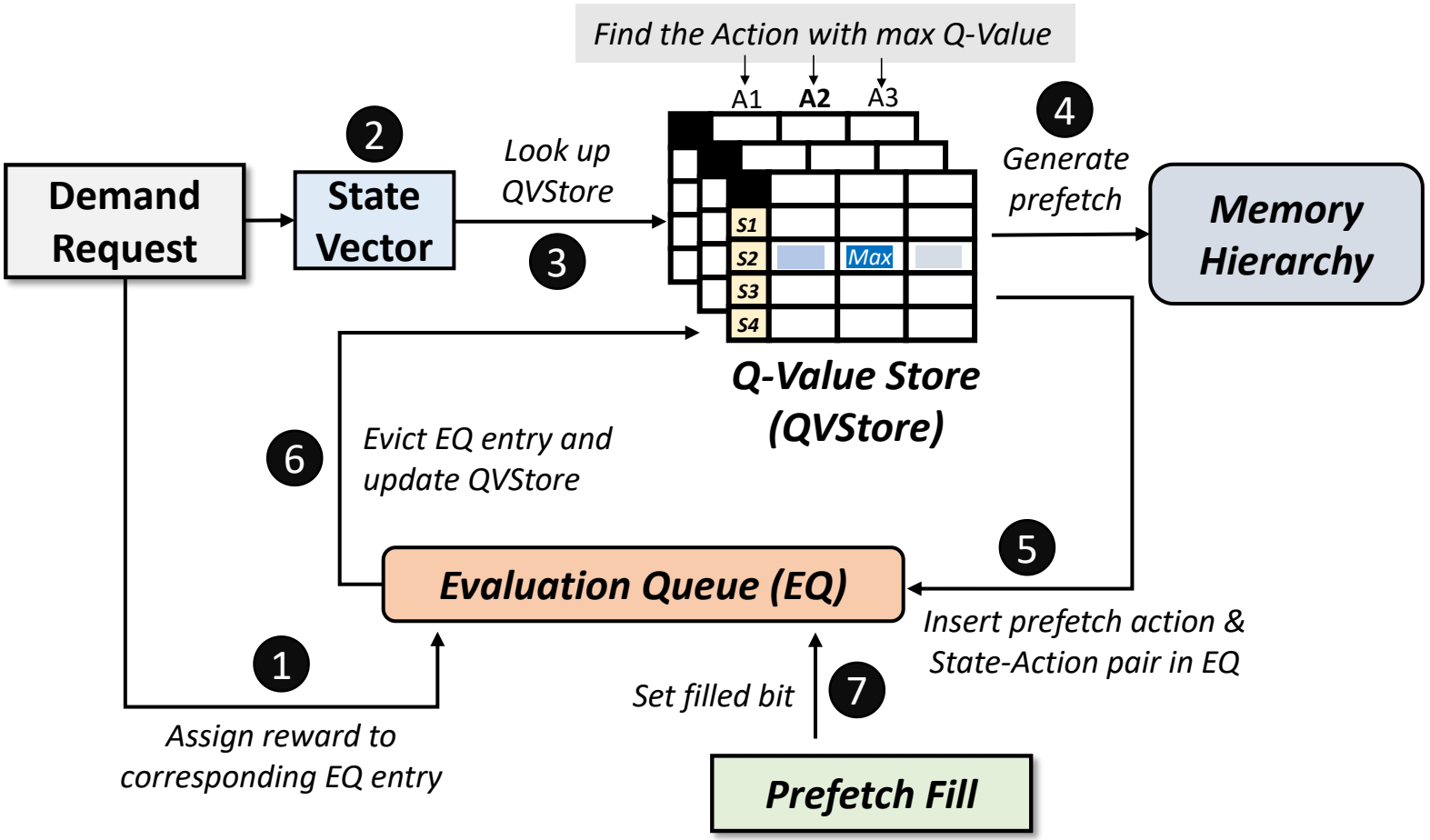
- **State:** 2 features
  - PC+Delta
  - Sequence of last-4 deltas

- **Actions:** 16 prefetch offsets
  - Ranging between -6 to +32. Including 0.

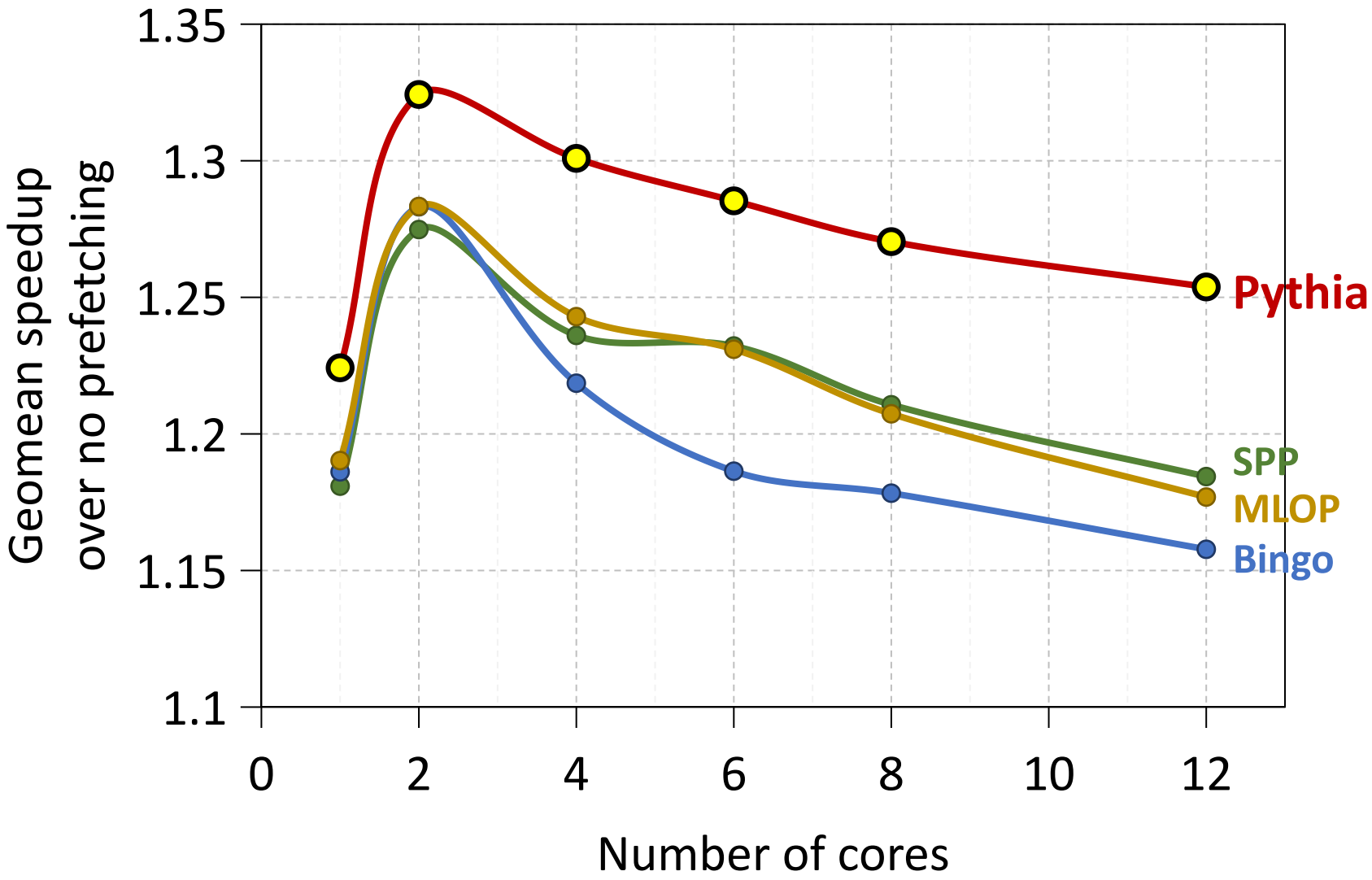
- **Rewards:**
  - $R_{AT}$ = +20; $R_{AL}$ = +12; $R_{NP}$-H=-2; $R_{NP}$-L=-4;
  - $R_{IN}$-H=-14; $R_{IN}$-L=-8; $R_{CL}$=-12

# More Detailed Pythia Overview

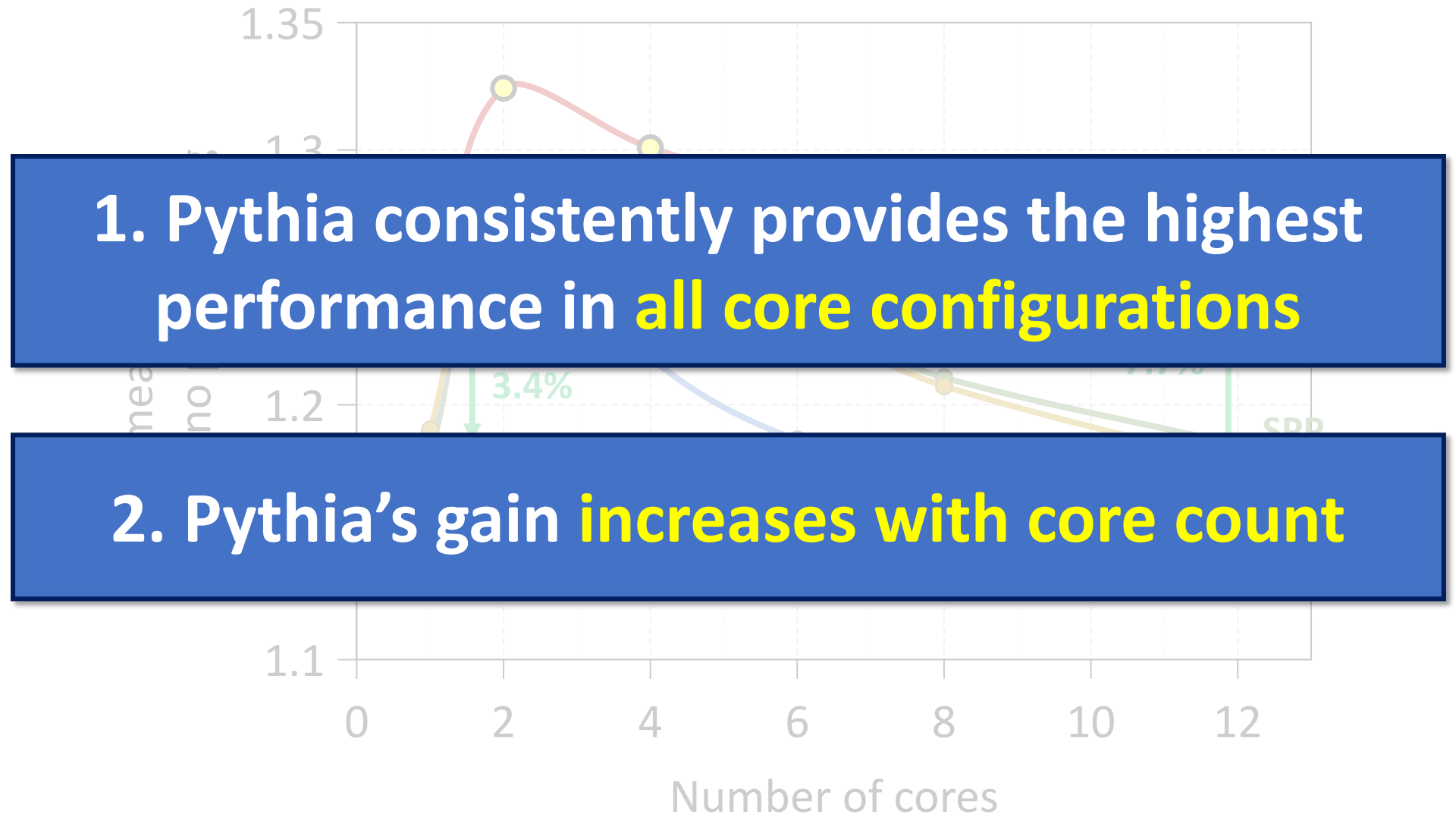
- **Q-Value Store**: Records Q-values for *all* state-action pairs
- **Evaluation Queue**: A FIFO queue of recently-taken actions



# Performance with Varying Core Count



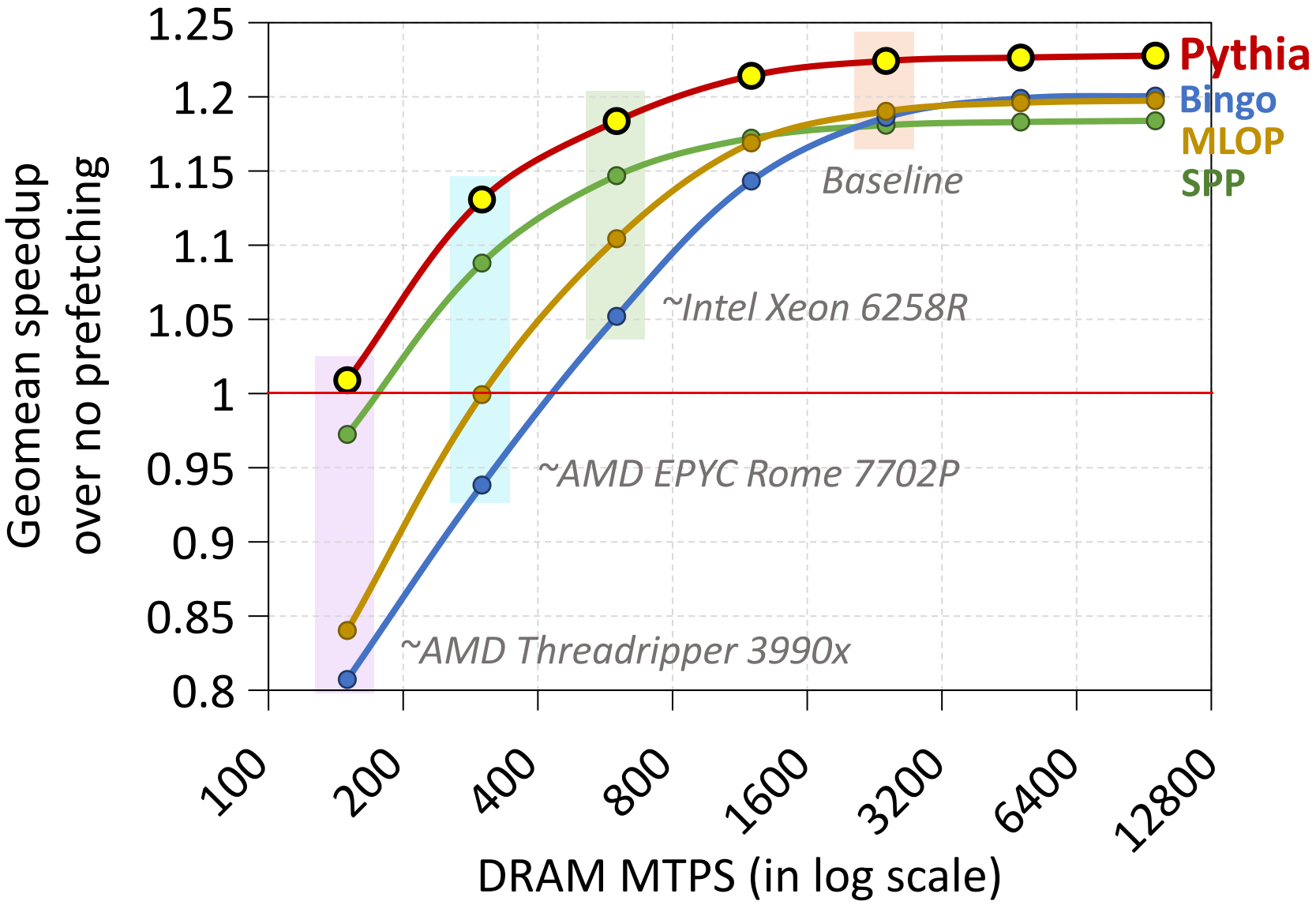
# Performance with Varying Core Count



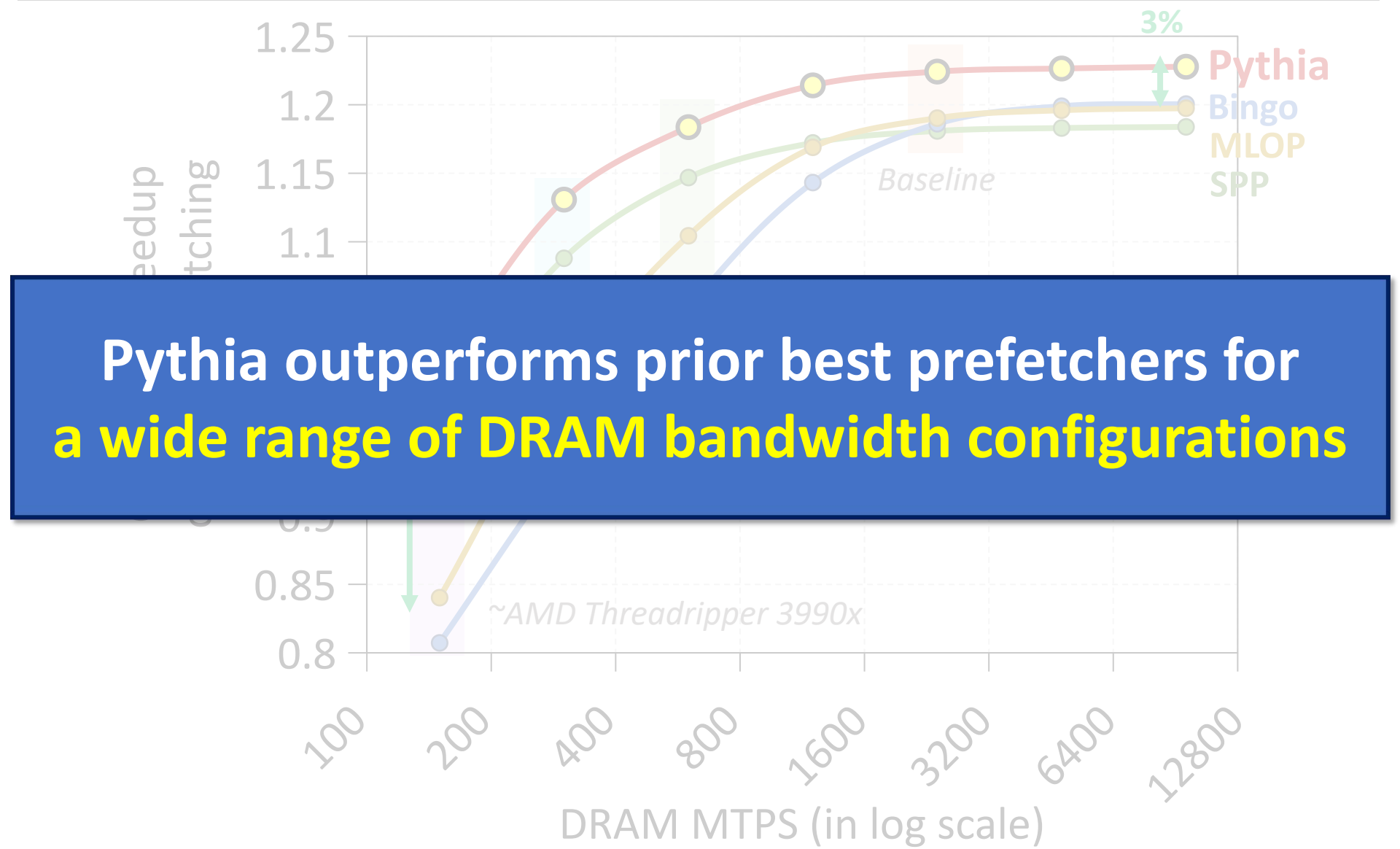

**1. Pythia consistently provides the highest performance in all core configurations**

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

# Performance with Varying DRAM Bandwidth



# Performance with Varying DRAM Bandwidth

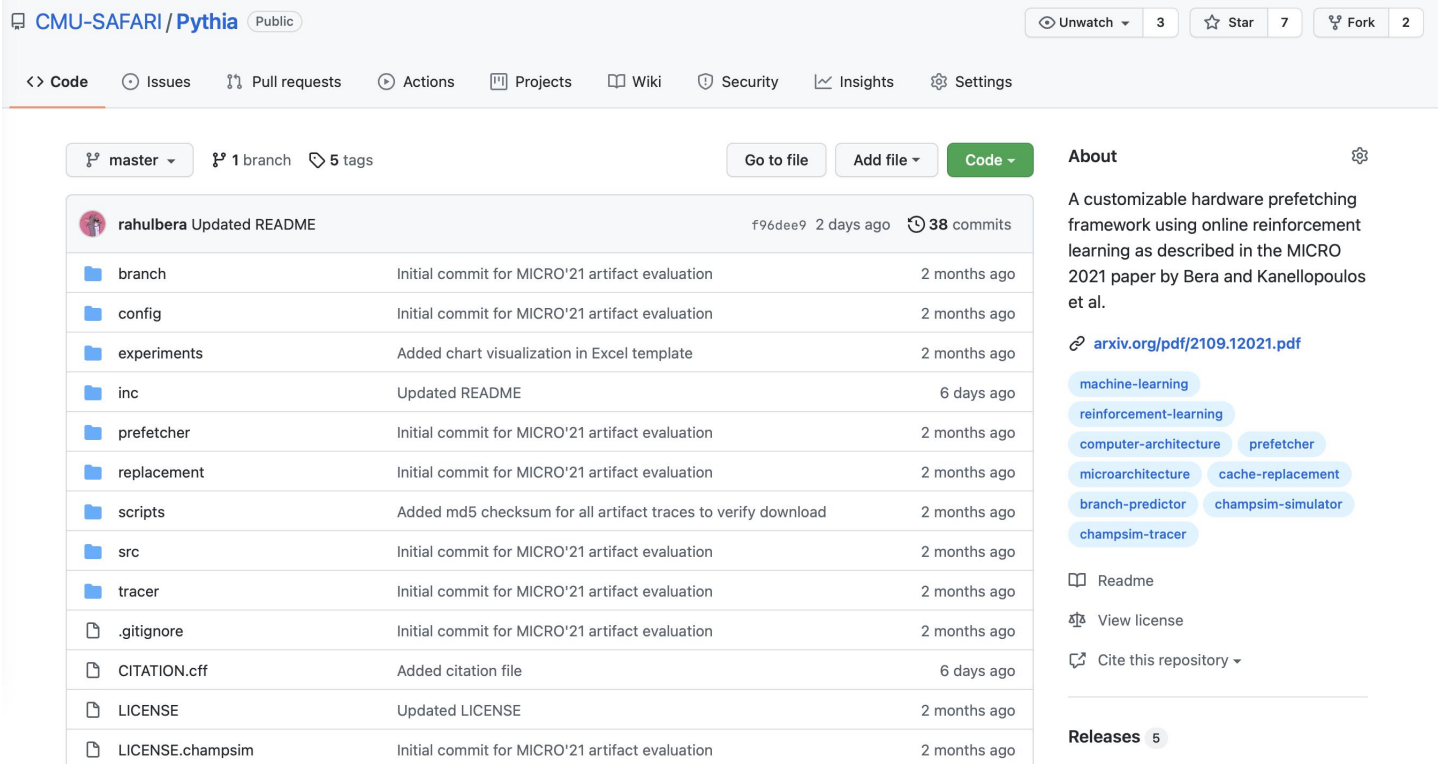



**Pythia outperforms prior best prefetchers for a wide range of DRAM bandwidth configurations**

# 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



# Real Systems: Hybrid Hardware Prefetchers

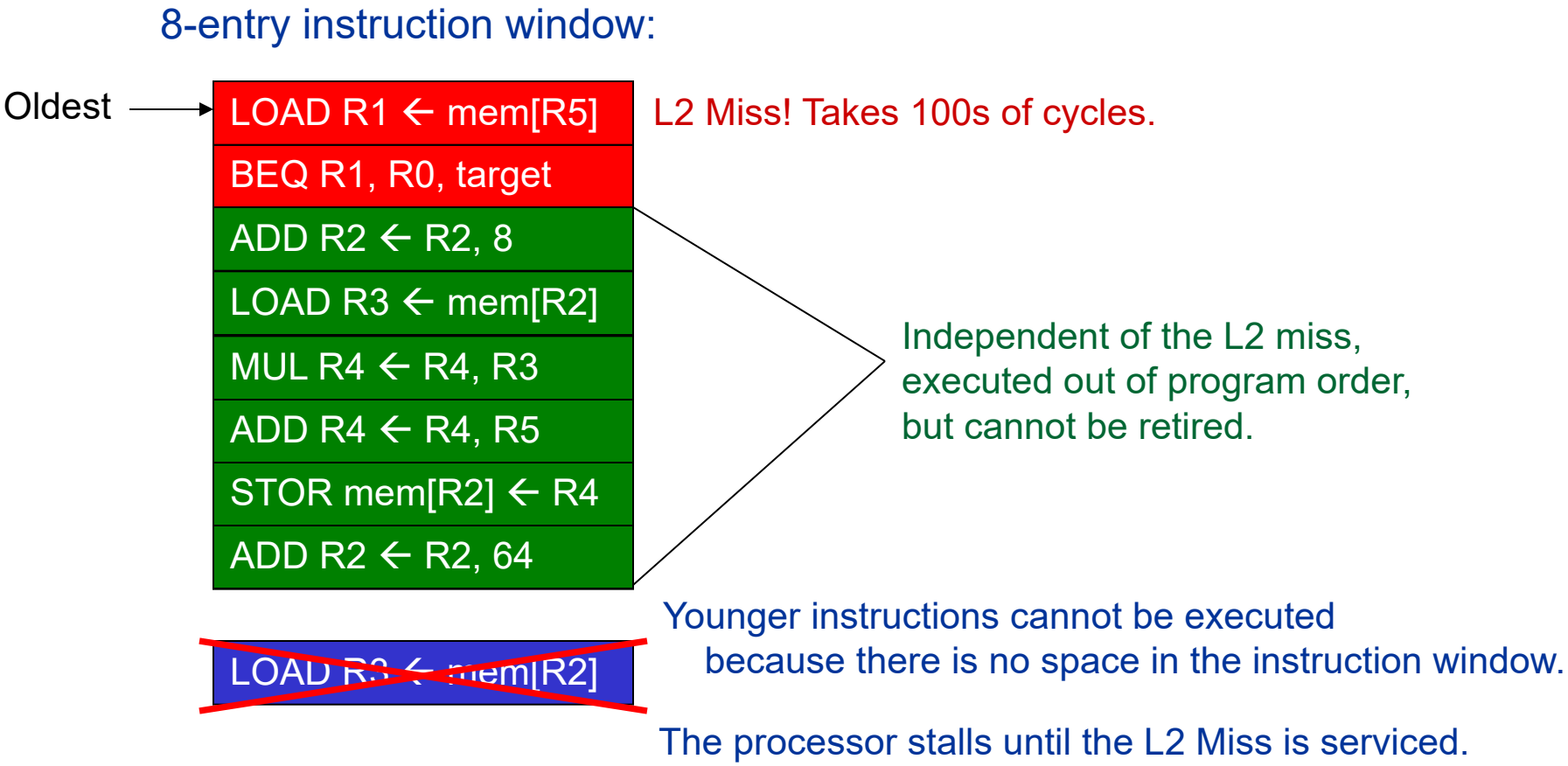
- Idea: Use multiple prefetchers to cover many memory access patterns

+ 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

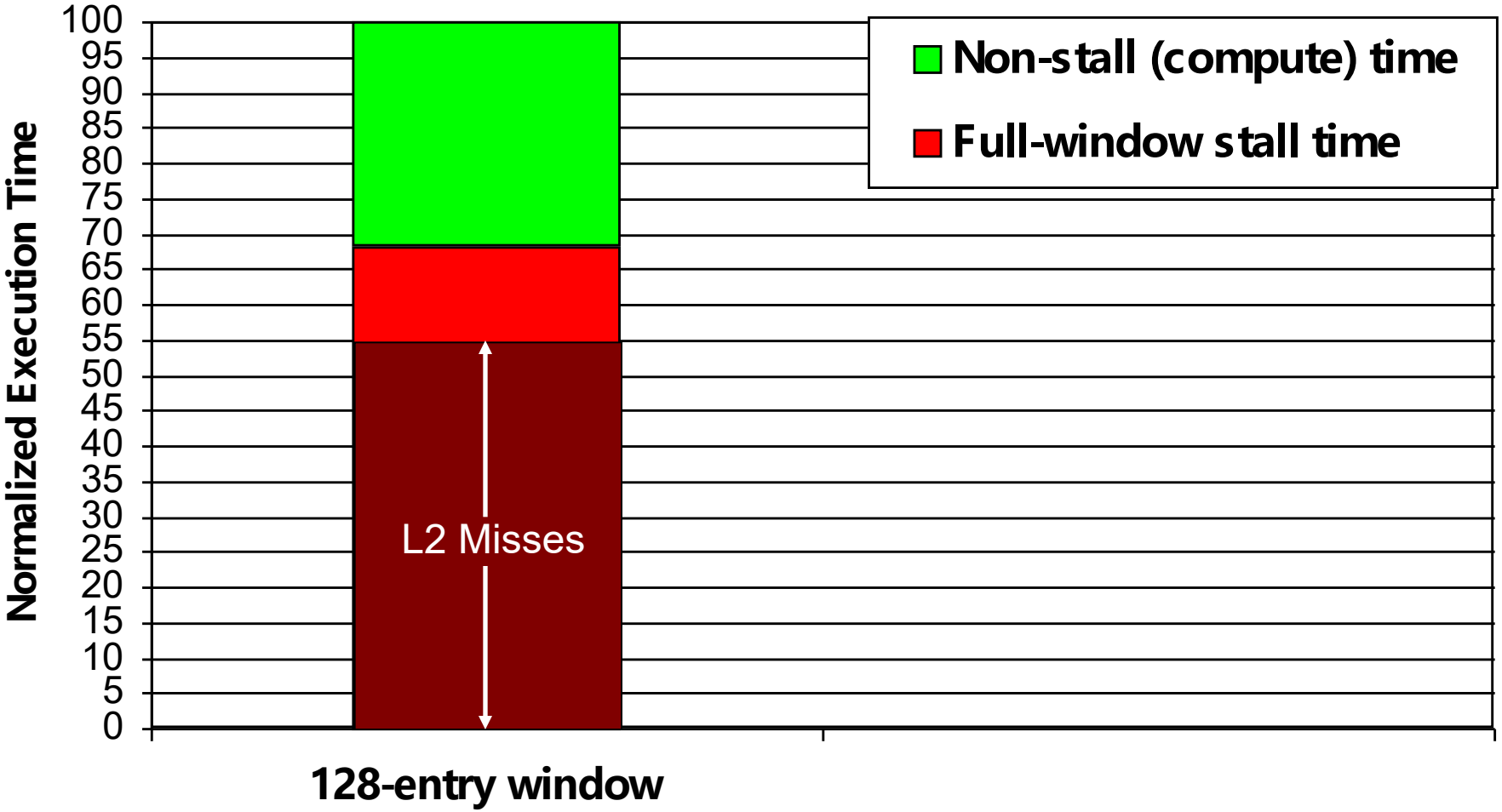
# Execution-Based Prefetcher: Runahead Execution

# Small Windows: Full-Window Stalls




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

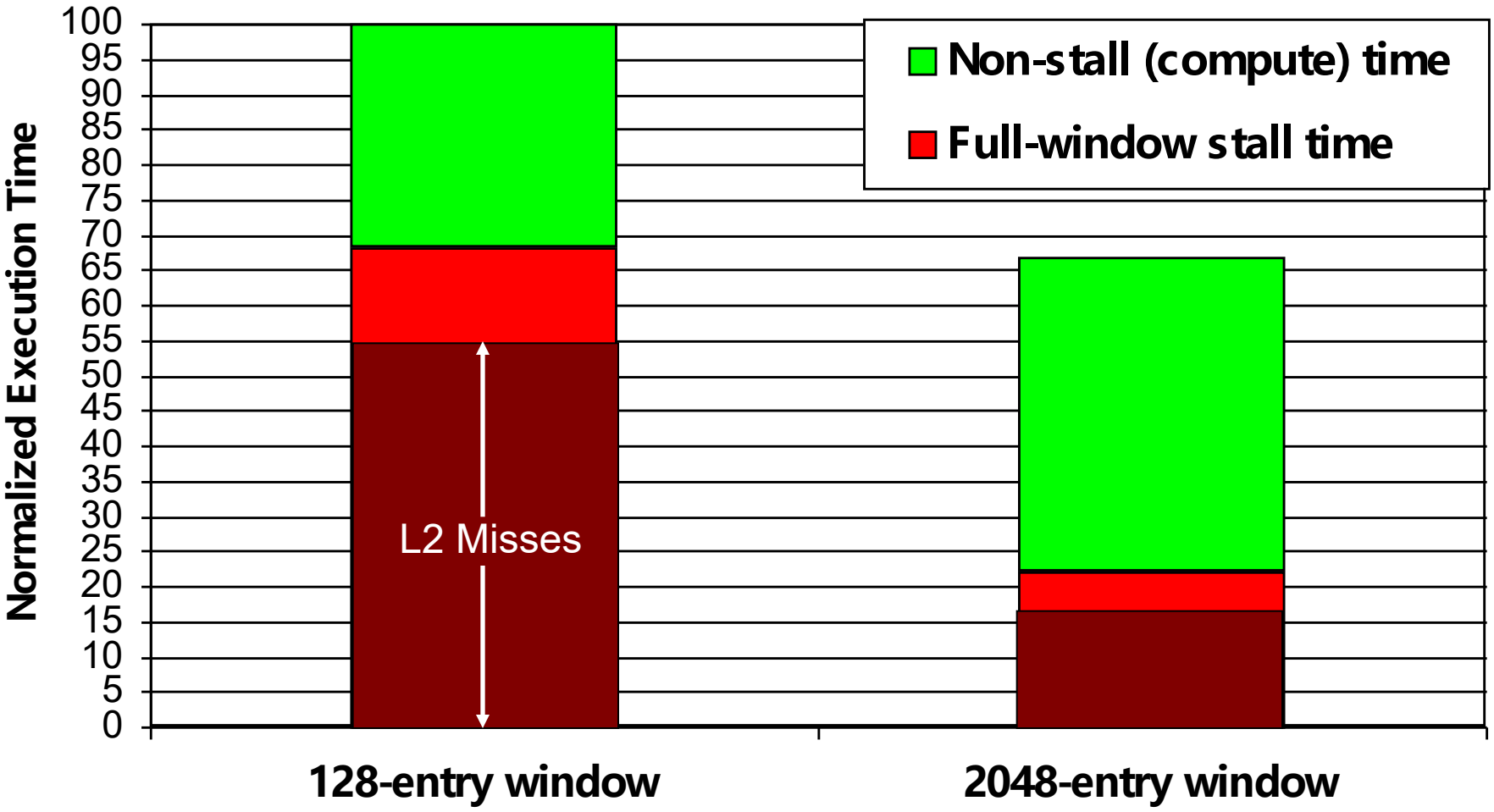
# Impact of Long-Latency Cache Misses




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

Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," HPCA 2003.

# 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

Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," HPCA 2003.

# 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

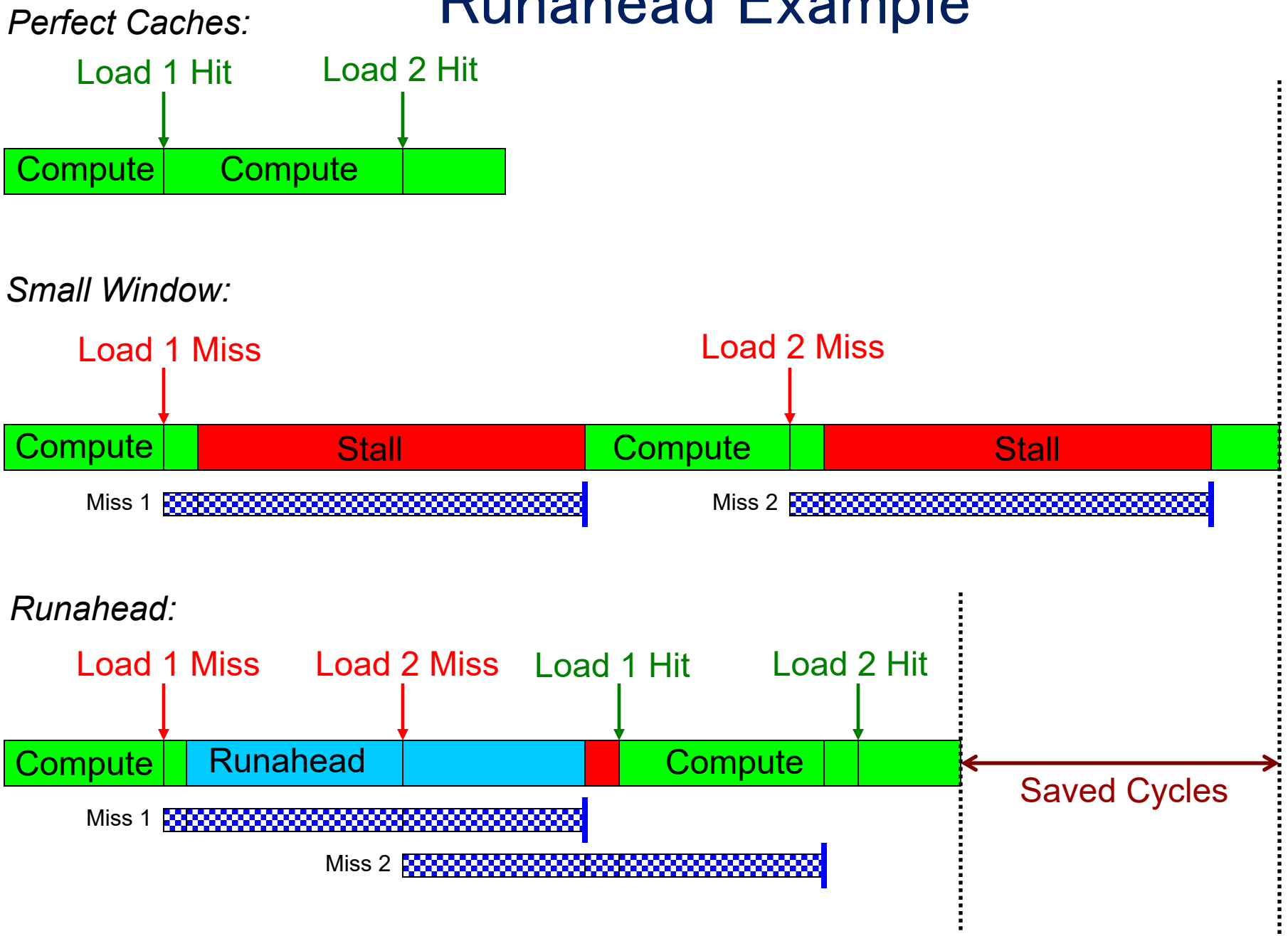
# Runahead Execution

- 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



# 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

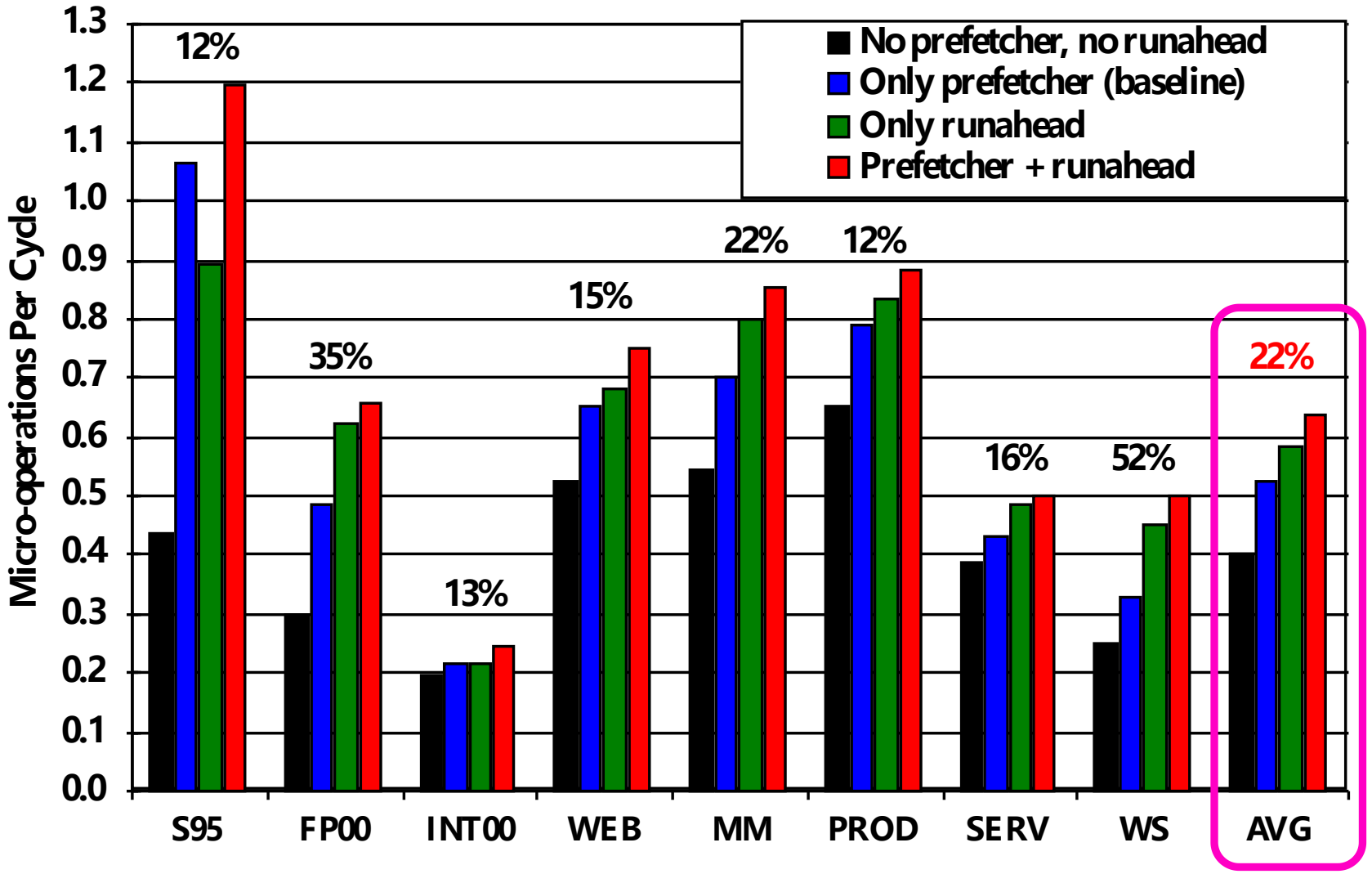
# Runahead Execution Pros and Cons

- 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

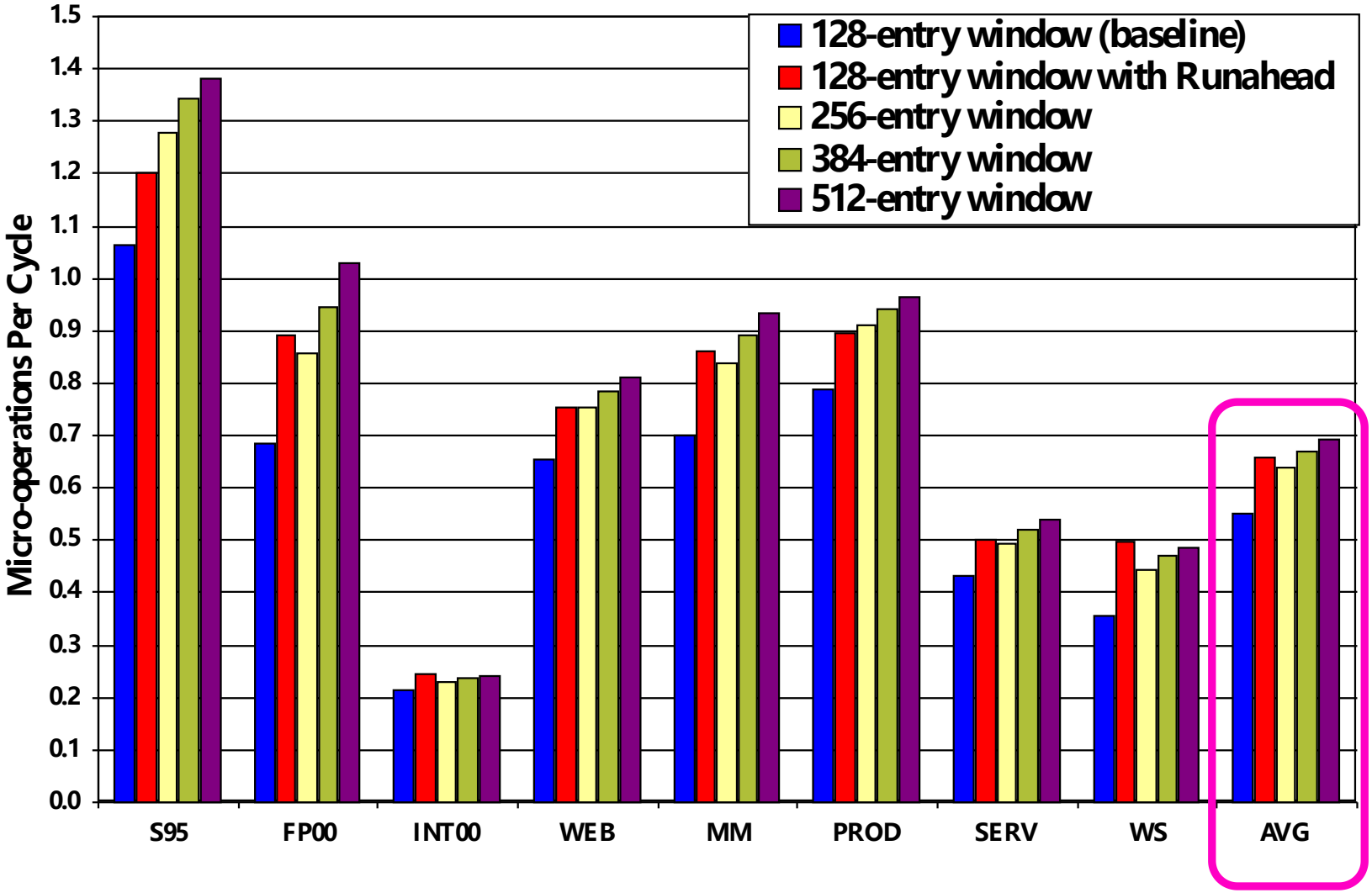
- Disadvantages/Limitations
  - -- Extra executed instructions
  - -- Limited by branch prediction accuracy
  - -- Cannot prefetch dependent cache misses
  - -- Prefetch distance (how far ahead to prefetch) limited by memory latency

- Implemented in Sun ROCK, IBM POWER6, NVIDIA Denver

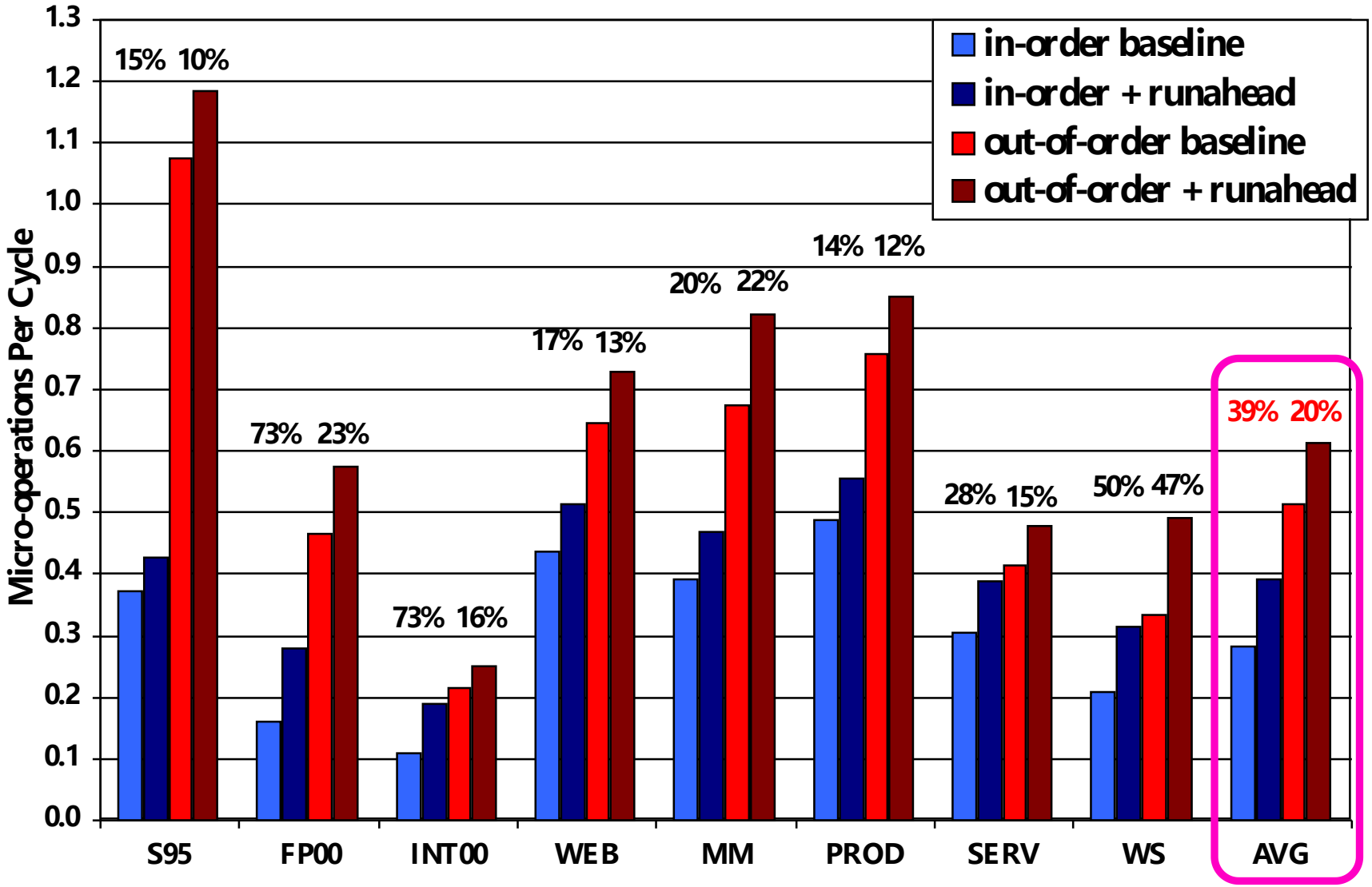
# Performance of Runahead Execution



# Runahead Execution vs. Large Windows

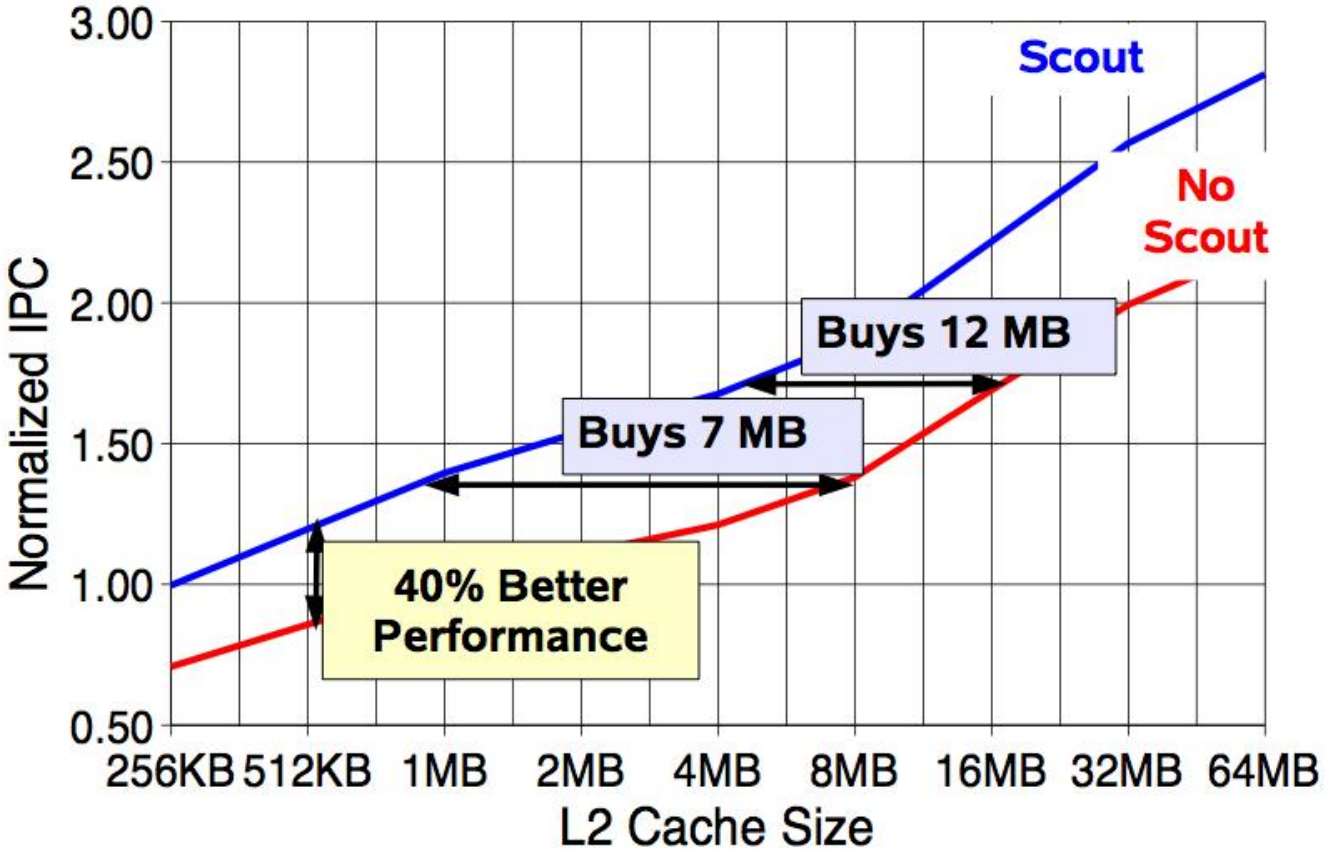


# Runahead on In-order vs. Out-of-order



# Effect of Runahead in Sun ROCK

- Shailender Chaudhry talk, Aug 2008.




**Effective prefetching can both improve performance and reduce hardware cost**

# 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]